Utrecht University

Master Thesis

Formalized Correctness Proofs of Automatic Differentiation in Coq

Student:
Curtis Chin Jen Sem

Supervisors:
Mathijs Vákár
Wouter Swierstra

Department of Information and Computing Science

Last updated: July 21, 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 between 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 [44]. 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 make the following core contributions:

- Formalize the proofs of both the forward-mode and continuation-based automatic differentiation algorithms specified by Huot, Staton, and Vákár [44] in *Coq*.
- Prove the semantic correctness of various useful compile-time optimizations techniques in the context of generating performant code for automatic differentiation.
- Extend the proofs with the array types and compile-time optimization rules by Shaikhha, et al.[37].
- Analyze the requirements for a formal proof of correctness for the combinator-based reverse-mode automatic differentiation algorithm by Vákár[45].

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

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

formalizations of optimization avenues using respectively program transformations and continuation-based reverse-mode automatic differentiation.

2 Background

2.1 Automatic differentiation

Automatic differentiation (*AD*) 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[29]. There are several different methods of implementing AD 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.

There are two main variants of AD, namely forward-mode and reverse-mode AD. In forward-mode AD, 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 calculating the partial derivatives of sub-terms is structure preserving with respect to the normal calculation of terms.

This approach to forward-mode AD can be explained by dual numbers as these are, mathematically seen, what we are calculating with[29]. Dual numbers are numbers of the form of

$$x + x'\epsilon$$

where $x, x' \in \mathcal{R}$ and ϵ is a nilpotent number, such that $\epsilon^2 = 0$ and $\epsilon \neq 0$. Notably, both primal and tangent values are tracked in this representation, namely the tangent value is present in the coefficients of ϵ. As an example, we can see that this is true for both addition and multiplication:

$$\begin{aligned}(x + x'\epsilon) + (y + y'\epsilon) &= (x + y) + (x' + y')\epsilon \\ (x + x'\epsilon)(y + y'\epsilon) &= (xy) + (xy' + yx')\epsilon\end{aligned}$$

Using the following scheme for function application:

$$f(x + x'\epsilon) = f(x) + f'(x)x'\epsilon$$

We can also see that it follows the chain rule for function composition.

$$\begin{aligned}f(g(x + x'\epsilon)) &= f(g(x) + g'(x)x'\epsilon)) \\ &= f(g(x)) + f'(g(x))g'(x)x'\epsilon\end{aligned}$$

Using this, we can essentially calculate the derivative of any derivable function by interpreting any non-dual number x as its dual number counterpart of either $x + 1\epsilon$ or $x + 0\epsilon$. We interpret constant values as $x + 0\epsilon$, while input variables we take the partial derivative of are interpreted as $x + 1\epsilon$.

To give a more elaborate example of how this works in forward-mode AD, take the function $f(x, y) = x^2 + (x - y)$ as an example. The dependencies between the terms and

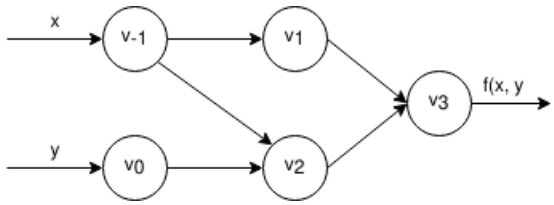

Figure 1: Computational graph of $f(x, y) = x^2 + (x - y)$

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

operations of the function is visible in the computational graph in fig. 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 $y' = 0$. Note that the calculation of the traces is structural. When calculating the primal value at a specific point, we can calculate the corresponding tangent value at that same point.

Reverse-mode automatic differentiation takes a drastically different approach. It starts by annotating one of the possibly many output variables $\frac{\partial y}{\partial y} = 1$ and working in the reverse direction, annotating each intermediate variable v_i with their adjoint

$$v'_i = \frac{\delta y_i}{\delta v_i}$$

To accomplish this, two passes are necessary. Like the forward-mode variant, a primal trace is needed. This first pass functions to determine the intermediate variables and their respective dependencies. The second pass in the algorithm calculates the partial-derivatives by working backwards from the output using the adjoints, also called the adjoint trace. When variables are used multiple times, also called fan-out, their adjoints are added in the adjoint trace.

The optimal choice between automatic differentiation variant is heavily dependent on the specific function being differentiated. Preference is given for forward-mode AD when the number of output variables exceeds 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 generally contain a small number of output variables.

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[42]. In this model, programs are regularly 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 $\perp$ that is lower in the ordering relation than any other element.

Automatic differentiation introduces a challenge in constructing a denotational semantics as the notion of differentiability needs to be included. If the language under consideration were to be restricted to real-typed terms, cartesian spaces would have been sufficient as any well-typed term $x_1 : \mathtt{R}, \ldots, x_n : \mathtt{R} \vdash t : \mathtt{R}$ could be interpreted as the corresponding smooth function $[\![t]\!] : \mathcal{R}^n \to \mathcal{R}$. Note that we use R as the syntactic type for real numbers, while $\mathcal{R}$ is its denotational counterpart. This, however, does not work when function types are added as their denotational equivalent, function spaces, are not supported by Cartesian spaces[44]. In the original pen and paper proof of automatic differentiation this thesis is based on by Huot, Staton and Vákár[44], 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 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 attempting to encode this by hand, we opted for the more comprehensive library `Coquelicot`[30], 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[3]. 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[4], dependent pattern matching[24] and advanced modular constructions for organizing large mathematical proofs[21][27].

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

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

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

$$\frac{\Gamma \vdash t_1 : \sigma \to \tau \qquad \Gamma \vdash t_2 : \sigma}{\Gamma \vdash t_1\ t_2 : \tau}\ \textsc{TApp}$$

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

middle, $\forall A, A \vee \neg A$. In most cases they can, however, be safely added to *Coq* without making its logic inconsistent. Many of these axioms 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, $(\forall x, f\ x = g\ x) \to f = g$.

2.3.1 Language representation

When defining a simply-typed lambda calculus, there are two main possibilities[40]. 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 by Pierce, et al.[33]. 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[15]. One example of such an approach is the nominal representation where every variable is named. Code listing 1 gives an example of how one would define a simply-typed lambda calculus in such a representation. While this does follow the standard format used in regular mathematics, problems such as alpha-conversion and capture-avoidance appears.

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

```
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 nominal extrinsic representation.

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[10][14][16], the specific formulation used in this thesis using separate substitution and renaming operations was fleshed out in *Coq* by Nick Benton, et al.[25], and was also used as one of the examples in the second POPLmark challenge which deals with logical relations[35]. 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 required 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 the 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 difficult 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 listing 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 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[46][18]. 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[24][39]. 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, 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.

```
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[26].

```
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`

```
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 normalization proof in the strongly-typed intrinsic representation

2.4 Logical relations

Logical relations arguments are a proof technique often employed when proving programming language properties of statically typed languages[38]. 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 reason about recursive types and general references[17], to open relations which enable working with terms of non-ground type[43][44]. 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 listing 5. Noteworthy is the case for function types, where one needs 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

AD Formalizations. While there are proofs of forward-mode AD algorithms[44][43] and many more implementations[37][36], there have been relative few attempts at formalized proofs in proof assistant. In 2002, M. Mayero did a formalized proof of an AD framework in *Fortran* in *Coq*[12]. Their minimal language example included assignments and sequences as language constructs, and excluded all forms of non-sequential control flow. They also restricted the terms in their language to first-order types.

Programming Language Metatheory. Much meta-theoretical research has been done on encoding programming languages in proof assistants[15]. Examples include the weak higher-order abstract syntax approach worked out in Coq by Despeyroux, et al.[7], which shallowly embeds abstractions as functions $abs : (var \to tm) \to tm$. The parametric HOAS variant by Chlipala[19], is an interesting polymorphic generalization of this technique as it, like the strongly-typed terms representation used in this thesis, avoids the problems of alpha-conversion and capture avoidance while still being somewhat user-friendly. The locally nameless approach introduced by many various authors[8][6][13] takes a hybrid approach and preserves names for free variables while using the De-Bruijn representation for bound variables.

With regards to denotational semantics, both Benton, et al.[22] and Dockins[28] present domain-theoretical libraries in *Coq*.

Forward-Mode AD. The earliest found description of an approach for forward-mode AD on functional languages is by Karczmarczuk[9], on first-order terms. Siskind and Pearlmutter presented a "nestable" variant of a forward-mode AD algorithm using the dual numbers representation. This same algorithm is used in the *F#* library, DiffSharp[29]. A nearly identical variation, implemented in *Haskell*, is given by Elliott[23]. This uncontroversial implementation of forward-mode AD is also discussed in the survey by Baydin et al.[29].

Reverse-Mode AD. There are many interpretations of reverse-mode AD on functional languages. Most well-known is the one by Pearlmutter and Siskind[20], which is one of the first attempts at reverse-mode AD in a functional context and introduces the practice of using various first-class operations to calculate derivatives. These operations very often involve maintaining some notion of state to keep track of adjoints. The specific approach by Pearlmutter and Siskind uses non-local program transformations as their primitive construct of choice. In the trend of define-by-run algorithms, whose main strategy involves building up the reverse pass of the algorithm during run-time, their primitive $\overleftarrow{\mathcal{J}}$ uses reflection to perform reverse-mode AD at run-time.

Abadi and Plotkin[34], and Wang, et al.[41] also make use of reverse-mode AD primitives. Abadi and Plotkin do so in the context of a define-by-run trace-based algorithm. The approach by Wang et al. using delimited continuations, tries to remedy this using multi-stage programming to reclaim some efficiencies.

Define-by-run algorithms, however, lose much of the optimization opportunities provided by the explicit compilation process involved with programming languages. One significant issue with defining define-then-run reverse-mode algorithms is how to treat

many of the various control-flow constructs such as conditionals, loops or higher-order types. Elliott[32] gave an interesting principled approach to reverse-mode AD from the perspective of category theory by formulating the algorithm as a functor. Their method, though, is still limited to first-order programs. An extension to higher-order types by Vákár[45] is discussed in section 5.2.

3 Formalizing Forward-Mode AD

We will explain our formalization of the forward-mode automatic differentiation macro in the following sections. The formal proof will start from a base simply-typed lambda calculus extended with product types and incrementally add both sum and array types. Also included in the final language are natural number types with a primitive recursion principle. Many of the theorems and lemmas introduced in section 3.1 do not change, as they are independent of the specific types and terms included in the language.

3.1 Simply Typed Lambda Calculus

As mentioned in the background section 2.3.1, we will make use of De-Bruijn indices in an intrinsic representation to formulate our language. We include both addition and multiplication as example operations on the real numbers, but the proofs are easily extensible to other primitive operations. Our base language consists of the classic simply-typed lambda calculus with product types and real numbers.

Both the language constructs and the typing rules for this language are common for a simply-typed lambda calculus, as shown in fig. 3. As expected, we include variables, applications, and abstractions in the language using, respectively, the `var`, `app`, and `abs` terms. We work with projection products, whose elimination rules are encoded in the `first` and `second` terms. The `tuple` term is used to represent the introduction rule. 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[25][5], instead of using indices into the list accompanied by 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 code listing 2. The cases for `app` and `abs` are as expected, where variables in the body of abstractions can reference their respective arguments.

Note that in the original proof by Huot, Staton, and Vákár [44], they made use of n-ary products accompanied by pattern matching expressions. We opted to implement binary projection products, as these are conceptually simpler while still retaining much of the same functionality expected of product types.

We use the same inductively defined macro on types and terms used by many previous

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

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

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

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

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

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

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

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

Code snippet 6: Terms in our language related to product and real types.

authors to implement the forward-mode automatic differentiation macro[44][43][37]. The forward-mode macro, $\overrightarrow{\mathcal{D}}$, keeps track of both primal and tangent traces using tuples as respectively its first and second elements. In most cases, the macro simply preserves the structure of the language. The cases for real numbers such as addition and multiplication are the exception. Here, the element encoding the tangent trace needs to contain the proper syntactic translation of the derivative of the operation.

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

$$\begin{aligned}
\overrightarrow{\mathcal{D}}(\mathtt{R}) &= \mathtt{R} \times \mathtt{R} \\
\overrightarrow{\mathcal{D}}(\tau \times \sigma) &= \overrightarrow{\mathcal{D}}(\tau) \times \overrightarrow{\mathcal{D}}(\sigma) \\
\overrightarrow{\mathcal{D}}(\tau \Rightarrow \sigma) &= \overrightarrow{\mathcal{D}}(\tau) \Rightarrow \overrightarrow{\mathcal{D}}(\sigma)
\end{aligned}$$

$$\begin{aligned}
\overrightarrow{\mathcal{D}}(\mathtt{rval}\ n) &= \mathtt{tuple}\ (\mathtt{rval}\ n)\ (\mathtt{rval}\ 0) \\
\overrightarrow{\mathcal{D}}(\mathtt{add}\ n\ m) &= \mathtt{tuple}\ (\mathtt{add}\ n\ m)\ (\mathtt{add}\ n'\ m') \\
\overrightarrow{\mathcal{D}}(\mathtt{mul}\ n\ m) &= \mathtt{tuple}\ (\mathtt{mul}\ n\ m) \\
&\qquad (\mathtt{add}\ (\mathtt{mul}\ n'\ m)\ (\mathtt{mul}\ m'\ n)))
\end{aligned}$$

Figure 4: Macro on base simply-typed lambda calculus

Applying the macro to a term gives the syntactic counterparts of both their primal and tangent denotations as a tuple. These terms can be accessed with projections to implement the various derivative implementations of the operations on real terms included in the language. Note that applying the macro to the case for variables does nothing as the macro is also applied to the typing context, so variables implicitly already reference macro-applied terms.

As we restrict 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 $\mathcal{R}, \rightarrow, \star$. Well-typed terms of type τ, given typing context Γ, will denotate to functions $[\![\Gamma]\!] \rightarrow [\![\tau]\!]$.

Denotating the terms in our language now corresponds to finding the appropriate inhabitants in the denotated types. As typing contexts, Γ, are represented by lists of types. The appropriate way to denotate these would be to map the denotation function over the list. The resulting heterogeneous list contains the denotations of each type in the list in the correct order. The specific implementation of heterogeneous lists used in the proof corresponds to the one given by Adam Chlipala[26]. In this implementation, heterogeneous lists consist of an underlying list of some type A and an accompanying function $A \rightarrow Set$, which in our use case are, respectively, the typing context and the denotation function.

When giving the constructs in our language their proper denotations, most of the cases are straightforward. Notable is the case for variables, where we made use of the inductively defined type evidence to type our terms. Remember that to type variables in our term

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

$$
\begin{aligned}
fst &= \lambda x.let\ (x, y) \coloneqq [\![t]\!](x)\ in\ x \\
snd &= \lambda x.let\ (x, y) \coloneqq [\![t]\!](x)\ in\ y
\end{aligned}
$$

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

language, we have to also give the exact position of the type we are referencing in the typing context. Similarly as denotations, we are able to transform this positional information to generate a specialized lookup function, which given a valid typing context, gives a term denotation with the correct type. Essentially, we do a lookup into the heterogeneous list of denotations corresponding to the typing context.

```
Equations denote_v Γ τ (v: τ ∈ Γ) : ⟦Γ⟧ → ⟦τ⟧ :=
denote_v (Top Γ τ) := hd;
denote_v (Pop Γ τ σ v) := denote_v v ∘ tl.
```

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.

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

□

Using the denotation rules in fig. 5, syntactically well-typed terms in our language of the form $x_1 : \mathtt{R}, \ldots, x_n : \mathtt{R} \vdash t : \mathtt{R}$ can be interpreted as their corresponding smooth functions $f : \mathcal{R}^n \to \mathcal{R}$. Intuitively, the free variables in the syntactic term t correspond to the parameters of the denotation function f.

Although Barthe, et al.[43] gave a syntactic proof of correctness of the macro, our formal proof follows the more denotational style of proof given by Huot, Staton and Vákár[44]. Likewise, our proof of correctness will follow a similar logical relations argument. While both approaches have their merits, the proof using the denotational semantics requires less technical bookkeeping of open and closed terms.

Informally, the correctness statement of the forward-mode macro will consists of the assertion that the denotation of any macro-applied term of type $x_1 : \mathsf{R}, \ldots, x_n : \mathsf{R} \vdash t : \mathsf{R}$ will result in a pair of both the denotation of the original term and the derivative of that denotation. Note that while both the free variables and result type of the term t are restricted to type R, t itself can consist of subterms of higher-order types.

The logical relation will ensure that both the smoothness property and the derivatives are preserved over higher-order types. We define the logical relation as a type-indexed relation between denotations of both terms and their macro-applied variants, so for any type 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 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 $R \to [\![\tau \times \sigma]\!]$.

Definition 1. *(Logical relation) Denotation functions f and their corresponding derivatives g are inductively defined on the structure of our types such that they follow the relation*

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

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 $x_1 : \mathsf{R}, \ldots, x_n : \mathsf{R} \vdash t : \tau$ and argument function $f : \mathcal{R} \to \mathcal{R}^n$ 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 the argument function $f : \mathcal{R} \to [\![\mathsf{R}^n]\!]$ into one that supplies both the original input value and its accompanying derivative. The full type signature of this auxiliary function is $\overrightarrow{\mathcal{D}}_n : (\mathcal{R} \to [\![\mathsf{R}^n]\!]) \to \mathcal{R} \to [\![\overrightarrow{\mathcal{D}}(\mathsf{R})^n]\!]$. Note that τ^n is used as syntactic sugar for

`repeat` $\tau\ n$ and is simply a typing context consisting of the type τ repeated n times.

$$\overrightarrow{\mathcal{D}}_n(f,x) = \begin{cases} f(x) & : n = 0 \\ ((hd \circ f)(x), \frac{\partial (hd \circ f)}{\partial x}(x)) :: \overrightarrow{\mathcal{D}}_{n'}(tl \circ f, x) & : n = n' + 1 \end{cases} \tag{2}$$

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 the relation is preserved when applying substitutions consisting of arbitrary terms, possibly containing higher-order constructs. In this style of proof, however, the same concept needs to be formulated in a denotational manner.

The key in formulating these denotationally lies in the argument function $f : \mathcal{R} \to \mathcal{R}^n$. Previously the function was used to indicate the open variables or function arguments. Generalized to $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$, this function can be interpreted as supplying for each open variable $x_1, \ldots, x_n$ a corresponding denotated term with type $[\![\tau_1]\!], \ldots, [\![\tau_n]\!]$. So the argument function now becomes the pair of functions $s : R \to [\![\Gamma]\!]$ and $s_D : R \to [\![\overrightarrow{\mathcal{D}}(\Gamma)]\!]$, which intuitively speaking, form the denotational counterparts of syntactic substitutions. Notably, for the functions s and s_D to be valid with respect to the logical relation, they are required to be built from the denotations of terms such that these denotations also follow the logical relation. We phrase this requirement as a definition.

Definition 2. *(Instantiation) Instantiation functions $s : R \to [\![\Gamma]\!]$ and $s_D : R \to [\![\overrightarrow{\mathcal{D}}(\Gamma)]\!]$ are inductively defined such that they follow*

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

Using this notion of instantiations we can now formulate our fundamental lemma. Informally, this states that given correct instantiation functions any well-typed term is semantically correct with respect to the logical relation.

Lemma 1 (Fundamental)**.** *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 $inst_\Gamma(s, s_D)$, we have that $S_\tau([\![t]\!] \circ s, [\![\overrightarrow{\mathcal{D}}(t)]\!] \circ s_D)$.*

Proof. This is proven 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 fundamental 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.

Corollary 1 (Fundamental property)**.** *For any term $x_1 : \mathsf{R}, \ldots, x_n : \mathsf{R} \vdash t : \mathsf{R}$, $[\![\overrightarrow{\mathcal{D}}(t)]\!]$ gives the dual number representation of $[\![t]\!]$, such that for any argument function $f : \mathcal{R} \to \mathcal{R}^n$, we have that $S_\tau([\![t]\!] \circ f, [\![\overrightarrow{\mathcal{D}}(t)]\!] \circ \overrightarrow{\mathcal{D}}_n \circ f)$.*

Proof. This follows from the fundamental lemma. We lastly need to prove $inst_{(\texttt{repeat R } n)}$. This is proven by induction on n. If $n = 0$, the goal is trivial due to the argument function f being extensionally equal to `const []`, which directly corresponds to $inst_{[]}$. 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 $x_1 : \mathsf{R}, \ldots, x_n : \mathsf{R} \vdash t : \mathsf{R}$, $[\![\overrightarrow{\mathcal{D}}(t)]\!]$ gives the dual number representation of $[\![t]\!]$, such that for any argument function $f : \mathcal{R} \to \mathcal{R}^n$, we have that $[\![\overrightarrow{\mathcal{D}}(t)]\!] \circ \overrightarrow{\mathcal{D}}_n \circ f = \lambda x.([\![t]\!] \circ f, {}^{\partial([\![t]\!] \circ f)}/_{\partial x})$.*

Proof. This is proven by showing that the goal follows from the logical relation which itself is implied by the fundamental property.

$$
\begin{aligned}
&[\![\overrightarrow{\mathcal{D}}(t)]\!] \circ \overrightarrow{\mathcal{D}}_n \circ f = \lambda x.([\![t]\!] \circ f, {}^{\partial([\![t]\!] \circ f)}/_{\partial x}) \\
&\qquad \Vdash \text{(By definition of } S_R \text{ with } f := [\![t]\!] \circ f \text{ and } g := [\![\overrightarrow{\mathcal{D}}(t)]\!] \circ \overrightarrow{\mathcal{D}}_n \circ f) \\
&\qquad S_R([\![t]\!] \circ f, [\![\overrightarrow{\mathcal{D}}(t)]\!] \circ \overrightarrow{\mathcal{D}}_n \circ f) \\
&\qquad \Vdash \text{(Fundamental property (corollary 1))}
\end{aligned}
$$

□

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 proof technique, natural number types and primitive recursion were also added. The inference rules for the new language constructs added for sum and number types are given in fig. 6.

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

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

$$\frac{\Gamma \vdash e : \tau <+> \sigma \qquad \Gamma \vdash t_1 : \tau \Rightarrow \rho \qquad \Gamma \vdash t_2 : \sigma \Rightarrow \rho}{\Gamma \vdash \mathsf{case}\ e\ t_1\ t_2 : \rho}\ \textsc{TCase}$$

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

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

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

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

```
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)
```

Code snippet 7: Terms in our language related to sum types.

```
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
```

Code snippet 8: Terms in our language related to natural number types.

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

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

Figure 7: Denotations of the sum and integer terms

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, these are largely ignored. More specifically, we do not keep track of derivatives at natural number types as the tangent space is 0-dimensional. For a similar reason, the logical relation at integer types only needs to establish that the denotations of integer terms tracked by the logical relation, which will be of type $\mathcal{R} \to \mathcal{N}$, 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.

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

Figure 8: Macro on the sum and integer terms

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

The only lemma or theorem that requires extension to deal with these new terms is the fundamental lemma, as the validity of every other statement is independent of the types or terms added to our language. With terms of integer type, the proof for the fundamental lemma is trivial using the definition of our denotation functions. The `nrec` case for primitive recursion is only slightly more difficult as we have to do case-analysis on the denotation of the iteration term. 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 corresponding cases.

3.3 Arrays

Automatic differentiation is rarely done on mono-valued real numbers, due to the massive computational power available on 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 pull-array formulation presented by Shaikhha, et al.[37].

This final iteration of our simply-typed lambda calculus differs from the $\tilde{F}$ language given by Shaikha, et al. only superficially. We can define both their `ifold` and `let` constructs

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

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

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

```
Inductive tm (Γ : Ctx) : ty → Type :=
  ...
  (* Arrays *)
  | build : forall n τ (f : Fin n → tm Γ τ), tm Γ (Array n τ)
  | get : forall n τ (i : Fin n), tm Γ (Array n τ) → tm Γ τ
```

Code snippet 9: The terms related to array types included in our language

as syntactic sugar:

$$\mathtt{let}\ e\ t \equiv \mathtt{app}\ (\mathtt{abs}\ t)\ e$$
$$\mathtt{ifold}\ f\ n\ d \equiv \mathtt{nrec}\ (\mathtt{app}\ (\mathtt{abs}\ (\mathtt{app}\ f\ (\mathtt{nsucc}\ (\mathtt{var}\ \mathtt{Top}))))\ (\mathtt{nval}\ 0))\ n\ d$$

Note that due to our nameless representation, `let` terms lose much of their original usefulness in writing programs. Nonetheless, we included it to help formulate some of the program optimizations used in the final system. We also omit much of their cardinality and indexing constructions due to our shallow embedding of arrays. The well-typed nature of our representation allows us to avoid much of the hairy details associated with bounds checking both indexing and array creation. To accomplish this, we use the `Fin` inductive datatype which is indexed by an upper-bound and represents for some n, the range $[1..n]$.

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

Similarly for denotations, the denotation function, instantiated to the correct 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 definition of $vect$, the Fin and nat types

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

Figure 10: Macro on array construction and indexing terms

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

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

Figure 11: Denotations of the array construction and indexing terms

are treated interchangeably, where $\texttt{Fin}\ n$ is treated as the type level integer corresponding to a $n : \mathcal{N}$. So every $n : \mathcal{N}$ is transformed to 1 in the corresponding $\texttt{Fin}\ n$, and any $i : \texttt{Fin}\ n$ is transformed to n.

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.

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

As was the case for sum types, only the proof of the fundamental lemma needs to be extended. The proof for the array terms proceeds as follows. For `build`, we first do induction on n, the length of the array. The base case is trivial, as $\texttt{Fin}\ 0$ 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.

4 Optimization

Shaikhha, et al. have presented a small system which has proven to be performant[37]. They empirically showed that it is possible for forward-mode AD to approach the performance of reverse-mode AD, even if the forward-mode algorithm has to be executed n times to calculate the n partial derivatives of a function. In section 3.3 we already formalized one of the critical components of their system, namely their usage of array types. Next, we will prove that the various program transformation rules they use to drastically reduce the number of calculations needed, are sound.

4.1 Program Transformations

The transformation rules consist of several algebraic identities, along with compile time optimization techniques such as partial evaluation and deforestation or loop fusion. We reuse the denotational semantics from section 3 we used to prove the forward-mode algorithm correct. As a small deviation from the rules given by Shaikhha, et al.[37], we explicitly include a set of simplification rules in the style of a natural semantics. figs. 12 and 13 show, respectively, the rewrite rules we included in our language and the inference rules we used for our simplification rules.

Before we can prove soundness of our rewrite rules, we have to prove soundness of our natural semantics.

Lemma 2 (Soundness of natural semantics)**.** *For any well-typed terms t, t' such that $t \Downarrow t'$ holds, we have $[\![t]\!] = [\![t']\!]$.*

Proof. This is proven by induction on the evaluation relation $\Downarrow$. All of the cases follow from the induction hypotheses after simplification. □

Theorem 2 (Soundness of program transformations)**.** *For any well-typed terms t, t' such that $t \rightsquigarrow t'$ holds, we have $[\![t]\!] = [\![t']\!]$.*

Proof. This is proven by induction on the rewriting relation $\Downarrow$. Most of the cases follow from the induction hypothesis. The rewrite rule where we incorporated the evaluation relation is proven by lemma 2. The rewrite rules associated with algebraic identities are proven by exactly those identities after simplifying using the denotational semantics.

For the loop fusion rule, we first do induction on i, the index being accessed. For $i = 0$, we use case-analysis on n along with simple rewriting to prove the goal. The induction step is proven by the induction hypothesis. Similarly for the loop fission rule, we have to do induction on the denotation of the term used to encode the number of iterations. The base case is trivial and the induction step is proven by the induction hypothesis. □

5 Towards Formalizing Reverse-Mode AD

As mentioned in section 2.5, there have been many attempts at reverse-mode algorithms that operate on functional languages. These algorithms, however, either fall short as they

$$\mathsf{add}\ t_1\ t_2 \rightsquigarrow \mathsf{add}\ t_2\ t_1$$
$$\mathsf{add}\ 0\ t \rightsquigarrow t$$
$$\mathsf{add}\ t\ (-t) \rightsquigarrow 0$$
$$\mathtt{mul}\ t_1\ t_2 \rightsquigarrow \mathtt{mul}\ t_2\ t_1$$
$$\mathsf{add}\ (\mathtt{mul}\ t\ t_1)\ (\mathtt{mul}\ t\ t_2) \rightsquigarrow \mathtt{mul}\ t\ (\mathsf{add}\ t_1\ t_2)$$
$$\mathtt{mul}\ 0\ t \rightsquigarrow 0$$
$$\mathtt{mul}\ 1\ t \rightsquigarrow t$$

(a) Algebraic laws

$$t \Downarrow t' \to t \rightsquigarrow t'$$

(b) Reuse rewriting rules from operational semantics

$$t \rightsquigarrow t' \to \mathsf{abs}\ t \rightsquigarrow \mathsf{abs}\ t'$$

(c) Partial evaluation on functions

$$\mathtt{get}\ i\ (\mathtt{build}\ n\ f) \rightsquigarrow f\ i$$

(d) Loop fusion

$$\begin{aligned}&\mathtt{ifold}\ (\mathsf{abs}\ (\mathsf{abs}\ (\mathtt{tuple}\\ &\quad(\mathsf{app}\ (\mathsf{app}\ f\ (\mathtt{var}\ (\mathtt{Pop}\ \mathtt{Top})))\ (\mathtt{first}\ (\mathtt{var}\ \mathtt{Top})))\\ &\quad(\mathsf{app}\ (\mathsf{app}\ f\ (\mathtt{var}\ (\mathtt{Pop}\ \mathtt{Top})))\ (\mathtt{second}\ (\mathtt{var}\ \mathtt{Top}))))))\ i\ (\mathtt{tuple}\ z_1\ z_2)\\ &\quad\rightsquigarrow \mathtt{tuple}\ (\mathtt{ifold}\ f\ i\ z_1)\ (\mathtt{ifold}\ f\ i\ z_2)\end{aligned}$$

(e) Loop fission

Figure 12: Included rewrite rules for the simply-typed lambda calculus extended with sum, product, number and array types

$$\frac{t_1 \Downarrow \mathsf{abs}\ t_1' \qquad t_2 \Downarrow t_2'}{\mathsf{app}\ t_1\ t_2 \Downarrow \mathtt{substitute}\ t_2'\ t_1'}\ \textsc{EVAppAbs} \qquad \frac{t \Downarrow \mathtt{nval}\ n}{\mathtt{nsucc}\ t \Downarrow \mathtt{nval}\ (n+1)}\ \textsc{EVSucc}$$

$$\frac{t_1 \Downarrow t_1' \qquad t_2 \Downarrow \mathtt{nval}\ 0 \qquad t_3 \Downarrow t_3'}{\mathtt{nrec}\ t_1\ t_2\ t_3 \Downarrow t_3}\ \textsc{EVNRec0}$$

$$\frac{t_1 \Downarrow t_1' \qquad t_2 \Downarrow \mathtt{nval}\ (n+1) \qquad t_3 \Downarrow t_3'}{\mathtt{nrec}\ t_1\ t_2\ t_3 \Downarrow \mathsf{app}\ t_1'\ (\mathtt{nrec}\ t_1'\ (\mathtt{nval}\ n)\ t_3')}\ \textsc{EVNRecS}$$

$$\frac{t_1 \Downarrow \mathtt{rval}\ r_1 \qquad t_2 \Downarrow \mathtt{rval}\ r_2}{\mathsf{add}\ t_1\ t_2 \Downarrow \mathtt{rval}\ r_1 + r_2}\ \textsc{EVAdd} \qquad \frac{t_1 \Downarrow \mathtt{rval}\ r_1 \qquad t_2 \Downarrow \mathtt{rval}\ r_2}{\mathtt{mul}\ t_1\ t_2 \Downarrow \mathtt{rval}\ r_1 * r_2}\ \textsc{EVMult}$$

$$\frac{t_1 \Downarrow t_1' \qquad t_2 \Downarrow t_2'}{\mathtt{tuple}\ t_1\ t_2 \Downarrow \mathtt{tuple}\ t_1'\ t_2'}\ \textsc{EVTuple}$$

$$\frac{t \Downarrow \mathtt{tuple}\ t_1\ t_2}{\mathtt{first}\ t \Downarrow t_1}\ \textsc{EVFst} \qquad \frac{t \Downarrow \mathtt{tuple}\ t_1\ t_2}{\mathtt{second}\ t \Downarrow t_2}\ \textsc{EVSnd}$$

$$\frac{t \Downarrow t'}{\mathtt{inl}\ t \Downarrow \mathtt{inl}\ t'}\ \textsc{EVInl} \qquad \frac{t \Downarrow t'}{\mathtt{inr}\ t \Downarrow \mathtt{inr}\ t'}\ \textsc{EVInr}$$

$$\frac{t \Downarrow \mathtt{inl}\ t' \qquad t_1 \Downarrow t_1' \qquad t_2 \Downarrow t_2'}{\mathtt{case}\ t\ t_1\ t_2 \Downarrow \mathsf{app}\ t_1'\ t'}\ \textsc{EVCaseInl}$$

$$\frac{t \Downarrow \mathtt{inr}\ t' \qquad t_1 \Downarrow t_1' \qquad t_2 \Downarrow t_2'}{\mathtt{case}\ t\ t_1\ t_2 \Downarrow \mathsf{app}\ t_2'\ t'}\ \textsc{EVCaseInr}$$

Figure 13: Inference rules for the evaluation relation

make use of unconventional semantics such as mutable state or delimited continuations, or they do not perform true reverse-mode AD. Much of the problem with defining an efficient reverse-mode algorithm on functional languages comes from the difficulty with ensuring that fan-out, the various usages of a variable, in the forward pass, correctly transform to addition in the reverse pass.

Huot, Staton and Vákár showed the versatility of their denotational semantics on a continuation-based AD algorithm, whose origin can be traced back to an description given by Karczmarczuk[11]. While this algorithm does calculate gradients, it is not a true reverse-mode AD algorithm as the final computation graph differs from what would be expected of reverse-mode AD[20]. The consequence of which is the excess usage of primitive operations. Nonetheless, it is useful to give a formal proof of correctness of this algorithm in terms of what we have already established, to show the versatility of the proof technique used, and by extension, our simple set-theoretic denotational semantics. This is done in section 5.1. A reverse-mode algorithm posed by Vákár along with an attempt at a formal proof in *Coq*, is discussed in section 5.2.

5.1 Formalizing Continuation-Based AD

The guiding principle in this algorithm is to build up the reverse pass as a continuation using a structure preserving macro on the program syntax. Like the forward-mode macro $\overrightarrow{\mathcal{D}}$, the continuation-based macro $\overrightarrow{\mathcal{D}}^c$ is structure preserving and works with tuples at ground type $\mathtt{R}$. But unlike the forward-mode macro, which uses these tuples to represent dual-numbers, the continuation-based macro pairs the primal values with a continuation calculating an array containing all of the perturbation variables. Notably, because the continuation-based macro, shown in fig. 14, calculates all of the perturbation values with respect to each of the input variables, it is additionally indexed by the number of input variables.

$$
\begin{aligned}
\overrightarrow{\mathcal{D}}^c_n(\mathsf{R}) &= \mathsf{R} \times (\mathsf{R} \Rightarrow \mathtt{Array}\ n\ \mathsf{R}) \\
\overrightarrow{\mathcal{D}}^c_n(\tau \times \sigma) &= \overrightarrow{\mathcal{D}}(\tau) \times \overrightarrow{\mathcal{D}}(\sigma) \\
\overrightarrow{\mathcal{D}}^c_n(\tau <+> \sigma) &= \overrightarrow{\mathcal{D}}(\tau) <+> \overrightarrow{\mathcal{D}}(\sigma) \\
\overrightarrow{\mathcal{D}}^c_n(\tau \Rightarrow \sigma) &= \overrightarrow{\mathcal{D}}(\tau) \Rightarrow \overrightarrow{\mathcal{D}}(\sigma) \\
\overrightarrow{\mathcal{D}}^c_n(\mathtt{Array}\ m\ \tau) &= \mathtt{Array}\ m\ \overrightarrow{\mathcal{D}}^c_n(\tau)
\end{aligned}
$$

$$
\begin{aligned}
\overrightarrow{\mathcal{D}}^c_n(\mathtt{rval}\ r) &= \mathtt{tuple}\ (\mathtt{rval}\ r) \\
&\quad\ (\mathtt{build}\ (\mathtt{const}\ (\mathtt{rval}\ 0))\ ) \\
\overrightarrow{\mathcal{D}}^c_n(\mathtt{add}\ r_1\ r_2) &= \mathtt{tuple}\ (\mathtt{add}\ r_1\ r_2)\ (\mathtt{abs}\ (\mathtt{vector\_add} \\
&\quad\ (\mathtt{app}\ r_1'\ (\mathtt{var}\ \mathtt{Top}))\ (\mathtt{app}\ r_2'\ (\mathtt{var}\ \mathtt{Top}))))) \\
\overrightarrow{\mathcal{D}}^c_n(\mathtt{mul}\ r_1\ r_2) &= \mathtt{tuple}\ (\mathtt{mul}\ r_1\ r_2)\ (\mathtt{abs}\ (\mathtt{vector\_add} \\
&\quad\ (\mathtt{app}\ r_1'\ (\mathtt{mul}\ r_2\ (\mathtt{var}\ \mathtt{Top}))) \\
&\quad\ (\mathtt{app}\ r_2'\ (\mathtt{mul}\ r_1\ (\mathtt{var}\ \mathtt{Top})))))
\end{aligned}
$$

Figure 14: Continuation-based macro on the simply-typed lambda calculus extended with array types

As we will phrase correctness in terms of the forward-mode macro, the forward-mode macro used in section 3.1 has to be slightly altered to calculate the partial derivatives with respect to all of the input variables. This change is slight, and merely swaps what was previously the tangent value for a vector of the tangent values corresponding to the input variables. The operations of reals are then swapped out for their vector element-wise counterparts. The modified forward-mode macro is shown in fig. 15.

As we work with the same denotational semantics as in our proof in section 3.1, the usage of an argument supplying function $f : \mathcal{R} \to \mathcal{R}^n$ makes a reappearance. Unlike last time, however, the arguments have to be massaged to fit both the forward and the reverse-mode macro. Note that for the forward-mode macro, input arguments we take the partial derivative of are supplied in a dual number format where the tangent value is equal to 1. Likewise, the tangent value of any other input argument should be set to 0. So each of the input arguments for the forward-mode macro are essentially coupled with a one-hot encoded vector.

$$
\overrightarrow{\mathcal{D}}_n\left(\begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}\right) \to \begin{bmatrix} (x_1, \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}) \\ (x_2, \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}) \\ (x_3, \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}) \end{bmatrix}^T
$$

Each of the one-hot encoded vectors function as indicating which partial derivative to calculate. The continuation-based macro requires a similar format, but instead of a one-hot

$$
\begin{aligned}
\overrightarrow{\mathcal{D}}_n(\mathtt{R}) &= \mathtt{R} \times \mathtt{Array}\ n\ \mathtt{R} \\
\overrightarrow{\mathcal{D}}_n(\tau \times \sigma) &= \overrightarrow{\mathcal{D}}(\tau) \times \overrightarrow{\mathcal{D}}(\sigma) \\
\overrightarrow{\mathcal{D}}_n(\tau <+> \sigma) &= \overrightarrow{\mathcal{D}}(\tau) <+> \overrightarrow{\mathcal{D}}(\sigma) \\
\overrightarrow{\mathcal{D}}_n(\tau \Rightarrow \sigma) &= \overrightarrow{\mathcal{D}}(\tau) \Rightarrow \overrightarrow{\mathcal{D}}(\sigma) \\
\overrightarrow{\mathcal{D}}_n(\mathtt{Array}\ m\ \tau) &= \mathtt{Array}\ m\ \overrightarrow{\mathcal{D}}^c_n(\tau)
\end{aligned}
$$

$$
\begin{aligned}
\overrightarrow{\mathcal{D}}_n(\mathtt{rval}\ r) &= \mathtt{tuple}\ (\mathtt{rval}\ r)\ (\mathtt{build}\ n\ (\mathtt{fun}\ \_\ \mathtt{=>}\ \mathtt{0})) \\
\overrightarrow{\mathcal{D}}_n(\mathtt{add}\ r_1\ r_2) &= \mathtt{tuple}\ (\mathtt{add}\ r_1\ r_2)\ (\mathtt{vector\_add}\ r_1'\ r_2') \\
\overrightarrow{\mathcal{D}}_n(\mathtt{mul}\ r_1\ r_2) &= \mathtt{tuple}\ (\mathtt{mul}\ r_1\ r_2) \\
&\qquad (\mathtt{vector\_add}\ (\mathtt{vector\_scale}\ r_2\ r_1')\ (\mathtt{vector\_scale}\ r_1\ r_2'))
\end{aligned}
$$

(a) Forward-mode macro calculating all partial derivatives

```
Definition vector_map Γ τ σ n ( f : tm Γ (τ ⇒ σ) )
( a : tm Γ (Array n τ) ) : tm Γ (Array n σ) :=
  build n (fun i => app f (get i a)).
Definition vector_map2 Γ τ σ ρ n
  ( a1 : tm Γ (Array n τ) ) ( a2 : tm Γ (Array n σ) )
  ( f : tm Γ (τ ⇒ σ ⇒ ρ) ) : tm Γ (Array n ρ) :=
  build n (fun i => app (app f (get i a1)) (get i a2)).
Definition vector_add Γ n
  ( a1 a2 : tm Γ (Array n R) ) : tm Γ (Array n R) :=
  vector_map2 a1 a2 (abs (abs (add (var Top) (var (Pop Top))))).
Definition vector_scale Γ n ( s : tm Γ R )
  ( a : tm Γ (Array n R) ) : tm Γ (Array n R) :=
  vector_map (abs (mul s (var Top))) a.
```

(b) Helper functions on array types

Figure 15: Consolidated forward-mode macro on the simply-typed lambda calculus extended with array types

encoded vector containing the value 1, the identity function is used.

$$\overrightarrow{\mathcal{D}}_n^c\left(\begin{bmatrix}x_1 & x_2 & x_3\end{bmatrix}\right) \to \begin{bmatrix}(x_1, \lambda x.\begin{bmatrix}x & 0 & 0\end{bmatrix}) \\ (x_2, \lambda x.\begin{bmatrix}0 & x & 0\end{bmatrix}) \\ (x_3, \lambda x.\begin{bmatrix}0 & 0 & x\end{bmatrix})\end{bmatrix}^T$$

Using these two input massaging functions, we can state correctness of our continuation-based macro with respect to our forward-mode macro.

Proposition 1 (Correctness of continuation-based AD)**.** *For any well-typed term* $x_1 : \mathsf{R}, \ldots, x_n : \mathsf{R} \vdash t : \mathsf{R}$ *and function arguments* $x : [\![\mathtt{repeat}\ \mathsf{R}\ n]\!]$ *we will take the partial derivatives relative to, we have that* $snd([\![\overrightarrow{\mathcal{D}}_n(t)]\!](\overrightarrow{\mathcal{D}}_n(x))) = snd([\![\overrightarrow{\mathcal{D}}_n^c(t)]\!](\overrightarrow{\mathcal{D}}_n^c(x)))(1)$.

This statement cannot be proven directly, however, we need a more general correctness statement. To be be specific, we need to establish equivalence between the macros regardless of the number we apply to the continuation-based macro, which corresponds to the coefficient of the forward-mode macro. This equivalence needs to be baked into the logical relation in our proof. The logical relation we will use in this proof for the most part stays the same as the one we used in the proof in section 3. The only case to differ significantly is the one for our ground type, the type of real numbers, R. Also note that, like the macro functions, the logical relation is also indexed by the number of input variables.

$$S_{n,\tau}(f,g) = \begin{cases} fst \circ f = fst \circ g \wedge & : \tau = \mathsf{R} \\ \quad \forall x.\lambda r.x * snd(f(r)) = \lambda r.(snd(g(r)))(x) & \end{cases} \tag{6}$$

As a recurring theme, many of the definitions and lemmas we used in the proof in section 3 reappear in this proof, differing only superficially in that they are additionally indexed by the number of partial derivatives. So we still make use of an instantiation relation like the one given in eq. (3) to establish that arbitrary typing context instantiations preserve the logical relation.

The proof of the fundamental lemma proceeds in much the same manner as the one given for lemma 1. As expected, the cases for the R-typed are exceptional. Each of these cases require a generalized version of the case to be proven. The problem here is that the number of partial derivatives is tightly coupled to the number of terms the operation is over. So induction on the number of terms involved in the operation necessary leads to issues with our relations, as these are indexed by the number of partial derivatives.

Proving this along with the corresponding fundamental property of the logical relation results in the following theorem.

Theorem 3 (Perturbation correctness)**.** *For any well-typed term* $x_1 : \mathsf{R}, \ldots, x_n : \mathsf{R} \vdash t : \mathsf{R}$*, function arguments* $x : [\![\mathtt{repeat}\ \mathsf{R}\ n]\!]$*, and real number* r*, we have* $r * snd(\overrightarrow{\mathcal{D}}_n(t)(\overrightarrow{\mathcal{D}}_n(x))) = snd(\overrightarrow{\mathcal{D}}_n^c(t)(\overrightarrow{\mathcal{D}}_n^c(x)))(r)$

Proof. This follows from the fundamental property of the logical relation in (eq. (6)). □

This is easily specialized to the original correctness statement we set out to prove.

Corollary 2 (Correctness of continuation-based AD)**.** *For any well-typed term* $x_1 : \mathtt{R}, \ldots, x_n : \mathtt{R} \vdash t : \mathtt{R}$ *and function arguments* $x : [\![\mathtt{repeat\ R}\ n]\!]$ *we will take the partial derivatives relative to, we have that* $snd([\![\overrightarrow{\mathcal{D}}_n(t)]\!](\overrightarrow{\mathcal{D}}_n(x))) = snd([\![\overrightarrow{\mathcal{D}}^c_n(t)]\!](\overrightarrow{\mathcal{D}}^c_n(x)))(1)$.

5.2 Combinator-Based Source Code Transformation

Eliott posed a novel and principled approach to defining an AD algorithm using *category theory*[32]. They take the category theoretical hammer to the problem by approaching it as an algorithm on categories, cartesian closed categories *(CCC)* to be exact, which are known to be equivalent to the simply-typed lambda calculi[31][2]. The algorithm is, however, still restricted to first order programs.

An extension of this technique is given by Vákár[45]. They go through first defining a small core combinator-based language. They then define a reverse-mode macro along with an additional auxiliary target language enriched with simple linear types.

5.2.1 Core Combinator Language

A well-known fact is the connection between CCC and simply-typed lambda calculi[2]. We can define a simple core combinator language inspired by the various categorical laws related to CCC. The requirement for a combinator language to be able to do reverse-mode AD comes from the need to make explicit, the contraction and weakening rules usually kept implicit in the typing contexts of typed lambda calculi. Translating from a simply-typed lambda calculus necessitates a translation between the implicit manipulation of the typing context to access variables, to one that is explicit in its usage of specific combinators.

The core combinator language we will be using is shown in fig. 16. Note the combinator language contains no typing context and are defined as terms $c : Hom(\tau, \sigma)$, where τ and σ are, respectively, the input and output types of the combinator c. As programming in the combinator language can quickly become cumbersome and unreadable, we make repeated usage of some syntactic niceties. We interchangeably use `;;` as the infix version of `seq` and $\langle A, B \rangle$ as a shortcut for `dupl;;cross` $A\ B$. We also use some helper functions for moving elements around in products, as shown below.

```
Definition assoc1 {A B C} : comb ((A × B) × C) (A × (B × C)) :=
  ⟨ exl;;exl, ⟨ exl;;exr, exr ⟩ ⟩.
Definition assoc2 {A B C} : comb (A × (B × C)) ((A × B) × C) :=
  ⟨ ⟨ exl, exr;;exl ⟩, exr;;exr ⟩.
Definition sym {A B} : comb (A × B) (B × A) :=
  ⟨ exr, exl ⟩.
```

We will next describe a translation from a simply-typed lambda calculus to this combinator language. Like the combinator language, the simply-typed lambda calculus will be restricted to function types, product types, and types for real numbers, vectors specialized to real numbers and unit, which is shown in fig. 17. Note that we omitted the terms related

```
Inductive ty : Type :=
  | R^ : nat → ty
  | R : ty
  | U : ty
  | ⇒ : ty → ty → ty
  | × : ty → ty → ty
.

Inductive comb : ty → ty → Type :=
  (* Category laws *)
  | id : forall A, comb A A
  | seq : forall A B C,
    comb A B → comb B C → comb A C
  (* Monoidal *)
  | cross : forall A B C D,
    comb A B → comb C D → comb (A × C) (B × D)
  (* Terminal *)
  | neg : forall A,
    comb A U
  (* Cartesian *)
  | exl : forall A B,
    comb (A × B) A
  | exr : forall A B,
    comb (A × B) B
  | dupl : forall A,
    comb A (A × A)
  (* Closed *)
  | ev : forall A B,
    comb ((A ⇒ B) × A) B
  | curry : forall A B C,
    comb (A × B) C → comb A (B ⇒ C)
  (* Reals *)
  | cplus : comb (R × R) R
  | crval : forall (r : ℛ), comb U R
  | cmplus : forall n, comb (R^n × R^n) (R^n)
  | cmrval : forall n (a : vector ℛ n), comb U (R^n)
.
```

Figure 16: Core combinator language inspired by Cartesian closed categories

```
Inductive tm (Γ : Ctx) : ty → Type :=
  ...
  (* Operations on reals *)
  | rval : forall r, → tm Γ R
  | plus :
    tm Γ R → tm Γ R → tm Γ R
  (* Operations on real vectors *)
  | mrval : forall n, vector ℛ n → tm Γ (R^n)
  | mplus : forall n,
    tm Γ (R^n) → tm Γ (R^n) → tm Γ (R^n)
  (* U *)
  | it : tm Γ U
.
```

Figure 17: Simply-typed lambda calculus with unit and specialized real arrays

to typing contexts, function types and product types, as these were identical to the ones used in section 3.

Using the same technique as Curien[2], we make use of an auxiliary language to smoothen the process. This auxiliary language, the typed categorical combinatory logic *(CCL)* will contain terms related to both the combinator language and the simply-typed lambda calculus, such as variable access, function application and function abstraction. While this CCL can be used to facilitate both back and forth translations, we will only describe the translation from the combinator language to the simply-typed lambda calculus.

The `ccl_env`combinator in CCL is most notable. In essence we transition each type in the typing context to become an additional argument in the resulting function type. Repeated usage of this specific combinator makes from the previously open term, a closed one.

The intrinsic representation we use in our definitions makes defining the exact translations a breeze as it then simply becomes an exercise in type-directed programming. The simply-typed lambda calculus to CCL translation specifically is straightforward, as the CCL language still has access to a typing context. Abstraction, where the argument type is added onto the typing context, is handled by the `ccl_env`construct. In cases where a value is introduced such as `var`, `rval` and , we make use of the `ccl_const`construct to ensure the term fits the type signature. Note that an additional domain type of `U` is added in the type signature of the translation function to accommodate the combinator-heavy auxiliary language. This translation is shown in listing 10.

As with defining the denotations of variables, we also need to define a mechanism to ensure the correct variable is still referenced when translating from the typed lambda calculi to CCL. Consider that the typing context, previously a list, is isomorphic to a single

```
Inductive ccl (Γ : Ctx) : ty → Type :=
(* Variables *)
| ccl_var : forall τ,
  τ ∈ Γ → ccl τ
(* Reals *)
| ccl_plus : ccl (R × R ⇒ R)
| ccl_rval : ℛ → ccl R
| ccl_mplus : forall n,
  ccl (R^n × R^n ⇒ R^n)
| ccl_mrval : forall n, vector ℛ n → ccl (R^n)
(* Category laws *)
| ccl_id : forall A, ccl (A ⇒ A)
| ccl_seq : forall A B C,
  ccl (A ⇒ B) → ccl (B ⇒ C) → ccl (A ⇒ C)
(* Cartesian *)
| ccl_exl : forall A B,
  ccl (A × B ⇒ A)
| ccl_exr : forall A B,
  ccl ((A × B) ⇒ B)
(* Monoidal *)
| ccl_cross : forall A B C,
  ccl (A ⇒ B) → ccl (A ⇒ C) → ccl (A ⇒ B × C)
(* Closed *)
| ccl_ev : forall A B,
  ccl ((A ⇒ B) × A ⇒ B)
| ccl_env : forall A B C,
  @ccl (A::Γ) (B ⇒ C) → @ccl Γ ((B ⇒ A ⇒ C))
(* Const *)
| ccl_const : forall A B,
  ccl A → ccl (B ⇒ A)
.
```

Figure 18: Auxilliary categorical combinatory logic language used in the translations

```
Fixpoint stlc_ccl {Γ τ} (t : tm Γ τ) : ccl Γ (U ⇒ τ) :=
  match t with
  (* Base *)
  | var v => ccl_const (ccl_var v)
  | app t1 t2 => ⟨ stlc_ccl t1, stlc_ccl t2 ⟩ ;; ccl_ev
  | abs t' => ccl_env (stlc_ccl t')
  (* Products *)
  | tuple t1 t2 => ⟨ stlc_ccl t1, stlc_ccl t2⟩
  | first t' => stlc_ccl t' ;; ccl_exl
  | second t => stlc_ccl t ;; ccl_exr
  (* Reals *)
  | plus t1 t2 => ⟨ stlc_ccl t1, stlc_ccl t2⟩ ;; ccl_plus
  | rval r => ccl_const (ccl_rval r)
  | mplus t1 t2 => ⟨ stlc_ccl t1, stlc_ccl t2⟩ ;; ccl_mplus
  | mrval r => ccl_const (ccl_mrval r)
  (* Unit *)
  | it => ccl_id
  end.
```

Code snippet 10: Simply-typed lambda calculus to CCL translation

nested product type. Empty lists correspond to the built-in unit type while concatenation becomes nested tupling. This isomorphism is what we use to translate the typing contexts to the input and output type format used by the combinator language.

```
Fixpoint translate_context (Γ : Ctx) : ty :=
  match Γ with
  | nil => U
  | τ :: Γ' => τ × translate_context Γ'
  end.
```

Doing a lookup in such a nested product type then reduces to applying the correct projection combinator.

```
Fixpoint fetch Γ τ (v : τ ∈ Γ) : comb (translate_context Γ) τ :=
  match v with
  | Top => exl
  | Pop v' => exr ;; fetch v'
  end.
```

We also define additional functions to correctly model weakening in the combinator language.

```
Definition weaken τ ρ σ (c : comb τ ρ): comb (σ × τ) ρ := exr ;; c.
Fixpoint weaken_ctx Γ τ (c : comb U τ): comb (translate_context Γ) τ :=
  match Γ with
  | nil => c
  | τ' :: Γ' => weaken τ' (weaken_ctx Γ' c)
  end.
```

The final translation function is the composition of both the translation functions in listings 10 and 11. Note that we have to remove the extra `U` type we added in the simply-typed lambda calculus to CCL translation shown listing 10.

```
Definition stlc_ccc Γ τ : tm Γ τ → comb (translate_context Γ) τ :=
  fun t => ⟨ ccl_ccc (stlc_ccl t), neg ⟩ ;; ev.
```

5.2.2 Defining the Macro and Target Language

The reverse-mode macro we will discuss in this chapter, like the forward-mode macro splits any type τ into tuples (τ_1, τ_2), where τ_1 and τ_2 represent, respectively, primal and tangent values. As expected, for τ_1 the macro essentially preserves the behavior expected of any of the combinators. The intuition what happens with reverse-mode AD gives a hint to

```
Fixpoint ccl_ccc Γ τ (c : @ccl Γ τ) : comb (translate_context Γ) τ :=
  match c with
  (* Base *)
  | ccl_var v => fetch v

  (* Reals *)
  | ccl_plus => curry (exr ;; cplus)
  | ccl_rval r => weaken_ctx Γ (crval r)
  | ccl_mplus => curry (exr ;; cmplus)
  | ccl_mrval r => weaken_ctx Γ (cmrval r)

  (* Category laws *)
  | ccl_id => curry exr
  | ccl_seq t1 t2 =>
    ⟨ ccl_ccc t2, ccl_ccc t1⟩ ;; curry (assoc1 ;; cross id ev ;; ev)

  (* Cartesian *)
  | ccl_exl => curry (exr ;; exl)
  | ccl_exr => curry (exr ;; exr)

  (* Monoidal *)
  | ccl_cross t1 t2 =>
    ⟨ ccl_ccc t1, ccl_ccc t2 ⟩ ;;
      curry (⟨ ⟨ exl;;exl, exr⟩, ⟨ exl;;exr, exr ⟩ ⟩;; cross ev ev)

  (* Closed *)
  | ccl_ev => curry (exr ;; ev)
  | ccl_env t' =>
    curry (curry (sym ;; assoc2 ;; cross (ccl_ccc t') id ;; ev))

  (* Const *)
  | ccl_const t' => ccl_ccc t';; curry exl
  end.
```

Code snippet 11: CCL to CCC translation

what τ_2. Essentially, starting from an input vector and the derivative of the output variable, reverse-mode AD calculates the derivatives of each of the input variables. On the type level, this can be formulated as transforming combinators $\mathsf{comb}\ \tau\ \sigma$ into a tuple of combinators $(\mathsf{comb}\ (fst(\overleftarrow{\mathcal{D}}(\tau)))\ (fst(\overleftarrow{\mathcal{D}}(\sigma))), \mathsf{comb}\ (fst(\overleftarrow{\mathcal{D}}(\tau)) \times snd(\overleftarrow{\mathcal{D}}(\sigma)))\ (snd(\overleftarrow{\mathcal{D}}(\tau))))$.

Defining this transformation poses an issue however, as we will illustrate with an example. Consider the combinators `exl`, `exr` and `dupl` from section 5.2.1.

$$\begin{aligned}\overleftarrow{\mathcal{D}}(\mathsf{exl}) &= (\mathsf{exl}, \mathsf{dupl};;\mathsf{cross}\ \mathsf{exr}\ (\mathsf{neg};;\bar{0}))\\ \overleftarrow{\mathcal{D}}(\mathsf{exr}) &= (\mathsf{exr}, \mathsf{dupl};;\mathsf{cross}\ (\mathsf{neg};;\bar{0})\ \mathsf{exr})\\ \overleftarrow{\mathcal{D}}(\mathsf{dupl}) &= (\mathsf{dupl}, \mathsf{exr};;\bar{+})\end{aligned}$$

Note the usage of the monoidal combinators $(\bar{0}, \bar{+})$. As mentioned in the previous section, these combinators encode the dual structures to the contraction and weakening rules present in programming language inference rules. This encodes how we will treat the fan-out problem, where $\bar{+} : \mathsf{comb}\ \tau \times \tau\ \tau$ combines two terms of type τ in a generic way. Likewise, $\bar{0} : \mathsf{comb}\ \mathsf{U}\ \tau$ formulates how to zero out variables of arbitrary type τ. We can define these combinators by induction on their types, where $\vec{0}$ is the regular vector consisting of just zeroes.

$$\bar{0}_\tau = \begin{cases} \mathsf{crval}\ 0 & : \tau = \mathsf{R} \\ \mathsf{cmrval}\ \vec{0} & : \tau = \mathsf{R\hat{}n} \\ \mathsf{neg} & : \tau = \mathsf{U} \\ \langle \bar{0}_\sigma, \bar{0}_\rho \rangle & : \tau = \sigma \times \rho \\ \mathsf{curry}\ (\mathsf{exl};;\bar{0}_\rho) & : \tau = \sigma \Rightarrow \rho \end{cases}$$

Defining $\bar{0}_\tau$ is straightforward as we only need to ensure that it preserves the structure of our types For ground types `R` and `R^n`, we generate their respective interpretations of zero. For function types, we can ignore the input of argument type and recursively call $\bar{0}$ on the result type.

$$\bar{+}_\tau = \begin{cases} \mathsf{cplus} & : \tau = \mathsf{R} \\ \mathsf{cmplus} & : \tau = \mathsf{R\hat{}n} \\ \mathsf{neg} & : \tau = \mathsf{U} \\ \langle\langle \mathsf{exl};;\mathsf{exl}, \mathsf{exr};;\mathsf{exl} \rangle;;\bar{+}_\sigma, & : \tau = \sigma \times \rho \\ \quad \langle \mathsf{exl};;\mathsf{exr}, \mathsf{exr};;\mathsf{exr} \rangle;;\bar{+}_\rho \rangle & \\ \mathsf{curry} & \\ (\langle\langle \mathsf{exl};;\mathsf{exl}, \mathsf{exr} \rangle;;\mathsf{ev}, & : \tau = \sigma \Rightarrow \rho \\ \quad \langle \mathsf{exl};;\mathsf{exr}, \mathsf{exr} \rangle;;\mathsf{ev} \rangle;;<+>_\rho) & \end{cases}$$

With $\bar{+}_\tau$, we can make recursive usage of the operator to combine subterms. For tuples, this involves creating a new tuple where the left and right components consist of, respectively, the combinations of all left and right projections. With function types, both left and right input functions need to be evaluated separately before they can be combined.

We can reuse the same reasoning we used to derive how the combinators should be transformed for reverse-mode, to define the macro on the function types. This reasoning leads to the incomplete and incorrect definition of

$$\overleftarrow{\mathcal{D}}(\tau \Rightarrow \sigma) = (fst(\overleftarrow{\mathcal{D}}(\tau)) \Rightarrow (fst(\overleftarrow{\mathcal{D}}(\sigma)) \times (snd(\overleftarrow{\mathcal{D}}(\sigma)) \Rightarrow snd(\overleftarrow{\mathcal{D}}(\tau)))), ?) \quad (7)$$

In a sense, the tangent value at function types needs to keep track of the possibly incomplete adjoints currently calculated as these can be combined with other adjoints down the line. Remember that in reverse-mode AD, the problem of fan-out involves combining each usage of a variable in the reverse pass. We solved the issue of combining variable usages using the $\bar{+}$ combinator, but still have to find something to keep track of which terms to combine.

Vákár pose the usage of a limited variant of linear types as the target of the reverse-mode macro[45]. They include both linear function types, $\multimap$, and tensor products, $!(-) \otimes (-)$.

5.2.3 Attempt at a Formalized Proof

6 Discussion

6.1 Future Work

6.2 Conclusion

References

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

[2] P.-L. Curien, "Typed categorical combinatory logic," in *Mathematical Foundations of Software Development*, H. Ehrig, C. Floyd, M. Nivat, and J. Thatcher, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1985, pp. 157–172, ISBN: 978-3-540-39302-3 (cit. on pp. 33, 35).

[3] 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).

[4] 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).

[5] 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).

[6] A. D. Gordon, "A mechanisation of name-carrying syntax up to alpha-conversion," in *Higher Order Logic Theorem Proving and Its Applications*, J. J. Joyce and C.-J. H. Seger, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 413–425, ISBN: 978-3-540-48346-5 (cit. on p. 13).

[7] J. Despeyroux, A. Felty, and A. Hirschowitz, "Higher-order abstract syntax in coq," in *Typed Lambda Calculi and Applications*, M. Dezani-Ciancaglini and G. Plotkin, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 124–138, ISBN: 978-3-540-49178-1 (cit. on p. 13).

[8] J. McKinna and R. Pollack, "Some lambda calculus and type theory formalized," *BRICS Report Series*, vol. 4, no. 51, Jun. 1997. DOI: 10.7146/brics.v4i51.19272. [Online]. Available: https://tidsskrift.dk/brics/article/view/19272 (cit. on p. 13).

[9] J. Karczmarczuk, "Functional differentiation of computer programs," in *Higher-Order and Symbolic Computation*, 1998, pp. 195–203 (cit. on p. 13).

[10] 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).

[11] J. Karczmarczuk, *Lazy time reversal, and automatic differentiation*, 2000. [Online]. Available: https://karczmarczuk.users.greyc.fr/arpap/revpearl.pdf (cit. on p. 29).

[12] M. Mayero, "Using theorem proving for numerical analysis correctness proof of an automatic differentiation algorithm," in *Theorem Proving in Higher Order Logics*, V. A. Carreño, C. A. Muñoz, and S. Tahar, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 246–262, ISBN: 978-3-540-45685-8 (cit. on p. 13).

[13] C. McBride and J. McKinna, "Functional pearl: I am not a number–i am a free variable," in *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, ser. Haskell '04, Snowbird, Utah, USA: Association for Computing Machinery, 2004, pp. 1–9, ISBN: 1581138504. DOI: 10.1145/1017472.1017477. [Online]. Available: https://doi.org/10.1145/1017472.1017477 (cit. on p. 13).

[14] ——, "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).

[15] 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 pp. 8, 13).

[16] 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).

[17] 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).

[18] 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. 10).

[19] A. Chlipala, "Parametric higher-order abstract syntax for mechanized semantics," in *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '08, Victoria, BC, Canada: Association for Computing Machinery, 2008, pp. 143–156, ISBN: 9781595939197. DOI: 10.1145/1411204.1411226. [Online]. Available: https://doi.org/10.1145/1411204.1411226 (cit. on p. 13).

[20] B. Pearlmutter and J. Siskind, "Reverse-mode ad in a functional framework: Lambda the ultimate backpropagator," *ACM Trans. Program. Lang. Syst.*, vol. 30, Jan. 2008 (cit. on pp. 13, 29).

[21] 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).

[22] 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. 13).

[23] C. Elliott, "Beautiful differentiation," in *International Conference on Functional Programming (ICFP)*, 2009. [Online]. Available: http://conal.net/papers/beautiful-differentiation (cit. on p. 13).

[24] 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, 10).

[25] 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).

[26] 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).

[27] 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).

[28] 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. 13).

[29] 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 pp. 5, 13).

[30] 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).

[31] C. Elliott, "Compiling to categories," in *Proceedings of the ACM on Programming Languages (ICFP)*, 2017. [Online]. Available: http://conal.net/papers/compiling-to-categories (cit. on p. 33).

[32] ——, "The simple essence of automatic differentiation," in *Proceedings of the ACM on Programming Languages (ICFP)*, 2018. [Online]. Available: http://conal.net/papers/essence-of-ad/ (cit. on pp. 14, 33).

[33] 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, ch. Stlc (cit. on p. 8).

[34] M. Abadi and G. D. Plotkin, "A simple differentiable programming language," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, Dec. 2019. DOI: 10.1145/3371106. [Online]. Available: https://doi.org/10.1145/3371106 (cit. on p. 13).

[35] 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).

[36] C. C. Margossian, "A review of automatic differentiation and its efficient implementation," *ArXiv*, vol. abs/1811.05031, 2019 (cit. on p. 13).

[37] 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, 13, 16, 23, 26).

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

[39] 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. 10).

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

[41] F. Wang, D. Zheng, J. Decker, X. Wu, G. M. Essertel, and T. Rompf, "Demystifying differentiable programming: Shift/reset the penultimate backpropagator," *Proc. ACM Program. Lang.*, vol. 3, no. ICFP, Jul. 2019. DOI: 10.1145/3341700. [Online]. Available: https://doi.org/10.1145/3341700 (cit. on p. 13).

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

[43] 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, 18).

[44] 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, 16, 18).

[45] M. Vákár, *Reverse ad at higher types: Pure, principled and denotationally correct*, 2020. arXiv: 2007.05283 [cs.PL] (cit. on pp. 4, 14, 33, 41).

[46] 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. 10).