Utrecht University

Master Thesis

Formalized Proof of Automatic Differentiation in Coq

Student:
Curtis Chin Jen Sem

Supervisors:
Mathijs Vákár
Wouter Swierstra

Department of Information and Computing Science

Last updated: June 16, 2020

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

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 [26]. 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 [26] in **Coq**.
- Prove that well-known compile-time optimizations such as the partial evaluation, are correct with respect to the semantics of automatic differentiation.
- Extend the proof with the array types and compile-time optimization rules by Shaikhha, et. al.[20].

[1] https://github.com/google/tangent
[2] https://github.com/HIPS/autograd

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. This gradient is used in the gradient descent algorithm to optimize objective functions by determining the direction of steepest descent[16]. Automatic differentiation has a long and rich history, where its driving motivation is to efficiently calculate the derivatives of functions in a manner that is both correct and fast. There are several different methods of implementing automatic differentiation algorithms, such as source-code transformations or operator overloading. These algorithms usually transform any program which implements some function to one that calculates its derivative. In addition to the standard semantics present in most programming languages, concepts relevant to differentiation such the chain rule are needed.

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 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 works

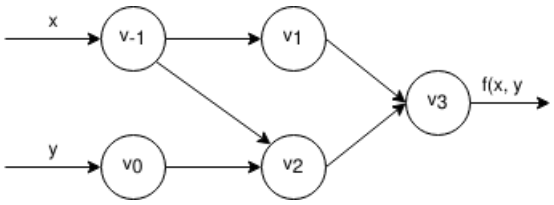

Figure 1: Computational graph of $f(x, y) = x^2 + (x - y)$

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 $f(x, y) = x^2 + (x - y)$

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, a primal trace is needed to determine the intermediate variables and subfunction dependencies. The second pass calculates the derivatives by working backwards from the output using the adjoints, also called the adjoint trace.

The optimal choice between automatic differentiation variant is heavily dependent on the specific 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 rerun for each partial derivative of the function. On the other hand, as reverse mode AD works backwards, the reverse-pass needs to be redone for each output variable. In machine learning research, reverse mode AD is generally preferred as the objective functions regularly contain a small number of output variables. How one does reverse mode automatic differentiation on a functional language is still an active area of research.

2.2 Denotational semantics

The notion of denotational semantics tries to find underlying mathematical models able to underpin the concepts known in programming languages. 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[24]. 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.

Automatic differentiation introduces a complication with picking a proper denotational model as the notion of derivability needs to be established. If the language under consideration were to be restricted to real-typed terms, cartesian spaces would have been sufficient as any well-typed term $x_1 : R, \ldots, x_n : R \vdash t : R$ could be interpreted as the corresponding smooth function $[\![t]\!] : R^n -> R$. This, however, does not work when function types are added as their denotational equivalent, function spaces, are not supported by cartesian spaces[26]. In the original pen and paper proof of automatic differentiation this thesis is based on by Huot, Staton and Vákár[26], the mathematical models used were diffeological spaces.

For the purpose of this thesis, however, we were able to avoid using diffeological spaces by directly encoding the property of differentiability in the logical relation itself. We were also able to avoid the domain theoretical models such as ω-cpos by excluding language constructs such as recursion and iteration where non-termination and partiality come into play. 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 also needed differentiability as the denotational result of applying the macro operation. Instead of encoding this by hand, we opted for the more comprehensive library `Coquelicot`[17], which contains many useful 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[12] and advanced modular constructions for organizing large mathematical proofs[11][15].

The core of this type theory is based on constructive logic and so many of the

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

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

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

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

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**, which states that functions are equal if they are extensionally equivalent.

2.3.1 Language representation

When defining a simply-typed lambda calculus, there are two main possibilities[23]. 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[18]. 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 possibili-

ties 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.[13], and was also used as one of the examples in the second POPLmark challenge which deals with logical relations[19]. 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 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 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 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

```
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[14].

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[27][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[12][22]. This introduces **Agda**-like dependent pattern matching with with-clauses. It does this by using a notion called coverings, where a 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,

```
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`

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[21]. 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[25][26]. 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-typed formulation is given in Snippet 5. Noteworthy is the case for function types, where one needs

```
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 representation

to prove that applying a function preserves the strong normalization property. If one were to attempt the proof of strong normalization without using logical relations, the proof would get stuck in the cases dealing with function types. More specifically when applying a function term to an argument term which terminates, the induction hypothesis is not strong enough to prove that substituting the argument into the body of the abstraction results in a terminating term.

2.5 Related work

3 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.

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. 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.

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 language using respectively the `var`, `app` and `abs` constructors. Product types are added to the language in the form of binary projections. The `first` and `second` constructors represent respectively the left and

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

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

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

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

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

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

Figure 3: Type-inference rules for the base simply-typed lambda calculus

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[13][4], 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 2. 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 [26], 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 product types.

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

We use the same inductively defined macro on types and terms used by many previous authors to implement the forward-mode automatic differentiation macro[26][25][20]. The forward-mode macro, $\overrightarrow{\mathcal{D}}$, keeps track of both primal and tangent traces using tuples as respectively its first and second elements. In most cases, the macro simply preserves the structure of the language. The cases for real numbers such as addition and multiplication are the exception. Here, the element encoding the tangent trace needs to contain the proper syntactic translation of the derivative of the operation. Due to the intrinsic nature of our language representation in the proof, the macro also needs to be applied to both the types and typing context to ensure that the terms stay well-typed.

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

Figure 4: Macro on base simply-typed lambda calculus

Applying the macro, to subterms give the syntactic counterparts of both their primal and tangent denotations as a tuple. Accessing these with projections, we can implement the specific derivative implementations of the operations on real terms. Note that applying the macro to the case for variables does nothing as the macro is also applied to the typing context, so variables implicitly already

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

Figure 5: Denotations of the base simply-typed lambda calculus

reference macro-applied terms.

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

Denotating the terms in our language now corresponds to finding the appropriate inhabitants in the denotated types. We denote our typing contexts Γ, lists of types, as heterogeneous lists containing their corresponding denotations. The specific implementation of heterogeneous lists used, correspond to the one given by Adam Chlipala[14]. In this implementation, heterogeneous lists consist of an underlying list of some type A and an accompanying function $A \to Set$. In our case these are respectively the typing context and the denotation function.

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

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

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

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

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

□

As we work with denotations, smooth functions $f : \mathsf{R}^n -> \mathsf{R}$ can be interpreted as the denotations of a corresponding syntactic term $x_1, \ldots, x_n \vdash t : R$. Intuitively, the free variables in the term t denote the usages of the parameters of the function and as such are restricted to terms of type R, same as each of the arguments in the function f. 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.[25] gave a syntactic proof of correctness of the macro, our proof follows the more denotational style of proof given by Huot, Staton and Vákár[26]. Correctness of the forward-mode macro consists of the assertion that the denotation of any macro-applied term of type $x_1 : R, \ldots, x_n : R \vdash t : \mathsf{R}$ will result in a pair of both the denotation of the original term and the derivative of that denotation.

Likewise, the proof of correctness will follow a logical relations argument, first generalizing the statement for both the typing contexts and the result type. The relation will ensure that both the smoothness property and the derivatives are preserved over higher-order types. We define the logical relation as a type-indexed relation between denotations of both terms and their macro-applied variants, so for any type τ, S_τ is the relation between functions $R \to [\![\tau]\!]$ and $R \to [\![\overrightarrow{\mathcal{D}}(\tau)]\!]$. When $\tau = \mathsf{R}$, the denotation of the macro-applied term should give both the original denotation and its derivative. With function types, as long as the relation is valid for the argument, applying these argument functions to the tracked denotations should preserve the relation. Some care has to be taken with in the case for products. Notably, the denotations of the subterms, $R \to [\![\tau]\!]$ and $R \to [\![\sigma]\!]$, should be existentially quantified as these are dependent on the original

denotation $R \to [\![\tau \times \sigma]\!]$.

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

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

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

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

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

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

To prove this statement, it first needs to be generalized to arbitrary substitutions. The key in formulating these denotationally lies in what was previously the argument function $f : R \to R^n$. Previously the function was used to indicate the open variables or function arguments. If generalized to $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$, this same function could be seen as a function which supplies terms foreach open

variable $x_1, \ldots, x_n$ with their appropriate types. So the argument function now becomes the pair of functions $s : R \to [\![\Gamma]\!]$ and $s_D : R \to [\![\overrightarrow{\mathcal{D}}(\Gamma)]\!]$. Note that the functions s and s_D are built out of the denotations of terms such that these same denotations follow the logical relation (1) for our language. We phrase this requirement as a definition.

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

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

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

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

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

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

Lemma 2 (Fundamental property)**.** *For any term $x_1 : R, ..., x_n : R \vdash t : R$, $[\![\overrightarrow{\mathcal{D}}(t)]\!]$ gives the dual number representation of $[\![t]\!]$, such that for any argument function $f : R \to R^n$, then $S_\tau([\![t]\!] \circ f, [\![\overrightarrow{\mathcal{D}}(t)]\!] \circ \overrightarrow{\mathcal{D}}_n \circ f)$.*

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

proven by both the induction hypothesis and the assumption that the denotations of the arguments supplied are smooth.

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

Proof. This is proven by showing that the goal follows from the logical relation which itself implied by the fundamental property (2) for well-typed terms.

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

□

3.2 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 6.

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

Binary sum types are included in the language using `inl` and `inr` as introducing terms. The `case` term encodes case-analysis given a function term for each

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

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

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

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

Figure 6: Type-inference rules for language constructs for sum types and primitive recursion

possibility. Primitive recursion is implemented using simple integers, where a endomorphic function is recursively applied a bounded number of times to a start value. For convenience, an additional successor function is added in the form of the nsucc term.

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

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

Both sums and integer terms are structure preserving with respect to the forward-mode macro. Note that we only take the derivative of values of type R, so when integers are encountered. More specifically, we do not keep track of derivatives at integer types as the tangent space is 0-dimensional.

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

Figure 7: Denotations of the sum and integer terms

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

Figure 8: Macro on the sum and integer terms

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

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

The only lemma or theorem that requires extension to deal with these new terms is the substitution lemma, as the validity of every other statement is independent of the additional types added to our language. With terms of integer type, the proof for the substitution lemma is trivial using the definition of our denotation functions. The `nrec` case for primitive recursion is only slightly more difficult as case-analysis on the denotation of the iteration term is required, where the 0 and $n + 1$ case are proven using the induction hypotheses derived from respectively the initial and function terms. As expected with the `case` term for sums, the denotation of the term under scrutiny needs to be destructed to properly apply the two disjunct induction hypotheses to their respective cases.

3.3 Arrays

Rarely is automatic differentiation done on mono-valued real numbers, due to the massive computational power available in GPUs in the form of array operations. So the next extension worth considering in our language are the array types. To be more specific we will be considering the array types and terms presented by Shaikhha, et. al.[20]. The well-typed nature of our presentation allows us to avoid much of the hairy details associated with bounds checking both indexing and array creation constructions. This is possible using the finite datatype, `Fin`, which is indexed by the upper-bound and represents for some n, the range of $[1..n]$.

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

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

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

Figure 9: Type-inference rules for array construction and indexing

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

Figure 10: Macro on array construction and indexing terms

composition of itself combined with the forward-mode macro, essentially applying the macro to every possible result of the function.

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

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

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

Figure 11: Denotations of the array construction and indexing terms

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

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

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

References

[1] D. Scott, "Outline of a mathematical theory of computation", *Kiberneticheskij Sbornik. Novaya Seriya*, vol. 14, Jan. 1977 (cit. on p. 6).

[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. 7).

[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. 7).

[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. 7).

[12] 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. 7, 11).

[13] 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).

[14] 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, 16).

[15] 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. 7).

[16] 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).

[17] 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. 7).

[18] 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).

[19] 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. 9).

[20] 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. 4, 15, 23).

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

[22] 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).

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

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

[25] 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, 15, 17).

[26] 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, 7, 12, 14, 15, 17).

[27] 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).