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

March 29, 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 very difficult problems very quickly. One of the principle techniques in practice is the use of the gradient descent algorithm, which takes an optimization problem and tries to find a solution by iteratively moving towards the optimum.

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.

Name examples of recent papers

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[13]. 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.

Fill in

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[9]. Automatic is a generalization of backpropagation. 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 which I will both discuss. These are namely forward and reverse mode AD. In this paper we will prove the semantics of a forward mode AD algorithm correct. The algorithm being a very simple macro on the syntax of a simply typed lambda calculus. Fill in

2.2 Denotational semantics

The notion of denotational semantics, created by Dana Scott and Christopher Strachey, tries to find underlying mathematical objects able to explain the properties of programming languages. Their original search for a solution for lambda calculi led them to well-known concepts such as partial orderings and least fixed points. In this model, programs are interpreted as partial functions and computation by taking fixpoint of such functions. Non-termination on the other hand is symbolized by a value bottom that is lower in the ordering relation than any other element. This search for an underlying mathematical foundation for languages is also known as domain theory.

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 paper was inspired by, the models used were diffeological spaces, which are the useful generalization of smooth manifolds and as such describe sets whose functions are always differentiable. For the purpose of this thesis, however, this was deemed overkill 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 denotation. Fill in

2.3 Coq

Coq is a proof assistant created by Thierry Coquand as an implementation of his calculus of constructions type theory[1]. 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[2], dependent pattern matching[6] and advanced modular constructions for organizing colossal mathematical proofs[5][8].

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, $\forall x, f(x) = g(x) \rightarrow f = g$. 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. 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. The other approach, also called an intrinsic representation or strongly typed terms, 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[3]. 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.

The approach used in the rest of this thesis is closely related to the de-bruijn representation, which numbers variables relative to the binding lambda term. Instead of using numbers to represent the distance, we make use of indices which should be present in the typing context. The specific formulation used in this thesis was fleshed out by Nick Benton, et. al.[7]. While this does subvert the problems present in the nominal representation, it unfortunately does have problems of its own. Variable substitutions are defined using two separate renaming and substitution operations. Renaming is formulated as a function from variables to variables just changing the typing context and substitution actually swaps the variables for terms. Due

to using indices from the context as variables, some lifting boilerplate is needed to manipulate the current context.

```
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 1: Basis of a simply typed λ-calculus using a strongly typed intrinsic formulation.

Extend example

2.3.2 Dependent types in Coq

In Coq, one can normally write function definitions using either case-analysis as 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. Writing definitions using tactics also has its limitations, the user is allowed definitions that later are uncovered to be unprovable, the definitions are opaque such that the standard 'simpl' tactic which invokes Coq's reduction mechanic is not able to reduce the term. This often requires the user to write and use proofs of the functions reducibility at its argument.

In [14] and [4] Matthieu Sozeau introduces an extensions to Coq via a new keyword 'Program' which allows the use of case-analysis in more complex definitions. To be more specific, it introduces a method called predicate subtyping to Coq which allows more permissive definitions with the proofs accompanying them being written separately. It does this in the language Russell, which contains a slightly weaker type system in that terms do not require proof terms for subset types. This definition is translated into Coq where the missing proofs are filled in. 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 can make debugging error messages even harder than it already is in a proof assistant.

Sozeau further improves on this in [6] and [11] 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 makes use of a notion called covering where a set of equations should be an exhaustive covering of a function signature. 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. This was invaluable when using well-scoped well-typed de-bruijn representation discussed in the previous section.

Example: Chlipala's Certified Programming with Dependent Types

Fill in

Fill in

2.4 Logical relations

Logical relations, otherwise known as Tait's method, is technique often employed when proving programming language properties[10]. They are defined as relations over the types of the language. 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.

The proof given in the paper this thesis is based on is a logical relations proof using diffeological spaces as the denotated domains[13]. A similar, independent proof of correctness was given in [12] using an syntactic open logical relation.

3 Preliminary Results

Fill in

3.1 Preliminary Proof

Fill in

3.2 Proof Extension

Fill in

4 Timetable and Planning

Fill in

4.1 Deadlines

Fill in

4.2 Extensions

Fill in

References

[1] 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. 4).

[2] 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. 4).

[3] 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. 5).

[4] 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. 6).

[5] 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. 4).

[6] 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. 4, 7).

[7] 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 p. 5).

[8] 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. 4).

[9] 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. 3).

[10] L. Skorstengaard, *An introduction to logical relations*, 2019. arXiv: 1907.11133 [cs.PL] (cit. on p. 7).

[11] 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. 7).

[12] 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. 7).

[13] 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, 7).

[14] 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. 6).