AI-DSL Technical Report (February to May 2021)

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

June 3, 2021

Abstract

Based on [15].

Contents

Chapter 1

Nil's work

Work done:

1. Implement `RealizedFunction` as described in [15].
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 [15], 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 [15] as well.

For that iteration we have implemented a simple version of `RealizedFunction` and `RealizedAttributes` in Idris2 [4]. 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 [2].

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 [2].

1.1.3 Future work

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

1.2 Network of Idris AI services

1.2.1 Description

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

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

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

- `incrementer . halfer . twicer`

with the objective of testing that such compositions were properly typed. The networking part was implemented based on the SingularityNET example service [10] mentioned in the SingularityNET tutorial [13]. 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 [3], 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 [8]. 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 [2].

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 AI algorithms, ideally making full use of dependent types in their specifications. Such endeavor was actually attempted over the AI service collective described in Section 2.1.2, but it was eventually concluded to be too ambitious for that iteration and was postponed for the next.

An intermediary step, before moving to actual AI algorithms, is to explore multi-facet specifications and their interactions with other AI services. That is, a service A might output a data type satisfying a combination of properties, such as, to tie that back to the trivial incrementer, twicer and halfer combination, an integer that is both even and within a certain interval. Then two other services, B and C, may have partial constraints instead. For instance B may require an even integer, while C may require that integer to be within a certain interval. In other words both can take as input the output of A, for different reasons. It's obviously not conceptual difficult to cast the output of A to match the input types of B and C, however it is still something we need to explore in practice with Idris.

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 [12]) 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 [2].

1.3.2 Objectives and achievements

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

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

1.3.3 Future work

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

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

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

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

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 platform and ecosystem. The discussions were based on [15, 11] 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 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 [16].

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 platform 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 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* [17, 18]. 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[1]. In order for the service mesh to be deployed, NuNet needs only a specification of program graph of the application. Note, that conceptually, construction of an application from several independent containers is almost equivalent to functionality explained in section 1.3 on AI-DSL Registry, namely performance of matching and composition of AI services. This is the main reason why we chose *Fake News Warning* application as a domain model for early development efforts of AI-DSL. However, we use this domain model solely for the 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

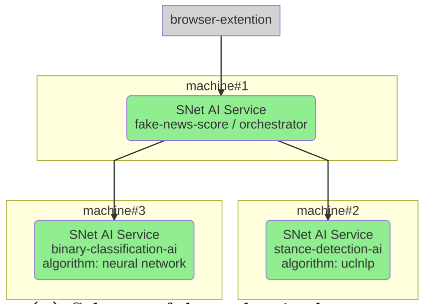
Table 2.1: Description of each component of `Fake News Warning` application.

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[9]. 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 1.2. Here is a simple representation of the program graph of *Fake News Warning* app:

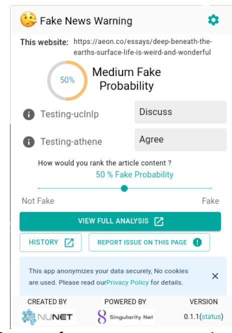
```
"dag": {
  "news-score" : ["uclnlp","binary-classification"]
  }
```

Figure 2.1: A directed acyclic graph (DAT) of *Fake News Warning* app prototype[6]. Meaning 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 this case is a Brave browser extension.



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



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

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

2.1.3 Ontology language and upper level ontology

After discussing several choices of ontology languages and reusing existing ontologies for designing AI-DSL ontology[3], we have opted to use SUO-KIF as an ontology language [24] and SUMO as an upper-level ontology [22]. 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[20]. Also, a conversion to Atomese – the OpenCog's language also employing a list-like syntax – has been successfully attempted in the past[4]. SUMO and the related ontology design tools [23] provide a convenient way for starting to design AI-DSL Ontology levels and their relations.

2.2 Objectives and achievements

2.2.1 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 incosistent 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.

[3]See Reusing Existing Ontologies discussion on AI-DSL Github repository[2]
[4]See the SUMO Importer in the OpenCog External Tools repository[7]

Component	Description	Dependencies	Extendability
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)
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 for 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.
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
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
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

Table 2.2: Description of each component of the AI-DSL Ontology prototope and links to related KIF files.

2.2.2 Ontology prototype

Using the ontology levels described in Table 2.2 and referenced files, we prototyped the ontology of *Fake News Warning* application.

kabir

include link to sigma server with the prototoype ontology installed (when ready)

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

Table 2.3: 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 2.3: 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

2.3 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 2.1) 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 2.4: Defining program graph as a formal ontology. This is equivalent to DAG of Figure 2.1.

Figure 2.4 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 2.5: 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 2.6: 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 2.7: 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 2.13), 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 2.8: Relation between SingularityNet and NuNet domain ontologies.

Figure 2.14 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 2.9: 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 2.16). 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 2.10: The definition of `NuNetEnabledComputer` in NuNet.kif requires available computing resources, computer type and operating system to be listed in the metadata.

A `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 2.11: 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		SingularityNet.kif
SNetAIService	Software package exposed via SNetPlatfrom and conforming to the special packaging rules	SingularityNet.kif
NuNetEnabledSNetAIService	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 2.4: 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.

kabir: Finished here 2021-06-03 03:55

2.2.3 The mechanism of dynamic workflow construction

1. Upper level SUMO ontology (Merge.kif);
2. Middle level SUMO ontology (Mid-level-ontology.kif);
3. Distributed computing hardware domain ontology in SUO-KIF (QoSontology.kif);

kabir: using ontology for agent communication in decentralized computing systems, based on [28]

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:

- In the long term, it may be ideal to develop a converter for converting it to KIF, since OWL may be representable in KIF [21] 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;

- Similarly we want to be able to convert SUO-KIF specifications into Idris, and vise versa, to take advantage of the strengths of each formalism. To the best of our knowledge there is no existing tools to automatically translating SUO-KIF to/from Idris, but there is a tool to translate SUO-KIF to FOL [25] and we have found a paper describing the translation from a Dependently Typed Language (DTL) to FOL [26]. Additionally, to start building understanding about that, we have manually ported the trivial AI services described in Section 1.3 to SUO-KIF, see TrivialServices.kif under the ontology of the AI-DSL repository [2]. As it turns out writing formal specifications of functions in SUO-KIF is reasonably straight forward. Here's the SUO-KIF implementation of the Twicer service to get a idea

```
(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 over the dependent type `Vect` [5] to SUO-KIF might not be as trivial, since the `domain` and `range` constructs cannot presumably represent such type dependence. This however 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) [14, 27, 19], 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 1.3 that Idris can fulfill that purpose.

2.2.4 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.

```
(instance uclnlp NuNetEnabledSNetAIService)

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

Figure 2.12: 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 2.13: 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 2.13), 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 2.14: Relation between SingularityNet and NuNet domain ontologies.

Figure 2.14 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 2.15: 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 2.16). 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 2.16: The definition of `NuNetEnabledComputer` in NuNet.kif requires available computing resources, computer type and operating system to be listed in the metadata.

A `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 2.17: 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		SingularityNet.kif
SNetAIService	Software package exposed via SNetPlatfrom and conforming to the special packaging rules	SingularityNet.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 2.5: 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.

kabir: Finished here 2021-06-03 03:55

2.2.5 The mechanism of dynamic workflow construction

1. Upper level SUMO ontology (Merge.kif);
2. Middle level SUMO ontology (Mid-level-ontology.kif);
3. Distributed computing hardware domain ontology in SUO-KIF (QoSontology.kif);

kabir: using ontology for agent communication in decentralized computing systems, based on [28]

2.3 Future work

- In the long term, it may be ideal to develop a converter for converting it to KIF, since OWL may be representable in KIF [21] 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;

- Similarly we want to be able to convert SUO-KIF specifications into Idris, and vise versa, to take advantage of the strengths of each formalism. To the best of our knowledge there is no existing tools to automatically translating SUO-KIF to/from Idris, but there is a tool to translate SUO-KIF to FOL [25] and we have found a paper describing the translation from a Dependently Typed Language (DTL) to FOL [26]. Additionally, to start building understanding about that, we have manually ported the trivial AI services described in Section 1.3 to SUO-KIF, see TrivialServices.kif under the ontology of the AI-DSL repository [2]. As it turns out writing formal specifications of functions in SUO-KIF is reasonably straight forward. Here's the SUO-KIF implementation of the Twicer service to get a idea

  ```
  (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 over the dependent type `Vect` [5] to SUO-KIF might not be as trivial, since the `domain` and `range` constructs cannot presumably represent such type dependence. This however 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) [14, 27, 19], 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 1.3 that Idris can fulfill that purpose.

Chapter 3

Eman's work

Chapter 4

Sam's work

4.1 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.

4.1.1 `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.

4.1.2 `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.

4.1.3 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.

As a test to

Bibliography

[1]

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

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

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

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

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

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

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

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

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

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

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

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

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

[15] 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

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

[17] 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

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

[19] 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)

[20] 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)

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

[22] 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

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

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

[25] 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

[26] 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)

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

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