AI-DSL Technical Report (February to May 2021)

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

May 26, 2021

Abstract

Based on [2].

Contents

Chapter 1

Nil's work

Work done:

1. Implement `RealizedFunction` as described in [2].
2. Implement a network of trivially simple AI services implemented in Idris2, and use Idris compiler to type check if they can properly connect to each other.
3. Implement a Registry prototype, as a proof-of-concept for querying AI services based on their dependently typed specifications.

1.1 Realized Function

1.1.1 Description

The `RealizedFunction` data structure, as introduced in [2], 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 [2] as well.

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

- `Costs`: as a triple of three constants, `financial`, `temporal` and `computational`,
- `Quality`: as a single `quality` value.

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

nil
is there a word for that?

```
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.

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

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

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

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

1.1.2 Objectives and achievements

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

1.1.3 Future work

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

1.2 Network of Idris AI services

1.2.1 Description

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

1. `incrementer`: increment an integer by 1
2. `twicer`: multiply an integer by 2
3. `halfer`: divide an integer by 2

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

- `incrementer . halfer . twicer`

with the objective of testing that such compositions were properly typed. The networking part was implemented based on the SingularityNET example mentioned in the SingularityNET tutorial. 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 [**?**], 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 Protobuf [**?**]. Thus one may see the usage of Idris in that context as adding an enhanced refined verification layer on top of gRPC making use of the expressive power of dependent types.

1.2.2 Objectives and achievements

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

- `halfer . incrementer . twicer`

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

nil

Add ref to Sam's work

```
-- 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.

1.2.3 Future work

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

nil

Ref to Kabir's fake news detector

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

1.3 AI-DSL Registry

1.3.1 Description

One important goal of the AI-DSL is to have a system that can perform autonomous matching and composition of AI services, so that provided the specification of an AI, it should suffice to find it, complete it or even entirely build it from scratch. We have implemented a proof-of-concept *registry* to start experimenting with such functionalities.

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

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

1. `retrieve`: find AI services on the network fulfilling a given specification.
2. `compose`: construct composite services fulfilling that specification. Useful when no such AI services can be found.

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

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

instead of

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

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

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

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

nil

Add ref to Sam's work

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

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

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

outputs

```
Registry.retreive
```

which is itself. Likewise one can query

```
Integer -> Integer
```

which outputs

```
Incrementer.incrementer
```

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

```
Integer -> EvenInteger
```

outputting

```
Twicer.twicer
```

Or equivalently provide the unwrapped dependent type signature

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

retrieving the correct service again

```
Twicer.twicer
```

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

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

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

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

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

1.3.2 Objectives and achievements

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

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

1.3.3 Future work

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

- Use structured data structures to represent type signatures instead of String.
- Return a list of services instead of the first one.
- Implement `compose` for autonomous composition.
- Use real AI services instead of trivially simple ones.

Also, as of right now, the registry was implemented in Python, 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.

Chapter 2

AI-DSL Ontology (Kabir's work)

2.1 Description

2.1.1 Design requirements

At the beginning of the current iteration of the AI-DSL project we had a round of discussions about the high level functional and design requirements for AI-DSL and its role in SingularityNET platfrom and ecosystem. The discussions were based on [2, 4] and are available online in their original form. Here is the summary of the preliminary design requirements informed by those discussions:

- AI-DSL is a language that allows AI agents/services running on SinglarityNET platfrom to declare their capabilities and needs for data to other AI agents in a rich and versatile machine readible form; This will enable different AI agents to search, find data sources and other AI services whithout human interaction;
- AI-DSL ontology defines data and service (task) types to be used by AI-DSL. Requirements for the ontology are shaped by the scope and specification of the AI-DSL itself;

High level requirements for AI-DSL are:

Extendability The ontology of data types and AI task types should be extendable in the sense that individual service providers / users should be able to create new types and tasks and make them available to the network. AI-DSL should be able to ingest these new types / tasks and immediately be able to do the type-checking job. In other words, AI-DSL ontology of types / tasks should be able to evolve. At the same time, extended ontologies should relate to existing basic AI-DSL ontology in a clear way, allowing AI agents to perform reasoning across the whole space of available ontologies (which, at lower levels, may be globally inconsistent). In order to ensure interoperability of lower level ontologies, AI-DSL ontology will define small kernel / vocabulary of globally accessible grounded types, which will be built-in into the platform at the deep level. Changing this kernel will most probably require some form of voting / global consensus on a platfrom level.

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

- *The globally accessible vocabulary/root ontology of grounded types.* This vocabulary can be seen as immutable (in short and medium term) kernel. It should be extendible in the long term, but the mechanisms of changing and extending it will be quite complex, most probably involving theorectical considerations and/or a strict procedures of reaching global consensus within the whole platform (a sort of voting);
- *A decentralized ontology of types and tasks* which each are based (i.e. type-dependent) on the root ontology/vocabulary, but can be extended in a decentralized manner – in the sense that each agent in the platfrom will be able to define, use and share derived types and task definitions at its own discretion without the need of global consensus.

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

- The globally accessible vocabulary / root ontology of grounded types will not allow for competing versions. In a sense, this level will be the true ontology, representable by a one and unique root / upper-level ontology of the network which users will not be able to modify directly;
- All other types and task definitions within the platform will be required to be derived from the root ontology (if they will want to be used for interaction with other agents); However, the platform whould not restrict the number of competing versions or define a global consensus of types and task descriptions on this level.
- Furthermore, the ontology and the AI-DSL logic should allow for some variant of 'soft matching' which would allow to find the type / service that does not satisfy all requirements exactly, but comes as closely as available in the platform.

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

Code-level / service-level APIs. It is important to ensure that the ontology is readable / writable by different components of the SingularityNET platform, at least between AI-DSL engine / data structures and each AI service separately. This is needed because some of the required descriptors of AI services will have to be dynamically calculated at the time of calling a service and will depend on the immediate context (e.g. price of service, a machine on which it is running, possibly reputation score, etc.). It is not clear at this point how much of this functionality will be possible (and practical) to implement on available type-dependent, ontology languages or even if it is possible to use single language. Even it if is possible to implement all AI-DSL purely on the current type-dependent language choice Idris, it will have to interface with the world, deal with indeterministic input from network and mutable states – operations that may fail in run-time no matter how careful type checking is done during compile time [3].

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

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

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

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

2.1.2 Domain model considerations

In order to attend to all high level design requirements. all levels of AI-DSL Ontology should be developed simulatneously, so that we could make sure that the work is aligned with the function and role of AI-DSL within SingularityNET platform and ecosystem. We therefore use the "AI/computer-scientific" perspective to ontology and ontology building – emphasizing *what an ontologoy is for* – rather than the "philosophical perspecive" dealing with *the study of what there is in terms of basic categories* [5, 6]. Therefore we first propose the mechanism of how different levels (upper, domain and the leaf- (or service)) of AI-DSL ontology will relate for facilitating interactions between AI services on the platfrom.

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

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

NuNet is the platfrom enabling dynamic deployment and up/down-scaling of SingularityNET AI Services on decentralized hardware devices of potentially any type. Importantly for the AI-DSL project, service discovery on NuNet is designed in a way that enables dynamic construction of application-specific service meshes from several SingularityNET AI services[8]. In order for the service mesh to be deployed, NuNet needs only a specification of program graph of the application. Note, that conceptually, construction of an application from several independent containers is almost equivalent to functionality explained in section 1.3 on AI-DSL Registry, namely performance of matching and composition of AI services. This is the main reason why we chose *Fake News Warning* application as a domain model for early development efforts of AI-DSL.

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

Figure 2.1: Program graph as defined and used in NuNet's fake-news-warning app prototype a the time of writing.

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

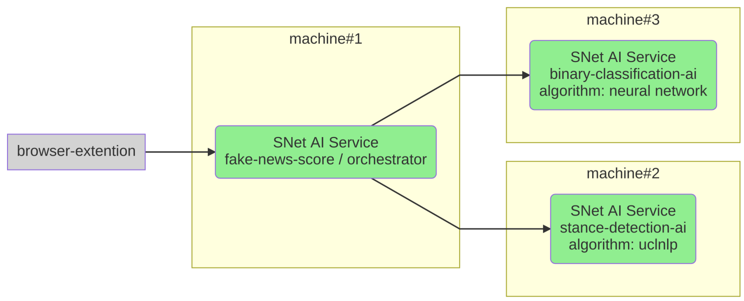


Figure 2.2: Schema of dependencies between components of the application.

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

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

2.1.3 Choice of existing ontologies

Based on:

1. discussion on https://github.com/singnet/ai-dsl/discussions/18 for the choice of SUMO and KIF;

2. Usage of:

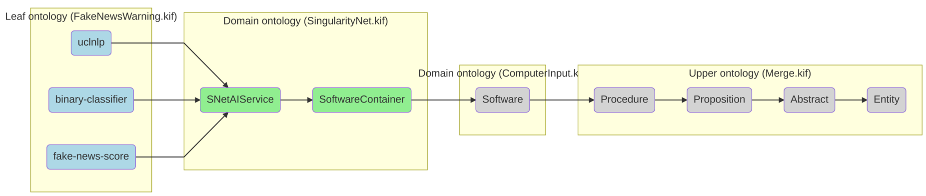
   - Upper level SUMO ontology (https://github.com/ontologyportal/sumo/blob/master/Merge.kif);
   - Middle level SUMO ontology (https://github.com/ontologyportal/sumo/blob/master/Mid-level-ontology.kif);
   - Distributed computing hardware domain ontology in SUO-KIF (https://github.com/ontologyportal/
   - https://github.com/allysonlister/swo in OWL. In the long term, it may be ideal to develop a converter for converting it to KIF, since OWL may be representable in KIF [7] using https://github.com/owlcs/owlapi; For the purpose of the ontology prototype, we will manually select parts of the this ontology in order to build the prototype and write them in SUO-KIF format;

2.1.4 Tools

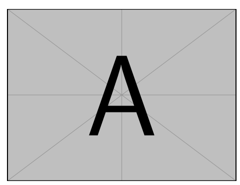
Intro to Sigma, SigmajEdit, etc. and how to install them.

2.2 Objectives and achievements

2.3 Relations between levels



(a) Class dependencies of the SNetAIService subclass within the ontology.



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

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

kabir: using ontology for agent communication in decentralized computing systems, based on [1]

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

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

2.4 Future work

2.4.1 Combining ontology with Idris

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

```
(subclass SNetAIService SoftwareContainer)
(documentation SNetAIService EnglishLanguage
"Software package exposed via SNetPlatfrom and conforming to the special packaging rules")

(subclass SNetAIServiceIO Descriptor)

(instance hasInput BinaryPredicate)
(domain hasInput 1 SNetAIService)
(domain hasInput 2 SNetAIServiceIO)

(instance hasOutput BinaryPredicate)
```

Figure 2.4: Contents of SingularityNet.kif

Chapter 3

Eman's work

Chapter 4

Sam's work

Bibliography

[1] Mastering agent communication in EMBASSI on the basis of a formal ontology. Tutorial and Research Workshop on Multi-Modal Dialogue in Mobile Environments, Fraunhofer Institute (2002)

[2] 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

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

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

[5] 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

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

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

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