AI-DSL Technical Report
(May to Septembre 2022)

DRAFT

Nil Geisweiller, Samuel Roberti, Matthew Iklé, Deborah Duong

January 12, 2023

Abstract

This report describes the second phase of the research that has taken place towards developing an AI-DSL for the SingularityNET platform. It focuses primarily on experimenting with the formalization of mathematical properties of AI algorithms, in particular a descending property for the gradient descent algorithm, using the Dependently Typed Language (DTL) Idris. It then turns to program synthesis applied to the problem of automatically composing AI services to meet a particular specification. In conclusion, it contains a description of related work as well as directions for future work.

Contents

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 [17].

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, can 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 is at the heart of this endeavor. First, for building the AI service assemblages themselves. Second, for achieving fuzzy matching, that is when an AI service almost fits a specification but not quite, or when AI services almost get together but not quite. And third, for potentially synthesizing AI services from the ground up.

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 of AI services and the AI-DSL evolve.

The report also contains an extensive, though somewhat packed, section of related work in the concluding chapter as well directions for the future phases.

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.

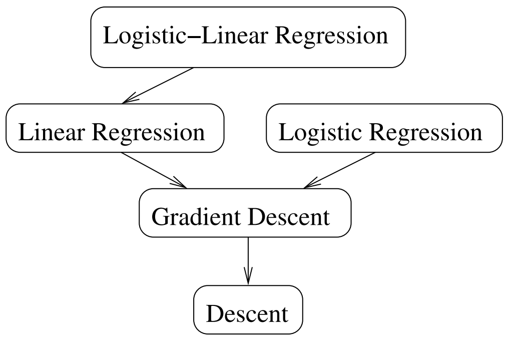

Figure 2.1: AI algorithms call graph

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 ambition of having our AI-DSL prototype discover an alternate way, possibly unforeseen by the AI practitioner, to perform certain AI tasks, here logistic regression. As we will see we did not come far enough to achieve that, but we certainly keep that goal on the side for the future.

Let us now describe in more details what each algorithm is doing, and then provide the descending property we have focused on during this work.

2.1.1 Descent

The Descent algorithm takes the following inputs:

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

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

2.1.2 Gradient Descent

The Gradient Descent algorithm takes the following inputs:

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.

It 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 the following inputs:

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.

It 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.

It 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 the following inputs:

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.

It 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 very broad. It could be verifying AI algorithms themselves, or abstract properties thereof, their output models, using either crisp mathematical properties or empirically based ones, or combinations thereof. In this work we have focused exclusively on crisp mathematical properties on AI algorithms.

2.2.1 Descending Property for Descent

The simplest property we could imagine in this situation 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, given enough steps left, 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 much so, for two reasons:

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

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

```
||| 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 `believe_me` function is provided by Idris especially for these situations. Also, it should be noted that equality properties like `(x <= y) === True` can be sugared into inequalities, but we decided to stay as close as possible to the Idris core language for starter. 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 to maintain as much control as possible.

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, is about twice the size of the simplified one, and can be found in file Descent.idr of the ai-dsl repository. Discovering such a proof automatically, or semi-automatically, would still remains relatively practical, either by requiring human intervention, using proof tactics or more sophisticated inference control techniques [19].

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 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 have access to

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 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
```

External functions found in libraries, or in our case remote AI services, would be represented as constructors of `Expr`, such as `Loss`, `Gradient` and `Descent`. Then one can ask Idris to fill the hole of a definition, such as

```
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 of linear regression we are looking for.

3.3.2 Program Synthesis with Variables

Let us now explore using environment variables to represent constants 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 successfully fills the hole by suggesting the following candidates

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

covering 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)
...
```

where `d`, `c` and `n` are variables referencing to the descent, cost and step functions respectively, and `i` is the initial candidate.

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 simplistic and too unconstrained to narrow down the search to the correct candidate, but the fact that they work at all 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, or, as mentioned in the 4.1 Section, program synthesis with Liquid Haskell.

3.4 Coevolutionary Intelligent Agent System

Another approach, derived from SISTER [13] and being currently developed in SingularityNET, not initially based on dependent types but heavily focused on collective emergent behaviors, including emergent communication and typing, is described below.

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 can leverage the AI-DSL strategy for agent typing. Hyperon's pattern miner, as well as more general forms of reasoning such as that offered by PLN [18], 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. 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 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.

4.1.1 Machine Learning Formal Verification

Over the recent years, and to the best of our knowledge, only a small number of publications have appeared on the application of formal verification to either machine learning algorithms or machine learned models. The most relevant work we have found so far is described in a paper entitled *Developing Bug-Free Machine Learning Systems With Formal Mathematics* [45]. In that paper a formal specification of a class of stochastic gradient descent algorithms operating on stochastic computation graphs is implemented in Lean [35], 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 its existing mathematical vocabulary as well newly introduced one 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 [23, 41].

Another related work presented in [5] aims to prove properties about learned models, as opposed to learning algorithms. Formalizing Hoeffding's inequality [26] in Coq [8], the authors show how to automatically prove the extend to which a given model, such as a perception, generalizes on unknown data.

4.1.2 Smart Contract Formal Verification

The subject of formal verification of smart contracts has gathered a lot more attention. This may not be so relevant to the AI aspect of the AI-DSL, but certainly is relevant to the smart contract interaction aspect that, even though has been somewhat neglected in our research until now, will likely play an important role eventually.

We first have come across a formal specification of the Ethereum Virtual Machine (EVM) provided in [25] using the $\mathbb{K}$ framework [12], enabling formal verification based on Reachability Logic [1]. That work focuses on a fairly low level to verify whether the EVM operates according to its specification, as opposed to verifying smart contracts themselves.

Approaches for doing formal verification on smart contracts, using higher level languages targeting the EVM, also exist. In 2016 a Master's thesis [39] was conducted to explore how Idris can be used to write smart contracts avoiding common failures such as stack overflow or lack of cryptographic encryption when required. On top of using Idris to detect such failures, the authors extended the Idris compiler to support Serpent [30], a high level language for the EVM. To our knowledge unfortunately that work has not been pursued any further. Similarly, a system called, ConCert [2] has been developed to do formal verification of smart contracts written in λ_{smart}, a fragment of the ACORN smart contract language, using Coq [8].

Another important aspect is the verification of smart contract compilers. For instance in [32] the authors describe an approach based on *Translation Certification* for Plutus smart contracts by providing certificates guarantying that the various stages of transformations from Plutus Intermediate Representation (PIR) to Plutus Core (PLC), such as dead-code elimination, inlining and more, do not invalidate high level properties. Although this approach is not as thorough as complete compilation certification, it has the advantage of being more adaptable to compiler evolution. The part providing the *Translation Certificates* is implemented in Coq.

Then they are blockchains that have been created to support formal verification from the beginning. Tezos [33], using Michelson, a low level stack-based smart contract language, comes with a formal verification tool called Mi-Cho-Coq [6] developed in Coq. Higher level languages targeting Michelson also exist such as Albert [7], an Intermediate Level Language, and Juvix [20] a Dependently Typed Language (DTL) based on Quantitative Type Theory (QTT) [4] that not only allows to reason about smart contracts at a high level, but also produces optimized Michelson code by taking advantage of the quantitative aspect of QTT. Likewise, ZILLIQA [49] comes with SCILLA [46], an intermediate-level language enabling formal verification with Coq. The Zen protocol [54] is another such example and supports formal verification via F* [48]. Notably, the type of verification described in its white paper includes computational resources, to formally guaranty that a contract will be allocated enough gaz before running. Another interesting blockchain specifically designed to support formal verification is RChain [42], further elaboration is provided in Section 4.1.4.

In addition, a few more publications can be found in that study [38], as well as other articles from that same book of proceedings. In conclusion, it appears to be a rapidly growing field and there is no doubt that formal verification will progressively become an indissociable part of the blockchain technology given the extremely strong financial incentive to create reliable systems in that context.

4.1.3 Program and Proof Synthesis

One of the most relevant work on program and proof synthesis for dependent types still seems to be the Idris hand book [9] as well as the more recent paper describing QTT, the new theoretical foundation of Idris2 [10]. The AGDA tutorial [37] is also frequently cited but we have yet to go through it.

Beside that, we have found a collection of relevant works aim to address the problem of *Component-Based Synthesis* [14], also called *Modular Synthesis* [24]. Meaning, how to synthesize functions by composing available functions from a large library, such as typically provided by programming language ecosystems, like C++ STL, Python Standard Library and so on. In [21], a system capable of composing RESTful APIs functions to fulfill a given specification is described. Since RESTful types are limited, the system described by the authors begins by enhancing type signatures with *Semantic Types*, exploiting the RESTful type descriptions to heuristically replace non-semantic types like `String` by semantic types like `User` or `Id`. Then performs component-based synthesis using *Type Transition Nets* (TTN) [14], a special kind of Petri Nets. Since Semantic Types are still rather limited, at least compared to Dependent Types, they also use input-output examples to further narrow down the search. TTN-based synthesis works well on *Concrete Data Types*, like `List Int`, but not so well on *Abstract Data Types*, like `List a` where `a` is a variable. To address that a variation of TTN is introduced in [22] called *Abstract Transition Network* (ATN). Such structure is used by HOOGLE+ [28], a search engine similar to HOOGLE [36], that is, it searches the entire Haskell Standard Library, but returns small function compositions satisfying a type signature, instead of standalone functions. Overall, even though that work is still limited to Non-Dependent Type, it is nonetheless quite relevant and contains parts that could potentially be reused for Dependent Types, such as ATN-based synthesis.

Work towards addressing component-based synthesis using more expressive types, approaching full blown dependent types, also exists. The most relevant work seems to be SYNQUID [40], a system that can perform component-based synthesis on functions formally specified with *Refinement Types* [15]. Refinement types are like regular types decorated with predicates. Below is an example of Refinement Type expressing the type of strictly positive integers

$$\{\nu : \texttt{Int} \mid 0 < \nu\}$$

They are generally less powerful than dependent types, although it may depend on the particular systems being compared. Unlike dependent types they do not require proof terms, meaning the function definitions are identical with or without refinement types, only the types, complemented with predicates when desired, differ. The advantage of such approach is that it makes adding refinement types to existing programs easier. The inconvenient is that the proofs are obfuscated, synthesized at compile time by a separate component like an SMT prover. In comparison dependent types treat proofs and programs on the exact same footing. It means that any synthesis method developed for programs de facto works on proofs and vice versa. Beside giving full visibility to the proofs, this also gives more freedom of action when synthesis requires assistance. That is said, refinement types can be very expressive and component-based synthesis already exists for that class thus they should be given carefully consideration.

Another aspect relates to formalizing computational costs and such, which can influence program synthesis by discarding candidates that do not operate within those constraints. The system Resyn [31] is an example of that where a more efficient algorithm to calculate the common elements between sets is discovered by incorporating computational costs is the formal description.

It is also worthwhile mentioning that work [16], aiming at providing a type-theoretic interpretation of example-directed. This provides some foundation for how formal specifications can be composed of abstract mathematical properties mixed with concrete examples. There is no doubt such mixture will be very useful for the AI-DSL. Of course in addition we will need to go further by introducing statistical properties as opposed to crisp mathematical properties, as theoretically explored in [52] for Dependent Types.

Finally, let us mention that work which is not about program synthesis per say but rather program specification synthesis using Liquid Types, short for *Logically Qualified Data Types* [44]. Liquid Types are a decidable fragment of Refinement Types. Their system is able to automatically infer the correct specifications of all functions of an Array OCAML library. A bug that had remained hidden for a long time was revealed and corrected in the process. This work could potentially be useful to infer the specifications of existing AI services that can then be manually corrected, which is easier than having to write a specification from scratch.

Most of the work cited above uses Haskell or extensions thereof. To the best of our knowledge there is no method for component-based synthesis using dependent types. Which is precisely what we need for the AI-DSL.

4.1.4 Languages for the AI-DSL

The AI-DSL will likely be made of at least two languages

1. a back-end language targeting the power user, doing the heavy lifting of program, proof, and possibly specification synthesis and checking;

2. a front-end language for the regular user.

So far we have explored using Idris as back-end language, but there are other choices. The reader may have noticed above languages such as Lean, Coq and F*. We still have to fully examine the pros and cons of each language. However already some observations can be made. Lean and Coq are more oriented towards proving mathematical theorems, while Idris is more oriented towards writing and verifying programs.

The advantage of Idris, as we have experienced first hand, is that proofs and programs are based on the same language, and share much vocabulary. For instance the function `map`, from the `Functor` interface, may be used inside proofs or programs alike. Beside being very elegant, it brings the notion of reusability to a whole new level. AGDA [37] also shares that quality, though seems to be more oriented towards mathematics and less towards programming compared to Idris. Finally, as we have seen, Juvix [20], although dedicated to smart contract programming, is foundationally close to Idris as it is also based on QTT [4]. It is a fairly new language and we will want to keep a close eye on it, because ultimately the AI-DSL will have to interface with smart contracts and Juvix already does that.

Lean and Coq, due to being more oriented towards mathematics, have also their advantages, such as having large libraries of mathematical theories that can be reused when proving properties of programs. There is no doubt that the Idris mathematical library will grow consequently over time, but as of today it is lagging behind.

All these languages above are Dependently Typed Languages. Another class of expressive languages are based on Refinement Types. Even though Dependent Types are generally superior to Refinement Types, as far as we have seen, tools for synthesizing programs and specifications using Refinement Types are currently more mature than tools using Dependent Types. Thus, although Dependently Typed Languages may be be better in the long run, languages based on Refinement Types like Liquid Haskell [50], may provide more benefits in the immediate term. Another, perhaps shallow but potentially meaningful reason to use Haskell, is that Plutus [27], Cardano's smart contract language, is heavily based on Haskell. More exploration on the matter is expected to take place during the next phases to measure the pros and cons of these various languages. But it should also be noted that our current choice, Idris, is planned to be eventually supplanted by MeTTa [51], own home brewed language being built for Hyperon, the next generation of the OpenCog Framework.

Finally, you may have noticed that all these languages are functional, but there are other classes of languages that could potentially be used as well. Process calculi, such as the π-calculus and the ρ-calculus [34] are some examples. These calculi make especially sense in highly distributed environments. In particular, it should be noted that the ρ-calculus is used as foundation of Rholang [43], a smart contract language for RChain [42], a blockchain technology that is heavily focused on concurrency and scaling. Process calculi also have strong type theory foundations, such as Behavioral and Spatial Types [11], as well as a novel approach called Native Types [53], that can auto-generate an expressive type systems given essentially any calculus. Finally, since it is clear that component-based synthesis is compositional by nature, process calculi also fit well that requirement. Other languages built on composition as their primitive operators are also worth exploring. For instance the Kihi language [29] is half-way between a *Concatenative Language*, highly compositional by nature, like Forth and a traditional functional programming language like Haskell. The advantage of Kihi over Forth is that no stack is required, which makes it more amenable to formal verification.

So there is a lot to explore. Fortunately we do not need to come to the end of what seems to be a never ending exploratory rabbit hole to start prototyping the AI-DSL, which we intend to do as soon as the next phase begin. We do intend however to keep exploring the various avenues put forth in that section in parallel of that prototyping work.

4.2 Future Work

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

4.2.1 Shortcomings and Solutions

Regarding formal specification of AI algorithms, what we have done in this work is very minimal. We have only formalized a descending property and a lot more is needed. For gradient descent, the exact set of properties we may 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 are sum of squared errors in the case of linear regression and cross-entropy in the case of logistic regression, measure information losses.
- Cost functions have certain topology, such as convex. This is useful to know for instance if the resulting candidate approximates the global optimum or not.
- The models are linear in the case of linear regression.

These are just examples of properties that are immediately applicable to our 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. This brings the need for an ontology as explained in the previous report [17]. Other properties driven by real world use could include for instance avoiding introducing backdoors during training [3]. There are many more examples, the breadth and utility of what can be formalized is simply enormous.

4.2.2 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 that should be addressed. 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, as done in [45]. 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 [18] can be used.

4.2.3 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 Markov chain Monte Carlo for Bayesian Deep Learning [55], to the most general, such as Solomonoff Universal Induction [47]. 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 outsourced 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 as well as to serve as didactic examples for the community.

4.2.4 Program Synthesis

For the long term is seems somewhat unavoidable we will need to build our own program synthesizers. That would probably give us the greatest amount of control which is especially important for addressing the inherent combinatorial explosion problems associated with program synthesis. Fortunately, that is exactly the sort of things that we, the broader SingularityNET research and development team, are working on via the development of OpenCog Hyperon and MeTTa. For the short term however, we want to rely on existing solutions, such as Idris Proof Search, which might entail investing some of our efforts to improve it, or discovering unconventional ways to use it to work around its limitations.

4.2.5 Work Planned for the Next Phase

The previous phases were mainly about building an understanding of the various challenges ahead, such as how to use dependent types and program synthesis to achieve autonomous AI services composition. In the next phase we intend to put this understanding into practice to ideally create a limited yet functional prototype of the AI-DSL. Below is a tentative, subject to change, description of the steps involved in creating such prototype.

1. Build the necessary tools for releasing an Idris-based SDK for the AI-DSL. Such SDK would initially target the power user. It may contain an Idris library exposing existing AI services from the SingularityNET market place. Such library should be automatically generated by crawling the market place and turning protobuf specifications into Idris type signatures. An offline AI-DSL Registry of some sort.

2. For starter, the AI service formal descriptions may be reduced to regular types, as that is what protobuf offers. This should already enable the user of the AI-DSL SDK to easily call and compose AI services. Such AI service compositions should then be publishable back to the SingularityNET market place. By extension, any AI algorithms implemented in Idris, like the ones implemented during phase 2, should be easily publishable to the market place as well.

3. The AI-DSL SDK could be tested on some of the service compositions described in the snet-service-assemblages.pdf document living within that same folder, such as the one converting an English song to a Chinese song as described in Section 2.5 of that referenced document, by splitting vocals and instruments, translating English singing into Chinese singing and layering the result back onto the instruments.

Once all this work has been accomplished, if time permits, improvements could be made on the following fronts.

1. Refine the protobuf-based specifications with Dependent Types. This may as well entail using an ontology, like SUMO, to replace general types like Bytes by more semantic types like Audio.

2. Integrate computational and financial cost in the formal specification, as well as performance evaluations and their compositional laws.

3. Improve Idris program synthesis to be geared towards composition-based synthesis involving a large number of functions.

4. Create a dedictated front-end language, the AI-DSL per se, as opposed to using Idris. Such language could be especially suited to represent composition, specifications, and designed to accommodate the regular user, not just the power user.

5. Make progress towards replacing the Idris back-end by, or complementing with, Hyperon/MeTTa.

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] Alechina, N., Immerman, N.: Reachability logic: an efficient fragment of transitive closure logic. Logic Journal of the IGPL **8**(3), 325–337 (05 2000). https://doi.org/10.1093/jigpal/8.3.325, `https://doi.org/10.1093/jigpal/8.3.325`

[2] Annenkov, D., Nielsen, J.B., Spitters, B.: Concert: A smart contract certification framework in coq. In: Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs. p. 215–228. CPP 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3372885.3373829, `https://doi.org/10.1145/3372885.3373829`

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

[4] Atkey, R.: Syntax and semantics of quantitative type theory. Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (2018)

[5] 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

[6] Bernardo, B., Cauderlier, R., Hu, Z., Pesin, B., Tesson, J.: Mi-cho-coq, a framework for certifying tezos smart contracts. CoRR **abs/1909.08671** (2019), `http://arxiv.org/abs/1909.08671`

[7] Bernardo, B., Cauderlier, R., Pesin, B., Tesson, J.: Albert, an intermediate smart-contract language for the tezos blockchain. In: Bernhard, M., Bracciali, A., Camp, L.J., Matsuo, S., Maurushat, A., Rønne, P.B., Sala, M. (eds.) Financial Cryptography and Data Security. pp. 584–598. Springer International Publishing, Cham (2020)

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

[9] Brady, E.: Type-Driven Development with Idris. Manning (2017)

[10] Brady, E.C.: Idris 2: Quantitative type theory in practice. CoRR **abs/2104.00480** (2021), `https://arxiv.org/abs/2104.00480`

[11] Caires, L.: Behavioral and spatial observations in a logic for the π-calculus. In: Walukiewicz, I. (ed.) Foundations of Software Science and Computation Structures. pp. 72–89. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)

[12] Chen, X., Roşu, G.: Matching μ-logic: Foundation of k framework. In: Roggenbach, M., Sokolova, A. (eds.) 8th Conference on Algebra and Coalgebra in Computer Science, CALCO 2019. Leibniz International Proceedings in Informatics, LIPIcs, Schloss Dagstuhl-Leibniz-Zentrum fur Informatik GmbH, Dagstuhl Publishing (Nov 2019). https://doi.org/10.4230/LIPIcs.CALCO.2019.1, publisher Copyright: © Xiaohong Chen and Grigore Roşu.; 8th Conference on Algebra and Coalgebra in Computer Science, CALCO 2019 ; Conference date: 03-06-2019 Through 06-06-2019

[13] Duong, D., Grefenstette, J.: Sister: a symbolic interactionist simulation of trade and emergent roles. Journal of Artificial Societies and Social Simulation **8** (01 2004)

[14] Feng, Y., Martins, R., Wang, Y., Dillig, I., Reps, T.W.: Component-based synthesis for complex apis. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. p. 599–612. POPL '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3009837.3009851, `https://doi.org/10.1145/3009837.3009851`

[15] Flanagan, C.: Hybrid type checking. In: ACM-SIGACT Symposium on Principles of Programming Languages (2006)

[16] Frankle, J., Osera, P.M., Walker, D., Zdancewic, S.: Example-directed synthesis: A type-theoretic interpretation. SIGPLAN Not. **51**(1), 802–815 (jan 2016). https://doi.org/10.1145/2914770.2837629, `https://doi.org/10.1145/2914770.2837629`

[17] 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`

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

[19] 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)

[20] Goes, C.: Compiling quantitative type theory to michelson for compile-time verification and run-time efficiency in juvix. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Applications. pp. 146–160. Springer International Publishing, Cham (2020)

[21] Guo, Z., Cao, D., Tjong, D., Yang, J., Schlesinger, C., Polikarpova, N.: Type-directed program synthesis for restful apis. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. p. 122–136. PLDI 2022, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3519939.3523450, `https://doi.org/10.1145/3519939.3523450`

[22] Guo, Z., James, M., Justo, D., Zhou, J., Wang, Z., Jhala, R., Polikarpova, N.: Program synthesis by type-guided abstraction refinement. Proc. ACM Program. Lang. **4**(POPL) (dec 2019). https://doi.org/10.1145/3371080, `https://doi.org/10.1145/3371080`

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

[24] Heineman, G.T., Bessai, J., Düdder, B., Rehof, J.: A long and winding road towards modular synthesis. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques. pp. 303–317. Springer International Publishing, Cham (2016)

[25] Hildenbrandt, E., Saxena, M., Rodrigues, N., Zhu, X., Daian, P., Guth, D., Moore, B., Park, D., Zhang, Y., Stefanescu, A., Rosu, G.: Kevm: A complete formal semantics of the ethereum virtual machine. In: 2018 IEEE 31st Computer Security Foundations Symposium (CSF). pp. 204–217 (2018). https://doi.org/10.1109/CSF.2018.00022

[26] 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`

[27] IOHK: Plutus core and plutus tx user guide (2022), `https://plutus.readthedocs.io/en/latest/`

[28] James, M.B., Guo, Z., Wang, Z., Doshi, S., Peleg, H., Jhala, R., Polikarpova, N.: Digging for fold: Synthesis-aided api discovery for haskell. Proc. ACM Program. Lang. **4**(OOPSLA) (nov 2020). https://doi.org/10.1145/3428273, `https://doi.org/10.1145/3428273`

[29] Jones, T., Homer, M.: The practice of a compositional functional programming language. In: Ryu, S. (ed.) Programming Languages and Systems. pp. 166–177. Springer International Publishing, Cham (2018)

[30] Kevin, D., Mitchell, A., Ahmed, K., Andrew, M., Elaine, S.: A programmer's guide to ethereum and serpent (2015), http://mc2-umd.github.io/ethereumlab/docs/serpent_tutorial.pdf

[31] Knoth, T., Wang, D., Polikarpova, N., Hoffmann, J.: Resource-guided program synthesis. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 253–268. PLDI 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3314221.3314602, https://doi.org/10.1145/3314221.3314602

[32] Krijnen, J.O.G., Chakravarty, M.M.T., Keller, G., Swierstra, W.: Translation certification for smart contracts. CoRR **abs/2201.04919** (2022), https://arxiv.org/abs/2201.04919

[33] L.M Goodman: Tezos – a self-amending crypto-ledger white paper (2014), https://tezos.com/whitepaper.pdf

[34] Meredith, L., Radestock, M.: A reflective higher-order calculus. Electronic Notes in Theoretical Computer Science **141**(5), 49–67 (2005). https://doi.org/https://doi.org/10.1016/j.entcs.2005.05.016, https://www.sciencedirect.com/science/article/pii/S1571066105051893, proceedings of the Workshop on the Foundations of Interactive Computation (FInCo 2005)

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

[36] Neil, M.: Hoogle (2004), https://hoogle.haskell.org

[37] Norell, U.: Dependently Typed Programming in Agda, pp. 230–266. Springer Berlin Heidelberg, Berlin, Heidelberg (2009), https://doi.org/10.1007/978-3-642-04652-0_5

[38] Pace, G.J., Sánchez, C., Schneider, G.: Reliable smart contracts. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Applications. pp. 3–8. Springer International Publishing, Cham (2020)

[39] Pettersson, J., Edström, R.: Safer smart contracts through type-driven development. In: Master's thesis (2016)

[40] Polikarpova, N., Kuraj, I., Solar-Lezama, A.: Program synthesis from polymorphic refinement types. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 522–538. PLDI '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2908080.2908093, https://doi.org/10.1145/2908080.2908093

[41] 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)

[42] RChain Cooperative: Rchain whitepaper (2021), `https://rchain.coop/whitepaper.html`

[43] RChain Cooperative: Rholang (2022), `https://rholang.github.io/docs/rholang/`

[44] Rondon, P.M., Kawaguchi, M., Jhala, R.: Liquid types. In: ACM-SIGPLAN Symposium on Programming Language Design and Implementation (2008)

[45] 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`

[46] Sergey, I., Kumar, A., Hobor, A.: Scilla: a smart contract intermediate-level language. CoRR **abs/1801.00687** (2018), `http://arxiv.org/abs/1801.00687`

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

[48] Swamy, N., Chen, J., Livshits, B.: Verifying higher-order programs with the dijkstra monad. In: ACM Programming Language Design and Implementation (PLDI) 2013. ACM (June 2013), `https://www.microsoft.com/en-us/research/publication/verifying-higher-order-programs-with-the-dijkstra-monad/`

[49] The Zilliqa Team: The Zilliqa technical paper (2017), `https://docs.zilliqa.com/whitepaper.pdf`

[50] Vazou, N., Seidel, E.L., Jhala, R., Vytiniotis, D., Peyton-Jones, S.: Refinement types for haskell. SIGPLAN Not. **49**(9), 269–282 (aug 2014). https://doi.org/10.1145/2692915.2628161, `https://doi.org/10.1145/2692915.2628161`

[51] Warrell, J., Potapov, A., Vandervorst, A., Goertzel, B.: A meta-probabilistic-programming language for bisimulation of probabilistic and non-well-founded type systems. AGI (2022)

[52] Warrell, J.H.: A probabilistic dependent type system based on non-deterministic beta reduction. ArXiv **abs/1602.06420** (2016)

[53] Williams, C., Stay, M.: Native type theory. In: Kishida, K. (ed.) Proceedings of the Fourth International Conference on Applied Category Theory, ACT 2021, Cambridge, United Kingdom, 12-16th July 2021. EPTCS,

vol. 372, pp. 116–132 (2021). https://doi.org/10.4204/EPTCS.372.9, https://doi.org/10.4204/EPTCS.372.9

[54] Zen Protocol Development: An introduction to the zen protocol (2017), https://www.zenprotocol.com/en/white_paper.pdf

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