From Category Theory to Functional Programming: A Formal Representation of Intent

Davide Borsatti
CIRI - ICT
University of Bologna, Italy
davide.borsatti@unibo.it

Walter Cerroni
Dept. of Electrical, Electronic and Information Engineering
University of Bologna, Italy
walter.cerroni@unibo.it

Stuart Clayman
Dept. of Electronic Engineering
University College London, London, UK
s.clayman@ucl.ac.uk

2022 IEEE 8th International Conference on Network Softwarization (NetSoft) | 978-1-6654-0694-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/NetSoft54395.2022.9844061

***Abstract*—The possibility of managing network infrastructures through software-based programmable interfaces is becoming a cornerstone in the evolution of communication networks. The Intent-Based Networking (IBN) paradigm is a novel declarative approach towards network management proposed by a few Standards Developing Organizations. This paradigm offers a high-level interface for network management that abstracts the underlying network infrastructure and allows the specification of network directives using natural language. Since the IBN concept is based on a declarative approach to network management and programmability, we argue that the use of declarative programming to achieve IBN could uncover valuable insights for this new network paradigm. This paper proposes a formalization of this declarative paradigm obtained with concepts from category theory. Taking this approach to Intent, an initial implementation of this formalization is presented using Haskell, a well-known functional programming language.**

***Index Terms*—Intent-Based Networking, Functional Programming, Haskell, Category Theory**

I. Introduction

The so-called *network softwarization* process represents a shift that has been taking place in the last decade towards the unprecedented and dominant role of software in communication networks. Among the many advantages that such an approach can bring to network and communication service management, one of the most relevant features is network *programmability*, i.e. the ability to view the network infrastructure, as well as the computing resources involved in service delivery, as a general purpose entities that can receive instructions by means of Application Programming Interfaces (APIs). Such APIs are typically offered by northbound interfaces of existing network controllers and/or service orchestrators, and as such the level of abstraction they provide depends on the specific solution adopted by the underlying platforms.

Intent-Based Networking (IBN) is emerging as one of the key technologies for abstracting network management and programmability operations. Through IBN, an automated network management platform can deploy a desired (or intended) state and enforce the required policies without having to detail specific steps and operational procedures. Then, once a given intent is enforced, an IBN system is supposed to continuously monitor its state and verify its consistency with the initial requirements, applying suitable corrective actions if needed. Within the Internet Research Task Force (IRTF), the Network Management Research Group (NMRG) defined an intent as "a set of operational goals (that a network should meet) and outcomes (that a network is supposed to deliver), defined in a *declarative* manner without specifying how to achieve or implement them" [1]. Therefore, intents are inherently a flexible and declarative way to express and compose network operations, and to program network infrastructures.

Declarative approaches to specifications and programming have been studied in computer science for some considerable time. Landin's paper [2] from 1966 started a trend that is still appropriate today. Declarative programs express *what* is to be computed rather than *how* it is to be computed. Such programs are made up of expressions rather than sequences of commands as in imperative programming. With this paradigm, repetitive executions are accomplished by recursion rather than by sequences of operations. There are two main types of declarative languages exist:

- *Functional (or applicative) languages*, in which the underlying model of computation is the mathematical concept of a function. During a computation, a function is applied to zero or more arguments to obtain a single result. In other words, the result is deterministic (or predictable) since a function will always return the same value if called with the same arguments. The most common purely functional programming language is Haskell, with Erlang, Scheme, and Common Lisp having a functional core.
- *Relational (or logic) languages*, in which the underlying model of computation is the mathematical concept of a relation (or a predicate). A computation is the (non-deterministic) association of a group of values with backtracking to resolve additional values. The main example of a logic programming language is Prolog.

Since the IBN concept is based on a declarative approach to network management and programmability, we argue that the use of declarative programming to achieve IBN could uncover valuable insights for this new network paradigm. In particular, we consider functional programming for IBN thanks to its mathematical foundation that enforces a rigorous approach to the design and implementation of the program. The notation used in functional languages is very close to that used in formal methods [3], hence any system designed using these methods can be implemented very rapidly as functional languages are often considered as executable specification languages [4], particularly with its close link to *category theory*. Category theory is a branch of mathematics that provides a powerful methodology to describe and work on abstract concepts, including math itself. For this reason, we believe that category theory could represent a useful mathematical tool and foundation to conceive an IBN system. In this work, we present a formal description of the IBN problem using tools from category theory. We then go on to show how this formalization can be implemented in Haskell, a well-known functional programming language.

The paper is organized as follows. Section II introduces related work in the fields of Applied Category Theory and IBN. Section III presents foundation concepts and definitions of category theory, then Section IV defines a category theory framework for expressing intents in an IBN system. Section V presents the connection between the theoretical concepts and their actuation by providing a preliminary implementation written in Haskell. Conclusions and future works are summarized in Section VI.

II. Related Work

Functional programming has been used by many researchers because of its theoretical basis and its mathematical aspects [5], and because functional programs are amenable to automatic machine-based reasoning. This area includes automatic program transformation [6], automatic program proving [7], and formal semantics [8]. Applied Category Theory is becoming a relevant field in research, by showing how category theory can be applied to different fields outside of "pure mathematics". For example, in [9] Coecke, Sadrzadeh, and Clark applied category theory to natural language processing, defining a model that characterizes natural language expressions and their meaning leveraging tools from category theory. These concepts are also adopted in the area of modeling cyber-physical systems [10] [11].

Recently, several research efforts investigated IBN and surveyed its different aspects [12] [13]. In particular, Jacobs et al. propose a process for intent refinement [14], which uses AI to process intent requests expressed in natural language and transforms them into an intermediate format called *Nile*, that can be fed back to the operator / user to be validated before its deployment. The same formalism is used and extended in [15] to cover a broader set of use cases, specifically the ones related to traffic rerouting and service traffic protection. The functional programming formalism presented in this work would still be valid to describe the data models proposed in the aforementioned approaches.

III. Category Theory and Functional Programming

The strength of category theory lies in the capability of reasoning with abstract concepts. In detail, it focuses on the relationships existing between objects rather than what these objects are. Furthermore, it can regulate how a "type" of object can be mapped to another one. Following the definition given by Fong and Spivak in [16], a *category* $\mathcal{C}$ is a collection of objects $Ob(\mathcal{C})$ such that:

i for each pair of objects $c, d \in Ob(\mathcal{C})$, a set $\mathcal{C}(c, d)$ is specified including elements called *morphisms from c to* d, or morphisms in $\mathcal{C}$;

ii for each object $c \in Ob(\mathcal{C})$, the *identity morphism on* c is specified as $id_c \in \mathcal{C}(c, c)$;

iii for any three objects $c, d, e \in Ob(\mathcal{C})$ and for any morphisms $f \in \mathcal{C}(c, d)$ and $g \in \mathcal{C}(d, e)$, the *composite morphism of* f *and* g is specified as $f \circ g \in \mathcal{C}(c, e)$.

A morphism $f \in \mathcal{C}(c, d)$ can be also denoted as $f : c \to d$. Here, c is called the domain of f and d is called the codomain of f. These constituents are required to satisfy two conditions:

a *unitality*: for any morphism $f : c \to d$, the composite morphism of f and any of the identity morphisms on c or d returns f, i.e.: $id_c \circ f = f$ and $f \circ id_d = f$.

b *associativity*: for any three morphisms $f : c_0 \to c_1$, $g : c_1 \to c_2$, and $h : c_2 \to c_3$, it holds: $(f \circ g) \circ h = f \circ (g \circ h) = f \circ g \circ h$.

Another important element is the concept of *functor*. A functor maps all objects of a category $\mathcal{C}$ to objects of a category $\mathcal{D}$, while preserving its structure (i.e., identities and composition). More formally from [16], let $\mathcal{C}$ and $\mathcal{D}$ be categories. A functor F from $\mathcal{C}$ to $\mathcal{D}$, denoted as $F : \mathcal{C} \to \mathcal{D}$, is such that:

i for every object $c \in Ob(\mathcal{C})$, an object $F(c) \in Ob(\mathcal{D})$ can be specified;

ii for every morphism $f : c_1 \to c_2$ in $\mathcal{C}$, a morphism $F(f) : F(c_1) \to F(c_2)$ in $\mathcal{D}$ can be specified.

A functor must satisfy two properties:

a for each object $c \in Ob(\mathcal{C})$, it holds $F(id_c) = id_{F(c)}$;

b for any three objects $c_1, c_2, c_3 \in Ob(\mathcal{C})$ and for any morphisms $f \in C(c_1, c_2)$ and $g \in C(c_2, c_3)$, the equation $F(f \circ g) = F(f) \circ F(g)$ holds in $\mathcal{D}$.

The link between Haskell, or functional programming in general, and category theory might not be easy to see. However, it is possible to construct a category **Hask** [17] in which $Ob($**Hask**$)$ contains all Haskell data types (e.g., `Int`, `Bool`, etc.) and morphisms between these are function between types (e.g., $isEven :: Int \to Bool$). This construction can be considered a category, up to some minor approximation. By considering **Hask** as a category we can of course use all the other constructions defined in category theory (e.g., functors, monads, etc.). For example, a new type definition in Haskell could be seen as an endofunctor on **Hask** (a functor from **Hask** to **Hask**), since it maps types to a new type and preserves morphisms (functions) between the starting types.

Therefore, given the wide variety of intents that could exist, with very different scopes and requirements, we thought that this kind of formalism would be beneficial to abstract these details while focusing on the transformations they need to undergo before being enforced in a network infrastructure (as described in the Intent Lifecycle [1]).

IV. The Intent Category

The focus of this work is to formalize an approach for an IBN system using categorical tools, to then be implemented with a functional language, namely Haskell. First, a category to represent intents is defined as $\mathcal{I}$. For this work, we will take into account only natural language-based intents. Nonetheless, the same considerations would still apply to other classes of intents (e.g., machine to machine intents). The objects of this category $Ob(\mathcal{I})$ can be seen as a subset of all possible sentences in English (or even in other languages) that express an intent. In other words, the objects are all possible well-formed intent expressions related to network management that can be constructed, e.g. using natural language. Morphisms can be defined to describe the relationships between "similar" intent requests. Specifically, an ordering relationship can be introduced between elements of this set, represented by the symbol $\leq$. As an example, intent I_1 can be "Deploy low-latency Service X" and I_2 can be "Deploy Service X": since the deployment of a service X with a low-latency requirement naturally *implies* the deployment of service, then $I_1 \leq I_2$.

We can prove that this construction is actually a category:

- For each intent I_i in the object set, we have $I_i \leq I_i$. This is trivial since it is clear that an Intent implies itself (identity morphism).
- For any three intents I_1, I_2, I_3 in the object set such that $I_1 \leq I_2$ and $I_2 \leq I_3$, we have $I_1 \leq I_3$. In other words, if I_1 implies I_2 and I_2 implies I_3, then of course I_1 implies I_3 (composition rule).

Furthermore, this category is a partially ordered category. It is ordered because, if a morphism exists between two objects, then they are related as described, and that morphism is unique. The ordering is partial since there might be objects that cannot be related with others.

Inside the intent category, a *product* operator $\otimes$ can be defined acting between objects. This operator could be used to link different intent expressions, resembling a logical `and`. For example, let intent I_1 be "Deploy a web server" and I_2 "Deploy a firewall", then $I_1 \otimes I_2$ would be "Deploy a web server *and* deploy a firewall". The product could be used to build a structure inside the category in which complex intent requests are linked to their constituent components through this operator. By defining an identity object for this operator, it is possible to prove that the intent category equipped with the intent product is a "monoidal preorder" (i.e., a free category obtained from a preorder set equipped with a monoidal product, see Sections 3.2.3 and 4.4.4 in [16]). The identity object for this operator should represent an intent request that, if multiplied or logically linked to any other intent, the resulting intent would not change. This identity object would be a form like a *Null Operation* intent. To prove that $\otimes$ is actually a monoidal operator, the following properties must hold:

- **Monotonicity:** For all $x_1, x_2, y_1, y_2 \in Ob(\mathcal{I})$, if $x_1 \leq y_1$ and $x_2 \leq y_2$, then $x_1 \otimes x_2 \leq y_1 \otimes y_2$.
- **Unitality:** Let id_I be the identity object for $\otimes$, then for all $x \in Ob(\mathcal{I})$ the left and right identities hold: $id_I \otimes x = x \otimes id_I = x$.
- **Associativity:** For all $x, y, z \in Ob(\mathcal{I})$, $(x \otimes y) \otimes z = x \otimes (y \otimes z)$.

Now consider an example of the first property, with $x_1 =$ "Deploy low-latency Service X", $y_1 =$ "Deploy Service X", $x_2 =$ "Activate a firewall between 8 am and 10 pm", $y_2 =$ "Activate a firewall". Following the definition, $x_1 \otimes x_2$ and $y_1 \otimes y_2$ would be "Deploy low-latency Service X *and* activate a firewall between 8 am and 10 pm" and "Deploy Service X *and* activate a firewall", respectively. Since $x_1 \leq y_1$ and $x_2 \leq y_2$, it is easy to see that also $x_1 \otimes x_2 \leq y_1 \otimes y_2$, thus satisfying the **monotonicity** property. Similar demonstrations can be given for **unitality** and **associativity**. An additional property that is of interest is **symmetry**, meaning that given $x, y \in Ob(\mathcal{I})$ it holds $x \otimes y = y \otimes x$.

Another definition of intent product could be also obtained through *universal construction*. It's an approach used in category theory to derive the properties of an object by looking at its relationships with other ones. For any Intent I_i having two projections towards I_1 and I_2, meaning I_i implies both I_1 and I_2, it exists an object $I_1 \otimes I_2$ such that there is a unique morphism from I_i to $I_1 \otimes I_2$ that makes the two "triangles" in Fig. 1 to commute (see Definition 3.71 in [16]). To simplify, $I_1 \otimes I_2$ could be seen as the "best object" having projections towards I_1 and I_2.

$$\begin{tikzcd}
I_i \arrow[r] \arrow[d] \arrow[dr, Rightarrow] & I_1 \\
I_2 & I_1 \otimes I_2 \arrow[l] \arrow[u]
\end{tikzcd}$$

Fig. 1. Commuting diagram for categorical product. The arrows in the figure represent implications between Intents.

This can be interpreted as if objects I_i are all the possible intent specifications requiring a service composed by the ones requested by I_1 and I_2, being $I_1 \otimes I_3$ the way of describing the composite service in which its components are "easier" to identify. An example may help clarify this: let I_1 be "Deploy a web server" and I_2 "Deploy a firewall", then $I_1 \otimes I_2$ would be "Deploy a web server *and* a firewall". In this example the "I_is" would be expressions like "Deploy a secured HTTP server" or "Deploy an HTTP server and secure it".

Having defined the category $\mathcal{I}$ representing the intent requests, the next step is to define a category $\mathcal{S}$ representing the specific services an intent requires. The intent category could then be linked to this new category $\mathcal{S}$ through a functor, which would map all objects from $\mathcal{I}$ to objects in $\mathcal{S}$, while preserving the structure of the starting category. This means that, if $F : \mathcal{I} \to \mathcal{S}$ is the functor between the two categories and $I_1, I_2 \in Ob(\mathcal{I})$ are such that $\exists\ f : I_1 \to I_2$, then $\exists\ F(f) : F(I_1) \to F(I_2)$. The objects of this category represent all the services that can be requested by an intent, while morphisms between these objects could embed composition rules between services. Examples of these composition rules will be given in Section V.

A similar approach could be followed to define another category $\mathcal{R}$ representing the service requirements. This category will embed all the modifiers that a particular intent could request, for instance specific Quality of Service (QoS) values to satisfy (e.g., minimum bandwidth, maximum latency, etc.) or given time periods in which the intent must be enforced (e.g., "every day", "all Mondays", "only between 8 am and 10 am)". Also in this case a functor from $\mathcal{I}$ to $\mathcal{R}$ can be defined, with the same properties as above.

A key aspect that is worth highlighting is that no details were given on the internal structure of categories $\mathcal{S}$ and $\mathcal{R}$, but only on the intent one. By leveraging the abstraction granted by category theory's tools, it is possible to reason and define a structure at the "natural language level" (i.e., at the intent level). This structure is preserved in the linked categories (i.e., at the service and requirement levels) through categorical relationships (e.g., functors), without the need of "looking inside" them. In other words, we could say that the services and requirements categories can be seen as intermediate steps between an intent reception and its actuation on the system, i.e. the Translation/Intent-Based System (IBS) Space in the intent lifecycle [1]. However, we do not need to define precise data models for these intermediate steps to derive their properties, since they are inherited from the structure of intents expressed in natural language.

In this view, the functors going from the intent category $\mathcal{I}$ to categories $\mathcal{S}$ and $\mathcal{R}$ can be seen as "meaning extracting" functions, i.e. extracting requested services and their requirements from natural language intent expressions.

V. Implementation of Categories in Haskell

Having defined the categories previously, and seen how they relate to each other, the next step is to consider each one of them and to define a syntax to start building the bridge between the theoretical construct and a first implementation in Haskell.

The objects in $\mathcal{I}$ could be described with a type defined as:

```
data Intent = Intent String
            | NullOperation
```

This example could also be used to explain how types can be defined in Haskell. The first `Intent` keyword here is a *type constructor* which identifies the type name. The other two keywords on the right side of the equality sign, `Intent` and `NullOperation`, are the data constructors, which specify how data of type `Intent` can be constructed. For this example, a value of type `Intent` can be constructed using either `NullOperation` without any parameter or `Intent` followed by a `String`. Here are two examples: `intent1 = Intent "Deploy a firewall"` or `nullOpIntent = NullOperation`. In other words, an object of the category $\mathcal{I}$ is either a string (i.e., an actual intent request expressed in natural language) or a null operation (i.e., the identity object defined for the internal monoidal product).

Then a similar type definition could be given for the objects of the $\mathcal{S}$ category:

```
data Action = Service (Phase,BasicService)
            | TemporalCompose [Action]
            | LogicalCompose [Action]
            | NoAction
```

This is a recursive data type, a common type definition in functional programming. The new type `Action` can be a single `Service`, specified through a pair of new data types (`Phase` and `BasicService`) that will be described later, or by a composition of a list of `Action`. Specifically, two different kinds of composition are defined, temporal and logical. The former represents a list of actions that need to be executed in a specific temporal order, e.g., deploy a firewall *and after* configure it. The latter describes an action that depends on a series of other logical components without a given order, e.g., the deployment of a 5G core depends on the deployment of a set of other functions such as AMF, SMF, PCF, etc. Lastly, the type constructor `NoAction` represents an "empty" `Action` object, i.e., an action that does nothing on the system, similar to empty list constructor `Nil` in classic recursive list definitions [3]. The `BasicService` type is used to represent basic services or actions. The idea here is that this new type definition should contain all possible "building blocks" that can be composed (temporally or logically) together to construct a complex service required by an intent. A basic definition of this new type is:

```
data BasicService = VnfId String
                  | RouterConfig (String,
                                  String)
                  | PathCreate (String,
                                String)
```

Of course, this type could be extended with other basic services that users may require (i.e., adding a new data constructor), or by using more complex structures as input to them. Finally, the `Phase` type simply defines the phase of the lifecycle in which that action will take place. A basic definition of this new type is:

```
data Phase = Add
           | Update
           | Restart
           | Remove
```

Having defined objects in $\mathcal{S}$, the next step would be to define the structure of this category, in other words, its morphisms. Morphisms can describe how services may be composed by other ones. For example, let $x, y \in Ob(\mathcal{S})$ with $x =$ "Deploy a 5G Core Network and $y =$ "Deploy an SMF function", then exists a morphism $f : x \to y$ that specifies that y is a component of x. To clarify, the definition of x and y used in the example is not completely correct, since they do not follow the formal definition given above. However, a functor from $\mathcal{I}$ to $\mathcal{S}$ has been defined, therefore each object in the former category is mapped to an object of the latter. For this

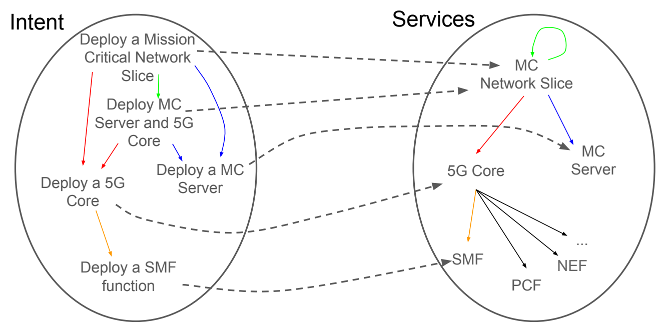

Fig. 2. Graphical representation of the functor between the two categories of intents $\mathcal{I}$ and services $\mathcal{S}$.

reason, x and y can be seen as the objects that are mapped by intents "Deploy a 5G Core Network" and "Deploy an SMF function", respectively.

Figure 2 is presented to better understand what the functor does between two categories $\mathcal{I}$ and $\mathcal{S}$. The dashed lines connect intents to the objects in $\mathcal{S}$, or in other words to the services they are requesting. The same color has been used to highlight how the functor maps morphisms between the two categories. From Fig. 2 it is possible to gain an insight into what is necessary to satisfy an intent request by taking into account the sub-tree having as root the mapped service. In other words, by knowing how to deploy the leaves of these trees and how to compose them, it should be possible to deploy the required service. For instance, the leaves in Fig. 2, using the type definition introduced above, can be defined as follows:

```
smf = Service (Add, VnfId "smfId")
```

meaning variable `smf` of type `Action` is defined using its data constructor `Service` since we can consider it as a single VNF, thus not requiring any composition. This data constructor takes as an argument a pair of objects. One of type `BasicService`, which was constructed using its data constructor `VnfId` followed by a string representing the identifier given to that specific function, `"smfId"` in the example. The other of type `Phase` built with its data constructor `Add` since the intent in the example is requiring the deployment of a function. The same procedure can also be applied to the other leaves represented in the figure, of course with the correct VNF identifiers. Then these leaves can be composed together to construct, for example, the object representing a 5G core deployment:

```
5gCore = LogicalCompose[
           (Service (Add, VnfId "smfId")),
           (Service (Add, VnfId "pcfId")),
           ...]
```

Here, the data constructor `LogicalCompose` is used to identify the list of VNFs that are part of the 5G core service (e.g., SMF, PCF, etc.). The logical composition is simply grouping them together without a particular order in which the VNFs need to be deployed (i.e., in NFV-MANO terminology, a network service composed of several VNFs without any specific Virtual Network Forwarding Graphs).

Of course, similar type definitions could be given for the $\mathcal{R}$ category as well. In this case, objects can be seen as a composition of basic "intent modifiers", such as: *Scope*, to whom the intent is targeted (e.g., network-wide, single user); *Time*, when the intent should be active (e.g. "everyday from 2pm to 5pm", "always"); *Latency* constraints, *Throughput* constraints. Using Haskell notation:

```
data SrvRequirement = Requirement BasicReq
                    | ComposeReq
                      [SrvRequirement]
                    | NoReq

data BasicReq = Latency _
              | Bandwidth _
              | Scope _
              | Uptime _}
```

As for the other data types, also `BasiqReq` could be expanded to cover a larger set of requirements. After the definition of these new types, the ordering has to be defined. This ordering will map the relationship between two variables of the same type, embedding the concept of morphism between objects in the categories from which the types derive. Since the categories were partially ordered, using the class `Ord` defined in Haskell's type system would not be the correct choice, as it is for total orderings. Therefore a new class has to be defined for them. Since in a partial order two objects may be related (e.g., less than, equal to, greater than) or not, a possible way to program this in Haskell is using a construct like `Maybe Ord.Ordering`. In other words, if the relationship exists the comparison of the two variables returns `Just Ord` (e.g., LT, EQ, GT), otherwise `Nothing`. An object of these new types would be less than another one if it is a component of the latter.

Having defined the types describing the three categories considered, the next step is the definition of the functors connecting them. In Haskell, this means defining functions taking as input objects of type `Intent` and returning `Action` or `SrvRequirement`. These functions should take natural language expressions and construct the tree representing the network operations they are asking for and their requirements. Finally, new functions are needed to enforce the requirements expressed in these new types on the underlying infrastructure.

A. Evaluation

To start testing these operations, we implemented a preliminary version of one actuation function, specifically, the one able to construct the Open Source Mano (OSM) Network Service Descriptor (NSD) required by a specific action. In detail, if the function is called with as input an action like `Service(Add, VnfId "vndfid")`, the Vnf identifier is added to the required fields of an OSM NSD data model, namely in the `vnfd-id` and `vnf-profile` lists. Alternatively, if the input of the function is a logical composition of a set of VNFs, then using a fold function (`foldr`) the behavior just described is applied recursively to all the elements in the list, accumulating all the results in the same network service descriptor. In a Haskell-like syntax:

```
generateNsd :: Action -> Nsd -> Nsd
generateNsd (Service (Add, VnfId vnfid))
   nsd = [...]
generateNsd (LogicalCompose actions) nsd =
   foldr generateNsd nsd actions
```

For example, assuming an input action like:

```
LogicalCompose[
        (Service (Add, VnfId "smfId")),
        (Service (Add, VnfId "pcfId"))]
```

the function constructs a valid OSM NSD containing the required VNFs. The relevant components of the descriptor generated from the previous example are:

```
nsd:
  nsd:
    df:
    - id: default-df
      vnf-profile:
      - id: pcfid
        virtual-link-connectivity:
        - constituent-cpd-id:
          - constituent-base-element-id:
              pcfid
            constituent-cpd-id: mgmt-ext
          virtual-link-profile-id: mgmtnet
        vnfd-id: pcfid
      - id: smfid
        virtual-link-connectivity:
        - constituent-cpd-id:
          - constituent-base-element-id:
              smfid
            constituent-cpd-id: mgmt-ext
          virtual-link-profile-id: mgmtnet
        vnfd-id: smfid
    vnf-id:
    - pcfid
    - smfid
    [...]
```

The NSD descriptor created with this function can then be onboarded on the OSM platform, and its deployment can be initialized. Furthermore, thanks to the type checking features of Haskell, the NSD format is automatically validated during every parsing and rendering of the YAML file, thus increasing the robustness of the code.

VI. Conclusion and Future Works

This work has directly applied category theory concepts to IBN in order to build a formal representation of the intent specifications. This representation aims to help with the reasoning about this new paradigm, while keeping a close relationship with functional programming and the possible implementations. In this paper we have shown a preliminary implementation of the categories for intent using Haskell. This implementation is able to take a definition of a service and constructs a valid OSM NSD containing the required VNFs. The relevant components of the descriptor are generated.

Further works can be made in both directions. For example, lenses in category theory [18] are used to describe in mathematical terms the concepts of *agent* and *environment*, thus they could serve as a valid tool to model the interaction between the Intent System and the network infrastructure it is managing. They are also widely used in Haskell, being one of the most powerful tools to access and modify data structures.

References

[1] A. Clemm, L. Ciavaglia, L. Z. Granville, and J. Tantsura, "Intent-Based Networking - Concepts and Definitions," IETF, Internet-Draft draft-irtf-nmrg-ibn-concepts-definitions-06, 12 2021. [Online]. Available: https://datatracker.ietf.org/doc/draft-irtf-nmrg-ibn-concepts-definitions

[2] P. J. Landin, "The Next 700 Programming Languages," *Communications of the ACM*, vol. 9, no. 3, p. 157–166, Mar 1966. [Online]. Available: https://doi.org/10.1145/365230.365257

[3] J. Hughes, "Why Functional Programming Matters," *Computer Journal*, vol. 32, no. 2, pp. 98–107, 1989.

[4] D. Turner, "Functional Programs as Executable Specifications," *Mathematical Logic and Programming Languages, ed. P. Sheperdson, Prentice Hall*, 1984.

[5] J. A. Stoy, "Some Mathematical Aspects of Functional Programming," *Functional Programming and its Applicationsm ed D.A. Turner, Cambridge University Press*, 1980.

[6] J. Darlington, P. G. Harrison, H. Khoshnevisan, L. McLoughlin, N. Perry, H. Pull, M. Reeve, K. Sephton, R. L. While, and S. Wright, "A functional programming environment supporting execution, partial execution and transformation," in *Proceedings of the Parallel Architectures and Languages Europe, Volume I: Parallel Architectures*, ser. PARLE '89. Berlin, Heidelberg: Springer-Verlag, 1989, p. 286–305.

[7] D. Turner, "Functional programming and proofs of program correctness," *Tools and Notions for Program Construction, ed. D. Neel, Cambridge University Press*, 1982.

[8] D. Schmidt, "Denotational Semantics - A Methodology for Language Development," 1986.

[9] B. Coecke, M. Sadrzadeh, and S. Clark, "Mathematical foundations for a compositional distributional model of meaning," 2010.

[10] A. Speranzon, D. I. Spivak, and S. Varadarajan, "Abstraction, composition and contracts: A sheaf theoretic approach," *CoRR*, vol. abs/1802.03080, 2018.

[11] G. Bakirtzis, C. Vasilakopoulou, and C. H. Fleming, "Compositional cyber-physical systems modeling," *Electronic Proceedings in Theoretical Computer Science*, vol. 333, p. 125–138, Feb 2021. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.333.9

[12] L. Pang, C. Yang, D. Chen, Y. Song, and M. Guizani, "A survey on intent-driven networks," *IEEE Access*, vol. 8, pp. 22 862–22 873, 2020.

[13] E. Zeydan and Y. Turk, "Recent advances in intent-based networking: A survey," in *2020 IEEE 91st Vehicular Technology Conference (VTC2020-Spring)*, 2020.

[14] A. S. Jacobs, R. J. Pfitscher, R. A. Ferreira, and L. Z. Granville, "Refining network intents for self-driving networks," in *Proceedings of the Afternoon Workshop on Self-Driving Networks*, ser. SelfDN 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 15–21. [Online]. Available: https://doi.org/10.1145/3229584.3229590

[15] M. Bezahaf, E. Davies, C. Rotsos, and N. Race, "To all intents and purposes: Towards flexible intent expression," in *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*, 2021, pp. 31–37.

[16] B. Fong and D. I. Spivak, "Seven sketches in compositionality: An invitation to applied category theory," 2018.

[17] Category hask. [Online]. Available: https://wiki.haskell.org/Hask

[18] D. Spivak. Lenses: applications and generalizations. [Online]. Available: http://math.ucr.edu/home/baez/ACTUCR2019/ACTUCR2019_spivak.pdf