AI-DSL Technical Report (February to May 2021)

Nil Geisweiller, Kabir Veitas, Eman Shemsu Asfaw, Samuel Roberti

June 17, 2021

Abstract

Based on [12].

Contents

Chapter 1

Nil's work

Work done:

1. Implement `RealizedFunction` as described in [12].

2. Implement a network of trivially simple AI services implemented in Idris2, and use Idris compiler to type check if they can properly connect to each other.

3. Implement a Registry prototype, as a proof-of-concept for querying AI services based on their dependently typed specifications.

1.1 Realized Function

1.1.1 Description

The `RealizedFunction` data structure, as introduced in [12], is a wrapper around a regular function to integrate aspects of its specifications pertaining to its execution on real physical substrates as opposed to just its algorithmic properties. For instance it contains descriptions of costs (financial, computational, etc) and performances (quality, etc) captured in the `RealizedAttributes` data structure, as introduced in [12] as well.

For that iteration we have implemented a simple version of `RealizedFunction` and `RealizedAttributes` in Idris2 [6]. The `RealizedAttributes` data structure contains

- `Costs`: as a triple of three constants, `financial`, `temporal` and `computational`,

- `Quality`: as a single `quality` value.

as well as an example of compositional law, `add_costs_min_quality`, where costs are additive and quality is infimumitive. Below is a small snippet of that code to give an idea of how it looks like

nil
is there a word for that?

2

```
record RealizedAttributes where
  constructor MkRealizedAttributes
  costs : Costs
  quality : Quality

add_costs_min_quality : RealizedAttributes ->
                        RealizedAttributes ->
                        RealizedAttributes
add_costs_min_quality f_attrs g_attrs = fg_attrs where
  fg_attrs : RealizedAttributes
  fg_attrs = MkRealizedAttributes (add_costs f_attrs.costs g_attrs.costs)
                                  (min f_attrs.quality g_attrs.quality)
```

The full implementation can be found in RealizedAttributes.idr, under the experimental/realized-function/ folder of the ai-dsl repository [4].

Then we have implemented `RealizedFunction` that essentially attaches a `RealizedAttributes` instance to a function. In addition we have implemented a composition (as in function composition) operating on `RealizedFunction` instead of regular function, making use of that compositional law above. Likewise below is a snippet of that code

```
data RealizedFunction : (t : Type) -> (attrs : RealizedAttributes) -> Type where
  MkRealizedFunction : (f : t) -> (attrs : RealizedAttributes) ->
                       RealizedFunction t attrs

compose : {a : Type} -> {b : Type} -> {c : Type} ->
          (RealizedFunction (b -> c) g_attrs) ->
          (RealizedFunction (a -> b) f_attrs) ->
          (RealizedFunction (a -> c) (add_costs_min_quality f_attrs g_attrs)) ->
compose (MkRealizedFunction g g_attrs) (MkRealizedFunction f f_attrs) =
  MkRealizedFunction (g . f) (add_costs_min_quality f_attrs g_attrs)
```

The full implementation can be found in RealizedFunction.idr under the same folder.

1.1.2   Objectives and achievements

The objectives of this work was to see if Idris2 was able to type check that the realized attributes of composed realized functions followed the defined compositional law. We have found that Idris2 is not only able to do that, but to our surprise much faster that Idris1 (instantaneous instead of seconds to minutes), by bypassing induction on numbers and using efficient function-driven rewriting on the realized attributes instead. That experiment can be found in RealizedFunction-test.idr, under the experimental/realized-function/ folder of the ai-dsl repository [4].

1.1.3   Future work

Experimenting with constants as realized attributes was the first step in our investigation. The subsequent steps will be to replace constants by functions, probability distributions and other sophisticated ways to represent costs and quality.

1.2   Network of Idris AI services

1.2.1   Description

In this work we have implemented a small network of trivially simple AI services, with the objective of testing if the Idris compiler could be used to type check the validity of their connections. Three primary services were implemented

1. `incrementer`: increment an integer by 1

2. `twicer`: multiply an integer by 2

3. `halfer`: divide an integer by 2

as well as composite services based on these primary services, such as

- `incrementer . halfer . twicer`

with the objective of testing that such compositions were properly typed. The networking part was implemented based on the SingularityNET example service [9] mentioned in the SingularityNET tutorial [11]. The specifics of that implementation are of little importance for that report and thus are largely ignored. The point was to try to be as close as possible to real networking conditions. For the part that matters to us we may mention that communications between AI services are handled by gRPC [5], which has some level of type checking by insuring that the data being exchanged fulfill some type structures (list of integers, union type of string and bool, etc) specified in Protocol Buffers [7]. Thus one may see the usage of Idris in that context as adding an enhanced refined verification layer on top of gRPC making use of the expressive power of dependent types.

1.2.2   Objectives and achievements

As mentioned above the objectives of such an experiment was to see how the Idris compiler can be used to type check combinations of AI services. It was initially envisioned to make use of dependent types by specifying that the `twicer` service outputs an even integer, as opposed to any integer, and that the `halfer` service only accepts an even integer as well. The idea was to prohibit certain combinations such as

- `halfer . incrementer . twicer`

Since the output of `incrementer . twicer` is provably odd, `halfer` should refuse it and such combination should be rejected. This objective was not reached in this experiment, but similar objectives were reached other experiments, see Section 1.3. The other objective was to type check that the compositions have realized attributes corresponding to the compositional law implemented in Section 1.1, which was fully achieved in this experiment. For instance by changing either the types, costs or quality of the following composition

```
-- Realized (twicer . incrementer).
rlz_compo1_attrs : RealizedAttributes
rlz_compo1_attrs = MkRealizedAttributes (MkCosts 300 30 3) 0.9
-- The following does not work because 301 /= 200+100
-- rlz_compo1_attrs = MkRealizedAttributes (MkCosts 301 30 3) 0.9
rlz_compo1 : RealizedFunction (Int -> Int) Compo1.rlz_compo1_attrs
rlz_compo1 = compose rlz_twicer rlz_incrementer
```

defined in experimental/simple-idris-services/service/Compo1.idr, the corresponding service would raise a type checking error at start up. More details on the experiment and how to run it can be found in the `README.md` under the `experimental/simple-idris-services/service/` folder of the ai-dsl repository [4].

1.2.3   Future work

Such experiment was good to explore how Idris can be integrated to a network of services. What we need to do next is experiment with actual AIs algorithms, ideally making full use of dependent types in their specifications. Such endeavor was actually attempted by using an existing set of cooperating AI services, but it was eventually concluded to be too ambitious for that iteration and was postponed for the next.

Obviously we want to be able to reuse existing AI services and write their enhanced specifications on top of them, as opposed to writing both specification and code in Idris/AI-DSL. To that end it was noted that having a Protobuf to/from Idris/AI-DSL converter would be useful, so that a developer can start from an existing AI service, specified in Protobuf, and enriched it with dependent types in Idris/AI-DSL. The other way around could be useful as well to enable a developer to implement AI services entirely in Idris/AI-DSL and expose their Protobuf specification to the network. To that end having an implementation of gRPC for Idris/AI-DSL could be handy as well.

1.3   AI-DSL Registry

1.3.1   Description

One important goal of the AI-DSL is to have a system that can perform autonomous matching and composition of AI services, so that provided the specification of an AI, it should suffice to find it, complete it or even entirely build

it from scratch. We have implemented a proof-of-concept *registry* to start experimenting with such functionalities.

So far we have two versions in the ai-dsl repository, one without dependent types support, under experimental/registry/, and a more recent one with dependent type support that can be found under experimental/registry-dtl/. We will focus our attention on the latter which is far more interesting.

The AI-DSL registry (reminiscent of the SingularityNET registry [**?**]) is itself an AI service with the following functions

1. `retrieve`: find AI services on the network fulfilling a given specification.

2. `compose`: construct composite services fulfilling that specification. Useful when no such AI services can be found.

The experiment contains the same `incrementer`, `twicer` and `halfer` services described in Section 1.2 with the important distinction that their specifications now utilize dependent types. For instance the type signature of `twicer` becomes

```
twicer : Integer -> EvenInteger
```

instead of

```
twicer : Integer -> Integer
```

where `EvenInteger` is actually a shorthand for the following dependent type

```
EvenInteger : Type
EvenInteger = (n : WFInt ** Parity n 2)
```

that is a *dependent pair* composed of a *well founded integer* of type `WFInt` and a dependent data structure, `Parity` containing a proof that the first element of the pair, `n`, is even. More details on that can be found in Section.

For now our prototype of AI-DSL registry implements the `retreive` function, which, given an Idris type signature, searches through a database of AI services and returns one fulfilling that type. In that experiment the database of AI services is composed of `incrementer`, `twicer`, `halfer`, the `registry` itself and `compo`, a composite service using previously listed services.

One can query each service via gRPC. For instance querying the `retreive` function of the `registry` service with the following input

```
String -> (String, String)
```

outputs

```
Registry.retreive
```

which is itself. Likewise one can query

```
Integer -> Integer
```

nil

Add ref to
Sam's work

6

which outputs

`Incrementer.incrementer`

corresponding to the `Incrementer` service with the `incrementer` function. Next one can provide a query involving dependent types, such as

`Integer -> EvenInteger`

outputting

`Twicer.twicer`

Or equivalently provide the unwrapped dependent type signature

`Integer -> (n : WFInt ** Parity n (Nat 2))`

retrieving the correct service again

`Twicer.twicer`

At the heart of it is Idris. Behind the scene the registry communicates the type signature to the Idris REPL and requests, via the :search meta function, all loaded functions matching the type signature. Then the registry just returns the first match.

Secondly, we can now write composite services with missing parts. The `compo` service illustrates this. This service essentially implements the following composition

`incrementer . halfer . (Registry.retrieve ?type)`

Thus upon execution queries the registry to fill the hole with the correct, according to its specification, service.

More details about this, including steps to reproduce it all, can be found in the `README.md` under the `experimental/simple-idris-services/service/` folder of the ai-dsl repository [4].

1.3.2   Objectives and achievements

As shown above we were able to implement a proof-of-concept of an AI-DSL registry. Only the `retrieve` function was implemented. The `compose` function still remains to be implemented, although the `compo` service is somewhat halfway there, with limitations, for instance the missing type, `?type`, was hardwired in the code, `Integer -> EvenInteger`. It should be noted however that Idris is in principle capable of inferring such information but more work is needed to more fully explore that functionality.

Of course it is a very simple example, in fact the simplest we could come up with, but we believe serves as a proof-of-concept, and demonstrates that AI services matching, using dependent types as formal specification language, is possible.

7

1.3.3   Future work

There a lot of possible future improvements for this work, in no particular order

- Use structured data structures to represent type signatures instead of String.

- Return a list of services instead of the first one.

- Implement `compose` for autonomous composition.

- Use real AI services instead of trivially simple ones.

Also, as of right now, the registry was implemented in Python[1], querying Idris when necessary. However it is likely that this should be better suited to Idris itself. Which leads us to an interesting possibility, maybe the registry, and in fact most (perhaps all) components and functions of the AI-DSL could or should be implemented in the AI-DSL itself.

---

[1] because the SingularityNET example it is derived from is written in Python

Chapter 2

AI-DSL Ontology (Kabir's work)

2.1 Description

2.1.1 Design requirements

At the beginning of the current iteration of the AI-DSL project we had a round of discussions about the high level functional and design requirements for AI-DSL and its role in SingularityNET platfrom and ecosystem. The discussions were based on [12, 10] and are available online in their original form. Here is the summary of the preliminary design requirements informed by those discussions:

- AI-DSL is a language that allows AI agents/services running on SinglarityNET platfrom to declare their capabilities and needs for data to other AI agents in a rich and versatile machine readible form; This will enable different AI agents to search, find data sources and other AI services whithout human interaction;

- AI-DSL ontology defines data and service (task) types to be used by AI-DSL. Requirements for the ontology are shaped by the scope and specification of the AI-DSL itself;

High level requirements for AI-DSL are:

Extendability The ontology of data types and AI task types should be extendable in the sense that individual service providers / users should be able to create new types and tasks and make them available to the network. AI-DSL should be able to ingest these new types / tasks and immediately be able to do the type-checking job. In other words, AI-DSL ontology of types / tasks should be able to evolve. At the same time, extended ontologies should relate to existing basic AI-DSL ontology in a clear way,

allowing AI agents to perform reasoning across the whole space of available ontologies (which, at lower levels, may be globally inconsistent). In order to ensure interoperability of lower level ontologies, AI-DSL ontology will define small kernel / vocabulary of globally accessible grounded types, which will be built-in into the platform at the deep level. Changing this kernel will most probably require some form of voting / global consensus on a platfrom level.

Therefore, it seems best to define AI-DSL Ontology and the mechanism of using it on two levels:

- *The globally accessible vocabulary/root ontology of grounded types.* This vocabulary can be seen as immutable (in short and medium term) kernel. It should be extendible in the long term, but the mechanisms of changing and extending it will be quite complex, most probably involving theorectical considerations and/or a strict procedures of reaching global consensus within the whole platform (a sort of voting);

- *A decentralized ontology of types and tasks* which each are based (i.e. type-dependent) on the root ontology/vocabulary, but can be extended in a decentralized manner – in the sense that each agent in the platfrom will be able to define, use and share derived types and task definitions at its own discretion without the need of global consensus.

Competing versions and consensus. We want both consistency (for enabling deterministic type checking – as much as it is possible) and flexibility (for enabling adaptation and support for innovation). This will be achieved by enforcing different restrictions for competing versions and consensus reaching on the two levels of ontology descrbed above:

- The globally accessible vocabulary / root ontology of grounded types will not allow for competing versions. In a sense, this level will be the true ontology, representable by a one and unique root / upper-level ontology of the network which users will not be able to modify directly;

- All other types and task definitions within the platform will be required to be derived from the root ontology (if they will want to be used for interaction with other agents); However, the platform whould not restrict the number of competing versions or define a global consensus of types and task descriptions on this level.

- Furthermore, the ontology and the AI-DSL logic should allow for some variant of 'soft matching' which would allow to find the type / service that does not satisfy all requirements exactly, but comes as closely as available in the platform.

- At the lowest level of describing each instance of AI service or data source on the platfrom, AI-DSL shall allow maximum extentibility in so that AI service providers and data providers will be able to describe and declare their services in the most flexible and unconstrained manner, facilitating competition and cooperation between them.

Code-level / service-level APIs. It is important to ensure that the ontology is readable / writable by different components of the SingularityNET platform, at least between AI-DSL engine / data structures and each AI service separately. This is needed because some of the required descriptors of AI services will have to be dynamically calculated at the time of calling a service and will depend on the immediate context (e.g. price of service, a machine on which it is running, possibly reputation score, etc.). It is not clear at this point how much of this functionality will be possible (and practical) to implement on available type-dependent, ontology languages or even if it is possible to use single language. Even it if is possible to implement all AI-DSL purely on the current type-dependent language choice Idris, it will have to interface with the world, deal with indeterministic input from network and mutable states – operations that may fail in run-time no matter how careful type checking is done during compile time [13].

Defining and maintaining code-level and service-level APIs will first of all enable interfacing SingularityNET agents to AI-DSL and therefore between themselves.

Key AI Agents properties We can distinguish two somewhat distinct (but yet interacting) levels of AI-DSL Ontology AI service description level and data description level. It seems that it may be best to start building the ontology from the service level, because data description language is even more open-ended than AI description language, which is already open enough. Initially, we may want to include into the description of each AI service at least these properties:

- Input and output data structures and types
- Financial cost of service
- Time of computation
- Computational resource cost
- Quality of results

Most probably it is possible to express and reason about this data with Idris. It is quite clear however, that in order to enable interaction with and between SingularityNET agents (and NuNet adapters) all above properties have to be made accessible outside Idris and therefore supported by the code-level / service-level APIs and the SingularityNET platfrom in general.

2.1.2 Domain model considerations

In order to attend to all high level design requirements. all levels of AI-DSL Ontology should be developed simulatneously, so that we could make sure that the work is aligned with the function and role of AI-DSL within SingularityNET platform and ecosystem. We therefore use the "AI/computer-scientific" perspective to ontology and ontology building – emphasizing *what an ontologoy is for* – rather than the "philosophical perspecive" dealing with *the study of what there is in terms of basic categories* [14, 15]. Therefore we first propose the mechanism of how different levels (upper, domain and the leaf- (or service)) of AI-DSL ontology will relate for facilitating interactions between AI services on the platfrom.

Note, that design principles of such mechanism relate to the question how abstract and consistent should relate to concrete and possibly inconsistent – something that may need a deeper conceptual understanding than is attempted during the project and presented here. We proceed in most practical manner for proposing the AI-DSL ontology prototype, being aware that it may need to (and possibly should) be subjected to more conceptual treatment in the future.

For a concrete domain model of AI-DSL ontologoy prototype we use the *Fake News Warning*[1] application being developed by NuNet – a currently incubated spinoff of SingularityNET[2].

NuNet is the platfrom enabling dynamic deployment and up/down-scaling of SingularityNET AI Services on decentralized hardware devices of potentially any type. Importantly for the AI-DSL project, service discovery on NuNet is designed in a way that enables dynamic construction of application-specific service meshes from several SingularityNET AI services[19]. In order for the service mesh to be deployed, NuNet needs only a specification of program graph of the application. Note, that conceptually, construction of an application from several independent containers is almost equivalent to functionality explained in section 1.3 on AI-DSL Registry, namely performance of matching and composition of AI services. This is the main reason why we chose *Fake News Warning* application as a domain model for early development efforts of AI-DSL. However, we use this domain model solely for the appication-independent design of AI-DSL and attend to its application specific aspects only as much as it informs the project.

The idea of dynamic service discovery is to enable application developers to construct working applications (or at least their backends) by simply passing a declarative definition of program graph to the special platform component ("network orchestrator") – which then searches for appropriate SingularityNET AI containers and connects them in to a single workflow (or workflows). Suppose, that the backend of *Fake News Warning* app consists of three SingularityNET AI containers *news_score*, *uclnlp* and *binary-classification*:

---

[1] https://gitlab.com/nunet/fake-news-detection
[2] https://nunet.io

Leaf item	Description	Input	Output	Source
binary-classification	A pre-trained binary classification model	English text of any length	1 – the text is categorized as fake; 0 – text is categorized as not-fake	©NuNet 2021
uclnlp	Forked and adapted component of stance detection algorithm (FNC third place winner)	Article title and text	Probabilities of the title *agreeing*, *disagreeing*, *discussing* or being *unrelated* to the text	©UCL Machine Reading 2017; ©NuNet 2021
news-score	Calls dependent services, calculates overall result and sends them to the frontend	URL of the content to be checked	Probability that the content in the URL is fake	©NuNet 2021

Table 2.1: Description of each component of `Fake News Warning` application.

Each component of application's backend is a SingularityNET AI Service registered on the platfrom. Note, that as SingularityNET AI services are defined through their specification and their metadata[8]. The main purpose of the AI-DSL Ontology is to be able to describe SNet AI Services in a manner that would allow them to search and match each other on the platfrom and compose into complex workflows – similarly to what is described in Section 1.2. Here is a simple represenation of the program graph of *Fake News Warning* app:

```
1        "dag": {
2          "news-score" : ["uclnlp","binary-classification"]
3          }
```

Figure 2.1: A directed acyclic graph (DAT) of *Fake News Warning* app prototype[3]

The schematic representation of the *Fake News Warning* app deployed as a result of processing the DAG is depicted below. The addition of NuNet platfrom to SingularityNET service discovery is that each service may be deployed on different hardware environments, sourced by NuNet. When the application backend is deployed, it can be accessed from the GUI interface, which in this case is a Brave browser extension.
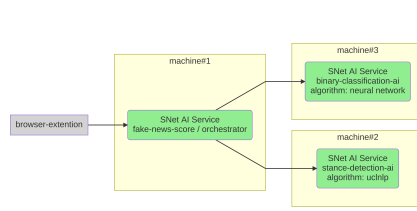
(a) Schema of dependencies between backend components of the application (SingularityNET AI services potentially running on different machines).

`../../../ontology/images/fake_news_detector_browse`

(b) Brave browser extension which calls the backend of *Fake News Warning* application on each invocation on new content displayed in browser tab.

We will use this application design principles as the domain model for the first design of AI-DSL Ontology and the prototype.

2.1.3 Ontology language and upper level ontology

After discussing several choices of ontology languages and reusing existing ontologies for designing AI-DSL ontology[3], we have opted to use SUO-KIF as an ontology language [21] and SUMO as an upper-level ontology [18]. The main motivation for this choice were the versatility of KIF/SUO-KIF (Knowledge Interchange Format) language, which essentially allows to express first order logic statements in a simple text format in terms of lisp-like syntax. Due to that, KIF can be easily converted to other formats[16]. Also, a conversion to Atomese – the OpenCog's language also employoing a list-like syntax – has been successfully attempted in the past[4] SUMO and the related ontology design tools [20] provide a convenient way for starting to design AI-DSL Ontology levels and their relations.

kabir: Finished here: 2021-05-27T01:37 CET

2.2 Objectives and achievements

2.3 Relations between levels

1. Upper level SUMO ontology (Merge.kif);

2. Middle level SUMO ontology (Mid-level-ontology.kif);

3. Distributed computing hardware domain ontology in SUO-KIF (QoSontology.kif);

kabir: using ontology for agent communication in decentralized computing systems, based on [22]

---

[3]See Reusing Existing Ontologies discussion on AI-DSL Github Repo[4]

[4]As informed by Ben Goertzel during one of the AI-DSL project meetings.
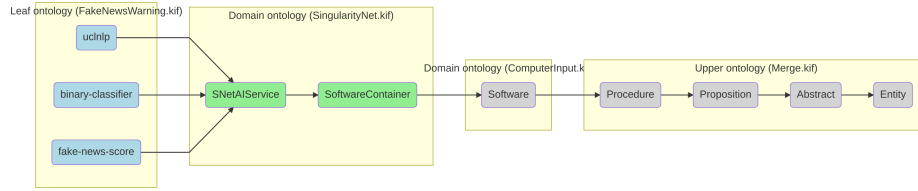
(a) Class dependencies of the SNetAIService subclass within the ontology.

(b) Class dependencies of the SNetAppBackend subclass (workflow definition) within the ontology.

Figure 2.3: Class dependences of AI-DSL Ontology (SingularityNet.kif)

2.3.1 Leaf ontology (fake-news-detection workflow definition)

The prototype will be the fake-news-detector leaf ontology based on the above listed upper and middle ontologies (SUMO) and domain ontologies of computer hardware and software.

```
2   (subclass SNetAIService SoftwareContainer)
3   (documentation SNetAIService EnglishLanguage
4   "Software package exposed via SNetPlatfrom and conforming to the special
    ↪   packaging rules")
5
6   (subclass SNetAIServiceIO Descriptor)
7
8   (instance hasInput BinaryPredicate)
9   (domain hasInput 1 SNetAIService)
10  (domain hasInput 2 SNetAIServiceIO)
11
12  (instance hasOutput BinaryPredicate)
```

Figure 2.4: Contents of SingularityNet.kif

nil: Very cool, I think we want to include the (subclass Boolean DataType) lines, etc, as well, as they start touching the what matters

kabir: Yes, we will. The kif files are quite large and I will have to think how to include the most important parts to test; but of course i will provide links to them here.

2.4   Future work

- In the long term, it may be ideal to develop a converter for converting it to KIF, since OWL may be representable in KIF [17] using https://github.com/owlcs/owlapi; For the purpose of the ontology prototype, we will manually select parts of the this ontology in order to build the prototype and write them in SUO-KIF format;

- Combining ontology with Idris

kabir

It would be good to have a section explaining ideas about that, but I cannot do this alone, so probably the best is to reserve it for the end of the monht, when all the other aspects of AI-DSL project (including Idris) are explained.

Chapter 3

Eman's work

Work done:

1. Implementing a simple fake new classifier task in Haskell and Idris1.

2. AI services aggregation with Idris2.

3.1 Fake News Classifier

3.1.1 Description

The idea is to develop an NLP model using word embedding and train it with fake news vs real news dataset [2] . I implemented a word embedding model based on an article shared by Eligijus Bujokas on medium [1]. Hence, given a dataset or a csv file containing these lines for instance.

1. The future king is the prince

2. Daughter is the princess

3. Son is the prince

4. Only a man can be a king

The goal is to make the computer understand the relationship between words in a sentence. Therefore, we want to create data points for all words in a sentence such that we have :- (The, future)(The, king) (future, king) (future,The)..etc

What we did here is that we took each word and created pairs between 2 words adjacent to it from its left and it's right. For example the word "King" in sentence 1 has 2 words on its left which are "future" and "The" and 2 words on its right "is" and "the". Therefore we have. (king,future) (king,The) (king,is) (king,the) as our data points. "King" here is our focus word and the points created are called context. For this example we took the context as only 2 words but we can make it 4,5,6... etc as we wish. This can be elegantly implemented in Haskel like so.

```
makeWordPairs :: [Str] -> [(Str,Str)]
makeWordPairs [] = []
makeWordPairs (x:y:z:h:xs) = (x,y):(x,z):(y,x):(y,z):(y,h):(z,x):(z,y):(z,h):(h,x):(h,y):(h
makeWordPairs (x:y:z:xs) = (x,y):(x,z):(y,x):(y,z):(z,x):(z,y):makeWordPairs xs
makeWordPairs (x:y:xs) =  (x,y):makeWordPairs xs
```

The corresponding Idris1 code is as follows

```
makeWordPairs : List String -> List (String, String)
makeWordPairs [] = []
makeWordPairs (x::y::z::h::xs) = (x,y)::(x,z)::(y,z)::(y,h)::(z,x)::(z,y)::(z,h)::(h,x)::(h
makeWordPairs (x::y::z::xs) = (x,y)::(x,z)::(y,z)::(z,x)::(z,y)::makeWordPairs xs
makeWordPairs (x::y::xs) = (x,y)::makeWordPairs xs
```

Our model's input will be the x values of our data points where as the targets
are the ys e.g in (king,future) (king, The)
    X Y King future King The . . .
    Before we make word pairs ,however, we perform som common dataset pre-
processing such as removing stop words and creating dictionaries

```
removeStopWords :: Str -> [Str]
removeStopWords line = filter (not . ((flip elem ) stopwords )) (words line)
```

Corresponding Idris1 code

```
export
removeStopWords : String -> List String
removeStopWords line = filter (not . ((flip elem) stopwords)) (words line)

createDictionary vocabs = zip  (sort . nub $ vocabs) [1..]
```

Corresponding Idris1 code

```
createDictionary xss = func 0 tmp
    where
        tmp = sort . nub $ xss
        func : Nat -> List String -> List (String, Nat)
        func _ [] = []
        func Z (x::xs) = (x, S Z) :: func (S (S Z)) xs
        func (S k) (x::xs) = (x, S k) :: func (S (S k) ) xs
```

The next step is passing the input to our model. To do so we need to first
one-hot encode the inputs and the targets.

```
oneHotEncoder :: (Num q) => [Int] -> [q] -> [[q]]
oneHotEncoder xs zs = foldr (\x acc -> insertAt zs 1 x: acc) [] xs
```

Corresponding Idris1 code

```
oneHotEncoder : List Nat -> List Double -> List (List Double)
oneHotEncoder ws zs = foldr(\x, acc => (insertAt zs 1.0 x)::acc) [] ws
```

Finally, we pass it to our model which has one Deep Neural Network of (n,n) Shape. We apply softmax activation to our output and compute cost using cross entropy loss function. model here returns newly learned weights.

```
model :: (Floating j) => [[j]] -> [[j]] -> [[j]] -> j -> j -> j -> j -> Int -> [[j]]
model [] _ _ _ _ _ _ _ = []
model _ [] _ _ _ _ _ _ = []
model _ _ [] _ _ _ _ _ = []
model [] [] [] _ _ _ _ _= []
model weights inputs targets alpha beta1 beta2 epsilon iterations
    |iterations == 0 = weights
    |otherwise = do
      let predicted = fullNNLayer inputs weights
      let full_cost = loss_func grad_CrossEntropy  predicted targets
      let new_weights = optimizer adam_Optimizer weights full_cost alpha 0.0 0.0 beta1 beta2
      model new_weights inputs targets alpha beta1 beta2 epsilon (iterations - 1)
```

Corresponding Idris1 code

```
export
model : List (List Double) -> List (List Double) -> List (List Double) -> Double -> Double -
model [] _ _ _ _ _ _ _ = []
model _ [] _ _ _ _ _ _ = []
model _ _ [] _ _ _ _ _ = []
model weights inputs targets alpha beta1 beta2 epsilon iterations = case iterations == 0 of
    True => weights
    False => do
      let predicted = fullNNLayer inputs weights
      let full_cost = loss_func grad_CrossEntropy predicted targets
      let new_weights = optimizer adam_Optimizer weights full_cost alpha 0.0 0.0 beta1 beta2
      model new_weights inputs targets alpha beta1 beta2 epsilon (iterations - 1)
```

3.1.2 Objectives and Achievements

The objective of the work was to explore Idris better and get familiar with it. Since Idris adopts Haskel's syntax I thought it would be better I got familiar with Haskel first. I decided to study it by reading the learn you Haskel for good by Miran Lipovača which is the best open source well written resource for Haskel. Writing the fake News classifier with Haskel was just my attempt of understanding Haskel better i.e is not related to the task. Both the Idris and Haskel model accepts a list of list of Strings which represents a dataset and return weights predicted by the models. The function that tests the model's prediction over a test set is not implemented yet but can be added in the future.

3.1.3 Future work

it's worth noting that the above models return the learned weights and not the accuracy of prediction or losses like in classical machine learning tasks.

19

Therefore, depending on the priority given to it, we can add these functionalities for the work to be fully complete.

3.2   AI services aggregation with Idris2

The idea is to showcase how Idris can facilitate AI services co-operations of tasks. For the purposes of demonstration, I developed an algorithm that can choose the best performing cost function for my fake news classifier model I discussed in the earlier section.

Let's call the algorithm that finds the best cost function the reducer and the fake news classifier as the classifier. The reducer ideally does the following:

Accepts a model and runs a comparison test on it over a selected list of cost functions . Run iteratively with say 50 iterations each time running the classifier with the cost functions and recording its performance. Please note that the overall training performance is costly hence the number of iterations of the reducer must be lowered.

Compare the losses and count 1 for the cost function with the lowest loss Repeat the process until you reach the last 10 iterations. For the last 10 we want to add the counts so far plus a term called accumulate and a coefficient of 0.4. We do so for each score gained at the last 10 iterations so that we put weights to the scores gained toward the end of the iteration as we assume that at this point the classifier must have learnt something. The points added on top of the original score are bonuses and they increase with a margin of accumulate + 0.4.

For example let's say we have two costs Cross entropy loss (CE) and Mean squared error (MST)

Let's say that at the end of the 39th iteration of this reducer function CE has scored 24 points and MST 15 points (points are gained each time the cost gives lower results than the other cost functions)

If MST scored a point at the 40th iteration we don't just add 1 to total 16 rather we add the bonus 1+ 0.4 so its 16.04. 1 here is the initial accumulate value. Whoever scores a point in the 41th Iteration is rewarded with accumulate + 0.4 . So for the next iteration we have 1.4 + 0.4 which gives 1.8 as our new accumulate value. The addition continues until the last iteration. This can give us the scores putting emphasis on the last iterations as it is assumed must have learnt a lot more compared to the initial iterations. The code in Idris for this algorithm is not yet implemented but I thought it's important to describe it since the updated version of the fake news classifier assumes there is such an algorithm that runs on it. The reducer algorithm is inspired from the concept of exponential weighted moving average and weight decay used in classical Machine learning.

As for the fake news classifier (classifier) model, I have converted the earlier version from Idris1 to Idris2. In addition, I've also changed some codes to take advantage of dependent types.

3.2.1 Future works

I intend to explore Idris better to write the code for the reducer algorithm I described above. This might reveal very interesting properties in both the language as well as the interaction between the reducer and classifier. Writing a reducer code tailored specifically to the classifier, however, is relatively straightforward and I understand it is not this task's requirement. From my experience, I can say that it requires a deeper understanding and familiarity with the language that I currently do not possess. Nonetheless, this is definitely something I would be excited to do in the future.

Chapter 4

Sam's work

Bibliography

[1] Creating word embedding: Coding the word2vec algorithm in python using deep learning

[2] Fake and real news dataset

[3] NuNet, Program Graph of fake News Warning Application

[4] AI-DSL, AI-DSL GitHub Repository (2021), `https://github.com/singnet/ai-dsl/`

[5] gRPC, gRPC Homepage (2021), `https://grpc.io/`

[6] Idris, Idris Homepage (2021), `https://www.idris-lang.org/`

[7] Protocol Buffers, Protocol Buffers Homepage (2021), `https://developers.google.com/protocol-buffers/`

[8] SingularityNET Developer Portal, Service Setup Web Page (2021), `https://dev.singularitynet.io/docs/ai-developers/service-setup/`

[9] SingularityNET example service, example-service GitHub Repository (2021), `https://github.com/singnet/example-service`

[10] SingularityNET Foundation, PhaseTwo Information Memorandum (Feb 2021), `https://rebrand.ly/SNPhase2`

[11] SingularityNET Tutorial, SingularityNET Tutorial Webpage (2021), `https://dev.singularitynet.io/tutorials/publish`

[12] Ben Goertzel, N.G.: Ai-dsl: Toward a general-purpose description language for ai agents, `https://blog.singularitynet.io/ai-dsl-toward-a-general-purpose-description-language-for-ai-agents-21459f691b9e`

[13] Brady, E.: Resource-Dependent Algebraic Effects. In: Hage, J., McCarthy, J. (eds.) Trends in Functional Programming. pp. 18–33. Springer International Publishing, Cham (2015)

[14] Gruber, T.R.: A translation approach to portable ontology specifications. Knowledge Acquisition **5**(2), 199–220 (1993). https://doi.org/https://doi.org/10.1006/knac.1993.1008, `https://www.sciencedirect.com/science/article/pii/S1042814383710083`

[15] Hofweber, T.: Logic and Ontology. In: Zalta, E.N. (ed.) The Stanford Encyclopedia of Philosophy. Metaphysics Research Lab, Stanford University, Spring 2021 edn. (2021)

[16] Kalibatiene, D., Vasilecas, O.: Survey on Ontology Languages. In: Grabis, J., Kirikova, M. (eds.) Perspectives in Business Informatics Research. pp. 124–141. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)

[17] Martin, P.: Translations between RDF+OWL, N3, KIF, UML, FL, FCG and FE, `http://www.webkb.org/doc/model/comparisons.html`

[18] Niles, I., Pease, A.: Towards a standard upper ontology. In: Proceedings of the International Conference on Formal Ontology in Information Systems - Volume 2001. p. 2–9. FOIS '01, Association for Computing Machinery, New York, NY, USA (2001). https://doi.org/10.1145/505168.505170, `https://doi.org/10.1145/505168.505170`

[19] NuNet: NuNet architecture and service discovery principles (for AI-DSL) (May 2021), `https://www.youtube.com/watch?v=GKH9C8pb3yw`

[20] Pease, A.: The Sigma Ontology Development Environment (2001), `http://www.ceur-ws.org/Vol-71/Pease.pdf`

[21] Pease, A.: Standard Upper Ontology Knowledge Interchange Format (Jun 2009), `http://ontolog.cim3.net/file/resource/reference/SIGMA-kee/suo-kif.pdf`

[22] Yves Forkl, M.H.: Mastering agent communication in EMBASSI on the basis of a formal ontology (2002), `https://core.ac.uk/display/57029957`