AI-DSL Technical Report
(May to Septembre 2022)

DRAFT

Nil Geisweiller, Samuel Roberti, Matthew Iklé, Deborah Duong

November 11, 2022

Abstract

Contents

1 Introduction **3**
1.1 Setting the Scene . . . 3
1.2 Work Accomplished . . . 3
1.3 Related Work . . . 4
1.3.1 Machine Learning Formal Verification . . . 4
1.3.2 Program and Proof Synthesis . . . 5

2 Implementation and Verification of AI Algorithms **6**
2.1 Implementation of AI Algorithms . . . 6
2.1.1 Descent . . . 7
2.1.2 Gradient Descent . . . 7
2.1.3 Linear Regression . . . 7
2.1.4 Logistic Regression . . . 8
2.1.5 Logistic-Linear Regression . . . 8
2.2 Verification of AI Algorithms . . . 8
2.2.1 Descending Property for Descent . . . 9
2.2.2 Descending Property for Other Algorithms . . . 11

3 Program Synthesis **13**
3.1 Language Framework . . . 13
3.2 Idris Elaboration . . . 13
3.3 Idris Proof Search . . . 13
3.3.1 Program Synthesis with Abstract Syntax Trees . . . 13
3.3.2 Program Synthesis with Variables . . . 14
3.4 Coevolutionary Intelligent Agent System . . . 15

4 Conclusion **17**
4.1 Future Work . . . 17
4.1.1 Shortcomings and Solutions . . . 17
4.1.2 Axioms and Uncertainty . . . 18
4.1.3 More, More and More Algorithms . . . 18
4.1.4 Program Synthesis . . . 18
4.1.5 AI-DSL Syntax and Users . . . 18

A Glossary **19**

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 [4].

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.

1.3.1 Machine Learning Formal Verification

The most relevant work we have found so far is described in a paper entitled *Developing Bug-Free Machine Learning Systems With Formal Mathematics* [11]. In that paper a formal specification of a class of stochastic gradient descent algorithms operating on stochastic computation graphs is implemented in Lean [9], alongside a property expressing that the back propagation correction points, in average, towards the gradient descent of the cost. Mathematically, this may be expressed by the following equality

$$\mathbb{E}_{g,\theta}[\texttt{bprop}(g, \theta, \mathbf{X})] = \nabla_\theta(\mathbb{E}_{g,\theta}[\texttt{cost}(g, \mathbf{X})])$$

where g is a stochastic computation graph parameterized by θ, $\mathbf{X}$ is a random vector describing the values sampled from g, `bprop` the is back propagation function and `cost` is the cost function. Such equality is formally expressed in Lean using existing and introduced mathematical vocabulary to express notions of probability and measure theory such as expectation, integration and derivation, then proved using tactics developed for that purpose. The authors admit that their prove assumes infinite-precision real numbers as opposed to finite-precision floating point numbers used in practice. However, they point to a couple of papers addressing the use of floating point numbers in the context of automatic theorem proving [7, 10].

Another related work presented in [2] aims to prove properties about learned models, as opposed to learning algorithms. Formalizing Hoeffding's inequality [8] in Coq [3], the authors show how to automatically prove the extend to which a given model, such as a perception, generalizes on unknown data.

TODO: mention various relevant Idris projects.

1.3.2 Program and Proof Synthesis

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.

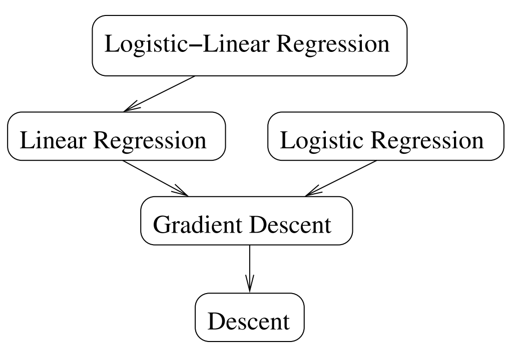

Figure 2.1: AI algorithms call graph

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

2.2.1 Descending Property for Descent

Here we focus on the simplest one we could possibly imagine 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, that 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. Note that `cnd_t` and `cost_t` are type variables, that is they may be substituted by any type, up to some constraints, at function call. 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 candidate[1]. Obviously such property should be trivial to prove given how the algorithm has been written. In practice however, it is not so, 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`.

2. Since the algorithm is recursive, it requires a recursive proof.

To address the first reason we added a number of functions formalizing the usual axioms of total strict and non-strict orders of `<`, `>`, `<=` and `>=`. A small snippet is given below:

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

```
/// Assume that < is irreflexive
lt_irreflexive : Ord a => {0 x : a} -> (x < x) === False
lt_irreflexive = believe_me ()

/// Assume that < is connected
lt_connected : Ord a => {0 x, y : a}
                     -> (x < y) === False
                     -> (y < x) === False
                     -> x === y
lt_connected _ _ = believe_me ()

/// Assume that <= is reflexive
le_reflexive : Ord a => {0 x : a} -> (x <= x) === True
le_reflexive = believe_me ()

/// Assume that <= is transitive
public export
le_transitive : Ord a => {0 x, y, z : a}
                      -> (x <= y) === True
                      -> (y <= z) === True
                      -> (x <= z) === True
le_transitive _ _ = believe_me ()
```

The whole list of axioms can be found in file OrdProofs.idr of the ai-dsl repository. We also attempted to use an existing library from Stefan Höck called idris2-prim, but decided to write our own for more flexibility.

The proof of `descent_le`, slightly simplified to suit our simplified version of `descent`, is presented below. Let us first deal with the base case where the number of allocated steps is zero:

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

In order to prove the descending property it suffices to invoke the reflexivity of `<=` since for that case `descent` merely becomes the identity function. Let us now examine the recursive 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 `<=` suffices for the same reason as above. If the condition is true then the proof needs to combine axioms about comparison with the recursion of `descent_le` and the transitivity of `<=`.

That simplified proof is already somewhat substantial, likely too substantial to be rapidly discovered by a greedy proof search algorithm. The non simplified version of Descent as well as the proof of its descending property, about the double the size of the simplified one, can be found in file Descent.idr of the ai-dsl repository. Discovering such a proof automatically or semi-automatically still remains relatively practical, either by requiring human intervention, using proof tactics or more sophisticated inference control techniques [6].

2.2.2 Descending Property for Other Algorithms

Once the descending property has been proved for Descent, proving it for the remaining algorithms is now truly trivial, for the most part anyway.

Let us provide an example for Gradient Descent, starting by recalling what is the gradient descent algorithm. Given a loss function L and a learning rate η, the gradient descent algorithm works by updating the candidate β as follows

$$\beta := \beta - \eta \nabla L(\beta)$$

in other words, the step function takes the opposite direction of the gradient by a factor of η. The Idris code of Gradient Descent is given below:

```
gradientDescent : (Ord a, Neg a) =>
  (cost : ColVect m a -> a) ->           -- Cost function
  (grd : ColVect m a -> ColVect m a) -> -- Gradient
  (eta : a) ->                           -- Learning rate
  (cas : (ColVect m a, Nat)) ->          -- Init candidate, steps
  (ColVect m a, Nat)                     -- Final candidate, steps
gradientDescent cost grd eta = descent cost (fsgrd grd eta)
```

where `fsgrd` is a function that takes a gradient, `grd`, a learning rate, `eta`, and produces the step function described above. The type of a candidate for Gradient Descent is now more specific. Instead of being the variable type `cnd_t`, it is a column vector of size `m` and type `a` represented by `ColVect m a`.

The descending property for Gradient Descent is expressed as follows:

```
gradientDescent_le : (Ord a, Neg a) =>
  (cost : ColVect m a -> a) ->            -- Cost function
  (grd : ColVect m a -> ColVect m a) ->  -- Gradient
  (eta : a) ->                            -- Step size
  (cas : (ColVect m a, Nat)) ->           -- Init candidate, steps
  (cost (fst (gradientDescent cost grd eta cas)) <= cost (fst cas))
   === True
```

And its proof is simply

```
gradientDescent_le cost grd eta = descent_le cost (fsgrd grd eta)
```

that is the proof of the descending property of Descent. Idris is able to directly reuse it because it automatically applies the rule of replacement in the type definition on the function calls present in it by using their definitions. So for instance

```
(cost (fst (gradientDescent cost grd eta cas)) <= cost (fst cas))
```

is automatically replaced by

```
(cost (fst (descent cost (fsgrd grd eta) cas)) <= cost (fst cas))
```

which is what `descent_le` proves.

Proving the descending properties on the other algorithms, with the exception of Logistic-Linear Regression, is equally trivial. Proving it for Logistic-Linear Regression requires an explicit use of the rule of replacement.

Chapter 3

Program Synthesis

3.1 Language Framework

3.2 Idris Elaboration

3.3 Idris Proof Search

Since recently, Idris2 has introduced a functionality called Proof Search. Contrary to what its name suggests however, it can be used for program synthesis, not just proof search – which should be no surprise to those familiar with the Curry-Howard correspondence. It has however, at the time of writing this document, a number of downsides. The main one being it can only access

1. data type constructors,

2. variables in its current environments.

Meaning, it does not have access to functions or constants defined in the current and imported modules. The other downsides are that it is poorly documented and difficult to control, likely due to having being introduced so recently.

Nonetheless, in this section we explore how such functionality can be used for program synthesis in spite of its current limitations.

3.3.1 Program Synthesis with Abstract Syntax Trees

The idea is to represent programs as Abstract Syntax Trees. Each operator can be represented as a constructor of that data structure of that Abstract Syntax Tree, which Idris can access to generate trees representing programs. Here is a minimal example:

```
/// Abstract Syntax Tree Types
data Ty = TyDouble | TyCandidate | TyFun Ty Ty
```

```
||| Abstract Syntax Tree Terms
data Expr : Ty -> Type where
    Candidate : Expr TyCandidate
    Loss : Expr (TyFun TyCandidate TyDouble)
    Gradient : Expr (TyFun TyCandidate TyCandidate)
    Descent : Expr (TyFun TyCandidate TyDouble) ->
              Expr (TyFun TyCandidate TyCandidate) ->
              Expr TyCandidate ->
              Expr TyCandidate
```

Then one can ask Idris to fill the hole of the following definition

```
linearRegression : Expr TyCandidate
linearRegression = ?hole
```

which it successfully does by suggesting a number of candidates to replace `?hole` by, such as

```
Candidate
Descent Loss Gradient Candidate
Descent Loss Gradient (Descent Loss Gradient Candidate)
...
```

The second suggestion corresponds to implementation we are looking for.

3.3.2 Program Synthesis with Variables

Let us now explore using environment variables to represent constant and functions instead of constructors. The meta-function `syn` described below:

```
syn : (a -> b -> c) ->
      (a -> b -> c) ->
      (a -> b -> c) ->
      a -> b -> c
syn f g h x y = ?hole
```

and takes 3 functions, `f`, `g` and `h`, as arguments, and outputs a function that takes 2 arguments of types `a` and `b` respectively. Idris can successfully attempt to can fill the hole by suggesting the following candidates

```
h x y
g x y
f x y
```

which cover all possibilities in that instance.

Here is another example attempting to reproduce the one using Abstract Syntax Trees provided in Section 3.3.1.

```
syn : (cnd -> cnd) ->                               -- Step function
      ((cnd -> cost) -> (cnd -> cnd) -> cnd -> cnd) -> -- Descent
      (cnd -> cost) ->                              -- Cost function
      cnd ->                                        -- Init candidate
      cnd                                           -- Final candidate
syn n d c i = ?hole
```

Idris again finds the candidate we are looking for, that is the second suggestion in the list below:

```
i
d c n i
d c n (d c n i)
...
```

The full experiments can be found in folder idris-proofsearch of the ai-dsl repository, and contain more attempts including unsuccessful ones using the `let` keyword not covered here. Of course these experiments are both very simplistic and too unconstrained but the fact that they work indicates that synthesizing programs, with more operators and types, including dependent types representing properties, should be possible with standalone Idris. And as Idris Proof Search functionality improves, it might even become a viable option in practice. Other options that would be worth exploring would be to experiment with the Proof Search functionalities of other DTLs such as AGDA and Coq.

3.4 Coevolutionary Intelligent Agent System

TODO: properly integrate Debbie paragraphs.

In the usecase for longevity, Singularity Net spinoff Rejuve.AI will use AI-DSL in tandem with a coevolutionary multi intelligent agent reinforcement learning algorithm, the Generative Cooperative Network (GCN), to combine crowdsourced models into a dynamic multiresolutional mechanistic model of the human body. Here it will do model synthesis rather than program synthesis, putting together generative Bayesian, neural and simulation models and data from separate studies into a coherent whole. The GCN will do implicit typing, that automatically categorizes models into groups of similar implicit requirements for sucess, where the measure of success is the amount of simulated tokens a model can win from multiple simulated challenges in a simulated market. Agents compete to win challenges but also cooperate in that they employ each others services, to delegate specialized knowledge to other "expert" models. Agents learn a system of signs that come to represent their emergent role category and the requirements that go along with those roles. The system of roles is the agent "culture", a functional semantic space that scaffolds other agents, including new agents that have not converged yet, along a path that leads them to the solutions that other agents have found in the past. However scaffolded, and however reachable by evolutionary computation, traveling along such a path is done by trial and error. Agents have classified themselves into types, the signs of which exist in a functional semantic space. However, the sign is limited in that it must basically be memorized. It is only rewarded when it is learned correctly, relative to other agents.

This sort of approach can carry us a significant distance toward modeling longevity related data, but it is likely to reach its limits. In order for the agent culture to contain open ended intelligence, it can not learn everything by trial and error: rather, the emergent type ontology will need to somehow be made explicit and carry with it explicit instructions on requirements. This becoming explicit is the fourth way in which signs encourage open ended emergence. For this we leverage the AI-DSL (Goertzel and Geisweiller 2020) strategy for agent typing. Hyperon's pattern miner will assist in finding what it is about the agents displaying a role sign which enables its teams to make a profit. PLN inference will express this in AI-DSL, which Hyperon will use to compose the answer from user contributed models, and formally verify exactly what those models do (Goertzel 2014). The implicit (emergent sign) and explicit AI-DSL methods that GCN agents use are complementary and help each other. The implicit sign method focuses selective pressure on agents long enough for choices to be objectified into institutions so that they are consistent and widespread enough for explication. Implicit signs supply the explicit algorithms with enough examples of emergent capabilities in the ecosystem to infer upon. Explication takes away some of the burden of memorization of implicit signs by trial and error for new agents, so signs can indicate emerging requirements while explicit rules indicate requirements that have already become objectified institutions. Agents and the signs that they display will be fed to the explicit algorithm which will use Hyperon's pattern mining to interpret the implicit sign's explicit meaning, through an examination of the behaviors of the agents that display the sign. Once explicit, the hyperon formalization of the sign is a directive that is implementable by agents new to an agent ecosystem, that no longer need to learn the meaning of those particular signs by trial and error.

Chapter 4

Conclusion

4.1 Future Work

This work is just scratching the surface. Let us explore the developments that the foreseeable future may bring.

4.1.1 Shortcomings and Solutions

The formal specifications of the algorithms that we have covered here is very minimal. We have only formalized the descending property and more is needed. The exact set of properties we want to formalize is yet to be determined, but could include:

- The gradient function provided to the gradient descent algorithm is, or approximates, the actual derivative of the cost function.
- Cost functions, such as sum of squared errors in the case of linear regression and cross-entropy in the case of logistic regression, measure information losses.
- The cost function has a certain topology, such as being convex. This is useful to know for instance if the resulting candidate approximates the global optimum or not.
- Linearity of the models in the case of linear regression.

These are just examples of properties that are immediately applicable to our five AI algorithms. More broadly there are many more properties of interest, pertaining not only to the algorithms themselves but as they interact with the real world. For instance a property could express the impracticality of introducing backdoors during training [1]. The breadth and utility of what can be formalized is simply enormous.

4.1.2 Axioms and Uncertainty

They are a few of problems regarding the assumptions about the comparison operators `<`, `>`, `<=` and `>=` that, in the case of `Double`, are known to be incorrect due to imprecision errors and handling of special cases such as `inf` and `nan`. It's not entirely clear yet how to address that. One way could be to replace `Double` by an arbitrary precision floating number data type. The problem is that such data type can have a high computational overhead, and most existing AI algorithms do not use that anyway. Another solution would be to refine the axioms of `Double` to account for these errors and special cases. A third solution would be to account for uncertainties, more on that below.

Then, the algorithms presented here are deterministic. However it is often the case that nondeterministic algorithms are preferable. For instance one may want to use stochastic gradient descent to avoid local optima. In that case the descending property should be replaced by a stochastic descending property. This brings us to the importance of supporting probabilistic specifications more generally. This is especially relevant to AI algorithms that are not only often nondeterministic but also have their performances typically measured in terms of their fitness to the real world, which is intrinsically and profoundly uncertain. Fortunately, to address that, logic frameworks such as Probabilistic Logic Networks [5] can be used.

4.1.3 More, More and More Algorithms

In the long run we want to provide formal specifications to *all* AI algorithms, ranging from the most specialized, such as Cyclical Stochastic Gradient MCMC for Bayesian Deep Learning [13], to the most general, such as Solomonoff Universal Induction [12]. Of course we, the members of the AI-DSL team, cannot do it all by ourselves. This is a monumental task that will have to be progressively crowdsourced to the community, and eventually to the network itself. The latter has the interesting ramification that the network should progressively take the role of an AI researcher conceiving its own AI algorithms in a justified and principled way. For starter, however, it is important that we provide an initial kernel that can be used in practical applications and serve as didactic example.

4.1.4 Program Synthesis

4.1.5 AI-DSL Syntax and Users

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] Artem, M.: An approach to generation triggers for parrying backdoor in neural networks. AGI (2022)

[2] Bagnall, A., Stewart, G.: Certifying the true error: Machine learning in coq with verified generalization guarantees. Proceedings of the AAAI Conference on Artificial Intelligence **33**, 2662–2669 (07 2019). https://doi.org/10.1609/aaai.v33i01.33012662

[3] Bertot, Y., Castéran, P.: Interactive theorem proving and program development. In: Texts in Theoretical Computer Science An EATCS Series (2004)

[4] Geisweiller, N., Veitas, K., Asfaw, E.S., Roberti, S., Ikle, M., Goertzel, B.: 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`

[5] Goertzel, B., Ikle, M., Goertzel, I.F., Heljakka, A.: Probabilistic Logic Networks. Springer US (2009)

[6] Goertzel, B., Pennachin, C., Geisweiller, N.: Engineering General Intelligence, Part 2: The CogPrime Architecture for Integrative, Embodied AGI, Chapter 18: Adaptive, Integrative Inference Control. Atlantis Press (2014)

[7] Harrison, J.: Floating-point verification using theorem proving. In: SFM (2006)

[8] Hoeffding, W.: Probability inequalities for sums of bounded random variables. Journal of the American Statistical Association **58**(301), 13–30 (1963), `http://www.jstor.org/stable/2282952`

[9] de Moura, L., Kong, S., Avigad, J., van Doorn, F., von Raumer, J.:

[10] Ramananandro, T., Mountcastle, P., Meister, B., Lethin, R.A.: A unified coq framework for verifying c programs with floating-point computations. Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs (2016)

[11] Selsam, D., Liang, P., Dill, D.L.: Developing bug-free machine learning systems with formal mathematics. CoRR **abs/1706.08605** (2017), `http://arxiv.org/abs/1706.08605`

[12] Solomonoff, R.: A formal theory of inductive inference. Information and Control **7** (1964)

[13] Zhang, R., Li, C., Zhang, J., Chen, C., Wilson, A.: Cyclical stochastic gradient MCMC for bayesian deep learning. ICLR (2020)