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 24, 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. But programming inside current frameworks is very limited~~. It has been used in fields such as computer vision, natural language processing, and as opponents in various games such as chess and Go. In machine learning and more specifically neural network research, researchers set up functions referred to as layers between the input and output data and through an algorithm called back propagation, try to optimize the network such that it learns how to solve the problem implied by the data. Back propagation makes heavy use of automatic differentiation, but programming in an environment which allows for automatic differentiation can be limited.

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

Programming language research has a rich history with many well-known both high- and ~~cumbersome. One possible solution is to create~~ low-level optimization techniques such as partial evaluation and deforestation. If instead of a framework, we were to have a programming language that ~~facilitates defining 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~~is able to facilitate automatic differentiation, we would be able to apply many of these techniques. Through the use of higher-order functions and type systems, we would also get additional benefits such as code-reusability and correctness.

~~We~~ In this thesis, we will aim to formalize an extendable correctness proof of an implementation of automatic differentiation on a ~~simply typed~~ 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,~~ Huot, Staton, and Vákár [29]. They proved, using a denotational model of diffeological spaces, that their forward mode emulating macro is correct when applied to a ~~simply typed~~ simply-typed lambda calculus with products, co-products and inductive types.

With this thesis we will aim for the following goals:

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

- ~~Contribute a formalized proof of forward-mode automatic differentiation~~ Formalize the proofs of both the forward mode and continuation-based automatic differentiation algorithms specified by Huot, Staton, and Vákár [29] in **Coq**.
- ~~Formulate the proofs such that it facilitates further extensions.~~
- ~~Extend the proof to polymorphic types.~~
- ~~Adapt the proof to a small imperative language.~~
- Prove that well-known compile-time optimizations such as the partial evaluation, are correct with respect to the semantics of automatic differentiation.
- ~~Prove the correctness of the continuation-based automatic differentiation algorithm.~~ Extend the proof with the array types and compile-time optimization rules by Shaikhha, et. al.[23].

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

2 Background

2.1 Automatic differentiation

One of the principal techniques used in machine learning is back propagation, which calculates the gradient of a function. The ~~idea being to use the gradient~~ gradient itself is used in the gradient descent algorithm to optimize an objective function by determining the direction of steepest descent[19]. Automatic differentiation has a long and rich history, where its ~~main purpose is to~~ driving motivation is to be able to automatically calculate the derivative of a function ~~, or more precisely, calculate this derivative of a function described by a program~~ in a manner that is both correct and fast. Through techniques such as source-code transformations or operator overloading, one is able to implement an automatic differentiation algorithm which can transform any program which implements some function to one that calculates its derivative. So in addition to the standard semantics present in most ~~functional~~ programming languages, ~~we also now deal with relevant concepts~~

concepts relevant to differentiation such as derivative values and the chain rule are needed.

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~~ variants of automatic differentiation~~. These are namely forward~~ , namely forward mode and reverse mode ~~AD. For the purposes of this paper, we will only discuss forward mode AD.~~ automatic differentiation.

In forward mode automatic differentiation every term in the function trace is ~~accompanied with a dual numbers representation which calculate the derivative of the function~~annotated with the corresponding derivative of that term. These are also known as the respectively the primal and tangent traces. So every partial derivative of every ~~sub function~~ 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~~Figure 1. The corresponding traces are filled in ~~table~~Table 1 for the input values $x = 2, y = 1$. We can calculate the partial derivative $\frac{\delta f}{\delta x}$ at this point by setting $x' = 1$ and $y' = 0$. In this paper we will prove the correctness of a simple forward mode automatic differentiation algorithm with respect to the semantics of a ~~simply typed~~ simply-typed lambda calculus.

Reverse mode automatic differentiation takes a different approach. It tries to work backwards from the output by annotating each intermediate variable v_i with an adjoint $v_i' = \frac{\delta y_i}{\delta v_i}$. To do this, two passes are necessary. Like the forward mode variant the primal trace is needed to determine the intermediate variables and function dependencies. These are recorded in the first pass. The second pass actually calculates the derivatives by working backwards from the output using the adjoints, also called the adjoint trace.

The choice between automatic differentiation variant is heavily dependent on the function being differentiated. The number of applications of the forward mode algorithm is dependent on the number of input variables, as it has to be redone for each possible partial derivative of the function. On the other hand, reverse mode AD has to work backwards from each output variable. In machine learning research, reverse mode AD is generally preferred as the objective functions regularly contain a very small number of output variables. How one does reverse mode automatic differentiation on a functional language is still an active area of research. Huot, Staton and Vákár have

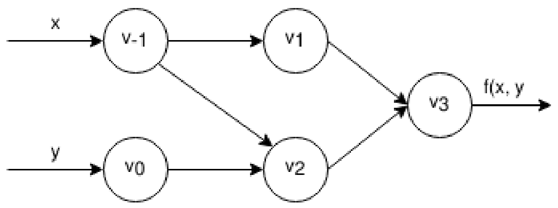

Figure 1: Computational graph of $f(x, y) = x^2 + (x - y)$

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

proposed a continuation-based algorithm which mimic much of the same ideas as reverse mode automatic differentiation[29].

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[27]. In this model, programs are interpreted as partial functions~~and~~ , and recursive computations as taking the fixpoint of such functions. Non-termination, on the other hand, is formalized as a value `bottom` that is lower in the ordering relation than any other element.

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~~ 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 a 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~~ we were able to avoid using diffeological spaces as recursion, iteration and concepts dealing with non-termination and partiality are left out of the scope of this thesis. **Coq** has very limited support for domain theoretical models. There are possible libraries which have resulted from experiments trying to encode domain theoretical models[13][18], but these are incompatible with recent versions of **Coq**. As ~~such, we chose to make use of~~ a part of its type system, **Coq** ~~'s existing types as denotations and base the relation on the denotations instead of the syntactic structures. Due to~~ contains a set-theoretical model available under the sort `Set`, which is satisfactory as 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~~denotational semantics for our language.

Because we use the real numbers as the ground type in our language, we also needed an encoding of the real numbers in Coq. The library for real numbers in **Coq** has improved in recent times from one based on a completely axiomatic definition to one involving Cauchy sequences[3] . For the purposes of this thesis, however, we needed differentiability as the denotational result of applying the macro operation. Instead of encoding this by hand, we opted for the more comprehensive library `Coquelicot`[20], which contains many general definitions for differentiating functions.

2.3 Coq

Coq is a proof assistant ~~created by Thierry Coquand as an implementation of his~~ based on the calculus of constructions type theory created by Thierry Coquand and Gérard Huet[2]. In the past 30 years since it has been released, research has contributed to extending the proof assistant with additional features such as inductive and co-inductive data types[3], dependent pattern matching[14] and advanced modular constructions for organizing ~~colossal~~ large mathematical proofs[12][17].

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~~ provable. An example includes the law of the excluded middle, $\forall A, A \vee \neg A$~~, or the law of functional extensionality,~~ ~~$(\forall x, f(x) = g(x)) \to f = g$~~~~. In most~~ . In some cases they can, however, be safely added to **Coq** without making its logic inconsistent. These are readily available in the standard library. Due to its usefulness in proving propositions over functions, we will make use of the functional extensionality axiom in **Coq**.

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

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

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

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

2.3.1 Language representation

When defining a ~~simply typed~~ simply-typed lambda calculus, there are two main possibilities[26]. The arguably simpler variant, known as an extrinsic representation, is traditionally the one introduced to new students learning **Coq**. In the ~~extensional~~ 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[21]. This, however, required many additional lemmas and machinery to be proved ~~.~~ to be able to work with both substitutions and contexts as these are defined separate from the terms. As an example, the preservation property which states that reduction does not change the type of a term, needs to be proven explicitly. The other approach, also called an intrinsic representation, makes use of just a single well-typed definition. Ill-typed terms are made impossible by the type checker. This representation, while beneficial in the proof load, however complicates much of the normal machinery involved in programming language theory. One example is how one would define operations such as substitutions or weakening.

But even when choosing an intrinsic representation, the problem of variable binding persists. Much meta-theoretical research has been done on possible approaches to this problem each with their own advantages and disadvantages. The POPLmark challenge gives a comprehensive overview of each of the possibilities in various proof assistants[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~~appears.

The approach used in the rest of this thesis is an extension of the ~~de-bruijn~~

```
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.

De-Bruijn representation which numbers variables relative to the binding lambda term. In this representation the variables are referred to as ~~de-bruijn~~ 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 ~~alteration~~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. ~~The~~ 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 separate substitutions and rename operations was fleshed out in Coq by Nick Benton, et. al.~~in~~ [15], and was also used as one of the examples in the second POPLmark challenge which deals with logical relations[22]. While this does ~~subvert~~ 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.

2.3.2 ~~Dependent~~ 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~~ 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 ~~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~~

```
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~~ simply-typed λ-calculus using a strongly typed intrinsic formulation.

~~signature is ambiguous, as it can be hard to recognize what the function then actually computes~~signature. This has been caused by the previously poor support in Coq for dependent pattern matching. Using the return keyword, one is able to vary the result type of a match expression. But due to requirement Coq used to have that case expressions be syntactically total, this could be very annoying to work with. One other possibility would be to write the function ~~using relations~~ as a relation between its input and output. This also has its limitations as ~~relations can be tricky to define~~you then lose computability as Coq treats these definitions opaquely. In this case ~~, the definitions are also opaque such that the~~ 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~~limited to lists of length at least one in Snippet 3. Using the **Coq** keyword return, it is possible to let the return type of a match ~~expression~~ expressions 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 oneinhabitant,~~ define an auxiliary function which, while total on the length of the list, has an incorrect return type. It namely returns the type unit if the input list had the length zero. We can then use this auxiliary function in the actual head function by specifying that the list has length at least one. It should be noted that more recent versions of Coq do not require that case expressions be syntactically total, so specifying that the input list has a length of at least zero is enough to eliminate the requirement for the zero-case.

~~In [30] and [11]~~ Mathieu Sozeau introduces an extension to **Coq** via a

```
  Inductive ilist : Type → nat → Type :=
    | nil : ∀ A, ilist A 0
    | cons : ∀ A n, A → ilist A n → ilist A (S n)

  Definition hd\DIFaddbegin \DIFadd{' }\DIFaddend {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.
\DIFaddbegin

  \DIFadd{Definition hd }{\DIFadd{A}} \DIFadd{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[16].

new keyword Program which allows the use of case-analysis in more complex definitions[30][11]. To be more specific, it allows definitions to be specified separately from ~~its~~ 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 ~~in [14] and [25] by~~ introducing a method for user-friendlier dependently-typed ~~programming~~ pattern matching in **Coq** ~~as the~~ in the form of the Equations library[14][25]. 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,

```
Equations hd {A n} (ls : ilist A n) (pf : n <> 0%DIF < nat) : A :=
\DIFaddbegin \DIFadd{) : A :=
}\DIFaddend hd nil pf with pf eq_refl := {\DIFdelbegin \DIFdel{| x :=! x }\DIFdelend
hd (cons h n) _ := h.
```

Code snippet 4: Definition of hd using `Equations`

much of this was invaluable when defining the substitution operators as the regular type checker in Coq often had difficulty ~~when recognizing type equalities~~ unifying dependently typed terms in certain cases.

2.4 Logical relations

Logical relations is a technique often employed when proving programming language properties of statically typed languages[24]. There are two main ways they are used, namely as unary and binary relations. Unary logical relations, also known as logical predicates, are predicates over single terms and are typically used to prove language characteristics such as type safety or strong normalization. Binary logical relations on the other hand are used to prove program equivalences, usually in the context of denotational semantics as we will do. There have been many variations on the versatile technique from syntactic step-indexed relations which have been used to solve recursive types[9], to open relations which enable working with terms of non-ground type[28][29]. Logical relations in essence are ~~simply relations~~ relations between terms defined by induction on ~~the~~ their types. A logical relations proof consists of 2 main steps. The first ~~usually states that well-typed terms~~ states the terms for which the property is expected to hold are in the relation, while the second states that the property of interest follows from the relation. The second step is easier to prove as it usually follows from the definition of the relation. The first on the other hand, will often require proving a generalized variant, called the fundamental property of the logical relation. In most cases this requires that the relation is correct with respect to applying substitutions.

A well-known logical relations proof is the proof of strong normalization of well-typed terms, which states that all ~~well-typed terms are either terminal values or can be reduced further~~terms eventually terminate. An example of a logical relation used in such a proof using the intrinsic strongly-typed formulation is given in ~~snippet~~Snippet 5. Noteworthy is the case for function types, which indicates that an application should maintain the strongly normalization ~~relation.~~ property. If one were to attempt the proof of

```
Equations SN {Γ} τ (t : tm Γ τ): Prop :=
SN unit t := halts t;
SN (τ ⇒ σ) t := halts t \DIFdelbegin \DIFdel{\wedge
    $\wedge$
  }\DIFaddend (\forall (s : tm \Gamma \tau), SN \tau s \rightarrow SN \sigma (app
```

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

strong normalization without using logical relations, they would get stuck in the cases dealing with function types. More specifically when reducing an application, the induction hypothesis is not strong enough to prove that substituting the argument into the body of the abstraction also results in a terminating term. 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[29]. A similar, independent proof of correctness was given ~~in [28] using an~~ by Barthe, et. al.[28] using a syntactic relation.

3 Preliminary Results

3.1 Language definitions

We currently mimic the ~~base~~ types used in the base language of the paper [29] extended with sum types, shown in ~~snippet~~Snippet 6. The paper itself initially makes use of the standard types found in a ~~simply typed~~ simply-typed lambda calculus extended with products and R as the only ground type. These are also the types used ~~in [28] which gives similar~~ by Barthe, et. al.[28] in their proof. In a later section ~~Stanton, Huot and~~ Huot and Staton, 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.

We have chosen the intrinsic strongly-typed formulation used ~~in~~ by Nick Benton et. al.[15] 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~~Snippet 2.

We have simplified a few of the language constructs used in the main

```
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

paper by Huot, Staton, and Vákár[29], shown in ~~snippet~~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 [29]~~ by Huot, Staton and Vákár[29] extended with sum types. 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 the set-theoretic types in **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~~Snippet 8 and 9, while ~~snippet~~Snippet 10 states the actual correctness theorem. During development, we also discovered that while the abstract proof was correct, the concrete fundamental lemma used in the proof of `Theorem 1` in the paper was incorrect.

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~~ by Huot, Staton, and Vákár[29]. 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~~

```
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 ~~i~~in Snippet 8

~~to go towards a SSA (static single assignment) representation which has many benefits as a well-known possible internal representation for use in compilers[10]. One other benefit is that this representation can also be transformed into a minimal functional language[5].~~ array types which should be better for performance than the n-tuples used in the original proof. We will be trying to extend the proof with the results achieved by Shaikhha, et. al.[23], where they also successfully implement several compile-time optimizations.

4.2 Deadlines

The hard deadlines for the first and second phases of the thesis are respectively May 1st and September 18th. The ambition is to follow the ~~following~~ framework of deadlines in Table 2. Note that until the proofs deadline, the proofs and paper will largely be written in parallel with each other.

```
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

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

Table 2: Framework of deadlines

References

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

[2] T. Coquand and G. Huet, "The calculus of constructions", *Information and Computation*, vol. 76, no. 2-3, pp. 95–120, Feb. 1988. DOI: 10.1016/0890-5401(88)90005-3. [Online]. Available: https://doi.org/10.1016/0890-5401(88)90005-3 (cit. on p. 7).

[3] T. Coquand and C. Paulin, "Inductively defined types", in *COLOG-88*, Springer Berlin Heidelberg, 1990, pp. 50–66. DOI: 10.1007/3-540-52335-9_47. [Online]. Available: https://doi.org/10.1007/3-540-52335-9_47 (cit. on p. 7).

[4] T. 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).

[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. 9).

[7] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic, "Mechanized metatheory for the masses: The PoplMark challenge", in *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2005, pp. 50–65. DOI: 10.1007/11541868_4. [Online]. Available: https://doi.org/10.1007/11541868_4 (cit. on p. 8).

[8] R. Adams, "Formalized metatheory with terms represented by an indexed family of types", in *Types for Proofs and Programs*, J.-C. Filliâtre, C. Paulin-Mohring, and B. Werner, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–16, ISBN: 978-3-540-31429-5 (cit. on p. 9).

[9] A. Ahmed, "Step-indexed syntactic logical relations for recursive and quantified types", in *Programming Languages and Systems*, P. Sestoft, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 69–83, ISBN: 978-3-540-33096-7 (cit. on p. 12).

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

[11] 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 pp. 10, 11).

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

[13] 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. 7).

[14] M. Sozeau, "Equations: A dependent pattern-matching compiler", in *Interactive Theorem Proving*, Springer Berlin Heidelberg, 2010, pp. 419–434. DOI: 10.1007/978-3-642-14052-5_29. [Online]. Available: https://doi.org/10.1007/978-3-642-14052-5_29 (cit. on pp. 7, 11).

[15] 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, 13).

[16] A. Chlipala, *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, 2013, ISBN: 0262026651 (cit. on p. 11).

[17] A. Mahboubi and E. Tassi, "Canonical structures for the working coq user", in *Interactive Theorem Proving*, Springer Berlin Heidelberg, 2013, pp. 19–34. DOI: 10.1007/978-3-642-39634-2_5. [Online]. Available: https://doi.org/10.1007/978-3-642-39634-2_5 (cit. on p. 7).

[18] 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. 7).

[19] 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).

[20] S. Boldo, C. Lelay, and G. Melquiond, "Coquelicot: A user-friendly library of real analysis for coq", *Mathematics in Computer Science*, vol. 9, pp. 41–62, 2015 (cit. on p. 7).

[21] B. C. Pierce, A. A. de Amorim, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, A. Tolmach, and B. Yorgey, *Programming Language Foundations*, ser. Software Foundations series, volume 2. Electronic textbook, May 2018 (cit. on p. 8).

[22] 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).

[23] 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, 17).

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

[25] M. Sozeau and C. Mangin, "Equations reloaded: High-level dependently-typed functional programming and proving in coq", *Proceedings of the ACM on Programming Languages*, vol. 3, no. ICFP, pp. 1–29, Jul. 2019. DOI: `10.1145/3341690`. [Online]. Available: `https://doi.org/10.1145/3341690` (cit. on p. 11).

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

[27] A. Aaby, "Introduction to programming languages", *Syntax Imperative Programming Concurrent Programming Object-Oriented Programming Evaluation*, vol. 3, Apr. 2020 (cit. on p. 6).

[28] G. Barthe, R. Crubillé, U. D. Lago, and F. Gavazzo, *On the versatility of open logical relations: Continuity, automatic differentiation, and a containment theorem*, 2020. arXiv: `2002.08489 [cs.PL]` (cit. on pp. 12, 13).

[29] 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, 4, 6, 12–14).

[30] 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 pp. 10, 11).