Utrecht University

Master Thesis

Formalized Correctness Proofs of Automatic Differentiation in Coq

Student:
Curtis Chin Jen Sem
5601118

Supervisors:
Matthijs Vákár
Wouter Swierstra

Department of Information and Computing Science

Abstract

In this thesis, we give a formalized proof of correctness of both a ubiquitous forward-mode and a continuation-based pseudo-reverse-mode automatic differentiation algorithm. We repeatedly do this using logical relations arguments accompanied by simple but effective language representations and denotational semantics. We also discuss and prove sound various program transformations, which in the context of efficient code generation for automatic differentiation, let forward-mode approach the performance of reverse-mode algorithms. Finally, we make preliminary steps towards a formalized proof of correctness of a real combinator-based reverse-mode algorithm.

Contents

1 Introduction

Automatic differentiation is a well-known technique within the scientific community with diverse applications such as Bayesian inference and solving systems of non-linear algebraic equations. It has received increased interest due to recent developments in machine learning research, where solving optimization problems is the primary goal. One of the algorithms involved in this area of research is known as backpropagation. Backpropagation directly corresponds to reverse-mode automatic differentiation, which, in most cases, is the most efficient method to compute the derivatives of a function, critical in optimization problems. However, programming in an environment that allows for automatic differentiation can be limiting.

Frameworks such as Tangent[1] or autograd[2] are define-by-run algorithms, whose main tactic is to build up the derivative calculation dynamically during runtime. This process can restrict which high-level optimizations one can 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. Exposing these optimization techniques to the world of automatic differentiation can be very fruitful as these calculations are costly and often require significant computing power to run. Through other concepts such as higher-order functions and type systems, we would also get additional benefits such as code-reusability and correctness.

While much research has already been done to integrate programming language theory with automatic differentiation, formalizations of these techniques are absent. In this thesis, we aim to formalize an extensible correctness proofs of various implementations 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 [48]. 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 [48] 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.[41].
- Analyze both the requirements of and issues involved with giving a formal proof of correctness for the combinator-based reverse-mode automatic differentiation algorithm by Vákár[50].

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

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

transformations and a continuation-based automatic differentiation algorithms. Finally, section 6 gives our attempt at a formal proof of the combinator-based reverse-mode automatic differentiation algorithm.

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

2 Background

2.1 Automatic differentiation

Automatic differentiation (*AD*) has a long and rich history, where its driving motivation is to efficiently calculate the derivatives of higher-dimensional functions in a manner that is both correct and efficient in terms of space and time[33]. 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, we annotate every term in the function trace with their corresponding derivative. These are also known as, respectively, the *primal* and *tangent* traces. So calculating the partial derivatives of sub-terms is structural with respect to normal calculations.

The core approach to forward-mode AD can be explained by what is mathematically known as dual numbers, as these are, in essence, what we are calculating[33]. Dual numbers are numbers in the form of

$$x + x'\epsilon$$

where $x, x' \in \mathcal{R}$ and ϵ is a formal construction, such that $\epsilon^2 = 0$ and $\epsilon \neq 0$. Using the standard ring axioms of addition and multiplication, we can uniquely evaluate any polynomial containing ϵ into the form $x + x'\epsilon$ where neither x nor x' contains ϵ. Notably, both the primal and tangent values are tracked in this representation as x and x'. 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 differentiable function by interpreting any non-dual number x as its dual number counterpart. Respectively, we

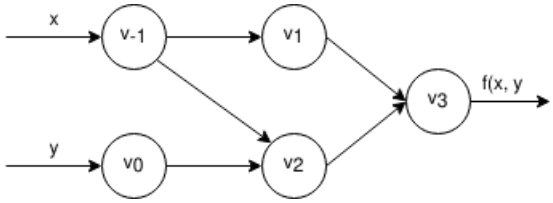

Figure 1: Computational graph of $f(x, y) = x^2 + (x - y)$

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

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 forward-mode AD works graphically, take the function $f(x, y) = x^2 + (x - y)$ as an example. The dependencies between the terms and operations of the function is visible in the computational graph in 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, which means that when we calculate 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, we require two separate passes. Like the forward-mode variant, a primal trace is needed. This first pass acts 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. The adjoints of multiple usages of the same variables, also called fan-out, are combined using addition in the adjoint trace.

The optimal choice between the automatic differentiation variants is heavily dependent on the function under consideration. 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, reverse-mode AD scales with the number of output variables as it works backwards from each one. For example, much machine learning research boils down to optimizing an objective function with just

a single output variable, but many input variables. As such, reverse-mode AD is the optimal choice.

Implementing reverse-mode AD is more problematic than the straightforward forward-mode AD algorithms. Correctly creating the reverse pass such that it handles both fan-out and zeroing in the reverse pass is complicated. As such, many frameworks targeting non-theoretical applications such as deep learning focus on imperative languages, which eases the process due to the additional concept of state.

There are two main styles of implementing reverse-mode AD, namely define-then-run and define-by-run. Define-then-run attempts to create the static computational graph during the compilation process that transforms the program to one that calculates the derivative of the function the program describes. One significant benefit to this approach is the added opportunity to apply optimization steps in the compilation process. Define-then-run frameworks do, however, have the restriction that programs defined within these frameworks are limited in their usage of intricate control flow as is apparent in the expressive power of TensorFlow[3] .

This restriction is lifted in the define-by-run style, which builds the computation graph during runtime. As such, conditionals and loops can now be used freely in the framework. On the other hand, as the computation graph is constructed during runtime, define-by-run frameworks can be slower due to the missing opportunities to apply optimizations. Frameworks using this style are Chainer[4] and PyTorch[5]

2.2 Denotational semantics

Denotational semantics enables reasoning about programs using formal mathematics. It also functions as a hotbed for new and innovative language designs and algorithms. The most well-known example is the domain theory model given by Dana Scott and Christopher Strachey[1] for lambda calculi. To be able to formalize non-termination and partiality, they thought to use concepts such as partial orderings and least fixed points[46]. 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 $\mathtt{R}$ as the syntactic type for real numbers, while $\mathcal{R}$ is its denotational counterpart. Using Cartesian spaces, however, does not work when function types are added as their denotational equivalent, function spaces, are not supported by Cartesian spaces[48]. In the original pen and paper proof of automatic differentiation this thesis is based on by Huot, Staton and Vákár[48], 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.

[3] https://www.tensorflow.org/
[4] https://chainer.org/
[5] https://pytorch.org/
[6] 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

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[6] . 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`[34], 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[28] and advanced modular constructions for organizing large mathematical proofs[25][31].

The core of this type theory is based on constructive logic and so many of the laws known in classical logic are not provable. An example includes the law of the excluded middle, $\forall A, A \vee \neg A$. In 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 heavy 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[44]. 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.[37]. This representation, 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[16]. One example of such an approach is the nominal representation where every variable is named. Code code snippet 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.

```
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.

The approach used in the rest of this thesis is an extension of the De-Bruijn representation which numbers variables relative to the binding lambda term. In this representation the variables are referred to as well-typed De-Bruijn indices. A significant benefit of this representation is that the problems of capture avoidance and alpha equivalence are avoided. As an alternative, instead of using numbers to represent the distance, indices within the typing context can be used to ensure that a variable is always well-typed and well-scoped. While the idea of using type indexed terms has been both described and used by many authors[11][14][17], the specific formulation used in this thesis using separate substitution and renaming operations was fleshed out in *Coq* by Nick Benton et al.[29], and was also used as one of the examples in the second POPLmark challenge which deals with logical relations[39]. 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.

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 complexity has

```
Inductive ∈ : 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.

historically been caused by the poor support in *Coq* for dependent pattern matching. Using the return keyword, one can 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 approach also has its limitations as you then lose computability as *Coq* treats these definitions opaquely. In this case, the standard `simpl` tactic which invokes the reduction mechanism is not able to reduce instances of the term. Instead the user is required to write many more proofs to be able to work with such 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 code snippet 3. Using the *Coq* keyword return, it is possible to let the return type of a match expressions depend on the result of one of the type arguments. This makes it possible to define an auxiliary function which, while total on the length of the list, has an incorrect return type. 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[19][20]. 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[28][43]. 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

```
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[30].

```
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`

correct, which is the approach used by Sozeau. Due to the intrinsic typeful representation this paper uses, much of this was invaluable when defining the substitution operators as the regular type checker in *Coq* often had difficulty unifying dependently typed terms in certain cases.

2.4 Logical relations

Logical relations arguments are a proof technique often employed when proving programming language properties of statically typed languages[42]. 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[18], to open relations which enable working with terms of non-ground type[47][48]. 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

```
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

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 code snippet 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 exists proofs of forward-mode AD algorithms[48][47][38] and many more implementations[41][40] in a functional setting, there have been relative few attempts at formalized proofs in proof assistant. In 2002, Mayero gave a formalized correctness proof of an AD framework implemented in *Fortran* in *Coq*[13]. Their minimal core language 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[16]. Examples include the weak higher-order abstract syntax approach worked out in Coq by Despeyroux et al.[8], which shallowly embeds abstractions as functions $abs : (var \to tm) \to tm$. The parametric HOAS variant by Chlipala[21], is an interesting polymorphic generalization of this technique. PHOAS, 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[9][7][15] 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.[26] and Dockins[32] present domain-theoretical libraries in *Coq*. Proof-wise, Huot, Staton and Vákár use diffeological spaces in their pen-and-paper correctness proof of automatic differentiation on a simply-typed lambda calculus. Contrastingly, Abadi and Plotkin use the more conventional ω-cpos to be able to support partiality[38].

Forward-Mode AD. The earliest found description of an approach for forward-mode AD on functional languages is by Karczmarczuk[10], on first-order terms. Siskind and Pearlmutter presented a nestable variant of the forward-mode AD algorithm using the dual numbers representation[23]. This same algorithm is used in the *F#* library, DiffSharp[33]. A nearly identical variation, implemented in *Haskell*, is given by Elliott[27]. This uncontroversial implementation of forward-mode AD is also discussed in the survey by Baydin et al.[33].

Reverse-Mode AD. There are many interpretations of reverse-mode AD on functional languages. Most well-known is the one by Pearlmutter and Siskind[24], 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 runtime, their primitive $\overleftarrow{\mathcal{J}}$ uses reflection to perform reverse-mode AD at runtime.

Define-by-run algorithms, however, lose much of the optimization opportunities provided by the explicit compilation process involved with programming languages. As an improvement on the approach by Pearlmutter and Siskind, Wang et al.[45] negate some of the issues associated with define-by-run algorithms by using multi-stage programming to reclaim some optimization opportunities. Their algorithm makes heavy usage of both delimited continuations as well as state by way of references.

Abadi and Plotkin[38] also make use of reverse-mode AD primitives, but do so in the context of a define-by-run trace-based algorithm. Their reverse-mode primitive is given special treatment in their operational semantics, essentially symbolically evaluating terms into so-called trace terms, which are devoid of control-flow constructs.

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[36] 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[50] is further discussed in section 6.

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

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

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

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

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

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

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

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

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. These are shown as typing judgments 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. The included operations are chosen for their simplicity, but the proof is able to accommodate any operation so long as it is total and differentiable.

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[29][6], 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 snippet 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 [48], 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. The code for implemented products and included operations on real numbers is given in code snippet 6.

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

```
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.

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

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

Figure 4: Macro on base simply-typed lambda calculus

authors[10][23][41] to implement the forward-mode automatic differentiation macro. 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, $\overrightarrow{\mathcal{D}}(\Gamma) \vdash \overrightarrow{\mathcal{D}}(t) : \overrightarrow{\mathcal{D}}(\tau)$.

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. The macro on types as well as real numbered values is given in fig. 4.

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,

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

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

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

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

Denotating the terms in our language now corresponds to finding the appropriate inhabitants in the denotated types. As typing contexts, 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[30]. In this implementation, heterogeneous lists consist of an underlying list of some type A and an accompanying function $A \to Set$, which in our use case are, respectively, the typing context and the denotation function.

When giving the constructs in our language their proper denotations, most of the cases are straightforward. Notable is the case for variables, where we made use of the inductively defined type evidence to type our terms. Remember that to type variables in our term 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. The denotations for the various types and terms in our base language is shown in fig. 5

```
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.[47] 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[48]. 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 $x_1 : \mathtt{R}, \ldots, x_n : \mathtt{R} \vdash t : \mathtt{R}$ 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 $\mathtt{R}$, t itself can consist of subterms of higher-order types.

Proving this correctness statement directly by induction on the structure of our small language, however, does not work. Instead, we use logical relations argument. The logical relation will ensure that both the smoothness and the derivative calculating properties are preserved over higher-order types. As expected, when a logical relations argument is required, the indicative issue lies in the cases for function application. To be specific, the induction hypothesis would indicate that both the argument and the function terms would satisfy our notion of correctness. The induction hypothesis, however, would be too weak to establish this property for the result of the application. To get the proof to go through, we need to strengthen the induction hypothesis at function types to include precisely the assertion that if the argument of our function application is valid with respect to the logical relation, then the result, too, is valid.

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 $R \to [\![\overrightarrow{\mathcal{D}}(\tau)]\!]$. For our ground types, $\mathtt{R}$, the logical relation should establish that the macro-applied term should be denotationally equivalent to the original denotation and its corresponding derivative. Some care has to be taken when defining the relation for product types. 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 $R \to [\![\tau \times \sigma]\!]$. One also has to supply proofs that both subterms also satisfy the logical relation.

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

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

The next step involves proving that the well-typed terms, which correspond to functions $\mathcal{R}^n \to \mathcal{R}$, are semantically correct with respect to the logical relation. In other words, the relation needs to be proven valid for any term $x_1 : \mathtt{R}, \ldots, x_n : \mathtt{R} \vdash t : \mathtt{R}$ and argument function $f : \mathcal{R} \to \mathcal{R}^n$ such that $S_\tau([\![t]\!] \circ f, [\![\overrightarrow{\mathcal{D}}(t)]\!] \circ \overrightarrow{\mathcal{D}}_n \circ f)$. This proposition is what we will use to derive our final correctness statement once we generalize it to arbitrary types, 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 $s : \mathcal{R} \to [\![\Gamma]\!]$ and $s_D : \mathcal{R} \to [\![\overrightarrow{\mathcal{D}}(\Gamma)]\!]$, 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) Argument functions $s : R \to [\![\Gamma]\!]$ and $s_D : R \to [\![\overrightarrow{\mathcal{D}}(\Gamma)]\!]$ are inductively defined such that they follow*

$$\mathit{inst}_\Gamma(f,g) = \begin{cases} f = (\lambda x.[\,]) \wedge g = (\lambda x.[\,]) & : \Gamma = [\,] \\ \forall f_1, f_2, g_1, g_2. & : \Gamma = (\tau :: \Gamma') \\ \quad \mathit{inst}_{\Gamma'}(f_1, g_1) \wedge S_\tau(f_2, g_2) & \\ \quad\quad \to f = (\lambda x.f_2(x) :: f_1(x)) \wedge & \\ \quad\quad\quad g = (\lambda x.g_2(x) :: g_1(x)) & \end{cases} \tag{1}$$

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

Lemma 1 (Fundamental lemma)**.** *For any well-typed term $\Gamma \vdash t : \tau$, and instantiation functions $s : \mathcal{R} \to [\![\Gamma]\!]$ and $s_D : \mathcal{R} \to [\![\overrightarrow{\mathcal{D}}(\Gamma)]\!]$ such that they follow $\mathit{inst}_\Gamma(s, s_D)$, 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 now step-by-step specialize the fundamental lemma to our final correctness theorem. The first step involves limiting the arguments of the functions to real numbers. The previous need to use the instantiation relation to establish that the typing context satisfies the logical relation disappears if we specialize these contexts to real numbers. In its place, we have to manually establish the connection between real numbers and their dual number counterparts. We establish this connection using a denotational translation from real to dual numbers in the definition of $\overrightarrow{\mathcal{D}}^{arg}$, as shown in definition 3. This function couples each primal value with their corresponding tangent value.

Definition 3 (Initialization of dual number representations)**.** *For any n-length argument function $f : \mathcal{R} \to [\![\mathsf{R}^n]\!]$, couple each argument with its corresponding tangent such that the resulting transformed function becomes $\overrightarrow{\mathcal{D}}^{arg}_n(f, x) : \mathcal{R} \to [\![\overrightarrow{\mathcal{D}}(\mathsf{R})^n]\!]$. This can be defined as the following higher-order function:*

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

Note that because we make use of the derivatives of our arguments, we also need to require that every argument be differentiable. This requirement is non-negotiable and will be present in the theorem. We formulate this requirement as definition 4.

Definition 4 (Differentiability of arguments)**.** *Any argument function $f : \mathcal{R} \to [\![\mathsf{R}^n]\!]$ with arity n, is inductively constructed such that it follows the following proposition.*

$$differentiable_n(f) = \begin{cases} differentiable_0(\lambda x.[]) & : n = 0 \\ \forall g, h. differentiable_{n'}(g) \to & : n = n' + 1 \\ \quad smooth\ h \to & \\ \quad f = (\lambda x. h(x) :: g(x)) & \end{cases} \tag{2}$$

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 $f : \mathcal{R} \to \mathcal{R}^n$ that follows differentiable$_n(f)$, 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 $\text{inst}_{\mathsf{R}^n}$. 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 $\text{inst}_{[]}$. The induction step is proven by both the induction hypothesis and the assumption that the denotations of the arguments supplied are smooth. □

The final step includes specializing the output variable of our functions to real numbers. Simplifying the logical relation for `R` gives the correctness statement we set out to prove.

Corollary 2 (Macro correctness)**.** *For any well-typed 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 $f : \mathcal{R} \to \mathcal{R}^n$ that follows differentiable$_n(f)$, we have that $[\![\overrightarrow{\mathcal{D}}(t)]\!] \circ \overrightarrow{\mathcal{D}}_n \circ f = \lambda x.([\![t]\!] \circ f, {}^{\partial([\![t]\!] \circ f)}/_{\partial x})$.*

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

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

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

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

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

Proof. This is proven by showing that the goal follows from the logical relation which itself is implied by corollary 1.

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

□

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 sum types. In the interest of testing the flexibility of both the representation and the proof technique, we also add natural number types and primitive recursion. The inference rules for the new language constructs added for sum and number types are given in fig. 6, while their intrinsic representations are shown in code snippet 7.

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 we first destruct the sum term 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. The specific denotations chosen is given in fig. 7.

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

```
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)
  (* 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 7: Terms in our language related to sum types.

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

Figure 7: Denotations of the sum and integer terms

$$
\begin{aligned}
&\dots \\
\overrightarrow{\mathcal{D}}(\tau <+> \sigma) &= \overrightarrow{\mathcal{D}}(\tau) <+> \overrightarrow{\mathcal{D}}(\sigma) \\
\overrightarrow{\mathcal{D}}(\mathsf{N}) &= \mathsf{N}
\end{aligned}
$$

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

Figure 8: Macro on the sum and integer terms

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, which we can neatly define using a logical disjunction.

$$
S_\tau(f,g) = \begin{cases}
\dots & \\
f = g \wedge \exists n. f = const(n) & : \tau = \mathsf{N} \\
(\exists f_1, f_2, & \\
\quad S_\sigma(f_1, f_2) \wedge f = inl \circ f_1 \wedge g = inl \circ g_1) & \\
\qquad \vee & : \tau = \sigma <+> \rho \\
(\exists f_1, f_2, & \\
\quad S_\rho(f_1, f_2) \wedge f = inr \circ f_1 \wedge g = inr \circ g_1) &
\end{cases} \tag{3}
$$

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

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

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

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 8: The terms related to array types included in our language

extension worth considering in our language are array types. To be more specific, we will be considering the pull array formulation used by Shaikhha et al.[41].

Pull-arrays are a well-known formulation which lends itself well to deforestation[5]. The most significant element of this concept is to encode arrays as functions, which in some cases can be composed with other similarly encoded arrays. The two operations we will be focussing on are `build` and `get`. Note that Ix represents some bounded integer type which should correspond to the arity of the arrays the operations are used on.

$$\text{build} : (Ix \to A) \to [A]$$
$$\text{get} : Ix \to [A] \to A$$
$$\text{get}\ i\ (\text{build}\ f) \equiv f(i)$$

This formulation is straightforward to encode in the intrinsic representation, as shown in code snippet 8. We are able to omit much of their cardinality and indexing type-level considerations due to our shallow embedding of pull-arrays into the existing types in *Coq*. The well-typed nature of our representation allows us to avoid much of the hairy details associated with bounds checking for both indexing and array creation. We accomplish this using the `Fin` n inductive datatype, which represents the range $[1..n]$.

The final iteration of our simply-typed lambda calculus including the array types is an expansion of the $\tilde{F}$ language by Shaikha et al. as ours also includes sum types. We can define the missing `ifold` and `let` constructs as syntactic sugar using the primitive recursion terms we added in section 3.2. Note that due to our nameless representation, `let` terms lose much of their original usefulness in writing programs.

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

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

Figure 10: Macro on array construction and indexing terms

$$
\begin{aligned}
&\ldots \\
[\![\texttt{Array}\ n\ \tau]\!] &= \text{vector}(n, [\![\tau]\!]) \\
[\![\texttt{build}\ n\ f]\!] &= \lambda x.\text{vect}(n, [\![]\!] \circ f, x) \\
[\![\texttt{get}\ i\ t]\!] &= \lambda x.[\![t]\!](x)\ !\ i \\
\text{vect}(i, f, x) &= \begin{cases} [\,] & : i = 0 \\ f(i)(x)\ ::\ \text{vect}(i', \lambda j.f(j+1), x) & : i = i' + 1 \end{cases}
\end{aligned}
$$

Figure 11: Denotations of the array construction and indexing terms

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. Instead, we substitute 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. In fig. 11 we formulate this as $\text{vector}(n, T)$, where n indicates the length of the vector and T is the contained type. 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 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 $1 : \texttt{Fin}\ n$, and any $i : \texttt{Fin}\ n$ is transformed to $n : \mathcal{N}$.

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

$$
S_\tau(f, g) = \begin{cases} \ldots & \\ \forall i.\exists f_1, g_1. & : \tau = \texttt{Array}\ n\ \sigma \\ \quad S_\sigma(f_1, g_1) \wedge & \\ \quad\quad f_1 = \lambda x.f(x)!i \wedge & \\ \quad\quad f_1 = \lambda x.g(x)!i & \end{cases} \tag{4}
$$

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

4 Optimizing through Program Transformations

Shaikhha et al. have presented a small system which has proven to be performant[41]. 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. One of the key reasons their system is performant is due to the usage of various program optimizations. In section 3.3 we already formalized one of the components of their system, namely their usage of array types. Next, we will prove that the various program transformation rules they use in their system are sound for our simple denotational semantics. Their seemingly low-level transformations facilitate various algebraic identities on matrices and vectors. The transformation rules consist of several algebraic identities on real numbers, along with compile-time optimization techniques such as partial evaluation and deforestation or loop fusion.

In this proof, we will reuse the denotational semantics from our proof of the forward-mode macro in section 3. As a small deviation from the rules given by Shaikhha et al.[41], we explicitly include a set of simplification rules in the style of a big-step operational semantics. We use these simplification rules to facilitate partial evaluation on non-functional terms. Note that partial evaluation on function terms is included as a separate transformation rule. figs. 12 and 13 shows, 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 the 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 1 (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,

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

(a) Algebraic laws

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

(b) Reuse rewriting rules from the evaluation relation

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

(c) Partial evaluation on functions

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

(d) Loop fusion

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

(e) Loop fission

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

$$\frac{t_1 \Downarrow \texttt{abs}\ t_1' \qquad t_2 \Downarrow t_2'}{\texttt{app}\ t_1\ t_2 \Downarrow \texttt{substitute}\ t_2'\ t_1'}\ \text{EVAppAbs}$$

$$\frac{t \Downarrow \texttt{nval}\ n}{\texttt{nsucc}\ t \Downarrow \texttt{nval}\ (n+1)}\ \text{EVSucc}$$

$$\frac{t_1 \Downarrow t_1' \qquad t_2 \Downarrow \texttt{nval}\ 0 \qquad t_3 \Downarrow t_3'}{\texttt{nrec}\ t_1\ t_2\ t_3 \Downarrow t_3}\ \text{EVNRec0}$$

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

$$\frac{t_1 \Downarrow \texttt{rval}\ r_1 \qquad t_2 \Downarrow \texttt{rval}\ r_2}{\texttt{add}\ t_1\ t_2 \Downarrow \texttt{rval}\ r_1 + r_2}\ \text{EVAdd} \qquad \frac{t_1 \Downarrow \texttt{rval}\ r_1 \qquad t_2 \Downarrow \texttt{rval}\ r_2}{\texttt{mul}\ t_1\ t_2 \Downarrow \texttt{rval}\ r_1 * r_2}\ \text{EVMult}$$

$$\frac{t_1 \Downarrow t_1' \qquad t_2 \Downarrow t_2'}{\texttt{tuple}\ t_1\ t_2 \Downarrow \texttt{tuple}\ t_1'\ t_2'}\ \text{EVTuple}$$

$$\frac{t \Downarrow \texttt{tuple}\ t_1\ t_2}{\texttt{first}\ t \Downarrow t_1}\ \text{EVFst} \qquad \frac{t \Downarrow \texttt{tuple}\ t_1\ t_2}{\texttt{second}\ t \Downarrow t_2}\ \text{EVSnd}$$

$$\frac{t \Downarrow t'}{\texttt{inl}\ t \Downarrow \texttt{inl}\ t'}\ \text{EVInl} \qquad \frac{t \Downarrow t'}{\texttt{inr}\ t \Downarrow \texttt{inr}\ t'}\ \text{EVInr}$$

$$\frac{t \Downarrow \texttt{inl}\ t' \qquad t_1 \Downarrow t_1' \qquad t_2 \Downarrow t_2'}{\texttt{case}\ t\ t_1\ t_2 \Downarrow \texttt{app}\ t_1'\ t'}\ \text{EVCaseInl}$$

$$\frac{t \Downarrow \texttt{inr}\ t' \qquad t_1 \Downarrow t_1' \qquad t_2 \Downarrow t_2'}{\texttt{case}\ t\ t_1\ t_2 \Downarrow \texttt{app}\ t_2'\ t'}\ \text{EVCaseInr}$$

Figure 13: Inference rules for the evaluation relation

$$
\begin{aligned}
\overrightarrow{\mathcal{D}}^c_n(\mathsf{R}) &= \mathsf{R} \times (\mathsf{R} \Rightarrow \mathtt{Array}\ n\ \mathsf{R}) \\
\overrightarrow{\mathcal{D}}^c_n(\tau \times \sigma) &= \overrightarrow{\mathcal{D}}(\tau) \times \overrightarrow{\mathcal{D}}(\sigma) \\
\overrightarrow{\mathcal{D}}^c_n(\tau <+> \sigma) &= \overrightarrow{\mathcal{D}}(\tau) <+> \overrightarrow{\mathcal{D}}(\sigma) \\
\overrightarrow{\mathcal{D}}^c_n(\tau \Rightarrow \sigma) &= \overrightarrow{\mathcal{D}}(\tau) \Rightarrow \overrightarrow{\mathcal{D}}(\sigma) \\
\overrightarrow{\mathcal{D}}^c_n(\mathtt{Array}\ m\ \tau) &= \mathtt{Array}\ m\ \overrightarrow{\mathcal{D}}^c_n(\tau) \\
&\cdots \\
\overrightarrow{\mathcal{D}}^c_n(\mathtt{rval}\ r) &= \mathtt{tuple}\ (\mathtt{rval}\ r) \\
&\quad (\mathtt{build}\ (\mathtt{const}\ (\mathtt{rval}\ 0)))\ ) \\
\overrightarrow{\mathcal{D}}^c_n(\mathtt{add}\ r_1\ r_2) &= \mathtt{tuple}\ (\mathtt{add}\ r_1\ r_2)\ (\mathtt{abs}\ (\mathtt{vector\_add} \\
&\quad (\mathtt{app}\ r'_1\ (\mathtt{var}\ \mathtt{Top}))\ (\mathtt{app}\ r'_2\ (\mathtt{var}\ \mathtt{Top})))) \\
\overrightarrow{\mathcal{D}}^c_n(\mathtt{mul}\ r_1\ r_2) &= \mathtt{tuple}\ (\mathtt{mul}\ r_1\ r_2)\ (\mathtt{abs}\ (\mathtt{vector\_add} \\
&\quad (\mathtt{app}\ r'_1\ (\mathtt{mul}\ r_2\ (\mathtt{var}\ \mathtt{Top}))) \\
&\quad (\mathtt{app}\ r'_2\ (\mathtt{mul}\ r_1\ (\mathtt{var}\ \mathtt{Top})))))
\end{aligned}
$$

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

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 Relating a Continuation-Based Algorithm

Huot, Staton and Vákár showed the versatility of their denotational semantics on a continuation-based AD algorithm whose origin can be traced back to a description given by Karczmarczuk[12]. 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[24]. A 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.

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 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.

As we will phrase correctness in terms of the forward-mode macro, the forward-mode

macro used in section 3.1 is 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 is essentially coupled with a one-hot encoded vector.

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

Each of the one-hot encoded vectors indicates which partial derivative to calculate. The continuation-based macro requires a similar format, but instead of a one-hot encoded vector containing the value 1, the identity function is used.

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

Using these two input massaging functions, we can state the correctness of our continuation-based macro relative 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* $\vec{x} : [\![\mathsf{R}^n]\!]$ *we will take the partial derivatives relative to, we have that* $snd([\![\overrightarrow{\mathcal{D}}_n(t)]\!](\overrightarrow{\mathcal{D}}_n^{arg}(\vec{x}))) = snd([\![\overrightarrow{\mathcal{D}}_n^c(t)]\!](\overrightarrow{\mathcal{D}}_n^{arg_c}(\vec{x})))(1)$.

Unfortunately, this statement cannot be proven directly. To see why, consider multiplication where we would have to prove

$$snd([\![\overrightarrow{\mathcal{D}}_n^c(\texttt{mul}\ t_1\ t_2)]\!] \circ f) = snd([\![\overrightarrow{\mathcal{D}}_n^c(\texttt{mul}\ t_1\ t_2)]\!] \circ g)(1)$$

This expression can be rewritten using the definitions of our denotations to

$$\begin{aligned} &\lambda a.\text{vect}(n, \lambda i, x. fst([\![\overrightarrow{\mathcal{D}}_n(t_2)]\!](x)) * snd([\![\overrightarrow{\mathcal{D}}_n(t_1)]\!](x))\ !\ i + \\ &\qquad fst([\![\overrightarrow{\mathcal{D}}_n(t_1)]\!](x)) * snd([\![\overrightarrow{\mathcal{D}}_n(t_2)]\!](x))\ !\ i, a) = \\ &\lambda a.\text{vect}(n, \lambda i, x. snd([\![\overrightarrow{\mathcal{D}}_n^c(t_1)]\!](tl(x)))(fst([\![\overrightarrow{\mathcal{D}}_n^c(t_2)]\!](x)))\ !\ i * hd(x) + \\ &\qquad snd([\![\overrightarrow{\mathcal{D}}_n^c(t_2)]\!](tl(x)))(fst([\![\overrightarrow{\mathcal{D}}_n^c(t_1)]\!](x)))\ !\ i * hd(x), 1\ ::\ a) \end{aligned}$$

Note on the right-hand side, that the first projection of the macro on terms, which contains the primal values, is passed along the continuation in the second projection. If

$$
\begin{aligned}
\overrightarrow{\mathcal{D}}_n(\mathsf{R}) &= \mathsf{R} \times \mathsf{Array}\ n\ \mathsf{R} \\
\overrightarrow{\mathcal{D}}_n(\tau \times \sigma) &= \overrightarrow{\mathcal{D}}(\tau) \times \overrightarrow{\mathcal{D}}(\sigma) \\
\overrightarrow{\mathcal{D}}_n(\tau <+> \sigma) &= \overrightarrow{\mathcal{D}}(\tau) <+> \overrightarrow{\mathcal{D}}(\sigma) \\
\overrightarrow{\mathcal{D}}_n(\tau \Rightarrow \sigma) &= \overrightarrow{\mathcal{D}}(\tau) \Rightarrow \overrightarrow{\mathcal{D}}(\sigma) \\
\overrightarrow{\mathcal{D}}_n(\mathsf{Array}\ m\ \tau) &= \mathsf{Array}\ m\ \overrightarrow{\mathcal{D}}_n(\tau) \\
&\dots \\
\overrightarrow{\mathcal{D}}_n(\mathsf{rval}\ r) &= \mathsf{tuple}\ (\mathsf{rval}\ r)\ (\mathsf{build}\ n\ (\mathsf{fun}\ \_ \Rightarrow \mathsf{0})) \\
\overrightarrow{\mathcal{D}}_n(\mathsf{add}\ r_1\ r_2) &= \mathsf{tuple}\ (\mathsf{add}\ r_1\ r_2)\ (\mathsf{vector\_add}\ r_1'\ r_2') \\
\overrightarrow{\mathcal{D}}_n(\mathsf{mul}\ r_1\ r_2) &= \mathsf{tuple}\ (\mathsf{mul}\ r_1\ r_2) \\
&\qquad (\mathsf{vector\_add}\ (\mathsf{vector\_scale}\ r_2\ r_1')\ (\mathsf{vector\_scale}\ r_1\ r_2'))
\end{aligned}
$$

(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

we attempt to base our logical relation on proposition 1, however, the proof of this case will fail. The corresponding induction hypothesis will only account for supplying the value 1 to our continuations, which is too restricting. This issue implies that we require some generalization in both our proof goal and logical relation.

To be specific, we need to establish equivalence between the macros regardless of the argument we apply to the continuation-based macro, which corresponds to the coefficient of our forward-mode macro. We bake this equivalence directly 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 type of real numbers, R. Also note that, like the macro functions, the logical relation is also indexed by the number of input variables.

$$S_{n,\tau}(f,g) = \begin{cases} \ldots & \\ fst \circ f = fst \circ g \wedge & : \tau = \mathsf{R} \\ \quad \forall x.\lambda r.x * snd(f(r)) = \lambda r.(snd(g(r)))(x) & \end{cases} \tag{5}$$

As a recurring theme, the definitions and lemmas we used in the proof in section 3 reappear in this proof. One of the main differences is that we now have to keep track of the number of partial derivatives. So we still use an instantiation relation like the one given in eq. (1) 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. The problem here is that the number of partial derivatives is tightly coupled to the number of terms involved in the operations. This tight coupling can be solved for each of these cases by proving the corresponding generalized variant. Proving these cases, along with the corresponding fundamental property of the logical relation, results in the following theorem.

Theorem 2 (Duality)**.** *For any well-typed term $x_1 : \mathsf{R}, \ldots, x_n : \mathsf{R} \vdash t : \mathsf{R}$, function arguments $\vec{x} : [\![\mathsf{R}^n]\!]$, and real number r, we have $r * snd(\overrightarrow{\mathcal{D}}_n(t)(\overrightarrow{\mathcal{D}}_n^{arg}(\vec{x}))) = snd(\overrightarrow{\mathcal{D}}_n^c(t)(\overrightarrow{\mathcal{D}}_n^{arg_c}(\vec{x})))(r)$*

Proof. This follows from the fundamental property of the logical relation in (eq. (5)). □

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

Corollary 3 (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 $\vec{x} : [\![\mathsf{R}^n]\!]$ we will take the partial derivatives relative to, we have that $snd([\![\overrightarrow{\mathcal{D}}_n(t)]\!](\overrightarrow{\mathcal{D}}_n^{arg}(\vec{x}))) = snd([\![\overrightarrow{\mathcal{D}}_n^c(t)]\!](\overrightarrow{\mathcal{D}}_n^{arg_c}(\vec{x})))(1)$.*

6 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 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.

Eliott posed a novel and principled approach to defining an AD algorithm using *category theory*[36]. 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[35][2]. The algorithm is, however, still restricted to first order programs.

An extension of this technique is given by Vákár[50]. 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.

6.1 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 the contraction and weakening rules usually kept implicit in the typing contexts of typed lambda calculi, explicit. 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 : `comb` τ σ, 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 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. So it will also include usage of a typing context and related constructs such as variable access and function abstraction. While this CCL can be used to facilitate both back and forth translations, we will restrict our focus to only translating 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

```
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

in the typing context to become an additional argument in the resulting function type. Repeated usage of this specific combinator turns any previously open term into a closed one.

The intrinsic representation we use in our definitions makes defining the exact translations a breeze as it then 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. We translate abstractions, where the argument type is added onto the typing context, using the `ccl_env` construct. In cases where we introduce new values, we make use of the `ccl_const` construct to ensure the terms fit 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 code snippet 9.

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 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.

```
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 9: Simply-typed lambda calculus to CCL translation

```
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 code snippets 9 and 10. Note that we have to remove the extra `U` type we added in the simply-typed lambda calculus to CCL translation shown code snippet 9.

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

```
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 10: CCL to CCC translation

6.2 Defining the Macro and Target Language

The reverse-mode macro we will discuss in this chapter splits any type τ into pairs (τ_1, τ_2), where τ_1 and τ_2 represent primal and tangent values. Defining this transformation poses an issue, however, as we will illustrate with an example. Consider the combinators `exl`, `exr` and `dupl` from section 6.1. Attempting to fill this in using type-directed programming following the type signature we mentioned previously, leads to the following:

$$\overleftarrow{\mathcal{D}}(\texttt{exl}) = (\texttt{exl}, \texttt{dupl;;cross exr (neg;;}\bar{0}\texttt{)})$$
$$\overleftarrow{\mathcal{D}}(\texttt{exr}) = (\texttt{exr}, \texttt{dupl;;cross (neg;;}\bar{0}\texttt{) exr})$$
$$\overleftarrow{\mathcal{D}}(\texttt{dupl}) = (\texttt{dupl}, \texttt{exr;;}\bar{+})$$

Note the usages of the $(\bar{0}, \bar{+})$ combinators. While $\bar{0}$ is used to explicitly zero out the unused terms in both `exl` and `exr`, $\bar{+}$ is used to recombine terms in `dupl`. 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 also appropriately encodes how we treat the fan-out problem, where $\bar{+} : \texttt{comb}\ (\tau \times \tau)\ \tau$ combines two terms of type τ generically. Likewise, $\bar{0} : \texttt{comb U}\ \tau$ 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.

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

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.

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

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 behavior of our macro on function types. This reasoning leads to the incomplete definition of

```
Inductive ty : Type :=
  ...
  | !_ ⊗ _ : ty → ty → ty
  | —∘  : ty → ty → ty
.
Inductive target : ty → ty → Type :=
  ...
  (* Linear *)
  | ev_l : forall A B, target ((A —∘ B) × A) B
  | curry_l : forall A B C,
    target (A × B) C → target A (B —∘ C)
  (* Tensor *)
  | mempty : forall A B,
  target U (!A ⊗ B)
  | msingleton : forall A B,
    target (A × B) (!A ⊗ B)
  | mplus : forall A B,
    target ((!A ⊗ B) × (!A ⊗ B)) (!A ⊗ B)
  | mfold : forall A B C,
    target (!A ⊗ B × (A × B —∘ C)) C
.
```

Figure 19

$$\overleftarrow{\mathcal{D}}(\tau \Rightarrow \sigma) = (fst(\overleftarrow{\mathcal{D}}(\tau)) \Rightarrow (fst(\overleftarrow{\mathcal{D}}(\sigma)) \times (snd(\overleftarrow{\mathcal{D}}(\sigma)) \Rightarrow snd(\overleftarrow{\mathcal{D}}(\tau))))), \\ fst(\overleftarrow{\mathcal{D}}(\tau)) \times snd(\overleftarrow{\mathcal{D}}(\sigma))) \quad (6)$$

In a sense, the tangent value at function types keeps track of the adjoints currently calculated so they 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. While 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.

In eq. (6), we attempt to keep track of the adjoints using product types, which are easily combined using $\bar{+}$. This, however, does not ensure that the linear properties of both $(\bar{0}, \bar{+})$ are preserved. Indeed, for any program consisting of combinators such that it has type `comb` τ `R`, if $[\![\bar{0}_\tau]\!]$ is given as input in the denotational sense, the resulting program should correspond to 0. Similarly, $[\![\bar{+}_\tau]\!]$ on inputs a and b should correspond to some notion of addition between a and b.

Vákár poses the usage of an auxiliary target language with a limited variant of linear types as the result of evaluating the reverse-mode macro[50]. The linear types used include both linear function types, $—\circ$, and tensor products, $!(-) \otimes (-)$. This target language would be almost identical to the source combinator language described in section 6.1. The added types and terms are shown in fig. 19.

Because we extended the target language with both linear functions and tensor

$$\overleftarrow{\mathcal{D}}(\tau \Rightarrow \sigma) = (fst(\overleftarrow{\mathcal{D}}(\tau)) \Rightarrow (fst(\overleftarrow{\mathcal{D}}(\sigma)) \times (snd(\overleftarrow{\mathcal{D}}(\sigma)) \multimap snd(\overleftarrow{\mathcal{D}}(\tau)))),\\ !fst(\overleftarrow{\mathcal{D}}(\tau)) \otimes snd(\overleftarrow{\mathcal{D}}(\sigma)))$$

$$\begin{aligned}
\overleftarrow{\mathcal{D}}(\texttt{id}) &= (\texttt{id}, \texttt{exr}) \\
\overleftarrow{\mathcal{D}}(t_1 ;; t_2) &= (d_1 ;; d_2, (\texttt{cross dupl id} ;; assoc1 ;; \\
&\quad (\texttt{cross id (cross } d_1 \texttt{ id)}) ;; (\texttt{cross id } (d_2' ;; d_1')))) \\
\overleftarrow{\mathcal{D}}(\texttt{cross } t_1\ t_2) &= (\texttt{cross } d_1\ d_2, assoc1 ;; \texttt{cross id } (assoc2 ;; \\
&\quad (\texttt{cross } sym \texttt{ id}) ;; assoc1) ;; assoc2 ;; \texttt{cross } d_1'\ d_2') \\
\overleftarrow{\mathcal{D}}(\texttt{neg}) &= (\texttt{neg}, \texttt{exr} ;; \bar{0}) \\
\overleftarrow{\mathcal{D}}(\texttt{exl}) &= (\texttt{exl}, \texttt{dupl} ;; \texttt{cross exr } (\texttt{neg} ;; \bar{0})) \\
\overleftarrow{\mathcal{D}}(\texttt{exr}) &= (\texttt{exr}, \texttt{dupl} ;; \texttt{cross } (\texttt{neg} ;; \bar{0})\ \texttt{exr}) \\
\overleftarrow{\mathcal{D}}(\texttt{dupl}) &= (\texttt{dupl}, \texttt{exr} ;; \bar{+}) \\
\overleftarrow{\mathcal{D}}(\texttt{ev}) &= (\texttt{ev} ;; \texttt{exl}, \langle \langle \texttt{exl} ;; \texttt{exr}, \texttt{exr} \rangle ;; \texttt{msingleton}, \\
&\quad \texttt{cross } (\texttt{ev} ;; \texttt{exr} ;; \texttt{curry ev_l})\ \texttt{id} ;; \texttt{ev} \rangle) \\
\overleftarrow{\mathcal{D}}(\texttt{curry } t) &= (\texttt{curry } (\langle d, \texttt{curry } d' ;; \texttt{exr} \rangle), \\
&\quad \texttt{cross id } (\langle \texttt{id}, \texttt{neg} ;; \texttt{curry exr} \rangle ;; \texttt{mfold}) ;; assoc2 ;; d' ;; \texttt{exl})
\end{aligned}$$

Figure 20: Macro on the combinator language

products, we also have to extend both $(\bar{0}, \bar{+})$ with these types.

$$\bar{0}_\tau = \begin{cases} \ldots & \\ \texttt{curry\_l } (\texttt{exl} ;; \bar{0}) & : \tau = \sigma \multimap \rho \\ \texttt{mempty} & : \tau = !\sigma \otimes \rho \end{cases}$$

$$\bar{+}_\tau = \begin{cases} \ldots & \\ \texttt{curry\_l} & \\ (\langle \langle \texttt{exl} ;; \texttt{exl}, \texttt{exr} \rangle ;; \texttt{ev\_l}, & : \tau = \sigma \multimap \rho \\ \quad \langle \texttt{exl} ;; \texttt{exr}, \texttt{exr} \rangle ;; \texttt{ev\_l} \rangle ;; \bar{+}_\rho) & \\ \texttt{mplus} & : \tau = !\sigma \otimes \rho \end{cases}$$

Using these additional constructs, we can finally define the macro as shown in fig. 20.

6.3 Attempt at a Formalized Proof

Temporarily ignoring the issue of denotational semantics of our target language, we can formulate the proof as a logical relations argument, as we have done in previous proofs. Unlike the previous proofs, however, we are this time working with a combinator language. Using a separate combinator language has the convenient consequence that we do not have to worry about typing contexts or substitutions in the proof of the fundamental lemma of our logical relation.

As previously, we can formulate our logical relation as a type indexed relation between

the curves related to our macro.

$$S_\tau : (\mathcal{R} \to [\![\tau]\!]) \to (\mathcal{R} \to [\![fst(\overleftarrow{\mathcal{D}}(\tau))]\!]) \to (\mathcal{R} \star [\![snd(\overleftarrow{\mathcal{D}}(\tau))]\!] \to \mathcal{R}) \to \text{Prop}$$

Note that we index the logical relation by the types of our source language. The most exceptional case, which will also give a clue as to where the difficulty with this proof lies, is the case for functions. Namely, we additionally have to establish that the second argument of the third curve of our relation is linear in its behavior with respect to $(\bar{0}, \bar{+})$. This requirement comes from the idea that the third curve, intuitively, is tracking the transposed derivatives of the respective function. As a consequence, its arguments should also obey the linearity of differentiation. We formulate this requirement as definition 5.

Definition 5. *Any transposed derivative calculating function is linear in its second argument such that it follows:*

$$\begin{aligned} linear\_second_\tau(h) =& (\forall r.h(r, [\![\bar{0}_\tau]\!]tt) = 0) \wedge \\ & (\forall r, a, b.h(r, [\![\bar{+}_\tau]\!](a, b) = h(r, a) + h(r, b)). \end{aligned}$$

Definition 6. *(Logical relation) Denotation functions f and their corresponding primal and tangent variants g and h are inductively defined on the structure of our types such that they follow the relation*

$$S_\tau(f,g,h) = \begin{cases} f = g \wedge h = \lambda x.0 & : \tau = \texttt{U} \\ smooth\ f \wedge f = g \wedge & : \tau = \texttt{R} \\ \quad h = \lambda x.(\frac{\partial f}{\partial x}(fst(x))) * snd(x) & \\ (\forall i.smooth\ (\lambda x.(f(x))\ !\ i)) \wedge f = g \wedge & : \tau = \texttt{R\^{}n} \\ \quad h = \lambda x.(\frac{\partial f}{\partial x}(fst(x))) \cdot snd(x) & \\ \exists f_1, f_2, f_3, g_1, g_2, g_3, & : \tau = \sigma \times \rho \\ \quad S_\sigma(f_1, f_2, f_3), S_\sigma(g_1, g_2, g_3). & \\ \quad f = \lambda x.(f_1(x), g_1(x)) \wedge & \\ \quad g = \lambda x.(f_2(x), g_2(x)) \wedge & \\ \quad h = \lambda x.(f_3(fst(x), fst(snd(x))) + & \\ \qquad g_3(fst(x), snd(snd(x)))) & \\ \forall f_1, f_2, f_3. & : \tau = \sigma \Rightarrow \rho \\ \quad linear\_second(h) & \\ \quad S_\sigma(f_1, f_2, f_3) \Rightarrow & \\ \quad S_\rho(\lambda x.f(x)(f_1(x)), \lambda x.fst(g(x)(f_2(x))), & \\ \qquad \lambda x.h(fstx, f_2(fst(x), snd(x))) + & \\ \qquad f_3(fst(x), snd(g(fst(x))(f_2(fst(x))))(snd(x)))) & \end{cases}$$

A useful property is that our notion of linearity in the second argument of the curve tracking the transposed derivatives, linear_second, follows from our logical relation.

Lemma 3. *For any type τ and argument functions $f : \mathcal{R} \to [\![\tau]\!]$, $g : \mathcal{R} \to [\![fst(\overleftarrow{\mathcal{D}}(\tau))]\!]$ and $h : \mathcal{R} \star [\![snd(\overleftarrow{\mathcal{D}}(\tau))]\!] \to \mathcal{R}$ such that they follow $S_\tau(f, g, h)$, then linear_second(h).*

Proof. This lemma is proven by induction on the type τ. The cases for R and R^n are proven by the distributive law, while the rest follows regularly from the induction hypotheses. □

We can plot down the skeleton of what the proof would likely require. We can state the correctness of the reverse-mode algorithm as the following proposition. Note the usage of the sum function to combine all partial derivatives linearly. We additionally use a macro-specific initialization function $\overleftarrow{\mathcal{D}}_n$ for our arguments akin to definition 3 used in section 3. The initialization function is only altered slightly to return a tuple of the original primal value and the tangent value defined as in definition 3.

$$\overleftarrow{\mathcal{D}}_n(f) = \begin{cases} (f, f) & : n = 0 \\ (\lambda x.(fst(f(x)), fst(\overleftarrow{\mathcal{D}}_{n'}(snd \circ f))(x)), \\ \quad \lambda x.\frac{\partial(fst \circ f)}{x}(x), snd(\overleftarrow{\mathcal{D}}_{n'}(snd \circ f))(x)) & : n = n' + 1 \end{cases}$$

We can now state correctness of the combinator-based macro as the assertion that the sum of the derivative values of the macro is equal to the gradient of the original function.

Proposition 2 (Correctness of reverse-mode AD)**.** *For any well-typed term* $x_1 : \mathsf{R}, \ldots, x_n : \mathsf{R} \vdash t : \mathsf{R}$ *and argument function* $f : \mathcal{R} \to [\![translate\_context(\mathsf{R}^n)]\!]$ *such that it follows differentiable*$_n(f)$*, then* $(\lambda x.sum([\![snd(\overleftarrow{\mathcal{D}}(stlc\_ccc(t)))]\!](fst(\overleftarrow{\mathcal{D}}_n(f))(fst(x)), 1))) = (\lambda x.{}^{\partial([\![stlc\_ccc(t)]\!] \circ f)}/_{\partial x}(fst(x)) * snd(x))$

As was the case for the proofs in sections 3 and 5, this statement follows from the fundamental lemma of the logical relation. We can state this lemma as:

Proposition 3 (Fundamental lemma)**.** *For any well-typed combinator* $c : \mathsf{comb}\ \tau\ \sigma$*, and instantiation functions* $f : \mathcal{R} \to [\![\tau]\!]$*,* $g : \mathcal{R} \to [\![fst(\overleftarrow{\mathcal{D}}(\tau))]\!]$ *and* $h : \mathcal{R} \star [\![snd(\overleftarrow{\mathcal{D}}(\tau))]\!] \to \mathcal{R}$ *such that they follow* $S_\tau(f, g, h)$*, we have that* $S_\sigma([\![c]\!] \circ f, [\![fst(\overleftarrow{\mathcal{D}}(c))]\!] \circ g, \lambda x.h(fst(x), ([\![snd(\overleftarrow{\mathcal{D}}(c))]\!](g(fst(x), snd(x))))))$.

The main issue we encountered was during the proof of proposition 3. As a first attempt, lists of products and regular functions were used as the denotations of, respectively, $!\_ \otimes \_$ and $\multimap$. As expected, however, both of these choices were insufficient to complete the proof. To be specific, we encountered issues with the case of `curry`, mainly in the proof of linear_second, whose goals are:

$$\forall r.h(r, [\![snd(\overleftarrow{\mathcal{D}}(\mathsf{curry}\ c))]\!](gr, [])) = 0 \tag{7}$$

$$\begin{aligned} \forall r, a, b.h(r, [\![snd(\overleftarrow{\mathcal{D}}(\mathsf{curry}\ c))]\!](g(r), a +\!\!+ b)) = \\ h(r, [\![snd(\overleftarrow{\mathcal{D}}(\mathsf{curry}\ c))]\!](gr, a)) + h(r, [\![snd(\overleftarrow{\mathcal{D}}(\mathsf{curry}\ c))]\!](gr, b)) \end{aligned} \tag{8}$$

Simplifying these statements we can see that we essentially have to prove that $[\![snd(\overleftarrow{\mathcal{D}}(c))]\!]$ is linear with respect to the denotations of $(\bar{0}, \bar{+})$.

$$\text{linear}([\![fst(\overleftarrow{\mathcal{D}}(c))]\!]) \wedge \text{linear}([\![snd(\overleftarrow{\mathcal{D}}(c))]\!]) \tag{9}$$

The (linear : ($[\![\tau]\!] \to [\![\sigma]\!]$) → Prop) proposition encodes our notion of linearity in the context of the types in our combinator language.

```
Definition linear τ σ (f : [[τ]] → [[σ]]) : Prop
  := f ([[0̄]] tt) = [[0̄]] tt /\
    forall a b, f ([[+̄]] (a, b)) = [[+̄]] (f a, f b).
```

Attempting to prove eq. (9) by induction on the combinator c lays bare the issues underlying the denotations we chose. As the denotation of $\multimap$, we picked regular functions. This choice proved much too unrestricted, as it does not ensure the linear behavior of the denotations of our monoidal combinators. The lists we used as the denotation of the tensor types were much more problematic. A better denotation would have been map types with some additional restrictions, which we can be state as:

$$\{x : \bar{[\![0]\!]}\} \equiv \emptyset \tag{10}$$
$$\{x : a\} \cap \{x : b\} \equiv \{x : a \ \bar{[\![+]\!]} \ b\} \tag{11}$$
$$m_1 \cup \bar{[\![0]\!]} \equiv m_1 \tag{12}$$
$$m_1 \cup m_2 \equiv m_2 \cup m_1 \tag{13}$$
$$m_1 \cup (m_2 \cup m_3) \equiv (m_1 \cup m_2) \cup m_3 \tag{14}$$

Of these, the list type denotations we used in the incomplete proof only satisfies eq. (14). Transitioning from lists to maps or bags would improve on this by also satisfying eq. (13). Equations (10) and (12), however, pose a unique difficulty in *Coq*. Note that $\bar{[\![0]\!]}$ is defined on the types of our target language, which includes these same tensor types. So when defining these maps as the denotations of tensor types, we require a notion of equivalence that is itself also dependent on those same denotations. This recursive definitional requirement is problematic in the total language of *Coq*. Another obstacle is the fact that maps require decidable equality on the types used as keys. Remember that equality on both function or real number types is not generally decidable, and as such, cannot be used as keys in our maps.

One presumed solution to the first problem is to define proper domains using canonical structures and setoids as the denotational semantics of our language. One would then be able to attach, at each type, an element corresponding to $\bar{[\![0]\!]}$. We could then take eqs. (10) and (12) into account in the equivalence relation. Embedding such a zero-element into the denotations directly, seemingly solve the first issue. How to concretely solve or avoid the second issue in a proof assistant is still unknown.

7 Discussion

In the original proof of correctness by Huot, Staton and Vákár[48], they encoded the smoothness requirements in the denotational semantics. This method meshed well with the existing mathematical literature about diffeological spaces. In contrast, we were able to encode this requirement directly within the logical relation used in the proof, while keeping the denotational semantics we needed as simple as possible. In retrospect, this reduced a significant amount of the proof load needed as we did not have to redefine many of the properties of functions on smooth functions. Unfortunately, however, the set-theoretic denotational semantics limits the proof possibilities to total programming languages.

The added value of creating the formalized correctness proof of the forward-mode automatic differentiation macro is immediately visible in the fact that we were also able to simplify the fundamental lemma in the original pen-and-paper proof. The exclusion of as much of the syntactic constructions as possible, such as substitutions, was critical in ensuring the proof went smoothly. If formulated incorrectly, the additional complexity involved with typing contexts and variable management muddies the corresponding cases

of app and abs in the proof of the fundamental lemma. An unfortunate amount of time was spent in this hybrid approach of both syntactic and denotational structures, before focussing on the full denotational approach.

Shaikhha et al.[41] showed that the performance of forward-mode AD can approach that of reverse-mode AD given the right optimizations. It is useful to realize that the various macros discussed in this thesis all ultimately calculate the same values, but do so using different expressions. These expressions differ merely in the number of primitive operations, and as such, are equal modulo the distributive law. It may then be reasonable to suggest that an appropriately intelligent compiler would be able to, in a sufficiently generic sense, optimize a forward-mode based gradient algorithm to the performance expected of a reverse-mode algorithm.

Vákár[50] gave a novel extension to the categorical approach to automatic differentiation by Elliott[36]. Unfortunately, however, we were unable to finish a formalized proof of correctness in time. One of the principal issues was the difficulty in applying the simple set-theoretic denotational semantics we have used so vehemently throughout our other proofs. Presumably, departing from this to a one based on quotient types using, for example, setoids, may be more fruitful, but more research is required.

8 Future Work

There are several extensions possible on both the proof discussed in this thesis, and directions of new proofs in the same vein.

Recursion and Iteration. Due to the limited nature of our chosen simply-typed lambda calculus, our contributions with respect to practical programming is still very limited. Both iteration and recursion are ubiquitous in programming. Their omission severely limits the usefulness of any functional programming language. As the set-theoretic denotational semantics we used in our proof is only able to support total languages, substituting this for the more domain-theoretical denotational semantics with ω-cpos would be an improvement in this direction. On the more mathematical side, Vákár[49] extends the original proof by Huot, Staton, and Vákár[48] with iteration and recursion, which should give an indication to which steps are needed in the corresponding formal proof. Also relevant is the proof given by Abadi and Plotkin[38], as their language also includes partial constructs.

Polymorphism and Impredicativity. The intrinsic representation we used in this thesis was derived from the presentation by Benton et al.[29]. One language aspect they also discussed, but we did not use, was polymorphism. Extending the proof to include polymorphic types or any other higher-order concepts such as rank-n types would help determine the feasibility of using this representation in eventual extensions to impredicativity. Due to the interactions between substitution and renaming, this may prove problematic with the proof load. In this case, alternative representations like the one described by Chlipala[22] may be a solution.

Reverse-Mode AD. Reverse-mode AD was briefly discussed in section 6, but we were unable to finish the proof. A possible research direction would be to give a completed formalized proof of correctness of the algorithm described by Vákár[50]. This algorithm has the added benefit of being a define-then-run algorithm, which further increases the value of such a formal proof. Other reverse-mode algorithms, such as the one described by Wang et al.[45], may also be possible valuable proof targets.

9 Conclusion

With this thesis, we were able to successfully give a formalized correctness proof of a ubiquitous forward-mode automatic differentiation algorithm. We were also able to expand this proof to include array types, oft used in practice, and GPU-based environments. Additionally, we proved that the various relevant compile-time optimizations for efficient code-generation are correct with respect to our denotational semantics. While we were unfortunately unable to finish the formalized proof of correctness of the reverse-mode combinator-based macro, we have identified multiple issues and requirements for a proper future formalized proof. The final proofs are simple, both in terms of definitions and proof strategy, and can easily be modified at any number of points to extend the language used.

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 p. 32).

[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. 8).

[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. 8).

[5] A. Gill, J. Launchbury, and S. L. Peyton Jones, "A short cut to deforestation," in *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, ser. FPCA '93, Copenhagen, Denmark: Association for Computing Machinery, 1993, pp. 223–232, ISBN: 089791595X. DOI: 10.1145/165180.165214. [Online]. Available: https://doi.org/10.1145/165180.165214 (cit. on p. 23).

[6] 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).

[7] 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. 12).

[8] 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. 12).

[9] 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. 12).

[10] J. Karczmarczuk, "Functional differentiation of computer programs," in *Higher-Order and Symbolic Computation*, 1998, pp. 195–203 (cit. on pp. 13, 15).

[11] 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).

[12] J. Karczmarczuk, *Lazy time reversal, and automatic differentiation*, 2000. [Online]. Available: https://karczmarczuk.users.greyc.fr/arpap/revpearl.pdf (cit. on p. 28).

[13] 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. 12).

[14] J. M. Conor McBride, "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] 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. 12).

[16] 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. 9, 12).

[17] 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).

[18] 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. 11).

[19] M. Sozeau, "Subset coercions in coq," in *Proceedings of the 2006 International Conference on Types for Proofs and Programs*, ser. TYPES'06, Nottingham, UK: Springer-Verlag, 2006, pp. 237–252, ISBN: 3540744630 (cit. on p. 10).

[20] 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).

[21] 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. 12).

[22] A. Chlipala, "Parametric higher-order abstract syntax for mechanized semantics," *SIGPLAN Not.*, vol. 43, no. 9, pp. 143–156, Sep. 2008, ISSN: 0362-1340. DOI: 10.1145/1411203.1411226. [Online]. Available: https://doi.org/10.1145/1411203.1411226 (cit. on p. 44).

[23] S. Mark and A. PearlmutterBarak, "Nesting forward-mode ad in a functional framework," 2008 (cit. on pp. 13, 15).

[24] 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, 28).

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

[26] 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. 12).

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

[28] M. Sozeau, "Equations: A dependent pattern-matching compiler," in *Interactive Theorem Proving*, Springer Berlin Heidelberg, 2010, pp. 419–434. DOI: 10.1007/978-3-642-14052-5_29. [Online]. Available: https://doi.org/10.1007/978-3-642-14052-5_29 (cit. on pp. 8, 10).

[29] 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, 44).

[30] 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).

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

[32] 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. 12).

[33] 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).

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

[35] 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. 32).

[36] C. Elliott, "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. 13, 32, 44).

[37] 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).

[38] 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 pp. 12, 13, 44).

[39] 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).

[40] C. C. Margossian, "A review of automatic differentiation and its efficient implementation," *ArXiv*, vol. abs/1811.05031, 2019 (cit. on p. 12).

[41] 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, 12, 15, 23, 25, 44).

[42] L. Skorstengaard, *An introduction to logical relations*, 2019. arXiv: 1907.11133 [cs.PL] (cit. on p. 11).

[43] 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).

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

[45] 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 pp. 13, 44).

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

[47] 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. 11, 12, 17).

[48] 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, 11, 12, 14, 17, 43, 44).

[49] M. Vákár, *Denotational correctness of foward-mode automatic differentiation for iteration and recursion*, 2020. arXiv: 2007.05282 [cs.PL] (cit. on p. 44).

[50] M. Vákár, *Reverse ad at higher types: Pure, principled and denotationally correct*, 2020. arXiv: 2007.05283 [cs.PL] (cit. on pp. 4, 13, 32, 39, 44).