AI-DSL Technical Report (February to May 2021)

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

July 5, 2021

Abstract

This document is a technical report of a work done between February and May 2021, based on the *AI-DSL Proposal* [22] published as a blogpost on the SingularityNET website in December 2020. It is the first iteration of a larger endeavor to create a system that enables autonomous interoperability between AI services over the network, more specifically over the SingularityNET-on-Cardano network. It presents in detail what has been accomplished so far as well as future plans for the continuation of that endeavor.

Contents

1 Introduction **3**
- 1.1 Autonomous Interoperability of AI Services . . . 3
- 1.2 Objectives and accomplishments . . . 4
- 1.3 Related work . . . 4

2 AI-DSL Registry **6**
- 2.1 Realized Function . . . 6
  - 2.1.1 Description . . . 6
  - 2.1.2 Objectives and achievements . . . 8
  - 2.1.3 Future work . . . 8
- 2.2 Network of Idris AI services . . . 8
  - 2.2.1 Description . . . 8
  - 2.2.2 Objectives and achievements . . . 9
  - 2.2.3 Future work . . . 9
- 2.3 Dependently Typed Registry . . . 10
  - 2.3.1 Description . . . 10
  - 2.3.2 Objectives and achievements . . . 12
  - 2.3.3 Future work . . . 12

3 Software Engineering Strategies **14**
- 3.1 Service Composition in Idris2 . . . 14
  - 3.1.1 `RealizedFunction` and `RealizedAttributes` . . . 14
  - 3.1.2 `Service` . . . 15
  - 3.1.3 A Look into Dependent Pairs . . . 15
  - 3.1.4 A Monadic DSL . . . 17
- 3.2 Depth of Embedding . . . 19

4 AI-DSL Ontology **21**
- 4.1 Description . . . 21
  - 4.1.1 Design requirements . . . 21
  - 4.1.2 Domain model considerations . . . 24
  - 4.1.3 Ontology language and upper level ontology . . . 26
  - 4.1.4 Tools . . . 26
- 4.2 Objectives and achievements . . . 27

4.2.1 Decentralized ontology 27
4.2.2 Ontology prototype 28
4.2.3 The mechanism of dynamic workflow construction 35
4.3 Future work 39

5 Conclusion **41**

A Glossary **43**

Chapter 1

Introduction

1.1 Autonomous Interoperability of AI Services

Among the wonders that the blockchain technology enables is the possibility to programmatically exchange values and services between parties. In the context of AI services it becomes especially relevant given the inherent complexity and decomposability of such systems. Moreover, the abundance of AI algorithms available on the Internet makes it that often creating a new AI solution consists of connecting together existing AI algorithms. For instance building a system to discover new drugs may consist of

1. a reasoner to extract background knowledge from biological databases,
2. a principal component analyzer to discover abstractions,
3. a feature selector to discard irrelevant information,
4. a learner to generate predictive models relating selected features and drug efficacy.

The task of composing such AI algorithms is, in most cases, done by humans and, as any AI practitioner knows, is tedious and time consuming. Not just the composition of the whole but also the search and the understanding required to find the parts. Facilitating and ultimately automating such process is the goal of the *Artificial Intelligence Domain Specific Language*, or *AI-DSL* for short.

Another important aspect is the management of resources, both financial and computational (CPUs, GPUs, etc). The AI-DSL is also intended to incorporate descriptions of such computational requirements as well as measures of expected result quality.

Finally, as described in the blogpost, the current plan is to have such system rely on Dependent Types [20] to express and validate the specifications of the AI services, including cost, quality and their relationships thereof. The reason Dependent Types have been chosen is because they are geared toward

program specification checking and program generation, which in our case comes close to AI services verification and combination if one sees AI services as functions. More specifically Idris has been chosen as our initial Dependently Typed Language (DTL) candidate, due to its efficiency and the fact that is has been primarily designed to verify and generate actual running programs as opposed to proofs [1].

1.2 Objectives and accomplishments

For that first iteration the goals were to

1. Experiment with matching and retrieval of AI services using Idris [8], a Dependently Typed Language (DTL) [1], equipped with a powerful type system to express function specifications. That work is described in Chapter 2.

2. Start building an AI ontology to ultimately provide a rich and extendable vocabulary for the AI-DSL. That work is described in Chapter 4.

3. Start building the AI-DSL itself, from its syntax to its semantics. Exploratory work on that is described in Chapter 3.

4. Integrate all the above into a holistic prototype, running on a real world test case of AI service assemblage, in real conditions, that is ideally on the SingularityNET-on-Cardano network. Preparatory work on that is described in Section 2.2.3.

All the objectives except the last one have been accomplished at least to some degree. The last, and the most ambitious, objective had to be postponed for the next iteration due to its complexity.

1.3 Related work

To the best of our knowledge there is no existing work on creating such AI-DSL system to enable autonomous AI services interoperability using Dependent Types, let alone running on a blockchain. There are however attempts using related methodologies, usually involving ontologies with more or less explicit forms of reasoning, outside of the context of the blockchain technology.

We are collecting a body of literature in a github issue [5] serving as a living document and being regularly updated with such related work. We will not describe the entire body here due to its volume (over 80 references at the time of writing) and the fact that we are still in the process of reviewing it. Here is however the most relevant work we have encountered so far.

[1] Programs are equivalent to proofs according to the Curry-Howard correspondence but some representations are more amenable to running actual programs than others

The most recent and also most relevant work we have found is the Function Ontology [6, 27]. Its goal is to define a standard for describing, both formally and informally, functions, their references to implementations, as well as developing tools for retrieving and executing them, remotely or locally. To the best of our knowledge it does not make use of dependent types or blockchain technology, however there is a lot of potential for reuse. More investigation will be conducted to precisely determine to what extent.

Still relevant work with potential for reuse if only conceptual is described below. Let us start with the multi-agent system (MAS) field. In [25] the authors describe a use case of a system called DESIRE, to formally define inputs and outputs of agents and their control flows. The logic seems somewhat limited but their goals align with ours. In [23] an extension of Computation Tree Logic (CTL) called multi-modal branching-time logic is defined to apply model checking on collections of agent given their description in some abstract programming language. It's not clear the specification language they use is very convenient and the model checking methodology is open-ended enough, but nevertheless the work is solid and relevant. In [29] is described an architecture for distributed multi-agent intelligent system. The description of the architecture is high level but the authors mention Agent Communication Language (ACL) standards [39] such as FIPA-ACL [3], used to communicate agents requests to each other. In the ontology field we have found the following papers. In [33] the authors introduce an ontology called EngMath for representing mathematical concepts useful for engineering, including quantity units, equations and more in KIF [10] to be used in SHADE [34] a system for collaborative intelligent agents. It is rather old and it seems development has halted, but could nevertheless be interesting to learn more about. OpenMath [19] is another old, but still maintained, ontology about mathematics. More recent mathematical ontologies worth mentioning are OntoMath [30] and OntoMathPRO [42], they seem rather high level but could still be useful. Broadening the scope, [49] is focused on validating the consistency of ontologies. Relatedly, in [54], a tool for analyzing and propagating changes across overlapping ontologies according to predefined inference rules, called FIDOE, is introduced. The authors mention standards such as SUMO [46] and SWRL [2] as well. Such work could be relevant towards the goal of facilitating the decentralization of the AI-DSL and its ontology.

Finally, it is important to mention a rather young field called Verified Artificial Intelligence (VAI) [50, 35, 41]. The objective of VAI is to bring formal verification to AI to insure that algorithms and models meet certain mathematical requirements. This includes work using dependent types to that effect [28, 48]. For instance [26] uses Idris to guaranty that layers of a neural network are arranged in a sound manner. This does not guaranty that the neural networks coming out of that program are good models of reality, but does eliminate certain types of memory corruption errors by forbidding, for instance, to connect an input vector of a size unequal to that of the first layer.

Chapter 2

AI-DSL Registry

In this chapter we describe a prototype of an AI-DSL registry, a service in charge of storing AI service specifications and returning matching AI services upon request, given a specification to fulfill. In Section 2.1 we describe experiments for implementing the `RealizedAttributes` and `RealizedFunction` data structures described in [22] used for capturing financial, computation costs, as well as measures of expected result quality. In Section 2.2 we describe the implementation of a network of trivially simple AI services implemented in Idris, and use the Idris compiler to type check if they can properly connect to each other. Finally, in Section 2.3 we describe the implementation of an AI-DSL Registry prototype, as a proof-of-concept for querying AI services based on their dependently typed specifications.

2.1 Realized Function

2.1.1 Description

The `RealizedFunction` data structure, as introduced in [22], 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 [22] as well.

For that iteration we have implemented a simple version of `RealizedFunction` and `RealizedAttributes` in Idris2 [8]. 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 infimum-itive. Below is a small snippet of that

code to give an idea of how it looks like

```
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.

Given such data structure we used the Idris compiler to type check if the realized attributes of realized functions, i.e. AI services, composed from other realized functions would follow the defined compositional law, here `add_costs_min_quality`. That is given for instance the realized attributes of an incrementer function

```
incrementer_attrs = MkRealizedAttributes (MkCosts 100 10 1) 1
```

and a twicer function

```
twicer_attrs = MkRealizedAttributes (MkCosts 200 20 2) 0.9
```

the realized attributes of their compositions must be

```
rlz_compo1_attrs = MkRealizedAttributes (MkCosts 300 30 3) 0.9
```

otherwise Idris detects a type error.

2.1.2 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 does it considerably 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].

An improvement of that work is also described in Section 3.1.

2.1.3 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.

2.2 Network of Idris AI services

2.2.1 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`

to test that such composition, for instance, is properly typed. The networking part was implemented based on the SingularityNET example service [15] mentioned in the SingularityNET tutorial [18]. 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 [7], 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 [13]. 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.

2.2.2 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 was reached in the experiments described in Sections 2.3 and 3.1.3. The other objective was to type check that the compositions have realized attributes corresponding to the compositional law implemented in Section 2.1, which was fully achieved in this experiment. For instance by changing either the input/output 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].

Thus besides the fact that dependent types were ignore in that experiment, the objectives were met. See Section 2.3 for a follow up experiment involving dependent types.

2.2.3 Future work

Such experiment was a good way to explore how Idris can be integrated to a network of services. What we need to do next is experiment with actual AI algorithms, making full use of dependent types in their specifications. Such endeavor was actually attempted over the Fake News Warning app described in Section 4.1.2, but it was eventually concluded to be too ambitious for that iteration and was postponed for the next. More about that is discussed in Section 2.3.3.

Also, we obviously want to be able to reuse existing AI services and write their specifications on top of them, as opposed to writing both specification and code in Idris (and ultimately the AI-DSL). To that end it was noted that having

a Protobuf to/from Idris converter would be useful, so that a developer can start from an existing AI service, specified in Protobuf, and enriched it with dependent types using Idris. The other way around could be useful as well to enable a developer to implement AI services entirely in Idris and expose their Protobuf specification to the network. Relatedly having directly an implementation of gRPC for Idris could be handy as well.

2.3 Dependently Typed Registry

2.3.1 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 an *AI-DSL Registry* prototype to start experimenting with such functionality.

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 [17]) 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 2.2 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 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 3.1.3.

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 actually itself (as the `retrieve` procedure of the `registry` service takes a string, a type signature, and returns two strings, the service and procedure names matching such type signature). Likewise one can query

```
Integer -> Integer
```

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, the `compo` AI service queries the registry to fill the hole with the correct service according to its specification, here `twicer`.

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

2.3.2 Objectives and achievements

As shown above we were able to implement a prototype of an AI-DSL registry. Only the `retrieve` function was implemented. The `compose` function still remains to be implemented, although the `compo` service is already somewhat halfway there, with the limitation that the missing type, `?type`, is hardwired in the code, `Integer -> EvenInteger`. It should be noted however that Idris should be capable of inferring such information but more work is needed to 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 and retrieval, using dependent types as formal specification language, is possible.

2.3.3 Future work

There are many possible future improvements for this work, falling into two main categories, the prototype itself, and its use cases.

Improve the prototype

Here is a list in no particular order of possible improvements of that AI-DSL prototype.

- Implement `compose` for autonomous composition.
- Use structured types to represent type signatures instead of String.
- Return a list of services instead of the first one.
- Allow fuzzy matching and infer *sophisticated casts* to automatically convert data in case of imperfect match between output and input types.
- Improve its implementation. The registry prototype is currently 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.

Improve the test cases

- First, we want to expand our trivial AI service assemblage by defining more complex properties, making use for instance of product and sum types (corresponding to the logical connectors $\wedge$ and $\vee$), and explore how to cast specialized properties into more abstract ones. For instance, given

[1] because the SingularityNET example it is derived from is written in Python, not because Python is considered to be the most suitable language for this purpose.

the AI services A, B and C, let's assume A's output satisfies a conjunction of properties, such as an integer that is both even and within a certain interval. Then B and C inputs may only need to partially fulfill such conjunction of properties. For instance B may require an even integer, while C may require that integer to be within a certain interval. In other words both services B and C may take the output of A as input but for different reasons. It is not conceptual difficult to cast the output of A to match the input types of B and C, however this is something we still need to explore in its full generality with Idris.

- Second, we want to adapt the Fake News Warning app described in Section 4.1.2 as test case for the AI-DSL Registry. To briefly explain, the Fake News Warning app is an AI service assemblage estimating if the headline of an article is consistent with its body. Such assemblage is composed of

  1. a collection of classifiers, each attempting to learn to recognize if the headline of an article is consistent with its body;
  2. an aggregator combining the outputs of all classifiers into a single answer.

  So what we would like to achieve is to

  1. formally specify the functions of each AI service above (which requires to enrich the leaf ontology described in Section 4.2.2 as well as the composition functions described in Sections 2.1 and 3.1);
  2. populate the AI-DSL registry described in Section 2.3 with these formal specifications;
  3. type check combinations of these AI services, rejecting illegal ones, such as two classifiers serially connected, and accepting legal ones, such as classifiers connected in parallel to the aggregator;
  4. automatically connect the AI services into a valid assemblage given its high level specification of such assemblage. Such high level specification should include its type signature, the overall financial, temporal and computational cost, as well as the overall expected result quality. Then the assemblage should be constructed in a way to simultaneously satisfy all the requirements. For instance in order to reach the expected result quality, the assemblage may require more classifiers, which may however increase the financial cost, etc.

  By now we have a good grasp of how such service assemblage works and some ideas of how to formally specify its subcomponents. As an intermediary step we have also started porting portions of the Fake News Warning app to Idris, see `ai-algorithms/NeuralNets/README.md` in the AI-DSL repository [4], with the intension of refining the type signatures of the various parts by taking advantage of dependent types.

Chapter 3

Software Engineering Strategies

3.1 Service Composition in Idris2

A key requirement of the AI-DSL is to provide both an ergonomic syntax for describing service properties and a robust process for using these descriptions to verify the correctness of composed services. This work involved investigating several different methods for meeting this requirement using Idris2.

3.1.1 `RealizedFunction` and `RealizedAttributes`

The `RealizedFunction` and `RealizedAttributes` data types were an early strategy for describing and composing AI services. They directly contained values representing the relevant properties of arbitrary Idris functions and made use of a `compose` function to compute the properties of the function resulting from the composition of two others.

While this approach worked to verify that a small, fixed set of attributes was correct for a composition of functions, it also presented several issues:

- The `RealizedFunction` definition contains only the raw data representing function properties, while using a separate function to represent composition logic. Because the composition logic is not part of the type definition, there is no way for Idris to prove that the correct logic was used to construct any given `RealizedFunction`.
- `RealizedAttributes` represents only a set of example properties. The syntax tree for the AI-DSL should be able to represent any properties specified by the user, assuming the composition laws for those properties are known.

3.1.2 `Service`

To address these problems, we implemented the `Service` type, which can be found in `experimental/realized-function/ServiceAttributes.idr`. It differs from `RealizedFunction` in two important ways:

- Composition logic is represented entirely at the type level as a second constructor for the `Service` type.
- Idris' `Num` interface is used as a generic representation of any attribute that can be added when two `Service`s are sequenced.

These changes were sufficient to solve the problems with our earlier approach, but we still needed to improve the expressiveness of our representation. Many important properties are too complex to be described using only the `Num` interface.

3.1.3 A Look into Dependent Pairs

Idris represents the intersection between a theorem proof assistant and a programming language. As such, it is often useful to think of types as logical propositions, and values as proofs of those propositions. Since our goal is to verify that a desired property is true of some value, we can use dependent types to describe a proposition parameterized by a specific value.

Idris provides a special syntax for this. `(x : a ** p)` can be read as "`x` is a value of type `a` such that proposition `p` holds true of `x`". This is called a dependent pair, and it can only be constructed by providing both a value and a proof that a desired property holds true for that specific value. In the context of service composition, we can use dependent pairs as a direct representation of input values that satisfy some condition.

To demonstrate the practicality of this pairing, consider the following types:

```
public export
data WFInt : Type where
     Nat : (n : Nat) -> WFInt
     Neg : (n : Nat) -> WFInt --Note: In the negative case, n=Z represents -1.

-- n-parity, i.e. proof that an integer a is evenly divisible by n (or not).
public export
data Parity : (a : WFInt) -> (n : WFInt) -> Type where
     -- a has even n-parity if there exists an integer multiple x s.t. x*n = a.
     Even : (x : WFInt ** (x * n) = a) -> Parity a n

public export
data OddParity : (a : WFInt) -> (n : WFInt) -> Type where
     -- a has odd n-parity if there exists
     Odd : (b : WFInt ** LT = compare (mag b) (mag n))
      -> (Parity (a + b) n) ->  OddParity a n
```

`WFInt` is a type describing a well-founded view of an integer. This alternate view is necessary in order to write more flexible inductive proofs for integer inputs.

`Parity` demonstrates the proof obligation necessary to show that one integer is evenly divisible by another. In plain English, it can be read as "If there exists some integer `x` such that `x * n = a`, then `a` can be said to have `n-parity`." `OddParity` is a type representing the opposite proposition, i.e. that dividing two integers will produce a remainder.

For services such as our Halfer example, this allows us to clearly express that inputs should be only even numbers, as shown in this function type signature:

```
halfer : (a : WFInt ** Parity a 2) -> WFInt
```

Similarly, the type signatures of the Twicer and Incrementer example services can express their properties with regards to the 2-parity of the integers they operate on:

```
-- Guaranteed to produce a value divisible by 2
twicer : (b : WFInt) -> (a : WFInt ** Parity a 2)

incrementer : (a : WFInt ** Parity a n) -> (b : WFInt ** OddParity b n)
```

Now that the relevant properties for verification are expressed entirely at the type level, the Idris2 typechecker can statically check the validity of service compositions.

```
-- A valid sequence of services that successfully typechecks.
compo1 : WFInt -> WFInt
compo1 = fst (incrementer . halfer . twicer)

-- An invalid sequence of services that will always fail typechecking
compo2 : WFInt -> WFInt
compo2 = fst (halfer . incrementer . twicer)
```

With dependent pairs, arbitrary properties of values can be encoded and formally verified. For an AI-DSL that may need to describe AI services in many different contexts, this ability to use custom types instead of a limited set of primitives is crucial. However, this method is not a complete solution, as it highlights major practical flaws.

An AI service developer making use of the AI-DSL should be able to adequately describe the necessary properties of data their service will take as input, but there should be no need for them to also encode the exact properties of their service's output data. A service developer is not likely to have any knowledge of how their service's outputs will be used by other services in the future, so the AI-DSL should not force them to describe their output data in any more detail than is possible[1]. In the examples above, the `incrementer` service was forced

[1] Of course given a full specification of the service, which is admittedly hard but possible to provide, any decidable correct property about its output data can be inferred, possibly at a prohibitively high cost.

to describe its inputs and outputs in terms of properties that are only relevant to other services.

3.1.4 A Monadic DSL

At this stage, there are two key problems which must be solved:

1. At the point of service creation, developers should not be expected to have knowledge of the properties that are only relevant to other services. They should be able to encode only the properties relevant to their own service.

2. Due to the limits of computability, some relevant properties of data will not be formally provable. However, some of these properties might still be safely assumed to hold in certain contexts, even if a formal proof is impossible. The AI-DSL should be able to represent such cases and provide the strongest possible guarantees.

For the first issue, we borrowed a well-established design pattern from strongly-typed functional programming and defined a new `Service` type around the `Monad` interface. [53] Monads are a class of types used to describe a context for operations, along with any custom logic necessary to combine those operations without imposing any requirement for tight coupling. This is perfect for the AI-DSL.

To address the issue of unproveable properties, we experimented with a conceptual model of smart contracts as a core language feature within the DSL. Because the actual implementation of logic to represent external smart contracts was outside the scope of this work, we made the assumption that such contracts could be used to represent a financially-backed assurance that some unproveable property holds. In theory, this could allow compositions of AI services to be analyzed for their overall financial risk.

The following type describes the Abstract Syntax Tree for a deeply-embedded DSL:

```
public export
data Service : Type -> Type where
     ||| A Service that is definitely of type `a`
     Val     : a -> Service a
     ||| A contract has promised a reward if `a` is not a Service b
     Promise : Contract a b ->  Service b
     ||| Application of a Service to another Service
     App     : Service (a -> b) -> Service a -> Service b
     ||| Explicitly construct a Service using monadic binding
     Bind    : Service a -> (a -> Service b) -> Service b
```

This `Service` type describes a context wherein values may be either native Idris2 values or a reference to an external smart contract.

Below are some simple definitions for the functions necessary for `Service` to be a member of the `Monad` typeclass, as well as superclasses `Functor` and `Applicative`.

```
public export
Functor Service where
        map f (Val a) = Val $ f a
        map f (Promise c) = Val $ f $ trustContract c
        map f s = App (Val f) s

public export
Applicative Service where
            pure = Val
            (<*>) = App

public export
Monad Service where
      (>>=) = Bind
      join m = !m
```

With these operations defined, sequencing services becomes much simpler. Idris2 provides a special /textttdo-notation for monads, as well as convenient syntax for pattern-matching on intermediate values. The following are several example composition scenarios, taken from `experimental/typed-dsl/Compo.idr`:

```
-- composition of Twicer and Incrementer
compo1 : Integer -> Service (Integer)
compo1 a = do
            n <- twicerService a
            incrementerService n



-- composition of Twicer, Halfer, and Incrementer
compo2 : Integer -> Service (Integer)
compo2 a = do
            i <- twicerService a
    -- Because twicerService does not provide its own proof that its
    -- output is always even, we use a Promise to provide a soft proof
    -- of this property.
            p <- Promise ?con
            j <- halferService (cast i ** p)
            incrementerService j


-- invalid composition of Twicer, Incrementer, and Halfer
compo3 : Integer -> Service (Integer)
```

```
compo3 a = do
            i <- twicerService a
            j <- incrementerService i
    -- The `resolve` hole shows that the programmer must create apply some logic
    -- of type Integer -> EvenNumber to resolve the mismatch here.
            halferService $ ?resolve j


-- A potential method to resolve the above mismatch
compo3sol : Integer -> Service (Integer)
compo3sol a = do
            i <- twicerService a
            j <- incrementerService i
    -- Because this function contains the actual composition of the various
    -- Services, this is the point where the programmer is best able to decide
    -- which measures are acceptable to resolve type mismatches.
    -- In this case, forceEven is used.
            halferService $ forceEven j


-- In this composition, we have no way to statically prove that
-- halferService is being passed an even number.
compo4 : Integer -> Service (Integer)
compo4 a = do
    -- i could be even or odd, depending on the value of a
            i <- incrementerService a
    -- We can pattern match on the result of a runtime test
    -- to create a branch in the logic of this Service.
            Just j <- pure $ maybeEven i
                 | Nothing => twicerService i
    -- If there is a Just EvenNumber, run the halferService on it.
    -- If there is no EvenNumber value to be found, run the twicerService on i.
            halferService j



-- Because all of the above examples are Integer -> Service (Integer),
-- it is relatively trivial to compose them.
compo5 : Integer -> Service (Integer)
compo5 a = (compo1 a) >>= compo2 >>= compo3 >>= compo4
```

3.2 Depth of Embedding

A domain-specific language requires not only a formal specification for its semantics, but a software implementation as well. The relationship between a

DSL and its implementation can vary, but most strategies fall into one or more of three categories:

1. `Independent Syntax:` A language may be designed completely separately from its implementation. Such languages typically require dedicated compilers or interpreters, as they are unable to borrow any functionality due to their lack of a host language.

2. `Shallow Embedding:` An embedded domain-specific language (eDSL) is written as a module or library for some host language. Data in the DSL's domain is represented directly as values in the host language. [31] Shallow embeddings tend to be easy to use and extend, but often suffer issues with performance and expressiveness. Programs written in a shallowly-embedded DSL can only describe operations in the domain of their host language, and thus are limited to a single interpretation.

3. `Deep Embedding:` Similarly to a shallowly-embedded DSL, an eDSL with a deep embedding is defined in some host language. Deep embeddings define a custom Generalized Algebraic Data Type (GADT) in the host language and represent all data as values of this type. [31] Because the entire Abstract Syntax Tree (AST) of a deeply-embedded program is a single type, it is simple to write functions in the host language that operate directly on the embedded program. This allows for automatic optimization of embedded programs, as well as multiple possible interpretations. However, any extensions to a deep eDSL require significant effort, as changes to the language's AST type incur a requirement to update every function that operates on that type.

For the AI-DSL, the most promising approach appears to be a hybrid method. The basic domain of the DSL can be defined as deep embedding, while more specialized features can be shallowly embedded as smaller DSLs within the main AI-DSL instead of directly in Idris2.

Chapter 4

AI-DSL Ontology

4.1 Description

4.1.1 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 platform and ecosystem. The discussions were based on [22, 16] 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 platform to declare their capabilities and needs for data to other AI agents in a rich and versatile machine readable form; This will enable different AI agents to search, find data sources and other AI services without 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 platform level.

Therefore, it seems best to define the 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 extendable in the long term, but the mechanisms of changing and extending it will be quite complex, most probably involving theoretical 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 platform 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 platform, AI-DSL shall allow maximum extendability 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 dependently typed, 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 dependently typed language choice Idris, it will have to interface with the world, deal with in-deterministic input from network and mutable states – operations that may fail in run-time no matter how careful type checking is done during compile time [24].

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

As demonstrated in Chapter 2 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 platform in general.

4.1.2 Domain model considerations

In order to attend to all high level design requirements. All levels of the AI-DSL Ontology should be developed simultaneously, 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* [32, 36]. 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 platform.

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 ontology prototype we use the `Fake News Warning`[1] application being developed by NuNet – a currently incubated spinoff of SingularityNET[2].

NuNet is the platform 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[44]. 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 2.3 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 application-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 back-ends) 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 back-end 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 pretrained 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 front-end	URL of the content to be checked	Probability that the content in the URL is fake	©NuNet 2021

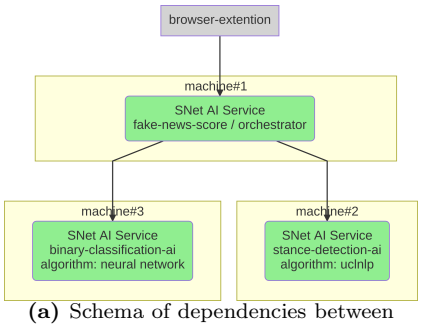
Table 4.1: Description of each component of `Fake News Warning` application.

Each component of application's back-end is a SingularityNET AI Service registered on the platform. Note, that as SingularityNET AI services are defined through their specification and their metadata[14]. 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 platform and compose into complex workflows – similarly to what is described in Section 2.2. Here is a simple representation of the program graph of `Fake News Warning` app:

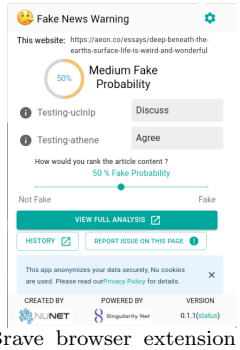
```
1       "dag": {
2         "news-score" : ["uclnlp","binary-classification"]
3         }
```

Figure 4.1: A directed acyclic graph (DAT) of `Fake News Warning` app prototype[11]. It simply says that `news-score` depends on `uclnlp` and `binary-classification`.

The schematic representation of the `Fake News Warning` app deployed as a result of processing the DAG is depicted below. The addition of NuNet platform to SingularityNET service discovery is that each service may be deployed on different hardware environments, sourced by NuNet. When the application back-end is deployed, it can be accessed from the GUI interface, which in case of `Fake News Warning` is a Brave browser extension.



(a) Schema of dependencies between backend components of the application (SingularityNET AI services potentially running on different machines).



(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 the AI-DSL Ontology and its prototype.

4.1.3 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 [46] and SUMO as an upper-level ontology [43]. 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 (FOL) statements in a simple text format in terms of lisp-like syntax. Due to that, KIF can be easily converted to other formats[38]. Also, a conversion to Atomese – the OpenCog's language also employing a lisp-like syntax – has been successfully attempted in the past[4]. SUMO and the related ontology design tools [45] provide a convenient way for starting to design AI-DSL Ontology levels and their relations.

4.1.4 Tools

For the purposes of design, inital validation and displaying relations between classes, subclasses and instances of the ontology, we have used software tools which come together with SUMO ontology [5]:

- Sigma IDE for SUMO[6] and
- jEdit plugin for SUMO [7]

[3] See Reusing Existing Ontologies discussion on AI-DSL Github repository[4]
[4] See the SUMO Importer in the OpenCog External Tools repository[12]
[5] https://ontologyportal.org
[6] https://github.com/ontologyportal/sigmakee
[7] https://github.com/ontologyportal/SUMOjEdit

The ontology prototype, presented here, is fully accessible for browsing and partial validation via the local Sigma installation[8].

4.2 Objectives and achievements

4.2.1 Decentralized ontology

In order to satisfy the *extendibility* requirement of ontology design, we are proposing a notion and design of a *decentralized ontology*, which enables us to work with globally consistent and locally inconsistent components within the same mechanism of AI-DSL. Based on our design, the full ontology of `Fake News Warning` application is constructed from a number of separate components, which operate at different level of decentralization. Table below describes each of these components.

[8]Can be temporarily accessed at http://nunetio.ddns.net:8080/sigma/KBs.jsp or installed and accessed locally by folowing these instructions

Component	Description	Dependencies	Extendability
Merge.kif	SUMO structural ontology, base ontology, numerical functions, set/class theory, temporal concepts and mereotopology	None - root ontology	Centralized and globally enforced – defined by ontologyportal.org
SingularityNet.kif	Defines global classes and types to be used for describing each SingularityNET AI Service	ComputerInput.kif, Merge.kif [,..]	Limited: versioning mechanism controlled by SingularityNET (to be defined)
FakeNewsScore.kif	SingularityNET service responsible for constructing the whole back-end of each `Fake News Warning` application instance i.e. program graph (DAG) of the application.	SingularityNET.kif [,..]	Fully decentralized: defined by application developers; Since `Fake News Warning` application is open source, any developer can fork it and define it otherwise; Technically, this would be a different application.
fnsBinaryClassifier.kif	A pre-trained binary classification model for fake news detection	SingularityNET.kif [,..]	Fully decentralized: defined by each algorithm developer independently. Technically, from the platform perspective, these will be different algorithms.
uclnlp.kif	Forked and adapted component of stance detection algorithm by UCL Machine Reading group	SingularityNET.kif [,..]	Fully decentralized: defined by each algorithm developer independently. Technically, from the platform perspective, these will be different algorithms
NuNetEnabledComputer.kif	Each NuNet enabled hardware resource will have to be described accordingly when onboarded to NuNet platform	NuNet.kif	Fully decentralized: independently defined by the owner of a hardware resource
NuNet.kif	Defines classes to be used for describing each hardware resource eligible for running SingularityNET AI Services via NuNet platform;	Merge.kif, SingularityNET.kif [,..]	Limited: versioning mechanism controlled by NuNet (to be defined)

Table 4.2: Description of each component of the AI-DSL Ontology prototype and links to related KIF files.

4.2.2 Ontology prototype

Using the ontology levels described in Table 4.2 and referenced files, we prototyped the ontology of `Fake News Warning` application[9].

[9]Can be temporarily accessed at http://nunetio.ddns.net:8080/sigma/KBs.jsp or installed and accessed locally by folowing these instructions

Architectural level	Class
SingularityNET platform	SNetAIService, SNetAIServiceIO, SNetAIServiceMetadata
NuNet platform	NuNetEnabledSNetAIService, NuNetEnabledComputer

Table 4.3: Main classes defined in AI-DSL ontology prototype per level of the `Fake News Warning` application's stack. Classes defined in SUMO are not included.

AI algorithms onboarded on the SNet platfrom are instances of `SNetAIService` class of sublasses of it. Services of `Fake News Warning` application are defined as follows:

```
(instance uclnlp NuNetEnabledSNetAIService)
(documentation uclnlp EnglishLanguage "Forked and adapted component of
↪  stance detection algorithm by UCL Machine Reading group.")
```

(a) Service description

```
(hasInput uclnlp uclnlpInput)
(hasInput uclnlp uclnlpOutput)

(instance uclnlpInputType DataType)
(instance uclnlpOutputType DataType)
```

(b) Descriptions of service input and output types.

```
(=>
  (and
    (hasField ?uclnlpInput titleText Text)
    (hasField ?uclnlpInput mainText Text)
  )
  (instance ?uclnlpInput uclnlpInputType)
)

(=>
  (and
    (hasField ?uclnlpOutput agree RealNumber)
    (hasField ?uclnlpOutput disagree RealNumber)
    (hasField ?uclnlpOutput discuss RealNumber)
    (hasField ?uclnlpOutput unrelated RealNumber)
  )
  (instance ?uclnlpOutput uclnlpOutputType)
)
```

(c) Definition of types and their dependencies.

Figure 4.3: SNet AI Service definition in KIF (uclnlp and binary-classification services are described in this way).

Type definitions and their dependency definitions are actually the domain of formal type-checking part of AI-DSL and Idris related research. However, irrespectively of which language will be eventually chosen for AI-DSL, Figure

4.3 expresses that we can:

1. define correct serviceInput and serviceOutput types (unique for each service);
2. potentially provide proofs that if a service data of correct type is provided on input, then it will output correctly typed data;
3. if the above is not possible (which may be the default option when actual service AI are not written in Idris):
   (a) check if input data is of correct type at run-time and refuse to start service if it is not;
   (b) check if output data is of correct type before sending it to the caller and raise error if it is not so;

`FakeNewsScore` AI Service is special in that it calls other dependent services (as described by program graph in Figure 4.2a) and combines their results. We can define the program graph in terms of dependencies between services in KIF as follows:

```
(instance fakeNewsScore NuNetEnabledSNetAIService)
(documentation fakeNewsScore EnglishLanguage "Calls dependent services,
↪  calculates summary result from their outputs
and calculates the overall probability that the provided content contains
↪  fake news")

(hasDependency fakeNewsScore uclnlp)
(hasInput fakeNewsScore uclnlpOutput)

(hasDependency fakeNewsScore fnsBinaryClassifier)
(hasInput fakeNewsScore fnsBinaryClassifierOutput)
```

Figure 4.4: Defining program graph as a formal ontology. This is similar to DAG of Figure 4.1.

Figure 4.4 demonstrates how a workflow of connected SingularityNET AI services can be statically defined and proven to work at compile time. However, we could go further and define dependencies as *subclasses* of services with the same input/output data types. In such case any instantiation of the subclass would be able to dynamically compile into the workflow. Therefore we would not need to describe concrete dependencies – they would be dynamically resolved at run-time by matching input and output types.

```
(instance fakeNewsScore NuNetEnabledSNetAIService)

(hasInputType fakeNewsScore StanceType)
(hasInputType fakeNewsScore BinaryClassificationType)

(hasOutput fakeNewsScore fakeNewsScoreOutput)
(instance fakeNewsScoreOutputType DataType)

(=>
  (and
    (hasField ?data agree RealNumber)
    (hasField ?data disagree RealNumber)
    (hasField ?data discuss RealNumber)
    (hasField ?data unrelated RealNumber)
  )
  (instance ?data StanceType)
)

(=>
  (hasField ?data fakeOrNot Boolean)
  (instance ?data BinaryClassificationType)
)
```

Figure 4.5: Defining generic input types instead of concrete dependencies in a `FakeNewsScoreDynamic` service.

Any AI service with output type matching input type of the `FakeNewsScoreDynamic` could be compiled into the workflow:

```
(instance uclnlp NuNetEnabledSNetAIService)

(hasInput uclnlp WebContentType)
(hasInput uclnlp StanceType)
```

Figure 4.6: Using static globally defined types of input and output data structures of matching services eligible for compilation into a workfow.

However, systems with dependent typing, like Idris, may allow to go even further and to find out if composite types are composed of the same components and primitive types – and thus match them.

```
(hasInput uclnlp SomeType)
(hasInput uclnlp SomeOtherType)

(=>
  (and
    (hasField ?data titleText Text)
    (hasField ?data mainText Text)
  )
  (instance ?data SomeType)
)

(=>
  (and
    (hasField ?data agree RealNumber)
    (hasField ?data disagree RealNumber)
    (hasField ?data discuss RealNumber)
    (hasField ?data unrelated RealNumber)
  )
  (instance ?data SomeOtherType)
)
```

Figure 4.7: Hypothetical usage of dynamic typing (most probably could be achieved in Idris, but not in KIF).

Primitive (or grounded) types (like `RealNumeber` and `Text` in Figure 4.7), however, should be globally accessible and unambiguously defined for this scheme to work.

All services of `Fake News Warning` application are instances of `NuNetEnablesSNetAIService` subclass, which, in turn, is a subclass of `SNetAIService` class:

```
(instance fakeNewsScore NuNetEnabledSNetAIService)
(instance uclnlp NuNetEnabledSNetAIService)
```

(a) Declaration of `FakeNewsScore` service in FakeNewsScore.kif and of `uclnlp` service in uclnlp.kif.

```
(subclass NuNetEnabledSNetAIService SNetAIService)
(documentation NuNetEnabledSNetAIService EnglishLanguage "SNetAIService
↪  which can be deployed on NuNetEnabledComputers and orchestrated via
↪  NuNet platfrom")
```

(b) Definition of `NuNetEnabledSNetAIService` in NuNet.kif.

Figure 4.8: Relation between SingularityNet and NuNet domain ontologies.

Figure 4.8 describes relation between SingularityNET and NuNet platforms. `SNetAIService` class, defined in SingularityNET.kif, contains all requirements for the metadata of the service to be published on SingularityNET platform. `NuNetEnabledSNetAIService` extends `SNetAIService` by adding metadata that is needed for this service to be deployed via NuNet APIs:

```
(=>
  (and
    (hasMetadata ?SNetAIServiceMetadata ?SNetAIService)
    (hasField ?RequiredComputingResources ?SNetAIServiceMetadata)
  )
  (instance ?SNetAIService NuNetEnabledSNetAIService)
)
; may include many other requirements
```

Figure 4.9: The definition of `NuNetEnabledSNetAIService` in NuNet.kif requires a service to have compute resource (and possibly other) requirements included in service metadata. The idea is that without required metadata fields, a service would not pass validation allowing it to be deployed via NuNet. An arbitrary amount of requirements could be defined here.

`NuNetEnabledSNetAIService`s can be deployed only on `NuNetEnabledComputer`s, which expose their available computing resources in a manner that the ability to run a service is automatically checked **before** a service is dynamically deployed on a computer and a service call is actually issued to it (see Figure 4.10). This formally described relation between SingularityNET and NuNet ontologies enables to prove at 'compile time' that a service will have enough computational resources to be executed. Recall, that SingularityNET ontology alone enables to prove that a service or a collection of services will return correct results when called with correct inputs.

```
(subclass NuNetEnabledComputer Computer)
(documentation NuNetEnabledComputer EnglishLanguage "A Computer which was
↪  onboarded to NuNet platfrom and complies to its requirements.")

(=>
  (and
    (hasRun ?NuNetEnabledComputer NuNetOnboardingScript)
    (hasMetadata ?NuNetEnabledComputer ?ComputerMetadata)
    (hasField AvailableComputingResources ?ComputerMetadata)
    (or
      (runsOS ?NuNetEnabledComputer Linux)
      (runsOs ?NuNetEnabledComputer Raspbian)
    )
    (or
      (hasHardware ?NuNetEnabledComputer PC)
      (hasHardware ?NuNetEnabledComputer RaspberyPi)
    )
  )
  (instance ?NuNetEnabledComputer NuNetEnabledComputer)
)
```

Figure 4.10: The definition of `NuNetEnabledComputer` in NuNet.kif requires available computing resources, computer type and operating system to be listed in the metadata.

An `SNetAIService` can only be deployed on `NuNetEnabledComputer` if available resources on the computer are not less than compute requirements of a service:

```
(=>
  (and
    (hasMetadata ?NuNetEnabledComputer ?ComputerMetadata)
    (hasField ?AvailableComputingResources ?ComputerMetadata)
    (hasMetadata ?NuNetEnabledSNetAIService ?SNetAIServiceMetadata)
    (hasField ?RequiredComputingResources ?SNetAIServiceMetadata)
    (lessThanOrEqualTo ?RequiredComputingResources
    ↪  ?AvailableComputingResources)
  )
  (canDeploy ?NuNetEnabledSNetAIService ?NuNetEnabledComputer)
)
```

Figure 4.11: Constraints on eligible match between `SNetAIService` and `NuNetEnabledComputer` defined in NuNet.kif and required for deployment of a service.

`SNetAIService` and `NuNetEnabledSNetAIService` classes are positioned within the SUMO ontology as follows:

Class, subclass or instance	Description	Where defined
Entity	The universal class of individuals. This is the root node of the ontology.	Merge.kif
Abstract	Properties or qualities as distinguished from any particular embodiment of the properties/ qualities in a physical medium. Instances of Abstract can be said to exist in the same sense as mathematical objects such as sets and relations, but they cannot exist at a particular place and time without some physical encoding or embodiment.	Merge.kif
Proposition	Propositions are Abstract entities that express a complete thought or a set of such thoughts. Note that propositions are not restricted to the content expressed by individual sentences of a Language. They may encompass the content expressed by theories, books, and even whole libraries. A Proposition is a piece of information, e.g. that the cat is on the mat, but a ContentBearingObject is an Object that represents this information. A Proposition is an abstraction that may have multiple representations: strings, sounds, icons, etc. For example, the Proposition that the cat is on the mat is represented here as a string of graphical characters displayed on a monitor and/ or printed on paper, but it can be represented by a sequence of sounds or by some non-latin alphabet or by some cryptographic form.	Merge.kif
Procedure	A sequence-dependent specification. Some examples are ComputerPrograms, finite-state machines, cooking recipes, musical scores, conference schedules, driving directions, and the scripts of plays and movies.	Merge.kif
ComputerProgram	A set of instructions in a computer programming language that can be executed by a computer.	Merge.kif
SoftwareContainer		Singularity-Net.kif
SNetAIService	Software package exposed via SNetPlatfrom and conforming to the special packaging rules	Singularity-Net.kif
NuNetEnabled-SNetAIService	SNetAIService which can be deployed on NuNetEnabledComputers and orchestrated via NuNet platfrom	NuNet.kif
`uclnlp`	Forked and adapted component of stance detection algorithm by UCL Machine Reading group.	uclnlp.kif

Table 4.4: Full hierarchy of dependencies of `uclnlp` SNet AI service instance within SUMO ontology. The same hierarchy applies to `binary-detection` and `fakeNewsScore` services used in the `Fake News Warning` app.

4.2.3 The mechanism of dynamic workflow construction

An important part of the *decentralized ontology* design is the mechanism which makes it work in actual scenarios. This mechanism was designed using the same domain model of `Fake News Warning` application. It also clarifies the reason why we propose this particular concept and design of *decentralized ontology*.

AI-DSL will allow to search, match, compile and execute independently developed AI components in terms of a single veritable workflow running on SingularityNET platform. AI components of the workflow may be developed using different programming languages by different people, have different licenses and, actually, may be developed with different initial goals. Furthermore, these workflows will be executed on the machines owned by different entities. In a decentralized system like this, each developer will be able to freely choose properties, capabilities and internal structure of their algorithms. Through the mechanism of dynamic workflow construction, AI-DSL will be able to pull together the information about each component of desired workflow when the execution of it is required.

The very high level view of SingularityNET's AI Service calls involving AI-DSL looks as follows:

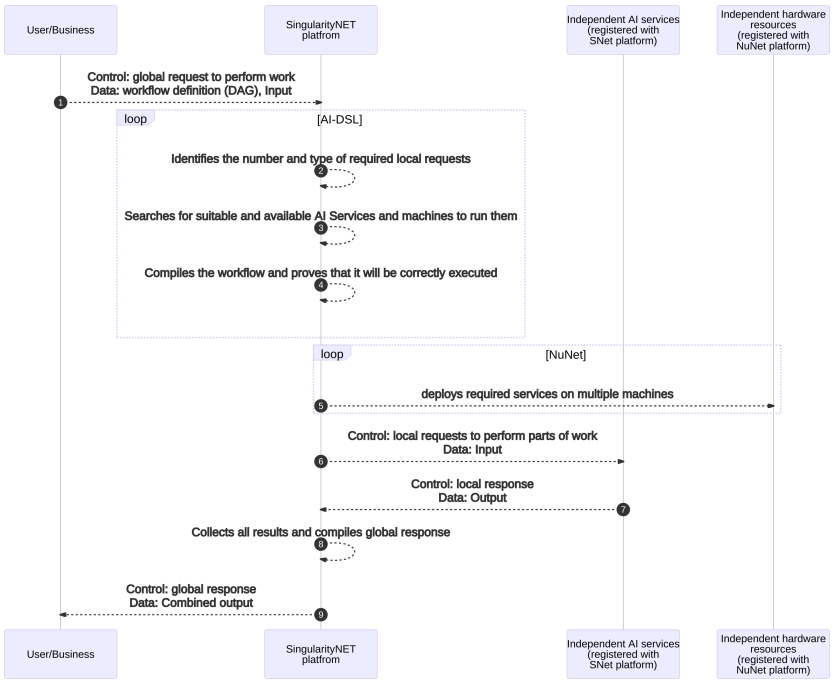


Figure 4.12: Bird's eye view of application-independent SingularityNET calls involving AI-DSL.

Within the domain model of `Fake News Warning` application (Figure 4.2a) this scheme works approximately in the following way:

1. *User/Business* accesses the platform via browser-extension by sending (a) the definition of the workflow to the platform in the form of a DAG (Figure 4.1) and (b) the web content to be checked for probability to contain fake news.

2. The AI-DSL engine reads the DAG and identifies the dependent SNet AI Services that need to be called.

3. If dependent services are indicated statically as in 4.1, then the platform knows immediately names of the services to be called. If, however, dependent services are described in terms of their input / output types (as in 4.4), the AI-DSL engine searches and matches services available in the platform that satisfy constraints defined there[10].

[10] In the future, the AI-DSL engine will aim to accommodate fuzzy service definitions and complex decision functions to search and match them, involving ability for an AI Service to choose its dependent services.

4. When matching services are found, the AI-DSL engine pulls their individual type signatures and other metadata from each service (note, that a decentralized system cannot be built with the assumption of availability of global registry; such registry can, however, be built as a secondary index of otherwise decentralized information sources) and compiles into a workflow. This operation may be done in a few stages:

    - When the AI-DSL engine requests metadata for the dependent service and receives it, it checks the received metadata for conformance to AI-DSL Ontology requirements (e.g. well-formed description in SUO-KIF and correct type dependencies as defined hierarchy in Table 4.4 and displayed graphically in Figure 4.13):

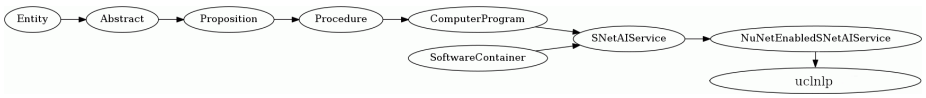


Figure 4.13: Graphical form of the hierarchy of dependencies of `uclnlp` SNet AI service instance of `Fake News Warning` application within SUMO ontology.

    - Note, that the correctness of type dependencies of decentralized components of the AI-DSL Ontology (`uclnlp`) will be checked against centralized components versioned by SingularityNET platform (Table 4.2). Defining versioning mechanism of global components of the ontology is not within the scope of this work. However, merely acknowledging the possible existence of different versions of root and middle level ontologies within the hierarchy requires to think about reasonable way to accommodate them into the system. A possibility is to include information about the version of global components of ontology when communicating decentralized components between each other, as suggested in [55]. In such case, the stage ❹ of the workflow construction in Figure 4.12 would look approximately like this:

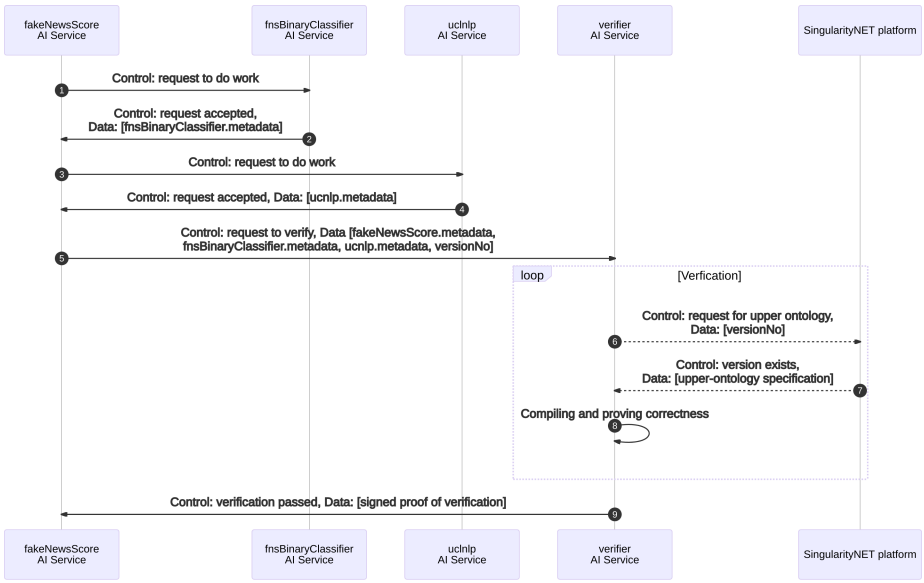


Figure 4.14: Stage ❹ of workflow construction sequence (fully depicted in Figure 4.12) – compiling the workflow and proving its correctness.

- Service metadata returned by calls ❷ and ❹ of Figure 4.14 include service definitions and the version number of the global AI-DSL Ontology that was used to build these definitions. For example, call ❹ may contain the following information:

```
{
  "sender": "uclnlp",
  "receiver": "fakeNewsScore",
  "upper-ai-dsl-ontology": "v0.1",
  "service-metadata": $(include sevice_definition.json),
  "leaf-ontology": $(include uclnlp.kif)
}
```

Figure 4.15: Metadata of the `uclnlp` service. Sample contents of included files can be seen separately for each service_definition.json and uclnlp.kif.

- When all service definitions are collected and the their versions checked to match, they can be checked for conformance with the global AI-DSL Ontology of respective version and, if service definitions include type signature – type-checked.
- The actual compilation of the workflow, compliance to the AI-DSL Ontology and type-checking need a dedicated and properly configured execution environment. In the context of this document, that execution environment may include the Idris compiler, an ontology

prover that is able to process SUO-KIF definitions (e.g. Sigma), their dependencies and possibly custom code. For that, it would be most logical to introduce a dedicated AI service into the platform – which is the `verifier` component denoted in Figure 4.14. The `verifier` service will be able to run any required verification procedures in order to provide a proof that the workflow constructed from services found in step ❸ is valid and can be correctly executed on the SingularityNET platform.

- After calls ❶, ❷, ❸ and ❹ of Figure 4.14 are completed, a call ❺ will be issued to `verifier` sending all metadata of each service along with the AI-DSL Ontology version's identifier. The `verifier` will then request all required dependencies (listed in Figure 4.2) from the central SingularityNET repository (or blockchain) and calculate the proof.
- After `verifier` calculates the proof at step ❽, the proof is sent to the service that has requested it (in the case of 4.12 – to `fakeNewsScore`). Additionally, the existence of independent verifiers would allow to optimize the overall computational costs of calculating proofs on the platform, by recording them into the blokchain and making searchable by other services that may require the same workflow. One way to do this would be to:
  (a) Calculate a hash from the metadata of each service of the workflow (i.e. *.kif or *.idr files);
  (b) Construct a Merkle tree [11] from those hashes which would exactly mirror the structure of the workflow defined in DAG (Figure 4.1);
  (c) Record the root hash of the tree into the blockchain with relevant metadata;
- Such setup would constitute an implicit reputation system of workflows in the sense that a workflow with most proofs of correctness on the blockchain could be trusted to work without recalculating the proof each time a workflow is constructed.

4.3 Future work

- In the long term, it may be ideal to develop a converter for converting OWL to KIF, since OWL may be representable in KIF [40] using OWL API; For the purpose of the ontology prototype, we are manually selecting parts of the existing ontologies in order to build the prototype and write them in SUO-KIF format.
- Similarly we want to be able to convert SUO-KIF specifications into Idris, and possibly vise versa, to take advantage of the strengths of each for-

[11] https://en.wikipedia.org/wiki/Merkle_tree

malism. To the best of our knowledge there are no existing tools to automatically translate SUO-KIF to/from Idris, however there is a tool to translate SUO-KIF to FOL [47] and a paper describing the translation from a Dependently Typed Language (DTL) to FOL [51]. Additionally, to start building an understanding about such process, we have manually ported the trivial AI services described in Section 2.3 to SUO-KIF, see TrivialServices.kif under the ontology of the AI-DSL repository [4]. As it turns out writing formal specifications of functions in SUO-KIF is reasonably straight forward. Here is for instance the SUO-KIF implementation of the Twicer service

```
(instance TwicerFn UnaryFunction)
(domain TwicerFn 1 Integer)
(range TwicerFn EvenInteger)
(=>
  (instance ?INTEGER Integer)
  (equal (TwicerFn ?INTEGER) (MultiplicationFn ?INTEGER 2))))
```

where `EvenInteger` happens to be predefined in Merge.kif of SUMO, partially recalled below

```
(=>
  (instance ?NUMBER EvenInteger)
  (equal (RemainderFn ?NUMBER 2) 0))
```

Thus one can see that is it easy to specify a function input type, using `domain`, and output type, using `range`, in SUO-KIF, as well its full or partial definition, using `=>`, `equal` and universally quantified variables such as `?NUMBER`. It should be noted however that the reason it works so well in that case is because the output type does not depend on the input value, the output is an even integer no matter what. It is expected that porting for instance the `append` function of the dependent type `Vect` [9] to SUO-KIF might not be as trivial, since the `domain` and `range` constructs may not be suitable to represent such dependence (i.e. that the size of the resulting vector of `append` is the sum of the sizes of the input vectors). However, given that dependent types are essentially functions, it might be possible to set the `domain` and `range` with such type functions. Alternatively such dependence can be moved to the function definition as offered by SUO-KIF expressiveness. Another aspect we need to explore is how tools, such as Automatic Theorem Provers (ATPs) [21, 52, 37], can be used to autonomously compose as well as retrieve functions given their input and output types. Obviously if ATP tools running over SUO-KIF turn out to be deficient in that respect, we already know from Section 2.3 that Idris can fulfill that purpose.

Chapter 5

Conclusion

The novel nature of such a project requires a large amount of exploration, which is what this iteration has been all about. From the start we agreed that we wanted to take a holistic approach, attempting to achieve a full albeit limited prototype as rapidly as possible in order to uncover hidden blocks as early as possible. Our exploration has reflected this approach. Let us summarize what we have accomplished so far and what are the next steps to bring us closer to a complete AI-DSL.

- We have started gathering and reviewing literature of related work to make sure we do not miss anything major and can take advantage of existing technologies. Even though we have found no such related project using dependent types and combined with the blockchain technology, there are related projects, described in Section 1.3, with potential for re-usability of ideas or implementations such as the Function Ontology, FIPA-ACL or more. We intend to keep studying the literature of related work.

- We have experimented with Idris to formalize and reason about realized function attributes such as costs and quality, see Section 2.1 for more details. We have done so in a limited manners, only considering additive cost and infimum-itive quality, but we have proven that it is possible and tractable to do in Idris. More work is required to expand the complexity of such realized function attributes, such as functional or distributional costs and quality.

- An AI-DSL Registry prototype have been build, to retrieve, match and connect AI services based on their specifications as dependent types using Idris meta-functions for function matching and retrieval as described in Section 2.3. This prototype has limits, such as returning only the first matching AI service and not performing fully autonomous composition of AI services, but none of these limits seem fundamentally hard to address and should only require more development time.

- In Chapter 3, we have experimented with Idris to formalize function properties as dependent types. This was done in a limited manner, using trivial properties such as evenness of numbers, but taught us that it is a possible to do in Idris, and provided us insights on how to expand that to more complex properties. Also, various approaches for defining AI-DSL as an Idris eDLS have explored, see Section 3.2. Additional an important idea was approached in this Chapter, the interaction between the AI-DSL and the tokenomics of the network as a means to provide soft guaranties when hard guaranties are difficult to obtain, see 3.1.4.

- In Chapter 4 we have explored ontologies with the goal of defining a rich and extendable vocabulary for specifying AI services, their algorithms, data types as well as their relationship to real world. For now the decision was made to build such AI ontology on top of SUMO due to its openness, breadth and quality, as well as the expressiveness of its representational language, SUO-KIF. The upper layer of the Fake News Warning app was translated into SUO-KIF as a SUMO extension. We also explored how to convert SUO-KIF knowledge into Idris, more work is required to automate such conversions. Something that has been discussed but remains to be fully explored is the use of Automatic Theorem Provers as complement (or possibly replacement) of Idris for matching and composing services.

- We have started building, though at the preparatory level, a real world AI service assemblage test case, based on Nunet Fake News Warning app, see Section 2.3.3. Such test case is going to be critical to put our AI-DSL prototypes to the test and build the understanding necessary to push it to the next level.

The work taking place during the next iterations will consist in continuing such exploration, refining our existing prototypes and bringing them together in a holistic system, guided by real world AI service assemblage test cases. Such test case will initially include the Nunet Fake News Warning collective. Then more test cases will be considered over various domains such as bio-informatics, finance, embodied agent control and more.

To conclude, even though there is a long way to go, we believe a lot of progress has been done already, and we are happy to say that no profound difficulties have been revealed so far. We must however remain cautious. One difficulty that is expected to eventually come up is the tractability of the verification and automated composition process of AI services. This is generally undecidable, and even when restricted to subclasses of functions (as is the case in Idris due to being based on Intuitionistic Logic) can still have explosive complexity. However, it is also expected that a tremendous amount of value will be created by having such system work with restricted applicability, or by relaxing the level of guaranties demanded. Ultimately it is expected that the AI-DSL will need to synergies with AGI systems to reach its full potential, which, fortunately, is one of the quintessential functions that SingularityNET aims to offer.

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.

Bibliography

[1] Wikipedia. category:dependently typed languages, https://en.wikipedia.org/wiki/Category:Dependently_typed_languages

[2] SWRL: A Semantic Web Rule Language Combining OWL and RuleML (2004), https://www.w3.org/Submission/SWRL/

[3] FIPA-ACL Standard Specification (2012), http://www.fipa.org/repository/standardspecs.html/

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

[5] AI-DSL Related Work, AI-DSL Related Work GitHub Issue 43 (2021), https://github.com/singnet/ai-dsl/issues/43

[6] Function Ontology, Function Ontology Homepage (2021), https://fno.io/

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

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

[9] Idris2, Idris2 Vectors Documentation (2021), https://idris2.readthedocs.io/en/latest/tutorial/typesfuns.html#vectors

[10] Knowledge Interchange Format (KIF) (2021), http://www-ksl.stanford.edu/knowledge-sharing/kif/

[11] NuNet, Program Graph of fake News Warning Application (2021), https://gitlab.com/nunet/fake-news-detection/fake_news_score/-/blob/master/service/dag.json

[12] OpenCog External Tools, OpenCog External Tools GitHub Repository (2021), https://github.com/opencog/external-tools/

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

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

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

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

[17] SingularityNET Registry, SingularityNET Registry Documentation Webpage (2021), https://dev.singularitynet.io/docs/concepts/registry/

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

[19] Abbott, J., Leeuwen, A., Strotmann, A.: Objectives of openmath: Towards a standard for exchanging mathematical information (07 1995)

[20] Altenkirch, T., Mcbride, C., Mckinna, J.: Why dependent types matter. In: In preparation, http://www.e-pig.org/downloads/ydtm.pdf (2005)

[21] Baumgartner, P., Suchanek, F.M.: Automated reasoning support for sumo/kif (2005)

[22] 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

[23] Bourahla, M., Benmohamed, M.: Formal specification and verification of multi-agent systems. Electronic Notes in Theoretical Computer Science **123**, 5–17 (2005). https://doi.org/https://doi.org/10.1016/j.entcs.2004.04.042, https://www.sciencedirect.com/science/article/pii/S1571066105000447, proceedings of the 11th Workshop on Logic, Language, Information and Computation (WoLLIC 2004)

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

[25] Brazier, F., Dunin-Keplicz, B., Jennings, N., Treur, J.: Formal specification of multi-agent systems: A real-world case. pp. 25–32 (01 1995)

[26] Chowdhury, S.: Github gist page of neural.idr, https://gist.github.com/mrkgnao/a45059869590d59f05100f4120595623

[27] De Meester, B., Seymoens, T., Dimou, A., Verborgh, R.: Implementation-independent function reuse. Future Generation Computer Systems **110**, 946–959 (2020). https://doi.org/https://doi.org/10.1016/j.future.2019.10.006, https://www.sciencedirect.com/science/article/pii/S0167739X19303723

[28] Diehl, L.: Verified stack-based genetic programming via dependent types (2011)

[29] El-Desouky, A., Ali, H., Elghamrawy, S.: A proposed architecture for distributed multi-agent intelligent system (dmais) (04 2007)

[30] Elizarov, A., Kirillovich, A., Lipachev, E., Nevzorova, O.: Digital ecosystem ontomath: Mathematical knowledge analytics and management. In: Kalinichenko, L., Kuznetsov, S.O., Manolopoulos, Y. (eds.) Data Analytics and Management in Data Intensive Domains. pp. 33–46. Springer International Publishing, Cham (2017)

[31] Gibbons, J., Wu, N.: Folding domain-specific languages: Deep and shallow embeddings (functional pearl). In: Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming. p. 339–347. ICFP '14, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2628136.2628138, https://doi.org/10.1145/2628136.2628138

[32] 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

[33] Gruber, T.R., Olsen, G.R.: An ontology for engineering mathematics abstract (1994)

[34] Gruber, T.R., Tenenbaum, J.M., Weber, J.C.: Toward a knowledge medium for collaborative product development. In: In J. S. Gero (Eds.), Artificial Intelligence in Design '92. pp. 413–432. Kluwer Academic Publishers (1992)

[35] Hand, D.J., Khan, S.: Validating and verifying ai systems. Patterns **1**(3), 100037 (2020). https://doi.org/https://doi.org/10.1016/j.patter.2020.100037, https://www.sciencedirect.com/science/article/pii/S2666389920300428

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

[37] Javier Álvez, Paqui Lucio, G.R.: Evaluating automated theorem provers using adimen-sumo. In: Proceedings of the 3rd Vampire Workshop at the 8th International Joint Conference on Automated Reasoning (IJCAR 2016). Springer International Publishing, Coimbra, Portugal (2016)

[38] 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)

[39] Labrou, Y., Finin, T., Peng, Y.: The current landscape of agent communication languages. INTELLIGENT SYSTEMS **14**, 45–52 (1999)

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

[41] Menzies, T., Pecheur, C.: Verification and validation and artificial intelligence. Advances in Computers, vol. 65, pp. 153–201. Elsevier (2005). https://doi.org/https://doi.org/10.1016/S0065-2458(05)65004-8, https://www.sciencedirect.com/science/article/pii/S0065245805650048

[42] Nevzorova, O., Zhiltsov, N., Kirillovich, A., Lipachev, E.: *ontomath*^{PRO} ontology: A linked data hub for mathematics (2014)

[43] 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

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

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

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

[47] Pease, A., Sutcliffe, G.: First order reasoning on a large ontology. CEUR Workshop Proceedings **257**, 61–70 (Dec 2007), 21st CADE 2007 Workshop on Empirically Successful Automated Reasoning in Large Theories, ESARLT 2007 ; Conference date: 17-07-2007 Through 17-07-2007

[48] Piñeyro, L., Pardo, A., Viera, M.: Structure verification of deep neural networks at compilation time using dependent types. pp. 46–53 (09 2019). https://doi.org/10.1145/3355378.3355379

[49] Roelofs, M., Vos, R.: Automatically inferring technology compatibility with an ontology and graph rewriting rules. Journal of Engineering Design **32**, 1–25 (12 2020). https://doi.org/10.1080/09544828.2020.1860202

[50] Seshia, S.A., Sadigh, D.: Towards verified artificial intelligence. CoRR **abs/1606.08514** (2016), http://arxiv.org/abs/1606.08514

[51] Sojakova, K., Rabe, F.: Translating a dependently-typed logic to first-order logic. In: Corradini, A., Montanari, U. (eds.) Recent Trends in Algebraic Development Techniques. pp. 326–341. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)

[52] Urban, J.: An overview of methods for large-theory automated theorem proving (2011)

[53] Wadler, P.: Monads for functional programming. In: Broy, M. (ed.) Program Design Calculi. pp. 233–264. Springer Berlin Heidelberg, Berlin, Heidelberg (1993)

[54] Witherell, P., Krishnamurty, S., Grosse, I., Wileden, J.: Fidoe: A framework for intelligent distributed ontologies in engineering. pp. 685–697 (01 2009). https://doi.org/10.1115/DETC2008-50099

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