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 19, 2020

Contents

1 Introduction

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

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

Programming language research has a rich history with many well-known both high- and low-level optimization techniques such as partial evaluation or deforestation. If instead of a framework, we were to have a programming language that is able to facilitate automatic differentiation, we would be able to apply many of these techniques. We would get benefits from other results from programming languages research such as: ease defining functions for use in a gradient descent optimization through higher order functions and correctness through the use of a possible type system.

In this thesis, we will 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 [24]. They proved, using a denotational model of diffeological spaces, that their forward mode emulating macro is correct when applied to a simply typed lambda calculus with products, co-products and inductive types.

With this thesis we will aim for the following goals:

- Formalize the proof of forward-mode automatic differentiation specified by Stanton, Huot, and Vákár [24] in **Coq**.
- Prove that well-known compile-time optimizations such as the partial evaluation, are correct with respect to automatic differentiation.

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

- Prove the correctness of the continuation-based automatic differentiation algorithm.
- Extend the proof with array types.

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[16]. Automatic differentiation has a long and rich history, where its main purpose is to automatically calculate the derivative of a function, or more precisely, calculate this derivative of a function described by a program. So in addition to the standard semantics present in most functional programming languages, we also now deal 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 $\frac{\delta f}{\delta x}$ at this point by setting $x' = 1$ and $y' = 0$. In this paper we will prove the correctness of a simple forward mode auto-

matic differentiation algorithm with respect to the semantics of a simply typed lambda calculus.

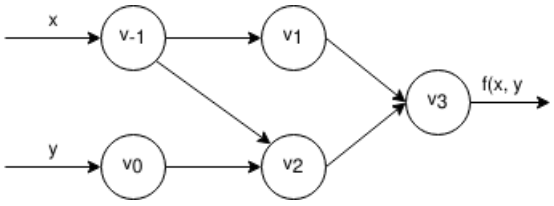

Figure 1: Computational graph of $f(x, y) = x^2 + (x - y)$

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 models able to underpin the concepts known in programming languages. The most well-known example is the solution given by Dana Scott and Christopher Strachey[1] for lambda calculi, also called domain theory. To be able to formalize non-termination and partiality, they thought to use concepts such as partial orderings and least fixed points[22]. In this model, programs are interpreted as partial functions and computations as taking the fixpoint of such functions. Non-termination, on the other hand, is formalized as a value `bottom` that is lower in the ordering relation than any other element.

In our specific case, we try to find a satisfactory model we can use to show that our implementation of forward mode automatic differentiation is correct when applied to a simply typed lambda calculus. In the original

pen and paper proof of automatic differentiation this thesis is based on, the mathematical models used were diffeological spaces, which are 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 currently expected that these would be needed.

2.3 Coq

Coq is a proof assistant based on the calculus of constructions type theory created by Thierry Coquand and Gérard Huet[2]. In the 30 years since it has been released, research has contributed to extending the proof assistant with additional features such as inductive and co-inductive data types[3], dependent pattern matching[12] and advanced modular constructions for organizing large mathematical proofs[11][15].

The core of this type theory is based on constructive logic and so many of the laws known in classical logic are not provable. 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 some 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 possibilities[21]. The arguably simpler variant, known as an extrinsic representation, is traditionally the one introduced to new students learning **Coq**. In the extrinsic representation, the terms themselves are untyped and typing judgments are defined separately as relations between the types and terms. A basic example of working with this is given in Software Foundations[17]. This, however, required many additional lemmas and machinery to be proved to be able to work with both substitutions and contexts as these

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

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

Figure 3: Type-inferrence rules for a simply typed lambda calculus using De-Bruijn indices

are defined separate from the terms. As an example, the preservation property which states that reduction does not change the type of a term, needs to be proven explicitly. The other approach, also called an intrinsic representation, makes use of just a single well-typed definition. Ill-typed terms are made impossible by the type checker. This representation, while beneficial in the proof load, however complicates much of the normal machinery involved in programming language theory. One example is how one would define operations such as substitutions or weakening.

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

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[4][6][8], the specific formulation used in this thesis using both substitutions and rename opera-

```
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.

tions was fleshed out in Coq by Nick Benton, et. al.[13], and was also used as one of the examples in the second POPLmark challenge which deals with logical relations[18]. While this does avoid the problems present in the nominal representation, it unfortunately does have some problems of its own. Variable substitutions have to be defined using two separate renaming and substitution operations. Renaming is formulated as extending the typing context of variables, while substitution actually swaps the variables for terms. Due to using indices from the context as variables, some lifting boilerplate is also needed to manipulate contexts.

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

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

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

2.3.2 Dependently-typed programming in Coq

In **Coq**, one can normally write function definitions using either case-analysis as is done in other functional languages, or using **Coq**'s tactics. If

```
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 Certified Programming with Dependent Types[14].

proof terms are present in the function definition, however, it is customary to write it using tactics because of the otherwise complicated and verbose code due to the previously poor support for dependent pattern matching in Coq. But if the functionality is not immediately apparent from the function signature, it can be hard to recognize what the function then computes. One other possibility would be to write the function as a relation between its input and output. This also has its limitations as you then lose computability as Coq treats these definitions opaquely. In this case the standard `simpl` tactic which invokes **Coq**'s reduction mechanism is not able to reduce instances of the term. This often requires the user to write 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, .

Mathieu Sozeau introduces an extension to **Coq** via a new keyword `Program` which allows the use of case-analysis in more complex definitions[25][10]. To be more specific, it allows definitions to be specified separately from its accompanying proofs, possibly filling them in automati-

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

Code snippet 4: Definition of hd using `Equations`

cally if possible. While this does improve on the previous situation, using the definitions in proofs can often be unwieldy due to the amount of boilerplate introduced. This makes debugging error messages even harder than it already is in a proof assistant. This approach was used by Benton in his formulation of strongly typed terms.

Sozeau further improves on this introducing a method for user-friendlier dependently-typed pattern matching in **Coq** in the form of the `Equations` library[12][20]. 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 type checking dependently typed terms in certain cases.

2.4 Logical relations

Logical relations is technique often employed when proving programming language properties of statically typed languages[19]. 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. Logical relations in essence are simply relations between terms defined by induction on the types. A logical relations proof consists of 2 steps. The first usually states that well-typed terms are in

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

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

the relation, while the second states that the property of interest follows from the relation. The second step is easier to prove as it usually follows from the definition of the relation. The first on the other hand, will often require proving a generalized variant, called the fundamental property of the logical relation. In most cases this requires that the relation is correct with respect to applying substitutions.

A well-known logical relations proof is the proof of strong normalization of well-typed terms, which states that all well-typed terms are either terminal values or can be reduced further. An example of a logical relation used in such a proof using the intrinsic strongly-typed formulation is given in snippet 5. Noteworthy is the case for function types, 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[24]. A similar, independent proof of correctness was given by Barthe, et. al.[23] using an syntactic relation.

3 Preliminary Results

3.1 Language definitions

We currently mimic the base types used in the paper [24] extended with sum types, shown in snippet 6. The paper itself initially makes use of the standard types found in a simply typed lambda calculus extended with products and R as the only ground type. These are also the types used by Barthe, et. al.[23] in their proof. In a later section Stanton, Huot and Vákár extended their language with sum and inductive types. 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.

```
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

We have chosen the intrinsic strongly-typed formulation used in [13] as the general framework to work in. This includes the various substitution and lifting operations for working with typing contexts. Typing contexts themselves consists of lists of types, while variables are typed indices into this list. We almost perfectly mimic the example shown in snippet 2.

We have simplified a few of the language constructs used in [24], 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 [24]. This consists of a formulation of semantic correctness of a forward-mode automatic 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.

```
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 the real numbers *)
  | 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

```
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 in snippet 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

4 Timetable and Planning

4.1 Extensions

We will be looking to extend the current prototype proof with the continuation-based AD macro mentioned in [24]. We will also try to extend the proof with a small imperative language we are able to translate into the simply typed lambda calculus. The goal is to work towards results usable in the context of program optimizations. One possibility is to go towards a SSA (static single assignment) representation which has many benefits as a well-known possible internal representation for use in compilers[9]. One other benefit is that this representation can also be transformed into a minimal functional language[5].

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", *Kibernetic heskij 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).

[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] 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. 7).

[5] M. M. Chakravarty, G. Keller, and P. Zadarnowski, "A functional perspective on ssa optimisation algorithms", *Electronic Notes in Theoretical Computer Science*, vol. 82, no. 2, pp. 347–361, 2004, COCV'03, Compiler Optimization Meets Compiler Verification, ISSN: 1571-0661. DOI: https://doi.org/10.1016/S1571-0661(05)82596-4. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1571066105825964 (cit. on p. 17).

[6] C. McBride and J. McKinna, "The view from the left", *J. Funct. Program.*, vol. 14, no. 1, pp. 69–111, Jan. 2004, ISSN: 0956-7968. DOI: 10.1017/S0956796803004829. [Online]. Available: https://doi.org/10.1017/S0956796803004829 (cit. on p. 7).

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

[8] R. Adams, "Formalized metatheory with terms represented by an indexed family of types", in *Types for Proofs and Programs*, J.-C. Filliâtre, C. Paulin-Mohring, and B. Werner, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–16, ISBN: 978-3-540-31429-5 (cit. on p. 7).

[9] S. Pop, "The ssa representation framework: Semantics, analyses and gcc implementation", Dec. 2006 (cit. on p. 17).

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

[11] M. Sozeau and N. Oury, "First-class type classes", in *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2008, pp. 278–293. DOI: 10.1007/978-3-540-71067-7_23. [Online]. Available: https://doi.org/10.1007/978-3-540-71067-7_23 (cit. on p. 6).

[12] M. Sozeau, "Equations: A dependent pattern-matching compiler", in *Interactive Theorem Proving*, Springer Berlin Heidelberg, 2010, pp. 419–434. DOI: 10.1007/978-3-642-14052-5_29. [Online]. Available: https://doi.org/10.1007/978-3-642-14052-5_29 (cit. on pp. 6, 10).

[13] N. Benton, C.-K. Hur, A. J. Kennedy, and C. McBride, "Strongly typed term representations in coq", *Journal of Automated Reasoning*, vol. 49, no. 2, pp. 141–159, Mar. 2011. DOI: 10.1007/s10817-011-9219-0. [Online]. Available: https://doi.org/10.1007/s10817-011-9219-0 (cit. on pp. 8, 12).

[14] 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).

[15] A. Mahboubi and E. Tassi, "Canonical structures for the working coq user", in *Interactive Theorem Proving*, Springer Berlin Heidelberg, 2013, pp. 19–34. DOI: 10.1007/978-3-642-39634-2_5. [Online]. Available: https://doi.org/10.1007/978-3-642-39634-2_5 (cit. on p. 6).

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

[17] 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).

[18] 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. 8).

[19] L. Skorstengaard, *An introduction to logical relations*, 2019. arXiv: 1907.11133 [cs.PL] (cit. on p. 10).

[20] 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).

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

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

[23] 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 p. 11).

[24] 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, 11, 12, 17).

[25] 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. 9).