Utrecht University

Master Thesis Proposal

Formalized Proof of Automatic Differentiation in Coq

Student:
Curtis Chin Jen Sem

Supervisors:
Mathijs Vákár
Wouter Swierstra

Department of Information and Computing Science

Last updated: April 9, 2020

Contents

1 Introduction

AI and machine learning research has sparked a lot of new interest in recent times due to its many applications and ability to solve complex problems very quickly.

This is regularly done using a technique called automatic differentiation. There has been a recent surge of interest in formulating languages for defining automatic differentiable functions. This could have many benefits such as both applying many of the established high and low level optimizations known in programming languages research, ease defining functions for use in a gradient descent optimization through higher order functions and correctness through the use of a possible type system.

We aim to formalize an extendable proof of an implementation of automatic differentiation on a simply typed lambda calculus in the **Coq** proof assistant, opening up further possibilities for formally proving the correctness of more complex language features in the future. Our formalization is based on a recent proof by Stanton Huot, and Vákár [19]. They proved, using a denotational model of diffeological spaces, that their forward mode emulating macro is correct when applied to a simply typed lambda calculus with products, co-products and inductive types.

With this thesis we will aim for the following goals:

- Contribute a formalized proof of forward-mode automatic differentiation in **Coq**.
- Formulate the proofs such that it facilitates further extensions.
- (TODO: UNLIKELY) Extend the original proof with iteration and possibly recursion.
- (TODO: MAYBE) Extend the original proof with inductive types.
- (TODO: MAYBE) Extend the original proof with polymorphism.
- (TODO: MAYBE) Adapt the proof to a small imperative language.
- (TODO: LIKELY) Prove that well-known optimizations such as the partial evaluation, are correct with respect to automatic differentiation.
- (TODO: LIKELY) Prove the correctness of the continuation-based automatic differentiation algorithm.

As a notational convention, we will use specialized notation in the definitions themselves. Coq requires that pretty printed notation be defined separately from the definitions they reference.

2 Background

2.1 Automatic differentiation

One of the principal techniques used in machine learning is back propagation, which calculates the gradient of a function. The idea being to use the gradient in the gradient descent algorithm[11]. Automatic differentiation has a long and rich history, where its purpose is to calculate the derivative of a function, or in other words, calculate the derivative of function described by an arbitrary program. So the semantics which one would normally expect in programming language is extended with relevant concepts such as derivative values and the chain rule.

Automatic or algorithmic differentiation is beneficial over other methods of automatically calculating the derivatives of functions such as numerical differentiation or symbolic differentiation due to its balance between speed and computational complexity. There are two main modes of automatic differentiation. These are namely forward and reverse mode AD. For the purposes of this paper, we will only discuss forward mode AD.

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

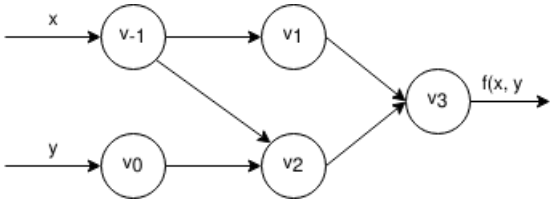

Figure 1: Computational graph of $f(x, y) = x^2 + (x - y)$

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

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

2.2 Denotational semantics

The notion of denotational semantics tries to find underlying mathematical objects able to explain the properties of programming languages. The most well-known example is the solution 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[17]. In this model, programs are interpreted as partial functions and computation by taking the fixpoint of such functions. Non-termination, on the other hand, is formalized as a value `bottom` that is lower in the ordering relation than any other element.

In our specific case, we try to find a satisfactory model we can use to show that our implementation of forward mode automatic differentiation is correct when applied to a simply typed lambda calculus. In the original pen and paper proof of automatic differentiation this thesis is based on, the mathematical models used were diffeological spaces, which are generalization of smooth manifolds. For the purpose of this thesis, however, this was deemed excessive and much too difficult and time consuming to implement in a mathematically sound manner in **Coq**. As such, we chose

to make use of **Coq**'s existing types as denotations and base the relation on the denotations instead of the syntactic structures. Due to the relative simplicity of the language, we did not yet require domain theoretical concepts. If recursion or iteration were to be added to the language, it is expected that these would be needed.

2.3 Coq

Coq is a proof assistant created by Thierry Coquand as an implementation of his calculus of constructions type theory[2]. In the 30 years since it has been released, research has contributed to extending the proof assistant with additional features such as inductive and co-inductive data types[3], dependent pattern matching[7] and advanced modular constructions for organizing colossal mathematical proofs[6][10].

The core of this type theory is based on constructive logic and so many of the laws known in classical logic are not present. Examples include the law of the excluded middle, $\forall A, A \vee \neg A$, or the law of functional extensionality, $(\forall x, f(x) = g(x)) \rightarrow f = g$. In most cases they can, however, be safely added to **Coq** without making its logic inconsistent. Due to its usefulness in proving propositions over functions, we will make use of the functional extensionality axiom in **Coq**.

2.3.1 Language representation

When defining a simply typed lambda calculus, there are two main representations[16]. The arguably simpler variant, known as an extrinsic representation, is traditionally the one introduced to new students learning **Coq**. In the extensional representation, the terms themselves are untyped and typing judgments are defined separately. A basic example of working with this is given in [12]. This required many additional lemmas and machinery to be proved, however. The other approach, also called an intrinsic representations 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.

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[4]. An example of an approach is the nominal representation where every variable is named. While this does follow the standard format used in regular mathematics, problems such as alpha-conversion and capture-avoidance arises.

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

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

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

The approach used in the rest of this thesis is an extension of the de-bruijn representation which numbers variables relative to the binding lambda term. In this representation the variables are referred to as de-bruijn indices. A significant benefit of this representation is that the problems of capture avoidance and alpha equivalence are avoided. As an alteration, 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. The specific formulation used in this thesis was fleshed out by Nick Benton, et. al. in [8], and was also used as one of the examples in the second POPLmark challenge which deals with logical relations[13]. While this does subvert the problems present in the nominal representation, it unfortunately does have some problems of its own. Variable substitutions have to be defined using two separate renaming and substitution operations. Renaming is formulated as extending the typing context of variables, while substitution actually swaps the variables for terms. Due to using indices from the context as variables, some lifting boilerplate is also needed to manipulate contexts.

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

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

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

2.3.2 Dependent 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. If proof terms are present in the function definition, however, it is customary to write it using tactics because of the otherwise cumbersome and verbose code needed to pattern-match on the arguments. But this can be troublesome in the cases where the function signature is ambiguous, as it can be hard to recognize what the function then actually computes. One other possibility would be to write the function using relations between its input and output. This also has its limitations as relations can be tricky to define. In this case, the definitions are also opaque such that the standard `simpl` tactic which invokes **Coq**'s reduction mechanism is not able to reduce the term. This often requires the user to write to write many more proofs to be able to work with the definitions.

As an example, we will work through defining a length indexed list and a corresponding head function, which is well known to be partial. Using the **Coq** keyword return, it is possible to let the return type of a match expression depend on the result of one of the type arguments. This makes it possible to specify what the return type of the empty list should be. In snippet 3, we use the unit type which contains just one inhabitant, .

In [20] and [5] Sozeau introduces an extension to **Coq** via a new keyword `Program` which allows the use of case-analysis in more complex definitions. To be more specific, it allows definitions to be specified separately from its accompanying proofs, possibly filling them in automatically if possible. While this does improve on the previous situation, using the definitions in

```
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.
```

Code snippet 3: Definition of a length indexed list and hd using the return keyword, adapted from [9].

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 in [7] and [15] by introducing a method for user-friendlier dependently-typed programming in **Coq** as the `Equations` library. This introduces **Agda**-like dependent pattern matching with with-clauses. It does this by using a notion called coverings, where a covering is a set of equations such that the pattern matchings of the type signature are exhaustive. There are two main ways to integrate this in a dependently typed environment, externally where it is integrated as high-level constructs in the pattern matching core as **Agda** does it, or internally by using the existing type theory and finding witnesses of the covering to prove the definition correct, which is the approach used by Sozeau. Due to the intrinsic typeful representation this paper uses, much of this was invaluable when defining the substitution operators as the regular type checker in Coq often had difficulty when recognizing type equalities in certain cases.

2.4 Logical relations

Logical relations, otherwise known as Tait's method, is technique often employed when proving programming language properties[14].

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

Code snippet 4: Definition of hd using `Equations`

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

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

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. A logical relations proof essentially consists of 2 steps. The first usually states that well-typed terms are in the relation, while the second states that the property of interest follows from the relation. It should be evident that the proof itself is usually routine compared to defining the relation itself.

A well-known logical relations proof is the proof of strong normalization. An example of a logical relation using the intrinsic strongly-typed formulation is given in 5. Noteworthy is the case for function types, which indicates that application should maintain the strongly normalization relation. The proof given in the paper this thesis is based on, is a logical relations proof on the denotation semantics using diffeological spaces as its domains[19]. A similar, independent proof of correctness was given in [18] using an syntactic relation.

```
Inductive ty : Type :=
  | Real : ty
  | ⇒ : ty → ty → ty
  | × : ty → ty → ty
  | <+> : ty → ty → ty.
```

Code snippet 6: Definition of the types present in the language

3 Preliminary Results

3.1 Language definitions

We currently mimic the base types used in the paper[19] extended with sum types, shown in snippet 6. The paper initially makes use of standard types found in a simply typed lambda calculus with products and R as the only ground type. These are also the types used in [18]. In a later section Stanton, Huot and Vákár extended their language with sum and inductive types. We have currently only extended our language with sums. Note that we use the unconventional symbol <+> for sum types instead of the more common +, because Coq already uses the latter for their own sum types.

We have chosen the intrinsic strongly-typed formulation used in [8] as the general framework to work in. This includes the various substitution and lifting operations for working with typing contexts included. Typing contexts themselves consists of lists of types, while variables are typed indices into this list shown in 2.

We have simplified a few of the language constructs used in [19], shown in snippet 7. For working with product types they make use of n-products and pattern matching, while we have opted for projection tuples. They proceeded to extended their base language with arbitrarily sized variant types, which we have substituted for standard sums reminiscent of the `Either` type in Haskell along with specialized case expressions. Both of these changes were intended to simplify the language as much as possible while still retaining the same core functionality and types.

3.2 Preliminary proofs

We have completed a preliminary proof of `Theorem 1` of [19]. This consists of a formulation of semantic correctness of a forward-mode automatic

```
Definition Ctx {x} : Type := list x.

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

  (* Operations on reals *)
  | const : R → tm Γ Real
  | add : tm Γ Real → tm Γ Real → tm Γ Real

  (* 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

  (* Sums *)
  | case : forall τ σ ρ, tm Γ (τ + σ) →
    tm Γ (τ ⇒ ρ) →
    tm Γ (σ ⇒ ρ) →
    tm Γ ρ
  | inl : forall τ σ, tm Γ τ → tm Γ (τ + σ)
  | inr : forall τ σ, tm Γ σ → tm Γ (τ + σ).
```

Code snippet 7: Definition of the language constructs present in the language

differentiation algorithm using a macro. The proof is done using a logical relation on the denotational semantics using **Coq**'s types as the underlying domain. The definition of the logical relation along with the lemma stating its fundamental property is shown in snippet 8 and 9, while snippet 10 states the actual correctness theorem.

4 Timetable and Planning

4.1 Extensions

We will be looking to extend the current prototype proof to include more complex language structures. The first of which we will be looking at hardcoded inductive types in the form of lists. This is not expected to require much time or code. More complex will be how we would be able to include iteration or recursion.

4.2 Deadlines

The hard deadlines for the first and second phases of the thesis are respectively May 1^{st} and September 18^{th}. The ambition is to follow the following framework of deadlines. Note that until the proofs deadline, the proofs and paper will largely be written in parallel with each other.

Deadline	Date
Proposal deadline	1/5/2020
Finish proofs	15/7/2020
Finish first draft	15/8/2020
Thesis deadline	18/9/2020

References

[1] D. Scott, "Outline of a mathematical theory of computation", *Kiberneticheskij Sbornik. Novaya Seriya*, vol. 14, Jan. 1977 (cit. on p. 5).

[2] T. Coquand and G. Huet, "The calculus of constructions", *Information and Computation*, vol. 76, no. 2-3, pp. 95–120, Feb. 1988. DOI: 10.1016/0890-5401(88)90005-3. [Online]. Available: https://doi.org/10.1016/0890-5401(88)90005-3 (cit. on p. 6).

```
Equations S τ :
  (R → ⟦ τ ⟧) → (R → ⟦ D τ ⟧) → Prop :=
S Real f g :=
  (∀ (x : R), ex_derive f x) ∧
  (fun r => g r) =
    (fun r => (f r, Derive f r));
S (σ × ρ) f g :=
  ∃ f1 f2 g1 g2,
  ∃ (s1 : S σ f1 f2) (s2 : S ρ g1 g2),
    (f = fun r => (f1 r, g1 r)) ∧
    (g = fun r => (f2 r, g2 r));
S (σ ⇒ ρ) f g :=
  ∀ (g1 : R → ⟦ σ ⟧),
  ∀ (g2 : R → ⟦ D σ ⟧),
    S σ g1 g2 →
    S ρ (fun x => f x (g1 x))
      (fun x => g x (g2 x));
S (σ <+> ρ) f g :=
  (∃ g1 g2,
    S σ g1 g2 ∧
      f = inl ∘ g1 ∧
      g = inl ∘ g2) ∨
  (∃ g1 g2,
    S ρ g1 g2 ∧
      f = inr ∘ g1 ∧
      g = inr ∘ g2).
```

Code snippet 8: Definition of the logical relation

```
Inductive instantiation : forall Γ,
    (R → ⟦ Γ ⟧) → (R → ⟦ D Γ ⟧) → Prop :=
| inst_empty : instantiation [] (const tt) (const tt)
| inst_cons :
  ∀ Γ τ g1 g2,
  ∀ (sb: R → ⟦ Γ ⟧) (Dsb: R → ⟦ D Γ ⟧),
    instantiation Γ sb Dsb →
    S τ g1 g2 →
    instantiation (τ::Γ)
      (fun r => (g1 r, sb r)) (fun r => (g2 r, Dsb r)).

Lemma fundamental :
    ∀ Γ τ (t : tm Γ τ),
    ∀ (sb : R → ⟦ Γ ⟧) (Dsb : R → ⟦ D Γ ⟧),
  instantiation Γ sb Dsb →
  S τ (⟦ t ⟧ ∘ sb)
    (⟦ Dtm t ⟧ ∘ Dsb).
```

Code snippet 9: Definition of the fundamental property of the logical relation i 8

```
Equations D n
  (f : R → ⟦ repeat Real n ⟧): R → ⟦ map Dt (repeat Real n) ⟧ :=
D 0 f r := f r;
D (S n) f r :=
  (((fst ∘ f) r, Derive (fst ∘ f) r), D n (snd ∘ f) r).

Inductive differentiable : ∀ n, (R → ⟦ repeat Real n ⟧) → Prop :=
| differentiable_0 : differentiable 0 (fun _ => tt)
| differentiable_Sn :
  ∀ n (f : R → ⟦ repeat Real n ⟧),
  ∀ (g : R → R),
    differentiable n f →
    (∀ x, ex_derive g x) →
    differentiable (S n) (fun x => (g x, f x)).

Theorem semantic_correct_R :
  ∀ n (t : tm (repeat Real n) Real),
  ∀ (f : R → ⟦ repeat Real n ⟧),
    differentiable n f →
    (⟦ Dtm t ⟧ ∘ D n f) =
      fun r => (⟦ t ⟧ (f r),
        Derive (fun (x : R) => ⟦ t ⟧ (f x)) r).
```

Code snippet 10: Definition of the correctness theorem

[3] T. Coquand and C. Paulin, "Inductively defined types", in *COLOG-88*, Springer Berlin Heidelberg, 1990, pp. 50–66. DOI: 10.1007/3-540-52335-9_47. [Online]. Available: https://doi.org/10.1007/3-540-52335-9_47 (cit. on p. 6).

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

[5] 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. 8).

[6] 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. 6).

[7] 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. 6, 9).

[8] 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. 7, 11).

[9] A. Chlipala, *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, 2013, ISBN: 0262026651 (cit. on p. 9).

[10] 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. 6).

[11] A. G. Baydin, B. A. Pearlmutter, A. Radul, and J. M. Siskind, "Automatic differentiation in machine learning: A survey", *J. Mach. Learn. Res.*, vol. 18, 153:1–153:43, 2015 (cit. on p. 4).

[12] B. C. Pierce, A. A. de Amorim, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, A. Tolmach, and B. Yorgey, *Programming Language Foundations*, ser. Software Foundations series, volume 2. Electronic textbook, May 2018 (cit. on p. 6).

[13] 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. 7).

[14] L. Skorstengaard, *An introduction to logical relations*, 2019. arXiv: 1907.11133 [cs.PL] (cit. on p. 9).

[15] 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. 9).

[16] P. Wadler and W. Kokke, *Programming Language Foundations in Agda*. 2019, Available at http://plfa.inf.ed.ac.uk/ (cit. on p. 6).

[17] A. Aaby, "Introduction to programming languages", *Syntax Imperative Programming Concurrent Programming Object-Oriented Programming Evaluation*, vol. 3, Apr. 2020 (cit. on p. 5).

[18] 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. 10, 11).

[19] 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. 3, 10, 11).

[20] M. Sozeau, "Subset coercions in coq", in *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 237–252. DOI: 10.1007/978-3-540-74464-1_16. [Online]. Available: https://doi.org/10.1007/978-3-540-74464-1_16 (cit. on p. 8).