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 2, 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 Introduction **4**

2 Background **5**
2.1 Automatic differentiation 5
2.2 Denotational semantics 6
2.3 Coq 8
2.3.1 Language representation 8
2.3.2 Dependently-typed programming in Coq 10
2.4 Logical relations 12

3 Formalizing Forward-Mode AD **13**
3.1 Simply Typed Lambda Calculus 13
3.2 Adding Sums and Primitive Recursion 28
3.3 Arrays 28

4 Optimization **28**
4.1 Program Transformations 28

5 Reverse-Mode AD **28**

6 Discussion **28**
6.1 Problems 28
6.2 Future Work 28

7 Conclusion **28**

A Language Definitions **28**

B Forward-Mode Macro **28**

C Denotations **28**

1 Introduction

AI and machine learning research has sparked a lot of new interest in recent times. It has been used in fields such as computer vision, natural language processing, and as opponents in various games such as chess and Go. In machine learning and more specifically neural network research, researchers set up functions referred to as layers between the input and output data and through an algorithm called back propagation, try to optimize the network such that it learns how to solve the problem implied by the data. Back propagation makes heavy use of automatic differentiation, but programming in an environment which allows for automatic differentiation can be limited.

Frameworks such as Tangent[1] or autograd[2] make use of source code transformations and operator overloading, which can restrict which high-level optimizations one is able to apply to generated code. Support for higher-order derivatives is also limited.

Programming language research has a rich history with many well-known both high- and low-level optimization techniques such as partial evaluation and deforestation. If instead of a framework, we were to have a programming language that is able to facilitate automatic differentiation, we would be able to apply many of these techniques. Through the use of higher-order functions and type systems, we would also get additional benefits such as code-reusability and correctness.

In this thesis, we will aim to formalize an extendable correctness proof of an implementation of automatic differentiation on a simply-typed lambda calculus in the **Coq** proof assistant, opening up further possibilities for formally proving the correctness of more complex language features in the future. Our formalization is based on a recent proof by Huot, Staton, and Vákár [28]. They proved, using a denotational model of diffeological spaces, that their forward mode emulating macro is correct when applied to a simply-typed lambda calculus with products, co-products and inductive types.

With this thesis we will aim for the following goals:

- Formalize the proofs of both the forward mode and continuation-based automatic differentiation algorithms specified by Huot, Staton, and Vákár [28] in **Coq**.
- Prove that well-known compile-time optimizations such as the partial evaluation, are correct with respect to the semantics of automatic differentiation.

[1] https://github.com/google/tangent

[2] https://github.com/HIPS/autograd

- Extend the proof with the array types and compile-time optimization rules by Shaikhha, et. al.[22].

Chapter 2 includes a background section explaining many of the topics and techniques used in this thesis. The formalization of forward-mode automatic differentiation is given in Chapter 3, starting from a base simply-typed lambda calculus extended with product types and incrementally adding new types and language constructs. Chapters 4 and 5 give formalizations of optimization avenues using respectively program transformations and continuation-based reverse-mode automatic differentiation.

As a notational convention, we will use specialized notation in the definitions themselves. Coq normally requires that pretty printed notations be defined separately from the definitions they reference. The letter Γ is used for typing contexts while lowercase Greek letters are usually used for types.

2 Background

2.1 Automatic differentiation

One of the principal techniques used in machine learning is back propagation, which calculates the gradient of a function. The gradient itself is used in the gradient descent algorithm to optimize an objective function by determining the direction of steepest descent[18]. Automatic differentiation has a long and rich history, where its driving motivation is to be able to automatically calculate the derivative of a function in a manner that is both correct and fast. Through techniques such as source-code transformations or operator overloading, one is able to implement an automatic differentiation algorithm which can transform any program which implements some function to one that calculates its derivative. So in addition to the standard semantics present in most programming languages, concepts relevant to differentiation such as derivative values and the chain rule are needed.

Automatic or algorithmic differentiation is beneficial over other methods of automatically calculating the derivatives of functions such as numerical differentiation or symbolic differentiation due to its balance between speed and computational complexity. There are two main variants of automatic differentiation, namely forward mode and reverse mode automatic differentiation.

In forward mode automatic differentiation every term in the function trace is annotated with the corresponding derivative of that term. These are also known as the respectively the primal and tangent traces. So every

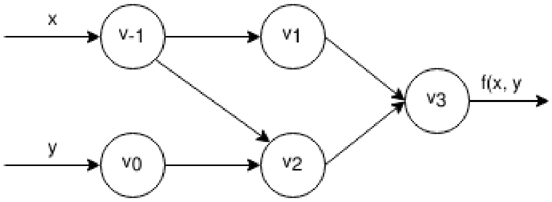

Figure 1: Computational graph of $f(x, y) = x^2 + (x - y)$

partial derivative of every sub-function is calculated parallel to its counterpart. We will take the function $f(x, y) = x^2 + (x - y)$ as an example. The dependencies between the terms and operations of the function is visible in the computational graph in Figure 1. The corresponding traces are filled in Table 1 for the input values $x = 2, y = 1$. We can calculate the partial derivative $\frac{\delta f}{\delta x}$ at this point by setting $x' = 1$ and $y' = 0$. In this paper we will prove the correctness of a simple forward mode automatic differentiation algorithm with respect to the semantics of a simply-typed lambda calculus.

Reverse mode automatic differentiation takes a different approach. It tries to work backwards from the output by annotating each intermediate variable v_i with an adjoint $v'_i = \frac{\delta y_i}{\delta v_i}$. To do this, two passes are necessary. Like the forward mode variant the primal trace is needed to determine the intermediate variables and function dependencies. These are recorded in the first pass. The second pass actually calculates the derivatives by working backwards from the output using the adjoints, also called the adjoint trace.

The choice between automatic differentiation variant is heavily dependent on the function being differentiated. The number of applications of the forward mode algorithm is dependent on the number of input variables, as it has to be redone for each possible partial derivative of the function. On the other hand, reverse mode AD has to work backwards from each output variable. In machine learning research, reverse mode AD is generally preferred as the objective functions regularly contain a very small number of output variables. How one does reverse mode automatic differentiation on a functional language is still an active area of research. Huot, Staton and Vákár have proposed a continuation-based algorithm which mimic much of the same ideas as reverse mode automatic differentiation[28].

2.2 Denotational semantics

The notion of denotational semantics tries to find underlying mathematical models able to underpin the concepts known in programming languages.

Primal trace					Tangent trace				
v_{-1}	$=$	x	$=$	2	v'_{-1}	$=$	x'	$=$	1
v_0	$=$	y	$=$	1	v'_0	$=$	y'	$=$	0
v_1	$=$	v_{-1}^2	$=$	4	v'_1	$=$	$2 * v_{-1}$	$=$	4
v_2	$=$	$v_{-1} - v_0$	$=$	1	v'_2	$=$	$v'_{-1} - v'_0$	$=$	1
v_3	$=$	$v_1 + v_2$	$=$	5	v'_3	$=$	$v'_1 + v'_2$	$=$	5
f	$=$	v_3	$=$	5	f'	$=$	v'_3	$=$	5

Table 1: Primal and tangent traces of $f(x, y) = x^2 + (x - y)$

The most well-known example is the solution given by Dana Scott and Christopher Strachey[1] for lambda calculi, also called domain theory. To be able to formalize non-termination and partiality, they thought to use concepts such as partial orderings and least fixed points[26]. In this model, programs are interpreted as partial functions, and recursive computations as taking the fixpoint of such functions. Non-termination, on the other hand, is formalized as a value `bottom` that is lower in the ordering relation than any other element.

In our specific case, we try to find a satisfactory model we can use to show that our implementation of forward mode automatic differentiation is correct when applied to a simply-typed lambda calculus. In the original pen and paper proof of automatic differentiation this thesis is based on, the mathematical models used were diffeological spaces, which are a generalization of smooth manifolds. For the purpose of this thesis, however, we were able to avoid using diffeological spaces as recursion, iteration and concepts dealing with non-termination and partiality are left out of the scope of this thesis. **Coq** has very limited support for domain theoretical models. There are possible libraries which have resulted from experiments trying to encode domain theoretical models[12][17], but these are incompatible with recent versions of **Coq**. As a part of its type system, **Coq** contains a set-theoretical model available under the sort `Set`, which is satisfactory as the denotational semantics for our language.

Because we use the real numbers as the ground type in our language, we also needed an encoding of the real numbers in Coq. The library for real numbers in **Coq** has improved in recent times from one based on a completely axiomatic definition to one involving Cauchy sequences[3] . For the purposes of this thesis, however, we needed differentiability as the denotational result of applying the macro operation. Instead of encoding this by hand, we opted

[3] https://coq.inria.fr/library/Coq.Reals.ConstructiveCauchyReals.html

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

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

Figure 2: Type-inferrence rules for a simply-typed lambda calculus using De-Bruijn indices

for the more comprehensive library `Coquelicot`[19], which contains many general definitions for differentiating functions.

2.3 Coq

Coq is a proof assistant based on the calculus of constructions type theory created by Thierry Coquand and Gérard Huet[2]. In the past 30 years since it has been released, research has contributed to extending the proof assistant with additional features such as inductive and co-inductive data types[3], dependent pattern matching[13] and advanced modular constructions for organizing large mathematical proofs[11][16].

The core of this type theory is based on constructive logic and so many of the laws known in classical logic are not provable. An example includes the law of the excluded middle, $\forall A, A \vee \neg A$. In some cases they can, however, be safely added to **Coq** without making its logic inconsistent. These are readily available in the standard library. Due to its usefulness in proving propositions over functions, we will make use of the functional extensionality axiom in **Coq**.

2.3.1 Language representation

When defining a simply-typed lambda calculus, there are two main possibilities[25]. The arguably simpler variant, known as an extrinsic representation, is traditionally the one introduced to new students learning **Coq**. In the extrinsic representation, the terms themselves are untyped and typing judgments are defined separately as relations between the types and terms. A basic example of working with this is given in Software Foundations[20]. This, however, required many additional lemmas and machinery to be proved to be able to work with both substitutions and contexts as these are defined separate from the terms. As an example, the preservation property which states that reduction does not change the type of a term, needs to be proven explicitly. The other approach, also called an intrinsic representation, makes use of just a single well-typed definition. Ill-typed terms are made impossible by the type checker. This representation, while beneficial in the proof load, however complicates much of the normal machinery involved in programming language theory. One example is how one would define operations such as substitutions or weakening.

But even when choosing an intrinsic representation, the problem of variable binding persists. Much meta-theoretical research has been done on possible approaches to this problem each with their own advantages and disadvantages. The POPLmark challenge gives a comprehensive overview of each of the possibilities in various proof assistants[7]. An example of an approach is the nominal representation where every variable is named. While this does follow the standard format used in regular mathematics, problems such as alpha-conversion and capture-avoidance appears.

```
Inductive ty : Type :=
  | unit : ty
  | ⇒ : ty → ty → ty.

Inductive tm : Type :=
  | var : string → tm
  | abs : string → ty → tm → tm
  | app : tm → tm → tm.
```

Code snippet 1: Simply typed λ-calculus using an extrinsic nominal representation.

The approach used in the rest of this thesis is an extension of the De-Bruijn representation which numbers variables relative to the binding lambda term. In this representation the variables are referred to as well-typed De-Bruijn indices. A significant benefit of this representation is that the problems of capture avoidance and alpha equivalence are avoided. As an alternative, instead of using numbers to represent the distance, indices within the typing context can be used to ensure that a variable is always well-typed and well-scoped. While the idea of using type indexed terms has been both described and used by many authors[5][6][8], the specific formulation used in this thesis using separate substitutions and rename operations was fleshed out in Coq by Nick Benton, et. al.[14], and was also used as one of the examples in the second POPLmark challenge which deals with logical relations[21]. While this does avoid the problems present in the nominal representation, it unfortunately does have some problems of its own. Variable substitutions have to be defined using two separate renaming and substitution operations. Renaming is formulated as extending the typing context of variables, while substitution actually swaps the variables for terms. Due to using indices from the context as variables, some lifting boilerplate is also needed to manipulate contexts.

```
Inductive τ ∈ Γ : Type :=
  | Top : ∀ Γ τ, τ ∈ (τ::Γ)
  | Pop : ∀ Γ τ σ, τ ∈ Γ → τ ∈ (σ::Γ).

Inductive tm Γ τ : Type :=
  | var : ∀ Γ τ, τ ∈ Γ → tm Γ τ
  | abs : ∀ Γ τ σ, tm (σ::Γ) τ → tm Γ (σ ⇒ τ)
  | app : ∀ Γ τ σ, tm Γ (σ ⇒ τ) → tm Γ σ → tm Γ τ.
```

Code snippet 2: Basis of a simply-typed λ-calculus using a strongly typed intrinsic formulation.

2.3.2 Dependently-typed programming in Coq

In **Coq**, one can normally write function definitions using either case-analysis as is done in other functional languages, or using **Coq**'s tactics language. Using the standard case-analysis functionality can cause the code to be complicated and verbose, even more so when proof terms are present in the function signature. This has been caused by the previously poor support in Coq for dependent pattern matching. Using the return keyword, one is able to vary the result type of a match expression. But due to requirement Coq used to have that case expressions be syntactically total, this could be very annoying to work with. One other possibility would be to write the function as a relation between its input and output. This also has its limitations as you then lose computability as Coq treats these definitions opaquely. In this case the standard `simpl` tactic which invokes **Coq**'s reduction mechanism is not able to reduce instances of the term. This often requires the user to write many more proofs to be able to work with the definitions.

As an example, we will work through defining a length indexed list and a corresponding head function limited to lists of length at least one in Snippet 3. Using the **Coq** keyword return, it is possible to let the return type

```
Inductive ilist : Type → nat → Type :=
  | nil : ∀ A, ilist A 0
  | cons : ∀ A n, A → ilist A n → ilist A (S n)

Definition hd' {A} n (ls : ilist A n) :=
  match ls in (ilist A n) return
    (match n with
    | O => unit
    | S _ => A end) with
  | nil => tt
  | cons h _ => h
end.

Definition hd {A} n (ls : ilist A (S n)) : A := hd' n ls.
```

Code snippet 3: Definition of a length indexed list and hd using the return keyword, adapted from Certified Programming with Dependent Types[15].

of a match expressions depend on the result of one of the type arguments. This makes it possible to define an auxiliary function which, while total on the length of the list, has an incorrect return type. It namely returns the type unit if the input list had the length zero. We can then use this auxiliary function in the actual head function by specifying that the list has length at least one. It should be noted that more recent versions of Coq do not require that case expressions be syntactically total, so specifying that the input list has a length of at least zero is enough to eliminate the requirement for the zero-case.

Mathieu Sozeau introduces an extension to **Coq** via a new keyword `Program` which allows the use of case-analysis in more complex definitions[29][10]. To be more specific, it allows definitions to be specified separately from their accompanying proofs, possibly filling them in automatically if possible. While this does improve on the previous situation, using the definitions in proofs can often be unwieldy due to the amount of boilerplate introduced. This makes debugging error messages even harder than it already is in a proof assistant. This approach was used by Benton in his formulation of strongly typed terms.

Sozeau further improves on this introducing a method for user-friendlier dependently-typed pattern matching in **Coq** in the form of the `Equations` library[13][24]. This introduces **Agda**-like dependent pattern matching with with-clauses. It does this by using a notion called coverings, where a

```
Equations hd {A n} (ls : ilist A n) (pf : n <> 0) : A :=
hd nil pf with pf eq_refl := {};
hd (cons h n) _ := h.
```

Code snippet 4: Definition of hd using `Equations`

covering is a set of equations such that the pattern matchings of the type signature are exhaustive. There are two main ways to integrate this in a dependently typed environment, externally where it is integrated as high-level constructs in the pattern matching core as **Agda** does it, or internally by using the existing type theory and finding witnesses of the covering to prove the definition correct, which is the approach used by Sozeau. Due to the intrinsic typeful representation this paper uses, much of this was invaluable when defining the substitution operators as the regular type checker in Coq often had difficulty unifying dependently typed terms in certain cases.

2.4 Logical relations

Logical relations is a technique often employed when proving programming language properties of statically typed languages[23]. There are two main ways they are used, namely as unary and binary relations. Unary logical relations, also known as logical predicates, are predicates over single terms and are typically used to prove language characteristics such as type safety or strong normalization. Binary logical relations on the other hand are used to prove program equivalences, usually in the context of denotational semantics as we will do. There have been many variations on the versatile technique from syntactic step-indexed relations which have been used to solve recursive types[9], to open relations which enable working with terms of non-ground type[27][28]. Logical relations in essence are relations between terms defined by induction on their types. A logical relations proof consists of 2 main steps. The first states the terms for which the property is expected to hold are in the relation, while the second states that the property of interest follows from the relation. The second step is easier to prove as it usually follows from the definition of the relation. The first on the other hand, will often require proving a generalized variant, called the fundamental property of the logical relation. In most cases this requires that the relation is correct with respect to applying substitutions.

A well-known logical relations proof is the proof of strong normalization of well-typed terms, which states that all terms eventually terminate. An example of a logical relation used in such a proof using the intrinsic strongly-

```
Equations SN {Γ} τ (t : tm Γ τ): Prop :=
SN unit t := halts t;
SN (τ ⇒ σ) t := halts t ∧
  (∀ (s : tm Γ τ), SN τ s → SN σ (app Γ σ τ t s));
```

Code snippet 5: Example of a logical predicate used in a strong normalizations proof in the intrinsic strongly-typed formulation

typed formulation is given in Snippet 5. Noteworthy is the case for function types, which indicates that an application should maintain the strongly normalization property. If one were to attempt the proof of strong normalization without using logical relations, they would get stuck in the cases dealing with function types. More specifically when reducing an application, the induction hypothesis is not strong enough to prove that substituting the argument into the body of the abstraction also results in a terminating term. The proof given in the paper this thesis is based on, is a logical relations proof on the denotation semantics using diffeological spaces as its domains[28]. A similar, independent proof of correctness was given by Barthe, et. al.[27] using a syntactic relation on the operational semantics.

3 Formalizing Forward-Mode AD

We will work out the formalization of automatic differentiation using a source code translating macro in the following sections. We start from a base simply-typed lambda calculus extended with product types and incrementally add both sum and array types.

3.1 Simply Typed Lambda Calculus

As mentioned in background section 2.3.1, we will make use of De-Bruijn indices in a intrinsic representation to formulate our language. Our base language consists of the typed lambda calculus with products and real numbers as ground type, $\Lambda_{\delta}^{\times,\to,R}$. As operations on the real numbers we include both addition and multiplication which are also the functions in δ.

Both the language constructs and the typing rules for this language are standard for a simply typed lambda calculus, as shown in 3. 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 fetch respectively the first and second com-

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

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

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

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

$$\frac{r \in \mathbb{R}}{\Gamma \vdash \mathit{rval}\ r : R}\ \textsc{TRval}$$

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

Figure 3: Type-inferrence rules for $\Lambda_{\delta}^{\times,\to,\mathbb{R}}$

ponents of tuples. For real numbers, rval is used to introduce constants and add and mul will be used to respectively encode addition and multiplication.

These can be translated into Coq definitions in a straigtforward 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[14][4], instead of using numbers accompanied with a proof, we make use of an inductively defined type evidence to type our variables as shown in 2. The cases for app and abs are as expected, where variables in the body of abstraction are able to reference their respective arguments.

In the original proof by Huot, Staton, and Vákár [28], 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 with product types.

```
Definition Ctx : Type := list ty.

Inductive tm (Γ : Ctx) : ty → Type :=
  (* Base STLC *)
  | var : forall τ,
    τ ∈ Γ → tm Γ τ
  | app : forall τ σ,
    tm Γ (σ ⇒ τ) →
    tm Γ σ →
    tm Γ τ
  | abs : forall τ σ,
    tm (σ::Γ) τ → tm Γ (σ ⇒ τ)

  (* Operations on real numbers *)
  | const : R → tm Γ Real
  | add : tm Γ Real → tm Γ Real → tm Γ Real
  | mul : tm Γ Real → tm Γ Real → tm Γ Real

  (* 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
```

Code snippet 6: **Coq** definition of the base lambda calculus

Code snippet 7: Forward-mode macro on the base simply-typed lambda calculus.

We use the same inductively defined macro on types and terms to implement forward-mode automatic differentiation as used by many previous authors[28][27][22]. The forward-mode macro $\overrightarrow{\mathcal{D}}$ keeps track of both primal and tangent traces using tuples as respectively the first and second subcomponents. In most cases, the macro simply preserves the structure of the language. In the cases for real numbers such as addition and multiplication, the appropriate implementation corresponding to the derivative needs to be given.

$$\begin{aligned}
\overrightarrow{\mathcal{D}}(\mathbb{R}) &= \mathbb{R} \times \mathbb{R} & \overrightarrow{\mathcal{D}}(n) &= (n, 0) \\
\overrightarrow{\mathcal{D}}(\tau \times \sigma) &= \overrightarrow{\mathcal{D}}(\tau) \times \overrightarrow{\mathcal{D}}(\sigma) & \overrightarrow{\mathcal{D}}(n + m) &= (n + m, n' + m') \\
\overrightarrow{\mathcal{D}}(\tau \to \sigma) &= \overrightarrow{\mathcal{D}}(\tau) \to \overrightarrow{\mathcal{D}}(\sigma) & \overrightarrow{\mathcal{D}}(n * m) &= (n * m, n' * m + m' * n)
\end{aligned}$$

This is implemented by destructing recursive calls to $\mathcal{D}$ to access the syntactic counterparts of the primal and tangent denotations. The **Coq** implementation requires seperate definitions for types, typing contexts and terms, as shown in 7. Note that applying the macro to a variable does nothing as we already apply the macro to the typing context, so variable implicitly reference macro-applied values.

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. Well-typed terms $\Gamma \vdash t : \tau$ will denotate to functions $[\![\Gamma]\!] \to [\![\tau]\!]$.

$$[\![t]\!] = \begin{cases}
\lambda x. lookup\ [\![v]\!]\ x & : t = \texttt{var}\ v \\
\lambda x. ([\![t1]\!](x))([\![t2]\!](x)) & : t = \texttt{app}\ t1\ t2 \\
\lambda xy. [\![t]\!](y :: x) & : t = \texttt{abs}\ t \\
const\ n & : t = \texttt{rval}\ n \\
\lambda x. [\![t1]\!](x) + [\![t2]\!](x) & : t = \texttt{add}\ t1\ t2 \\
\lambda x. [\![t1]\!](x) * [\![t2]\!](x) & : t = \texttt{mul}\ t1\ t2 \\
\lambda x. ([\![t1]\!](x), [\![t2]\!](x)) & : t = \texttt{tuple}\ t1\ t2 \\
\lambda x. let\ (x, y) = [\![t]\!](x)\ in\ x & : t = \texttt{first}\ t \\
\lambda x. let\ (x, y) = [\![t]\!](x)\ in\ y & : t = \texttt{second}\ t
\end{cases} \tag{1}$$

For any type τ, simply swap out the syntactic type to its corresponding **Coq** variant in `Type`. We denotate our typing contexts Γ, lists of types, as

Code snippet 8: Denotatonal semantics for the base simply-typed lambda calculus.

heterogeneous lists containing their corresponding denotations. The specific implementation of heterogeneous lists used, correspond to the one given by Adam Chlipala[15]. When working through giving the constructs in our language the proper denotations, most of the cases are straigtforward. 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 our heterogeneous lists to their appropriate types.

As mentioned by by Barthe, et. al.[27], this small calculus, $\Lambda_{\delta}^{\times,\rightarrow,\mathbb{R}}$, accompanied with a very simple denotational semantics is expressive enough to encode all higher-order polynomial functions 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 $f : \mathbb{R}^n - > \mathbb{R}$ can be interpreted as the denotations of a corresponding syntactic term $x_1, \ldots, x_n \vdash t : R$. The idea here is that 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. Note that while both the arguents and result type of t are restricted to R, t itself can contain higher order types.

Although Barthe, et. al.[27] gave a syntactic proof of correctness of the macro, our proof follows the more denotational style of proof given by Huot, Staton and Vákár[28]. We state correctness as the denotation of a macro-applied term will give a pair of both the original function the term represents along with its corresponding derivative.

The proof of correctness will follow a logical relations argument.Intuitively, the relation will encapsulate idea that derivability is preserved over higher-order types. We define a type-indexed logical 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 $R \to [\![\overrightarrow{\mathcal{D}}(\tau)]\!]$.

$$S_\tau(f,g) = \begin{cases} \mathit{smooth}\ f \wedge g = \lambda x.(f(x), \frac{\partial f}{\partial x}(x)) & : \tau = R \\ \exists f_1, f_2, g_1, g_2, & : \tau = \sigma \times \rho \\ \quad S_\sigma(f_1, f_2), S_\sigma(g_1, g_2). & \\ \quad f = \lambda x.(f_1(x), g_1(x)) \wedge & \\ \quad g = \lambda x.(f_2(x), g_2(x)) & \\ \forall f_1, f_2. & : \tau = \sigma \to \rho \\ \quad S_\sigma(f_1, f_2) \Rightarrow & \\ \quad S_\rho(\lambda x.f(x)(f_1(x)), \lambda x.f(x)(f_2(x))) & \end{cases} \tag{2}$$

The next step involves proving that syntactically well-typed terms are semantically correct. In other words, the relation is valid for any term $x_1, \ldots, x_n \vdash t : \tau$ and argument function $f : R \to R^n$ such that $S_\tau([\![t]\!] \circ f, [\![\overrightarrow{\mathcal{D}}(t)]\!] \circ \overrightarrow{\mathcal{D}}_f \circ f)$. Note the need to apply a function $\overrightarrow{\mathcal{D}}_f$ which essentially accompanies each argument supplied by f with its accompanying derivative, 0.

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*

$$inst_\Gamma(f, g) = \begin{cases} f = const([]) \wedge g = const([]) & : \Gamma = [] \\ \forall s, s_D. & : \Gamma = (\tau :: \Gamma') \\ \quad inst_{\Gamma'}(s, s_D) \wedge S_\tau(f, g) & \end{cases} \qquad (3)$$

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

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

Proof. This is proven by induction on the typing derivation of t. Unless otherwise specified, the type of s and s_D are respectively $R \to [\![\Gamma]\!]$ and $R \to [\![\overrightarrow{\mathcal{D}}(\Gamma)]\!]$.

1. `(var)`

   Prove: $S_\tau([\![var\ v]\!] \circ s, [\![\overrightarrow{\mathcal{D}}(var\ v)]\!] \circ s_D)$.

   Proceed by induction on the type evidence v.

   - `(Top)` Base case

     Prove: $S_\tau([\![var\ Top]\!] \circ s, [\![\overrightarrow{\mathcal{D}}(var\ Top)]\!] \circ s_D)$, where $s : R \to [\![\tau :: \Gamma]\!]$ and $s_D : R \to [\![\tau :: \Gamma]\!]$

     In this case the referenced τ exists at the top of the list. So both $[\![var\ Top]\!]$ and $[\![\overrightarrow{\mathcal{D}}(var\ Top)]\!]$ denotate to fetching the top term. This is now proven by definition of $inst$, which states that the the term is semantically well-typed.

     $$\begin{aligned} & S_\tau([\![var\ Top]\!] \circ s, [\![\overrightarrow{\mathcal{D}}(var\ Top)]\!] \circ s_D) \\ & \quad \Vdash \text{(Definition of } \overrightarrow{\mathcal{D}}) \\ & \quad S_\tau([\![var\ Top]\!] \circ s, [\![var\ Top]\!] \circ s_D) \\ & \quad \Vdash \text{(Definition of } \circ) \\ & \quad S_\tau(\lambda x.[\![var\ Top]\!](s(x)), \lambda x.[\![var\ Top]\!](s_D(x))) \\ & \quad \Vdash \text{(Definition of } [\![]\!]) \\ & \quad S_\tau(\lambda x.lookup\ [\![Top]\!](s(x)), \lambda x.lookup\ [\![Top]\!](s_D(x))) \end{aligned}$$

$\Vdash$ (Rewrite using $s = \lambda x.hd(s(x)) :: tl(s(x))$)

$S_\tau(\lambda x.lookup\ [\![Top]\!](hd(s(x)) :: tl(s(x))),$
$\lambda x.lookup\ [\![Top]\!](hd(s_D(x)) :: tl(s_D(x))))$

$\Vdash$ (Simplify with lookup and $[\![Top]\!]$)

$S_\tau(\lambda x.hd(s(x)), \lambda x.hd(s_D(x)))$

$\Vdash$ (By definition of $inst_{\tau::\Gamma}$)

□

- (Pop) Induction step

  Prove: $S_\tau([\![var\ (Pop\ v)]\!] \circ s, [\![\overrightarrow{\mathcal{D}}(var\ (Pop\ v))]\!] \circ s_D)$, where $s : R \to [\![\sigma :: \Gamma]\!]$ and $s_D : R \to [\![\overrightarrow{\mathcal{D}}(\sigma :: \Gamma)]\!]$.

  Induction hypothesis:

  (a) $\forall (f : R \to [\![\Gamma]\!]), (g : R \to [\![\overrightarrow{\mathcal{D}}(\Gamma)]\!]).$
  $S_\tau([\![var\ v]\!] \circ f, [\![\overrightarrow{\mathcal{D}}(var\ v)]\!] \circ g)$

  Note that the var term now denotates to ignoring the arbitrary unrelated type σ and looking up v in the rest of the list Γ. So $S_\tau([\![var\ v]\!] \circ tl \circ s, [\![\overrightarrow{\mathcal{D}}(var\ v)]\!] \circ tl \circ s_D)$, which is proven using the induction hypothesis by respectively instantiating f and g as $tl \circ s$ and $tl \circ s_D$.

$S_\tau([\![var\ (Pop\ v)]\!] \circ s, [\![\overrightarrow{\mathcal{D}}(var\ (Pop\ v))]\!] \circ s_D)$

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

$S_\tau([\![var\ (Pop\ v)]\!] \circ s, [\![var\ (Pop\ v)]\!] \circ s_D)$

$\Vdash$ (Definition of $\circ$)

$S_\tau(\lambda x.[\![var\ (Pop\ v)]\!](s(x)), \lambda x.[\![var\ (Pop\ v)]\!](s_D(x)))$

$\Vdash$ (Definition of $[\![]\!]$)

$S_\tau(\lambda x.lookup\ [\![Pop\ v]\!](s(x)), \lambda x.lookup\ [\![Pop\ v]\!](s_D(x)))$

$\Vdash$ (Rewrite using $s = \lambda x.hd(s(x)) :: tl(s(x))$)

$S_\tau(\lambda x.lookup\ [\![Pop\ v]\!](hd(s(x)) :: tl(s(x))),$
$\lambda x.lookup\ [\![Pop\ v]\!](hd(s_D(x)) :: tl(s_D(x))))$

$\Vdash$ (Simplify with lookup and $[\![Pop\ v]\!]$)

$S_\tau(\lambda x.lookup\ [\![v]\!](tl(s(x))), \lambda x.lookup\ [\![v]\!](tl(s_D(x))))$

$\Vdash$ (Use IH. 1 with $f = tl(s(x))$ and $g = tl(s_D(x)))$

□

2. (app)

   Prove: $S_\tau([\![app\ t_1\ t_2]\!] \circ s, [\![\overrightarrow{\mathcal{D}}(app\ t_1\ t_2)]\!] \circ s_D)$

   Induction hypotheses:

   (a) $S_{\sigma\to\tau}([\![t_1]\!] \circ s, [\![\overrightarrow{\mathcal{D}}(t_1)]\!] \circ s_D)$

   (b) $S_\sigma([\![t_2]\!] \circ s, [\![\overrightarrow{\mathcal{D}}(t_2)]\!] \circ s_D)$

   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.

$S_{\sigma\to\tau}([\![t_1]\!] \circ s, [\![\overrightarrow{\mathcal{D}}(t_1)]\!] \circ s_D)$

$\Vdash$ (Definition of $\circ$)

$S_{\sigma\to\tau}(\lambda x.[\![t_1]\!](s(x)), \lambda x.[\![\overrightarrow{\mathcal{D}}(t_1)]\!](s_D(x)))$

$\Vdash$ (Definition of $S_\to$)

$\forall f_1, f_2.S_\sigma(f1, f2) \to$

$S_\tau(\lambda x.([\![t_1]\!](s(x)))(f_1(x)), \lambda x.([\![\overrightarrow{\mathcal{D}}(t_1)]\!](s_D(x)))(f_2(x)))$

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 satify its premise.

$S_\tau([\![app\ t_1\ t_2]\!] \circ s, [\![\overrightarrow{\mathcal{D}}(app\ t_1\ t_2)]\!] \circ s_D)$

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

$S_\tau([\![app\ t_1\ t_2]\!] \circ s, [\![app\ \overrightarrow{\mathcal{D}}(t_1)\ \overrightarrow{\mathcal{D}}(t_2)]\!] \circ s_D)$

$\Vdash$ (Definition of $\circ$)

$S_\tau(\lambda x.[\![app\ t_1\ t_2]\!](s(x)), \lambda x.[\![app\ \overrightarrow{\mathcal{D}}(t_1)\ \overrightarrow{\mathcal{D}}(t_2)]\!](s_D(x)))$

$\Vdash$ (Definition of $[\![]\!]$)

$S_\tau(\lambda x.([\![t_1\ ]\!](s(x)))([\![t_2]\!](s(x))), \lambda x.([\![\overrightarrow{\mathcal{D}}(t_1)\ ]\!](s_D(x)))([\![\overrightarrow{\mathcal{D}}(t_2)]\!](s_D(x)))$

$\Vdash$ (Induction hypothesis 2a)

$S_\sigma(\lambda x.[\![t_2]\!](s(x)), \lambda x.[\![\overrightarrow{\mathcal{D}}(t_2)]\!] \circ (s_D(x)))$
$\Vdash$ (Induction hypothesis 2b)

□

3. (abs)

   Prove: $S_{\sigma\to\tau}([\![abs\ t]\!] \circ s, [\![\overrightarrow{\mathcal{D}}(abs\ t)]\!] \circ s_D)$

   Induction hypothesis:

   (a) $S_\sigma([\![t]\!] \circ s, [\![\overrightarrow{\mathcal{D}}(t)]\!] \circ s_D)$, where $s : R \to [\![\sigma :: \Gamma]\!]$ and $s_D : R \to [\![\sigma :: \Gamma]\!]$

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

   Prove: $S_\tau(\lambda x.([\![abs\ t]\!](s(x)))(f_1(x)), \lambda x.([\![\overrightarrow{\mathcal{D}}(abs\ t)]\!](s_D(x)))(f_2(x)))$

   Assume:

   (a) $f_1 : R \to [\![\sigma]\!]$
   (b) $f_2 : R \to [\![\overrightarrow{\mathcal{D}}(\sigma)]\!]$
   (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.

   $S_\tau(\lambda x.([\![abs\ t]\!](s(x)))(f_1(x)), \lambda x.([\![\overrightarrow{\mathcal{D}}(abs\ t)]\!](s_D(x)))(f_2(x)))$
   $\Vdash$ (Definition of $\overrightarrow{\mathcal{D}}$)
   $S_\tau(\lambda x.([\![abs\ t]\!](s(x)))(f_1(x)), \lambda x.([\![abs\ \overrightarrow{\mathcal{D}}(t)]\!](s_D(x)))(f_2(x)))$
   $\Vdash$ (Definition of $[\![]\!]$)
   $S_\tau(\lambda x.([\![t]\!](f_1(x) :: s(x))), \lambda x.([\![\overrightarrow{\mathcal{D}}(t)]\!](f_2(x) :: s_D(x))))$
   $\Vdash$ (Induction hypothesis 2a)

   □

4. (rval)

   Prove: $S_R([\![rval\ n]\!] \circ s, [\![\overrightarrow{\mathcal{D}}(rval\ n)]\!] \circ s_D)$

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.

$S_R([\![rval\ n]\!] \circ s, [\![\overrightarrow{\mathcal{D}}(rval\ n)]\!] \circ s_D)$

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

$S_R([\![rval\ n]\!] \circ s, [\![tuple\ (rval\ n)\ (rval\ 0)]\!] \circ s_D)$

$\Vdash$ (Definition of $[\![]\!]$)

$S_R(const\ n, (const\ n, const\ 0))$

$\Vdash$ (Definition of S_R)

$smooth\ (const\ n) \wedge const\ 0 = {}^{\partial const\ n}/_{\partial x}$

$\Vdash$ (split goals: goal 1)

$smooth\ (const\ n)$

$\Vdash$ ($f(x) = n$ is continuously differentiable)

$\Vdash$ (split goals: goal 2)

$const\ 0 = {}^{\partial const\ n}/_{\partial x}$

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

□

5. (`add`)

Prove: $S_R([\![add\ t_1\ t_2]\!] \circ s, [\![\overrightarrow{\mathcal{D}}(add\ t_1\ t_2)]\!] \circ s_D)$

Induction hypotheses:

(a) $S_R([\![t_1]\!] \circ s, [\![\overrightarrow{\mathcal{D}}(t_1)]\!] \circ s_D)$

(b) $S_R([\![t_2]\!] \circ s, [\![\overrightarrow{\mathcal{D}}(t_2)]\!] \circ s_D)$

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

$S_R([\![add\ t_1\ t_2]\!] \circ s, [\![\overrightarrow{\mathcal{D}}(add\ t_1\ t_2)]\!] \circ s_D)$

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

$S_R([\![add\ t_1\ t_2]\!] \circ s, [\![tuple$

$(add\ (first\ \overrightarrow{\mathcal{D}}(t_1))(first\ \overrightarrow{\mathcal{D}}(t_2)))$

$(add\ (second\overrightarrow{\mathcal{D}}(t_1))(second\overrightarrow{\mathcal{D}}(t_2))))]\!] \circ s_D)$

$\Vdash$ (Definition of $[\![]\!]$, using

$(d_1, d_1') = [\![\overrightarrow{\mathcal{D}}(t_1)]\!]s(x)$ and $(d_2, d_2') = [\![\overrightarrow{\mathcal{D}}(t_2)]\!]s_D(x))$

$S_R(\lambda x.d_1(x) + d_2(x), \lambda x.(d_1(x) + d_2(x), d_1'(x) + d_2'(x)))$

$\Vdash$ (Definition of S_R)

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

$\lambda x.d_1'(x) + d_2'(x) = {}^{\partial(\lambda x.d_1'(x)\,+\,d_2'(x))}/_{\partial x}$

$\Vdash$ (split goals: goal 1)

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

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

$smooth\ d_1 \wedge smooth\ d_2$

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

$\Vdash$ (split goals: goal 2)

$\lambda x.d_1'(x) + d_2'(x) = {}^{\partial(\lambda x.d_1'(x)\,+\,d_2'(x))}/_{\partial x}$

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

$d_1' = {}^{\partial d_1}/_{\partial x} \wedge d_2' = {}^{\partial d_2}/_{\partial x}$

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

□

6. `(mul)`

Prove: $S_R([\![mul\ t_1\ t_2]\!] \circ s, [\![\overrightarrow{\mathcal{D}}(mul\ t_1\ t_2)]\!] \circ s_D)$

Induction hypotheses:

(a) $S_R([\![t_1]\!] \circ s, [\![\overrightarrow{\mathcal{D}}(t_1)]\!] \circ s_D)$

(b) $S_R([\![t_2]\!] \circ s, [\![\overrightarrow{\mathcal{D}}(t_2)]\!] \circ s_D)$

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

$S_R([\![mul\ t_1\ t_2]\!] \circ s, [\![\overrightarrow{\mathcal{D}}(mul\ t_1\ t_2)]\!] \circ s_D)$

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

$S_R([\![mul\ t_1\ t_2]\!] \circ s, [\![tuple$

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

$(add$

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

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

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

$\quad (d_1, d_1') = \llbracket \overrightarrow{\mathcal{D}}(t_1)\rrbracket s(x)$ and $(d_2, d_2') = \llbracket \overrightarrow{\mathcal{D}}(t_2)\rrbracket s_D(x))$

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

$\quad \lambda x.(d_1(x) * d_2(x), d_1(x) * d_2'(x) + (d_2(x) * d_1'(x))))$

$\Vdash$ (Definition of S_R)

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

$\quad \lambda x.d_1(x) * d_2'(x) + d_2(x) * d_1'(x) = {}^{\partial(\lambda x.(d_1(x) * d_2(x))}/_{\partial x}$

$\Vdash$ (split goals: goal 1)

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

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

$\quad smooth\ d_1 \wedge smooth\ d_2$

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

$\Vdash$ (split goals: goal 2)

$\quad \lambda x.d_1(x) * d_2'(x) + d_2(x) * d_1'(x) = {}^{\partial(\lambda x.(d_1(x) * d_2(x))}/_{\partial x}$

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

$\quad d_1' = {}^{\partial d_1}/_{\partial x} \wedge d_2' = {}^{\partial d_2}/_{\partial x}$

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

□

7. `(tuple)`

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

   Induction hypotheses:

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

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

   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_τ and S_σ 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.

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

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

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

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

$$\begin{aligned}&S_(\tau \times \sigma)(\lambda x.(\llbracket t_1 \rrbracket(s(x)), \llbracket t_2 \rrbracket(s(x))),\\ &\quad \lambda x.(\llbracket \overrightarrow{\mathcal{D}}(t_1) \rrbracket(s'(x)), \llbracket \overrightarrow{\mathcal{D}}(t_2) \rrbracket(s'(x))))\end{aligned}$$

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

$$\begin{aligned}&\exists f_1, f_2, g_1, g_2,\\ &\quad S_\tau(f_1, f_2), S_\sigma(g_1, g_2).\\ &\quad \lambda x.(\llbracket t_1 \rrbracket(s(x)), \llbracket t_2 \rrbracket(s(x))) = \lambda x.(f_1(x), g_1(x)) \wedge\\ &\quad \lambda x.(\llbracket \overrightarrow{\mathcal{D}}(t_1) \rrbracket(s'(x)), \llbracket \overrightarrow{\mathcal{D}}(t_2) \rrbracket(s'(x))) = \lambda x.(f_2(x), g_2(x))\end{aligned}$$

$\Vdash$ (Give witnesses: $f_1 := \llbracket t_1 \rrbracket \circ s$, $f_2 := \llbracket t_2 \rrbracket \circ s$,
$g_1 := \llbracket \overrightarrow{\mathcal{D}}(t_1) \rrbracket \circ s'$, $g_2 := \llbracket \overrightarrow{\mathcal{D}}(t_2) \rrbracket \circ s'$)

$$\begin{aligned}&\exists S_\tau(f_1, f_2), S_\sigma(g_1, g_2).\\ &\quad \lambda x.(\llbracket t_1 \rrbracket(s(x)), \llbracket t_2 \rrbracket(s(x)))\\ &\qquad = \lambda x.(\llbracket t_1 \rrbracket(s(x)), \llbracket t_2 \rrbracket(s(x))) \wedge\\ &\quad \lambda x.(\llbracket \overrightarrow{\mathcal{D}}(t_1) \rrbracket(s'(x)), \llbracket \overrightarrow{\mathcal{D}}(t_2) \rrbracket(s'(x)))\\ &\qquad = \lambda x.(\llbracket \overrightarrow{\mathcal{D}}(t_1) \rrbracket(s'(x)), \llbracket \overrightarrow{\mathcal{D}}(t_2) \rrbracket(s'(x)))\end{aligned}$$

$\Vdash$ (Give witnesses of S_τ and S_σ using respective IHs 7a and 7b)

$$\begin{aligned}&\quad \lambda x.(\llbracket t_1 \rrbracket(s(x)), \llbracket t_2 \rrbracket(s(x)))\\ &\qquad = \lambda x.(\llbracket t_1 \rrbracket(s(x)), \llbracket t_2 \rrbracket(s(x))) \wedge\\ &\quad \lambda x.(\llbracket \overrightarrow{\mathcal{D}}(t_1) \rrbracket(s'(x)), \llbracket \overrightarrow{\mathcal{D}}(t_2) \rrbracket(s'(x)))\\ &\qquad = \lambda x.(\llbracket \overrightarrow{\mathcal{D}}(t_1) \rrbracket(s'(x)), \llbracket \overrightarrow{\mathcal{D}}(t_2) \rrbracket(s'(x)))\end{aligned}$$

$\Vdash$ (Reflexivity)

□

8. (`first`)

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

   Induction hypotheses:

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

   Simplifying the induction hypothesis 8a using the definition of $S_{\tau\times\sigma}$ gives rise to a number of usefull assumptions:

   (a)

9. (`second`)

□

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 $\llbracket t\rrbracket$.*

Proof. □

3.2 Adding Sums and Primitive Recursion

3.3 Arrays

4 Optimization

4.1 Program Transformations

5 Reverse-Mode AD

6 Discussion

6.1 Problems

6.2 Future Work

7 Conclusion

A Language Definitions

B Forward-Mode Macro

C Denotations

References

[1] D. Scott, "Outline of a mathematical theory of computation", *Kiberneticheskij Sbornik. Novaya Seriya*, vol. 14, Jan. 1977 (cit. on p. 7).

[2] T. Coquand and G. Huet, "The calculus of constructions", *Information and Computation*, vol. 76, no. 2-3, pp. 95–120, Feb. 1988. DOI: 10.1016/0890-5401(88)90005-3. [Online]. Available: https://doi.org/10.1016/0890-5401(88)90005-3 (cit. on p. 8).

[3] T. Coquand and C. Paulin, "Inductively defined types", in *COLOG-88*, Springer Berlin Heidelberg, 1990, pp. 50–66. DOI: 10.1007/3-540-52335-9_47. [Online]. Available: https://doi.org/10.1007/3-540-52335-9_47 (cit. on p. 8).

[4] 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. 14).

[5] T. Altenkirch and B. Reus, “Monadic presentations of lambda terms using generalized inductive types”, in *Proceedings of the 13th International Workshop and 8th Annual Conference of the EACSL on Computer Science Logic*, ser. CSL ’99, Berlin, Heidelberg: Springer-Verlag, 1999, pp. 453–468, ISBN: 3540665366 (cit. on p. 9).

[6] C. McBride and J. McKinna, “The view from the left”, *J. Funct. Program.*, vol. 14, no. 1, pp. 69–111, Jan. 2004, ISSN: 0956-7968. DOI: 10.1017/S0956796803004829. [Online]. Available: https://doi.org/10.1017/S0956796803004829 (cit. on p. 9).

[7] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic, “Mechanized metatheory for the masses: The PoplMark challenge”, in *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2005, pp. 50–65. DOI: 10.1007/11541868_4. [Online]. Available: https://doi.org/10.1007/11541868_4 (cit. on p. 9).

[8] R. Adams, “Formalized metatheory with terms represented by an indexed family of types”, in *Types for Proofs and Programs*, J.-C. Filliâtre, C. Paulin-Mohring, and B. Werner, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–16, ISBN: 978-3-540-31429-5 (cit. on p. 9).

[9] A. Ahmed, “Step-indexed syntactic logical relations for recursive and quantified types”, in *Programming Languages and Systems*, P. Sestoft, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 69–83, ISBN: 978-3-540-33096-7 (cit. on p. 12).

[10] M. Sozeau, “Program-ing finger trees in coq”, *SIGPLAN Not.*, vol. 42, no. 9, pp. 13–24, Oct. 2007, ISSN: 0362-1340. DOI: 10.1145/1291220.1291156. [Online]. Available: https://doi.org/10.1145/1291220.1291156 (cit. on p. 11).

[11] M. Sozeau and N. Oury, “First-class type classes”, in *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2008, pp. 278–293. DOI: 10.1007/978-3-540-71067-7_23. [Online]. Available: https://doi.org/10.1007/978-3-540-71067-7_23 (cit. on p. 8).

[12] N. Benton, A. Kennedy, and C. Varming, "Some domain theory and denotational semantics in coq", vol. 5674, Aug. 2009, pp. 115–130. doi: `10.1007/978-3-642-03359-9_10` (cit. on p. 7).

[13] M. Sozeau, "Equations: A dependent pattern-matching compiler", in *Interactive Theorem Proving*, Springer Berlin Heidelberg, 2010, pp. 419–434. doi: `10.1007/978-3-642-14052-5_29`. [Online]. Available: `https://doi.org/10.1007/978-3-642-14052-5_29` (cit. on pp. 8, 11).

[14] 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 pp. 9, 14).

[15] A. Chlipala, *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, 2013, isbn: 0262026651 (cit. on pp. 11, 17).

[16] A. Mahboubi and E. Tassi, "Canonical structures for the working coq user", in *Interactive Theorem Proving*, Springer Berlin Heidelberg, 2013, pp. 19–34. doi: `10.1007/978-3-642-39634-2_5`. [Online]. Available: `https://doi.org/10.1007/978-3-642-39634-2_5` (cit. on p. 8).

[17] R. Dockins, "Formalized, effective domain theory in coq", in *Interactive Theorem Proving*, G. Klein and R. Gamboa, Eds., Cham: Springer International Publishing, 2014, pp. 209–225, isbn: 978-3-319-08970-6 (cit. on p. 7).

[18] A. G. Baydin, B. A. Pearlmutter, A. Radul, and J. M. Siskind, "Automatic differentiation in machine learning: A survey", *J. Mach. Learn. Res.*, vol. 18, 153:1–153:43, 2015 (cit. on p. 5).

[19] S. Boldo, C. Lelay, and G. Melquiond, "Coquelicot: A user-friendly library of real analysis for coq", *Mathematics in Computer Science*, vol. 9, pp. 41–62, 2015 (cit. on p. 8).

[20] B. C. Pierce, A. A. de Amorim, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, A. Tolmach, and B. Yorgey, *Programming Language Foundations*, ser. Software Foundations series, volume 2. Electronic textbook, May 2018 (cit. on p. 8).

[21] A. Abel, G. Allais, A. Hameer, B. Pientka, A. Momigliano, S. Schäfer, and K. Stark, "Poplmark reloaded: Mechanizing proofs by logical relations", *Journal of Functional Programming*, vol. 29, e19, 2019. DOI: 10.1017/S0956796819000170 (cit. on p. 10).

[22] 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. 5, 16).

[23] L. Skorstengaard, *An introduction to logical relations*, 2019. arXiv: 1907.11133 [cs.PL] (cit. on p. 12).

[24] M. Sozeau and C. Mangin, "Equations reloaded: High-level dependently-typed functional programming and proving in coq", *Proceedings of the ACM on Programming Languages*, vol. 3, no. ICFP, pp. 1–29, Jul. 2019. DOI: 10.1145/3341690. [Online]. Available: https://doi.org/10.1145/3341690 (cit. on p. 11).

[25] P. Wadler and W. Kokke, *Programming Language Foundations in Agda*. 2019, Available at http://plfa.inf.ed.ac.uk/ (cit. on p. 8).

[26] A. Aaby, "Introduction to programming languages", *Syntax Imperative Programming Concurrent Programming Object-Oriented Programming Evaluation*, vol. 3, Apr. 2020 (cit. on p. 7).

[27] 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. 12, 13, 16, 17).

[28] 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, 12–14, 16, 17).

[29] M. Sozeau, "Subset coercions in coq", in *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 237–252. DOI: 10.1007/978-3-540-74464-1_16. [Online]. Available: https://doi.org/10.1007/978-3-540-74464-1_16 (cit. on p. 11).