

AI-DSL Technical Report (May to Septembre 2022)

Nil Geisweiller, Samuel Roberti

October 19, 2022

Abstract

Contents

1	Introduction	2
1.1	Setting the Scene	2
1.2	Work Accomplished	2
1.3	Related Work	3
2	Implementation and Verification of AI Algorithms	4
2.1	Implementation of AI Algorithms	4
2.1.1	Descent	5
2.1.2	Gradient Descent	5
2.1.3	Linear Regression	5
2.1.4	Logistic Regression	6
2.1.5	Logistic-Linear Regression	6
2.2	Verification of AI Algorithms	6
3	Program Synthesis	9
3.1	Language Framework	9
3.2	Idris Elaboration	9
3.3	Idris Proof Search	9
4	Conclusion	10
A	Glossary	11

Chapter 1

Introduction

1.1 Setting the Scene

In the previous iteration we explored using Dependent Types to express formal specifications of AI services, with the ultimate goal of building a language for easily writing those specifications, the AI-DSL itself, as well as services to automatically connect AI services together, the AI-DSL Registry [1].

Back then we experimented with trivial AI services, computing simple arithmetic, described by trivial properties, such as the parity of their inputs/outputs. We were able to demonstrate that Idris, our DTL of choice, could be used to verify the correctness of such AI service assemblages. The approach seemed promising, but to really put the idea to the test we had to make progress on two fronts:

1. Replace trivial AI services by actual AI algorithms.
2. Explore program synthesis, as it became clear that it was at the heart of this endeavor. First, for building the AI service assemblages themselves. Second, for achieving fuzzy matching, that is when AI services almost fit together but not quite yet. And third, for completing assemblages when some AI services are outright missing.

That is what we have done during that iteration.

1.2 Work Accomplished

First we have implemented three AI algorithms in Idris:

1. Gradient descent
2. Linear regression
3. Logistic regression

These algorithms were chosen because they are relatively simple, yet extensively use in real world applications, as well as tightly related to each other. Linear regression can be framed as a gradient descent problem, and logistic regression can be framed both as gradient descent and linear regression problems, thus constituting an excellent case study for the AI-DSL. Alongside these implementations, a descending property was formulated and formally proved for each algorithm.

Finally, we have explored ways to perform program synthesis of dependently typed programs. While we have only achieved partial success as far as program synthesis is concerned, we were able to demonstrate its feasibility within the Idris ecosystem. It was clear from the start anyway that to be done well and fully, program synthesis essentially requires achieving AGI. Indeed, it is one of these AI-complete problems. That is any problem can be framed as a program synthesis problem and vice versa. The idea being that such functionality can be progressively grown, deferred less and less to human intervention, as the network and the AI-DSL evolve.

1.3 Related Work

Here's a list of projects and publications we have discovered along the way that relate to the work done during that iteration. TODO

Chapter 2

Implementation and Verification of AI Algorithms

2.1 Implementation of AI Algorithms

We have implemented the following AI algorithms in Idris:

1. Descent: a generic descending algorithm.
2. Gradient Descent: a gradient descent algorithm using Descent.
3. Linear Regression: a linear regression algorithm using Gradient Descent.
4. Logistic Regression: a logistic regression algorithm using Gradient Descent.
5. Logistic-Linear Regression: a logistic regression algorithm using Linear Regression.

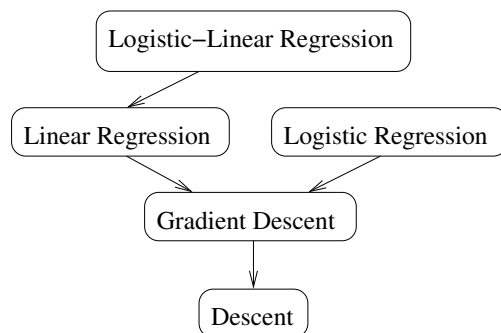


Figure 2.1: AI algorithms call graph

A call graph is provided in Figure 2.1. Each algorithm may be viewed as an AI service, together forming a network of AI services delegating work to one another when possible.

The idea of performing logistic regression via two paths, either directly via calling Gradient Descent, or indirectly via calling Linear Regression, came from the ambitious goal of having our AI-DSL prototype discover an alternate way, possibly unforeseen by the AI practitioner, to perform a certain AI tasks, here logistic regression. As we will see we did not come far enough to achieve that goal, but we certainly keep on the side for the future.

Let us now describe in more details what these algorithms are doing, and then provide the descending property we have focused on in this work.

2.1.1 Descent

The Descent algorithm takes in input:

1. a cost function to minimize;
2. a step function to jump from candidate to candidate;
3. an initial candidate to start the search from;
4. a maximum number of steps allocated to the search;

and outputs the final candidate as well as the remaining unallocated steps.

2.1.2 Gradient Descent

The Gradient Descent algorithm takes in input:

1. a loss function;
2. a gradient function;
3. a learning rate, also called step size;
4. an initial candidate to start the search from;
5. a maximum number of steps allocated to the search;

converts the gradient function and the learning rate into a step function, calls the Descent algorithm and returns the final candidate as well as the remaining unallocated steps.

2.1.3 Linear Regression

The Linear Regression algorithm takes in input:

1. a data set to explain, a matrix of inputs and a column vector of outputs;
2. a learning rate, also called step size;

3. an initial candidate to start the search from;
4. a maximum number of steps allocated to the search;

defines a sum-of-squared-errors-based loss and gradient functions for that data set, calls Gradient Descent and returns the final candidate as well as the remaining unallocated steps.

2.1.4 Logistic Regression

The Logistic Regression algorithm takes in input:

1. a data set to explain, a matrix of inputs and a Boolean column vector of outputs;
2. a learning rate, also called step size;
3. an initial candidate to start the search from;
4. a maximum number of steps allocated to the search;

defines a cross-entropy-based loss and gradient functions for that data set, calls Gradient Descent and returns the final candidate as well as the remaining unallocated steps.

2.1.5 Logistic-Linear Regression

The Logistic-Linear Regression algorithm takes in input:

1. a data set to explain, a Boolean matrix of inputs and a Boolean column vector of outputs;
2. a learning rate, also called step size;
3. an initial candidate to start the search from;
4. a maximum number of steps allocated to the search;

transforms the data set so that the column vector of outputs represents the odds of outputting True instead of a Boolean value, calls linear regression on that transformed data set and returns the final candidate as well as the remaining unallocated steps.

2.2 Verification of AI Algorithms

The concept of verifying properties of AI algorithms is a very broad one, could be verifying the AI algorithms themselves, or their output models, either using crisp mathematical properties, or empirical fuzzy NEXT

Here we focus on the simplest one we could possibly come up with in this situation, which is that the algorithm must descend, or at least not ascend.

In other words, that the final candidate must be better, or at least not worse, than the initial one. This may seem like an overly simplistic property, and it is. However, as we will see, working with that was already quite an educational journey.

Let us begin by showing the Idris implementation of Descent, or rather a slightly simplified version modified for expository purpose:

```
descent : Ord cost_t =>
  (cnd_t -> cost_t) ->      -- Cost function
  (cnd_t -> cnd_t) ->      -- Step function
  (cnd_t, Nat) ->          -- Init candidate, steps
  (cnd_t, Nat)             -- Final candidate, steps
descent _ _ (cnd, Z) = (cnd, Z)
descent cost next (cnd, S k) = if cost (next cnd) < cost cnd
                               then descent cost next (next cnd, k)
                               else (cnd, (S k))
```

It essentially expresses that if the cost of the next candidate is less than the cost of the initial candidate, it should recursively descend from the next candidate, otherwise return the initial candidate. The descending property can then be formalized as follows:

```
descent_le : Ord cost_t =>
  (cost : cnd_t -> cost_t) -> -- Cost function
  (next : cnd_t -> cnd_t) -> -- Step function
  (cas : (cnd_t, Nat)) ->    -- Init candidate, steps
  (cost (fst (descent cost next cas)) <= cost (fst cas)) == True
```

which expresses that the cost of the final candidate should be less than or equal to the cost of the initial candidate¹. Obviously such property should be trivial to proof given how the algorithm has been written. In practice however, it is not so trivial, for two reasons:

1. Idris makes no assumption about the comparison operators <, >, <= and >=. The interface `Ord` guaranties that `cost_t` implements these operators, but not how they should behave. Thus one needs to encode these assumptions and make sure that they are true for the types of interest, which is not always easy, or even possible, especially for primitive types like `Double`. NEXT: Stefane prim vs implementing ourselves.
2. As the algorithm is recursive, it requires a recursive proof.

Here is a proof of `descent_le`, slightly simplified as `descent`. Let us first deal with the case where the number of allocated steps is zero:

```
descent_le _ _ (_, Z) = le_reflexive
```

¹For information, `==` denotes the equality type, a dependent type with `Ref1` as sole constructor corresponding to the reflexivity axiom of equality.

In order to proof the descending property it suffices to invoke the reflexivity of \leq since for that case `descent` merely becomes the identity function. Let us now examine the case where the number of allocated steps is greater than zero:

```
descent_le cost next (cnd, S k)
  with ((cost (next cnd)) < (cost cnd)) proof eq
_ | True = let des_le_nxtcst = descent_le cost next (next cnd, k)
            nxtcst_le_cst = le_reflexive_closure_lt (Left eq)
            in le_transitive des_le_nxtcst nxtcst_le_cst
_ | False = le_reflexive
```

The proof considers the two branches of the conditional. If the condition is false then invoking the reflexivity of \leq suffices for the same reason as above. If the condition is true then the proof needs to combine more axioms pertaining to comparison with the recursion of `descent_le`. That proof, even simplified, is already somewhat substantial, likely too substantial to be rapidly discovered by greedy proof search. And the non simplified version, which can be found here [?] is about twice that size.

NEXT: other algorithms.

Chapter 3

Program Synthesis

3.1 Language Framework

3.2 Idris Elaboration

3.3 Idris Proof Search

Idris proof search, contrary to what its name may suggest, can be used for program synthesis in general, not just proof search – which is no surprise to those familiar with the Curry-Howard correspondence.

Chapter 4

Conclusion

Appendix A

Glossary

- **AI service assemblage:** collection of AI services interacting together to fulfill a given function. Example of such AI service assemblage would be the Nunet Fake News Warning system.
- **Dependent Types:** types depending on values. Instead of being limited to constants such as `Integer` or `String`, dependent types are essentially functions that take values and return types. A dependent type is usually expressed as a term containing free variables. An example of dependent type is `Vect n a`, representing the class of vectors containing `n` elements of type `a`.
- **Dependently Typed Language:** functional programming language using dependent types. Examples of such languages are Idris, AGDA and Coq.
- **DTL:** Shorthand for Dependently Typed Language.

Bibliography

- [1] Nil Geisweiller, Kabir Veitas, E.S.A.S.R.M.I.B.G.: AI-DSL Technical Report (February to May 2021) (2021), https://github.com/singnet/ai-dsl/blob/master/doc/technical-reports/2021-May/ai-dsl-techrep-2021-05_may.pdf