Da teoria das categorias à programação funcional: Uma representação formal da intenção

Davide Borsatti
CIRI - Universidade TIC de Bolonha, Itália davide.borsatti@unibo.it

Walter Cerroni
Departamento de Engenharia Elétrica, Eletrônica e da Informação Universidade de Bolonha, Itália walter.cerroni@unibo.it

Stuart Clayman
Departamento de Engenharia Eletrônica University College London, Londres, Reino Unido s.clayman@ucl.ac.uk

Resumo—A possibilidade de gerenciar infraestruturas de rede por meio de interfaces programáveis baseadas em software está se tornando um marco na evolução das redes de comunicação. O O paradigma Intent-Based Networking (IBN) é uma nova abordagem declarativa para o gerenciamento de rede proposta por algumas organizações de desenvolvimento de padrões. Este paradigma oferece uma interface de alto nível para gerenciamento de rede que abstrai a infraestrutura de rede subjacente e permite a especificação de diretivas de rede usando linguagem natural. Uma vez que o conceito IBN é baseado em uma abordagem declarativa para gerenciamento e programabilidade de rede, argumentamos que o uso de programação declarativa para alcançar IBN poderia revelar informações valiosas para este novo paradigma de rede. Este artigo propõe uma formalização desse paradigma declarativo obtido com conceitos da teoria das categorias. Tomando esta abordagem para Intent, uma implementação inicial desta formalização é apresentada usando Haskell, uma conhecida linguagem de programação funcional.

Termos do índice—Rede baseada em intenção, Programação funcional, Haskell, Teoria da categoria

I. INTRODUÇÃO

O chamado processo de softwarização de redes representa uma mudança que vem ocorrendo na última década em direção ao papel inédito e dominante do software nas redes de comunicação. Dentre as inúmeras vantagens que tal abordagem pode trazer para o gerenciamento de redes e serviços de comunicação, uma das características mais relevantes é a programabilidade da rede, ou seja, a capacidade de visualizar a infraestrutura da rede, bem como os recursos computacionais envolvidos na entrega do serviço, como um entidades de propósito geral que podem receber instruções por meio de Application Programming Interfaces (APIs).
Essas APIs são normalmente oferecidas por interfaces de direção norte de controladores de rede existentes e/ou orquestradores de serviço e, como tal, o nível de abstração que fornecem depende da solução específica adotada pelas plataformas subjacentes.

A Intent-Based Networking (IBN) está emergindo como uma das principais tecnologias para abstrair o gerenciamento de rede e as operações de programação. Por meio do IBN, uma plataforma automatizada de gerenciamento de rede pode implantar um estado desejado (ou pretendido) e aplicar as políticas necessárias sem precisar detalhar etapas específicas e procedimentos operacionais. Então, uma vez que uma determinada intenção é aplicada, um sistema IBN deve monitorar continuamente seu estado e verificar sua consistência com os requisitos iniciais, aplicando ações corretivas adequadas, se necessário.
Dentro da Internet Research Task Force (IRTF), a Rede O Management Research Group (NMRG) definiu uma intenção como "um conjunto de objetivos operacionais (que uma rede deve atender) e resultados (que uma rede deve entregar), definidos de maneira declarativa, sem especificar como alcançá-los ou implementá-los" [1]. Portanto, as intenções são inerentemente uma maneira flexível e declarativa de expressar e compor operações de rede e programar infraestruturas de rede.

Abordagens declarativas para especificações e programação têm sido estudadas na ciência da computação por um tempo considerável. O artigo de Landin [2] de 1966 iniciou uma tendência que ainda é apropriada hoje. Os programas declarativos expressam o que deve ser calculado, em vez de como deve ser calculado. Esses programas são compostos de expressões em vez de sequências de comandos como na programação imperativa. Com esse paradigma, as execuções repetitivas são realizadas por recursão em vez de sequências de operações. Existem dois tipos principais de linguagens declarativas:

- Linguagens funcionais (ou aplicativas), nas quais o modelo subjacente de computação é o conceito matemático de uma função. Durante uma computação, uma função é aplicada a zero ou mais argumentos para obter um único resultado. Em outras palavras, o resultado é determinístico (ou previsível), pois uma função sempre retornará o mesmo valor se chamada com os mesmos argumentos. A linguagem de programação puramente funcional mais comum é Haskell, com Erlang, Scheme e Common Lisp tendo um núcleo funcional.

- Linguagens relacionais (ou lógicas), nas quais o modelo subjacente de computação é o conceito matemático de uma relação (ou um predicado). Uma computação é a associação (não determinística) de um grupo de valores com rastreamento de retorno para resolver valores adicionais. O principal exemplo de uma linguagem de programação lógica é o Prolog.

Uma vez que o conceito IBN é baseado em uma abordagem declarativa para gerenciamento e programabilidade de rede, argumentamos que o uso de programação declarativa para alcançar IBN pode revelar informações valiosas para esse novo paradigma de rede. Em particular, consideramos a programação funcional para IBN graças à sua base matemática que impõe uma abordagem rigorosa ao design e implementação do programa.
A notação usada em linguagens funcionais é muito próxima daquela

978-1-6654-0694-9/22/$31,00 ©2022 IEEE

usado em métodos formais [3], portanto, qualquer sistema projetado usando esses métodos pode ser implementado muito rapidamente, pois as linguagens funcionais são frequentemente consideradas como linguagens de especificação executáveis [4], particularmente com sua estreita ligação com a teoria da categoria. A teoria das categorias é um ramo da matemática que fornece uma metodologia poderosa para descrever e trabalhar em conceitos abstratos, incluindo a própria matemática. Por esta razão, acreditamos que a teoria das categorias pode representar uma ferramenta matemática útil e uma base para conceber um sistema IBN. Neste trabalho, apresentamos uma descrição formal do problema IBN usando ferramentas da teoria das categorias. Em seguida, mostramos como essa formalização pode ser implementada em Haskell, uma conhecida linguagem de programação funcional.

O artigo está organizado da seguinte forma. A Seção II apresenta trabalhos relacionados nas áreas de Teoria de Categoria Aplicada e IBN. A Seção III apresenta os conceitos básicos e as definições da teoria da categoria, então a Seção IV define uma estrutura da teoria da categoria para expressar intenções em um sistema IBN. A Seção V apresenta a conexão entre os conceitos teóricos e sua atuação, fornecendo uma implementação preliminar escrita em Haskell. As conclusões e trabalhos futuros são resumidos na Seção VI.

II. TRABALHO RELACIONADO

A programação funcional tem sido utilizada por muitos pesquisadores por causa de sua base teórica e seus aspectos matemáticos [5], e porque os programas funcionais são passíveis de raciocínio automático baseado em máquinas. Esta área inclui a transformação automática do programa [6], prova automática do programa [7] e semântica formal [8]. A Teoria das Categorias Aplicadas está se tornando um campo relevante na pesquisa, mostrando como a teoria das categorias pode ser aplicada a diferentes campos fora da "matemática pura". Por exemplo, em [9] Coecke, Sadrzadeh e Clark aplicaram a teoria de categorias ao processamento de linguagem natural, definindo um modelo que caracteriza expressões de linguagem natural e seu significado, alavancando ferramentas da teoria de categorias.

Esses conceitos também são adotados na área de modelagem de sistemas físicos cibernéticos [10] [11].

Recentemente, vários esforços de pesquisa investigaram IBN e examinaram seus diferentes aspectos [12] [13]. Em particular, Jacobs et al. propõem um processo de refinamento de intenções [14], que usa IA para processar solicitações de intenções expressas em linguagem natural e as transforma em um formato intermediário chamado Nile, que pode ser realimentado ao operador/usuário para ser validado antes de sua implantação. O mesmo formalismo é usado e estendido em [15] para cobrir um conjunto mais amplo de casos de uso, especificamente aqueles relacionados ao reencaminhamento de tráfego e proteção de tráfego de serviço.
O formalismo de programação funcional apresentado neste trabalho ainda seria válido para descrever os modelos de dados propostos nas abordagens mencionadas.

III. CATEGORIA TEORIA E FUNCIONAL PROGRAMAÇÃO

A força da teoria das categorias reside na capacidade de raciocinar com conceitos abstratos. Em detalhes, ele se concentra nas relações existentes entre os objetos, e não no que essas objetos são. Além disso, pode regular como um "tipo" de objeto pode ser mapeado para outro. Seguindo a definição dada por Fong e Spivak em [16], uma categoria C é uma coleção de objetos Ob(C) tal que: i para cada par de objetos c, d ÿ Ob(C), um

conjunto C(c, d) é especificado incluindo elementos chamados morfismos de c a d, ou morfismos em C; ii para cada objeto c ÿ Ob(C), o morfismo de identidade em c

é especificado como idc ÿ C(c, c);
iii para quaisquer três objetos c, d, e ÿ Ob(C) e para quaisquer morfismos f ÿ C(c, d) eg ÿ C(d, e), o morfismo composto de f e g é especificado como f ÿ g ÿ C(c, e).

Um morfismo f ÿ C(c, d) também pode ser denotado como f : c ÿ d.
Aqui, c é chamado de domínio de f e d é chamado de contradomínio de f. Esses constituintes são necessários para satisfazer duas condições: uma unitalidade:

para qualquer morfismo f : c ÿ d, o morfismo composto de f e qualquer um dos morfismos de identidade em c ou d retorna f, ou seja: idc ÿ f = f e f ÿ idd = f. b associatividade: para quaisquer três morfismos f : c0 ÿ c1, g : c1 ÿ c2 e h : c2 ÿ c3, vale: (f ÿ g) ÿ h = f ÿ (g ÿ h) = f ÿ g ÿ h .

Outro elemento importante é o conceito de functor. Um functor mapeia todos os objetos de uma categoria C para objetos de uma categoria D, enquanto preserva sua estrutura (ou seja, identidades e composição).
Mais formalmente de [16], sejam C e D categorias. Um funtor F de C a D, denotado como F : C ÿ D, é tal que:

i para cada objeto c ÿ Ob(C), um objeto F(c) ÿ Ob(D) pode ser especificado; ii para cada morfismo
f : c1 ÿ c2 em C, um morfismo F(f) : F(c1) ÿ F(c2) em D pode ser especificado.

Um functor deve satisfazer duas propriedades:

a para cada objeto c ÿ Ob(C), contém F(idc) = idF (c) ; b para quaisquer três objetos c1, c2, c3 ÿ Ob(C) e para quaisquer morfismos f ÿ C(c1, c2) eg ÿ C(c2, c3), a equação F(f ÿ g) = F(f) ÿ F(g) vale em D.

A ligação entre Haskell, ou programação funcional em geral, e a teoria das categorias pode não ser fácil de ver. No entanto, é possível construir uma categoria **Hask** [17] na qual **Ob(Hask)** contém todos os tipos de dados Haskell (por exemplo, Int, Bool, etc.) bool). Esta construção pode ser considerada uma categoria, até uma pequena aproximação. Ao considerar **Hask** como uma categoria, podemos usar todas as outras construções definidas na teoria da categoria (por exemplo, functors, mônadas, etc.). Por exemplo, uma nova definição de tipo em Haskell pode ser vista como um endofunctor em **Hask** (um functor de **Hask** para **Hask),** pois mapeia tipos para um novo tipo e preserva morfismos (funções) entre os tipos iniciais.

Portanto, dada a grande variedade de intenções que podem existir, com escopos e requisitos muito diferentes, pensamos que esse tipo de formalismo seria benéfico para abstrair esses detalhes, focando nas transformações pelas quais eles precisam passar antes de serem aplicados em uma infraestrutura de rede (conforme descrito no Ciclo de vida da intenção [1]).

4. A CATEGORIA DE INTENÇÃO

O foco deste trabalho é formalizar uma abordagem para um sistema IBN utilizando ferramentas categóricas, para depois ser implementado com uma linguagem funcional, nomeadamente Haskell. Primeiro, uma categoria para representar intents é definida como I. Para este trabalho, levaremos em conta apenas intents baseados em linguagem natural. No entanto, as mesmas considerações ainda se aplicam a outras classes de intenções (por exemplo, intenções de máquina para máquina). Os objetos desta categoria Ob(I) podem ser vistos como um subconjunto de todas as sentenças possíveis em inglês (ou mesmo em outras línguas) que expressam uma intenção. Em outras palavras, os objetos são todas as possíveis expressões de intenção bem formadas relacionadas ao gerenciamento de rede que podem ser construídas, por exemplo, usando linguagem natural. Os morfismos podem ser definidos para descrever as relações entre solicitações de intenção “semelhantes”. Especificamente, pode-se introduzir uma relação de ordenação entre os elementos desse conjunto, representada pelo símbolo ÿ. Como exemplo, a intenção I1 pode ser “Implantar Serviço X de baixa latência” e I2 pode ser “Implantar Serviço X”: como a implantação de um serviço X com um requisito de baixa latência implica naturalmente na implantação de serviço, então I1 ÿ I2.

Podemos provar que esta construção é na verdade uma categoria:

- Para cada intenção Ii no conjunto de objetos, temos Ii ÿ Ii . Isso é trivial, pois é claro que um Intent implica a si mesmo (morfismo de identidade).
- Para quaisquer três intenções I1, I2, I3 no conjunto de objetos tal que I1 ÿ I2 e I2 ÿ I3, temos I1 ÿ I3. Em outras palavras, se I1 implica I2 e I2 implica I3, então é claro que I1 implica I3 (regra de composição).

Além disso, esta categoria é uma categoria parcialmente ordenada. É ordenado porque, se existe um morfismo entre dois objetos, eles estão relacionados conforme descrito e esse morfismo é único. A ordenação é parcial, pois pode haver objetos que não podem ser relacionados com outros.

Dentro da categoria de intenção, um operador de produto ÿ pode ser definido atuando entre objetos. Esse operador pode ser usado para vincular diferentes expressões de intenção, semelhante a um e lógico. Por exemplo, deixe a intenção I1 ser “Implantar um servidor web” e I2 “Implantar um firewall”, então I1 ÿ I2 seria “Implantar um servidor web e implantar um firewall”. O produto pode ser usado para construir uma estrutura dentro da categoria na qual solicitações de intenção complexas são vinculadas a seus componentes constituintes por meio desse operador. Ao definir um objeto de identidade para este operador, é possível provar que a categoria de intenção equipada com o produto de intenção é uma “pré-encomenda monoidal” (ou seja, uma categoria livre obtida de um conjunto de pré-encomenda equipado com um produto monoidal, consulte as Seções 3.2 .3 e 4.4.4 em [16]). O objeto de identidade para esse operador deve representar uma solicitação de intenção que, se multiplicada ou vinculada logicamente a qualquer outra intenção, a intenção resultante não mudaria. Esse objeto de identidade seria um formulário como uma intenção de operação nula. Para provar que ÿ é realmente um operador monoidal, as seguintes propriedades devem ser mantidas:

- **Monotonicidade:** Para todo x1, x2, y1, y2 ÿ Ob(I), se x1 ÿ y1 e x2 ÿ y2, então x1 ÿ x2 ÿ y1 ÿ y2.
- **Unitalidade:** Seja idI o objeto identidade para ÿ, então para todo x ÿ Ob(I) as identidades esquerda e direita valem: idI ÿx = x ÿ idI = x. •

Associatividade: Para todo x, y, z ÿ Ob(I), (x ÿ y) ÿ z = x ÿ (y ÿ z).

Agora considere um exemplo da primeira propriedade, com x1 = “Implantar serviço X de baixa latência”, y1 = “Implantar serviço X”, x2 = “Ativar um firewall entre 8h e 22h”, y2 = “Ativar um firewall” . Seguindo a definição, x1 ÿ x2 e y1 ÿy2 seriam “Implantar o Serviço X de baixa latência e ativar um firewall entre 8h e 22h” e “Implantar o Serviço X e ativar um firewall”, respectivamente. Como x1 ÿ y1 e x2 ÿ y2, é fácil ver que também x1 ÿ x2 ÿ y1 ÿ y2, satisfazendo assim a propriedade **de monotonicidade .** Demonstrações semelhantes podem ser dadas para **unidade** e **associatividade.** Uma propriedade adicional que é interessante é **a simetria,** o que significa que dado x, y ÿ Ob(I) ele contém x ÿ y = y ÿ x.

Outra definição de produto intencional também pode ser obtida por meio da construção universal. É uma abordagem usada na teoria da categoria para derivar as propriedades de um objeto observando suas relações com outros objetos. Para qualquer Intent Ii tendo duas projeções em direção a I1 e I2, ou seja, Ii implica tanto I1 quanto I2, existe um objeto I1 ÿ I2 tal que existe um único morfismo de Ii para I1 ÿI2 que faz os dois “triângulos” na Fig. 1 para comutar (ver Definição 3.71 em [16]). Para simplificar, I1 ÿ I2 pode ser visto como o “melhor objeto” com projeções em direção a I1 e I2.

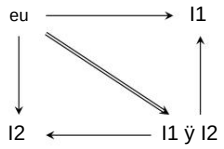

Fig. 1. Diagrama de comutação para produto categórico. As setas na figura representam implicações entre intenções.

Isso pode ser interpretado como se os objetos Ii fossem todas as possíveis especificações de intenção que requerem um serviço composto pelas solicitadas por I1 e I2, sendo I1 ÿ I3 a forma de descrever o serviço composto em que seus componentes são “mais fáceis” de identificar. Um exemplo pode ajudar a esclarecer isso: seja I1 “Implantar um servidor web” e I2 “Implantar um firewall”, então I1 ÿI2 seria “Implantar um servidor web e um firewall”. Neste exemplo, o “Iis” seria expressões como “Implantar um servidor HTTP seguro” ou “Implantar um servidor HTTP e protegê-lo”.

Tendo definido a categoria I que representa os pedidos de intenção, o próximo passo é definir uma categoria S que representa os serviços específicos que uma intenção requer. A categoria de intenção poderia então ser vinculada a essa nova categoria S por meio de um functor, que mapearia todos os objetos de I para objetos em S, preservando a estrutura da categoria inicial. Isso significa que, se F : I ÿ S é o funtor entre as duas categorias e I1, I2 ÿ Ob(I) são tais que ÿ f : I1 ÿ I2, então ÿ F(f) : F(I1) ÿ F(I2). Os objetos desta categoria representam todos os serviços que podem ser solicitados por uma intenção, enquanto

morfismos entre esses objetos poderiam incorporar regras de composição entre serviços. Exemplos dessas regras de composição serão dados na Seção V.

Uma abordagem semelhante pode ser seguida para definir outra categoria R representando os requisitos de serviço. Essa categoria incorporará todos os modificadores que uma determinada intenção pode solicitar, por exemplo, valores específicos de Qualidade de Serviço (QoS) a serem satisfeitos (por exemplo, largura de banda mínima, latência máxima, etc.) ou determinados períodos de tempo em que a intenção deve ser aplicada (por exemplo, “todos os dias”, “todas as segundas-feiras”, “somente entre 8h e 10h)”. Também neste caso pode ser definido um functor de I a R, com as mesmas propriedades acima.

Um aspecto importante que vale destacar é que não foram dados detalhes sobre a estrutura interna das categorias S e R, mas apenas sobre a intenção. Aproveitando a abstração concedida pelas ferramentas da teoria das categorias, é possível raciocinar e definir uma estrutura no “nível da linguagem natural” (isto é, no nível da intenção). Essa estrutura é preservada nas categorias vinculadas (ou seja, nos níveis de serviço e requisito) por meio de relacionamentos categóricos (por exemplo, functors), sem a necessidade de “olhar para dentro” delas. Em outras palavras, poderíamos dizer que as categorias de serviços e requisitos podem ser vistas como etapas intermediárias entre a recepção de um intent e sua atuação no sistema, ou seja, o Translation/Intent-Based System (IBS) Space no ciclo de vida do intent [1]. No entanto, não precisamos definir modelos de dados precisos para essas etapas intermediárias para derivar suas propriedades, pois são herdadas da estrutura de intenções expressas em linguagem natural.

Nessa visão, os functores que vão da categoria de intenção I para as categorias S e R podem ser vistos como funções de “extração de significado”, ou seja, extração de serviços solicitados e seus requisitos de expressões de intenção de linguagem natural.

V. IMPLEMENTAÇÃO DE CATEGORIAS EM HASKELL

Definidas previamente as categorias, e visto como elas se relacionam, o próximo passo é considerar cada uma delas e definir uma sintaxe para começar a construir a ponte entre a construção teórica e uma primeira implementação em Haskell.

Os objetos em I podem ser descritos com um tipo definido como:

```
Intenção de dados = String de intenção
                  | NullOperation
```

Este exemplo também pode ser usado para explicar como os tipos podem ser definidos em Haskell. A primeira palavra-chave Intent aqui é um construtor de tipo que identifica o nome do tipo. As outras duas palavras-chave no lado direito do sinal de igualdade, Intent e NullOperation, são os construtores de dados, que especificam como os dados do tipo Intent podem ser construídos.

Para este exemplo, um valor do tipo Intent pode ser construído usando NullOperation sem qualquer parâmetro ou Intent seguido por uma String. Aqui estão dois exemplos: intent1 = Intenção "Implantar um firewall" ou nullOpIntent = NullOperation. Em outras palavras, um objeto da categoria I é uma string (ou seja, uma solicitação de intenção real expressa em linguagem natural) ou uma operação nula (ou seja, o objeto de identidade definido para o produto monoidal interno).

Então, uma definição de tipo semelhante poderia ser dada para os objetos da categoria S:

```
Ação de dados = Serviço (Fase,BasicService)
              | TemporalCompose [Ação]
              | LogicalCompose [Ação]
              | Sem ação
```

Este é um tipo de dados recursivo, uma definição de tipo comum em programação funcional. O novo tipo Action pode ser um único Service, especificado através de um par de novos tipos de dados (Phase e BasicService) que serão descritos posteriormente, ou por uma composição de uma lista de Action. Especificamente, dois tipos diferentes de composição são definidos, temporal e lógico.

O primeiro representa uma lista de ações que precisam ser executadas em uma ordem temporal específica, por exemplo, implantar um firewall e depois configurá-lo. Este último descreve uma ação que depende de uma série de outros componentes lógicos sem uma ordem dada, por exemplo, a implantação de um núcleo 5G depende da implantação de um conjunto de outras funções como AMF, SMF, PCF, etc.

Por último, o construtor de tipo NoAction representa um “vazio” Objeto de ação, ou seja, uma ação que não faz nada no sistema, semelhante ao construtor de lista vazia Nil nas definições clássicas de lista recursiva [3]. O tipo BasicService é usado para representar serviços ou ações básicas. A ideia aqui é que essa nova definição de tipo deve conter todos os “blocos de construção” possíveis que podem ser compostos (temporal ou logicamente) juntos para construir um serviço complexo exigido por uma intenção. Uma definição básica desse novo tipo é:

```
dados BasicService = String VnfId
                   | RouterConfig (String,
                                   Corda)
                   | PathCreate (String,
                                 Corda)
```

Obviamente, esse tipo pode ser estendido com outros serviços básicos que os usuários possam exigir (ou seja, adicionar um novo construtor de dados) ou usar estruturas mais complexas como entrada para eles. Por fim, o tipo Fase simplesmente define a fase do ciclo de vida em que essa ação ocorrerá. Uma definição básica desse novo tipo é:

```
Fase de dados = Adicionar
              | Atualizar
              | Reiniciar
              | Remover
```

Definidos os objetos em S, o próximo passo seria definir a estrutura dessa categoria, ou seja, seus morfismos.

Os morfismos podem descrever como os serviços podem ser compostos por outros. Por exemplo, seja x, y ÿ Ob(S) com x = “Deploy a 5G Core Network e y = “Deploy an SMF function”, então existe um morfismo f : x ÿ y que especifica que y é um componente de x. Para esclarecer, a definição de x e y usada no exemplo não é totalmente correta, pois não segue a definição formal fornecida acima. No entanto, um functor de I a S foi definido, portanto, cada objeto da primeira categoria é mapeado para um objeto da segunda. Por esta

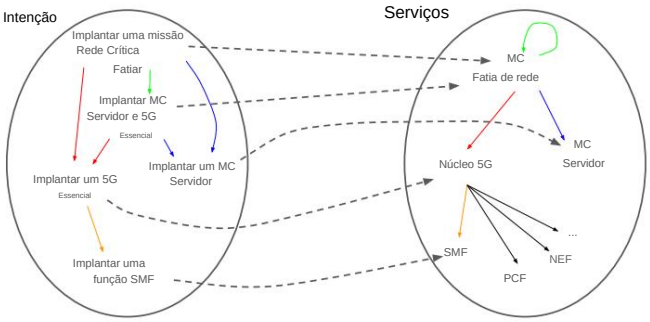

Fig. 2. Representação gráfica do functor entre as duas categorias de intenções I e serviços S.

razão, x e y podem ser vistos como os objetos que são mapeados pelas intenções “Implantar uma rede principal 5G” e “Implantar uma função SMF”, respectivamente.

A Figura 2 é apresentada para entender melhor o que o functor faz entre duas categorias I e S. As linhas tracejadas conectam os intents aos objetos em S, ou seja, aos serviços que estão solicitando. A mesma cor foi usada para destacar como o functor mapeia morfismos entre as duas categorias. A partir da Fig. 2 é possível ter uma visão do que é necessário para satisfazer uma requisição de intenção levando em consideração a sub-árvore tendo como raiz o serviço mapeado. Ou seja, sabendo como distribuir as folhas dessas árvores e como compô-las, deve ser possível implantar o serviço requerido. Por exemplo, as folhas da Fig. 2, usando a definição de tipo apresentada acima, podem ser definidas da seguinte forma:

```
smf = Serviço (Adicionar, VnfId "smfId")
```

ou seja, a variável smf do tipo Action é definida usando seu construtor de dados Service, pois podemos considerá-la como um único VNF, não exigindo, portanto, nenhuma composição. Esse construtor de dados recebe como argumento um par de objetos. Um do tipo BasicService, que foi construído usando seu construtor de dados VnfId seguido por uma string representando o identificador fornecido para essa função específica, "smfId" no exemplo. O outro do tipo Fase construído com seu construtor de dados Add, pois a intenção no exemplo é exigir a implantação de uma função. O mesmo procedimento também pode ser aplicado às outras folhas representadas na figura, claro que com os identificadores VNF corretos. Então essas folhas podem ser compostas juntas para construir, por exemplo, o objeto que representa uma implantação de núcleo 5G:

```
5gCore = LogicalCompose[ (Serviço
                (Adicionar, VnfId "smfId")), (Serviço (Adicionar, VnfId
                "pcfId")), ...]
```

Aqui, o construtor de dados LogicalCompose é usado para identificar a lista de VNFs que fazem parte do serviço principal 5G (por exemplo, SMF, PCF, etc.). A composição lógica é simplesmente agrupá-los sem uma ordem específica na qual os VNFs precisam ser implantados (ou seja, na terminologia NFV-MANO, um serviço de rede composto por vários VNFs sem nenhum gráfico de encaminhamento de rede virtual específico).

Obviamente, definições de tipo semelhantes também podem ser dadas para a categoria R. Nesse caso, os objetos podem ser vistos como uma composição de “modificadores de intenção” básicos, como: Escopo, para quem a intenção é direcionada (por exemplo, toda a rede, usuário único); Hora, quando a intenção deve estar ativa (ex. “todos os dias das 14h às 17h”, “sempre”); Restrições de latência, restrições de throughput. Usando a notação Haskell:

```
dados SrvRequirement = Requisito BasicReq
                     | ComposeReq
                          [SrvRequirement]
                     | NoReq

dados BasicReq = Latência _
               | largura de banda    _
               | Escopo _
               | Tempo de atividade _}
```

Quanto aos outros tipos de dados, BasiqReq também pode ser expandido para cobrir um conjunto maior de requisitos. Após a definição desses novos tipos, a ordenação deve ser definida.
Essa ordenação mapeará a relação entre duas variáveis do mesmo tipo, incorporando o conceito de morfismo entre objetos nas categorias das quais os tipos derivam. Como as categorias foram parcialmente ordenadas, usar a classe Ord definida no sistema de tipos de Haskell não seria a escolha correta, como é para ordenações totais. Portanto, uma nova classe deve ser definida para eles. Como em uma ordem parcial dois objetos podem estar relacionados (por exemplo, menor que, igual a, maior que) ou não, uma forma possível de programar isso em Haskell é usando uma construção como Maybe Ord.Ordering. Em outras palavras, se a relação existe, a comparação das duas variáveis retorna Just Ord (por exemplo, LT, EQ, GT), caso contrário, Nothing. Um objeto desses novos tipos seria menor que outro se fosse um componente deste último.

Definidos os tipos que descrevem as três categorias consideradas, o próximo passo é a definição dos functores que as conectam. Em Haskell, isso significa definir funções tomando como entrada objetos do tipo Intent e retornando Action ou SrvRequirement. Essas funções devem receber expressões de linguagem natural e construir a árvore que representa as operações de rede que estão solicitando e seus requisitos.

Por fim, novas funções são necessárias para impor os requisitos expressos nesses novos tipos na infraestrutura subjacente.

A. Avaliação

Para começar a testar essas operações, implementamos uma versão preliminar de uma função de atuação, especificamente, aquela capaz de construir o Network Service Descriptor (NSD) Open Source Mano (OSM) requerido por uma ação específica. Em detalhes, se a função for chamada com uma ação de entrada como Service(Add, VnfId "vndfid"), o identificador Vnf é adicionado aos campos obrigatórios de um modelo de dados OSM NSD, ou seja, no vnfd-id e vnf-profile listas. Alternativamente, se a entrada da função for uma composição lógica de um

conjunto de VNFs, então usando uma função fold (foldr) o comportamento descrito é aplicado recursivamente a todos os elementos da lista, acumulando todos os resultados no mesmo descritor de serviço de rede. Em uma sintaxe semelhante a Haskell:

```
generateNsd :: Ação -> Nsd -> Nsd generateNsd (Serviço
(Adicionar, VnfId vnfid))
    nsd = [...]
generateNsd (ações do LogicalCompose) nsd = foldr generateNsd
    ações nsd
```

Por exemplo, assumindo uma ação de entrada como:

```
LogicalCompose[
            (Serviço (Adicionar, VnfId "smfId")),
            (Serviço (Adicionar, VnfId "pcfId"))]
```

a função constrói um OSM NSD válido contendo os VNFs necessários. Os componentes relevantes do descritor gerado a partir do exemplo anterior são:

```
nsd:
  nsd:
    df:
    - id: default-df vnf-profile: -
      id: pcfid virtual-link-
      connectivity: -
        constituinte-cpd-id:

          - constituinte-base-elemento-id:
              pcfid
            constituinte-cpd-id: mgmt-ext virtual-link-profile-
          id: mgmtnet vnfd-id: pcfid - id: smfid


        virtual-link-conectividade: - constituinte-cpd-
        id:
          - constituinte-base-elemento-id:
              smfid
            constituinte-cpd-id: mgmt-ext virtual-link-profile-
          id: mgmtnet
        vnfd-id: smfid
    vnf-id:
    - pcfid -
    smfid
     [...]
```

O descritor NSD criado com esta função pode então ser integrado na plataforma OSM e sua implantação pode ser inicializada. Além disso, graças aos recursos de verificação de tipo do Haskell, o formato NSD é validado automaticamente durante cada análise e renderização do arquivo YAML, aumentando assim a robustez do código.

VI. CONCLUSÃO E TRABALHOS FUTUROS

Este trabalho aplicou diretamente os conceitos da teoria das categorias ao IBN para construir uma representação formal das especificações de intenção. Esta representação pretende ajudar no raciocínio sobre este novo paradigma, mantendo uma estreita relação com a programação funcional e as possíveis implementações. Neste artigo, mostramos uma implementação preliminar das categorias de intenção usando Haskell. Essa implementação é capaz de obter uma definição de um serviço e construir um OSM NSD válido contendo os VNFs necessários. Os componentes relevantes do descritor são gerados.

Outros trabalhos podem ser feitos em ambas as direções. Por exemplo, lentes na teoria das categorias [18] são usadas para descrever em termos matemáticos os conceitos de agente e ambiente, podendo assim servir como uma ferramenta válida para modelar a interação entre o Intent System e a infraestrutura de rede que está gerenciando. Eles também são amplamente utilizados em Haskell, sendo uma das ferramentas mais poderosas para acessar e modificar estruturas de dados.

REFERÊNCIAS

[1] A. Clemm, L. Ciavaglia, LZ Granville e J. Tantsura, “Intent-Based Networking - Concepts and Definitions”, IETF, rascunho do Internet-Rascunho irtf-nmrg-ibn-concepts-definitions-06, 12 2021 . [On-line]. Disponível: https://datatracker.ietf.org/doc/draft-irtf-nmrg-ibn-concepts-definitions [2] PJ Landin, “The Next 700 Programming Languages,” Communications of the ACM, vol. 9, não. 3, pág. 157–166, março de 1966. [Online]. Disponível: https://doi.org/10.1145/365230.365257

[3] J. Hughes, "Por que a programação funcional é importante", Computer Journal, vol. 32, nº. 2, pp. 98–107, 1989.

[4] D. Turner, “Programas Funcionais como Especificações Executáveis,” Lógica Matemática e Linguagens de Programação, ed. P. Sheperdson, Prentice Hall, 1984.

[5] JA Stoy, “Alguns Aspectos Matemáticos da Programação Funcional,” Functional Programming and its Applicationsm ed DA Turner, Cam bridge University Press, 1980.

[6] J. Darlington, PG Harrison, H. Khoshnevisan, L. McLoughlin, N. Perry, H. Pull, M. Reeve, K. Sephton, RL While e S. Wright, “Um ambiente de programação funcional que suporta a execução, Execução e transformação parcial,” em Proceedings of the Parallel Architectures and Languages Europe, Volume I: Parallel Architectures, ser. PARLAMENTO '89. Berlim, Heidelberg: Springer-Verlag, 1989, p. 286–305.

[7] D. Turner, “Programação funcional e provas de correção do programa,” Ferramentas e Noções para a Construção de Programas, ed. D. Neel, Cambridge University Press, 1982.

[8] D. Schmidt, "Semântica denotacional - uma metodologia para o desenvolvimento da linguagem", 1986.

[9] B. Coecke, M. Sadrzadeh e S. Clark, "Fundamentos matemáticos para um modelo de distribuição composicional de significado", 2010.

[10] A. Speranzon, DI Spivak e S. Varadarajan, “Abstração, composição e contratos: uma abordagem teórica do feixe,” CoRR, vol. abs/1802.03080, 2018.

[11] G. Bakirtzis, C. Vasilakopoulou e CH Fleming, “Modelagem de sistemas ciberfísicos composicionais,” Procedimentos Eletrônicos em Ciência da Computação Teórica, vol. 333, pág. 125–138, fevereiro de 2021. [On-line]. Disponível: http://dx.doi.org/10.4204/EPTCS.333.9 [12] L. Pang, C. Yang, D. Chen, Y. Song e M. Guizani, “Uma pesquisa sobre redes orientadas por intenção, ” Acesso IEEE, vol. 8, pp. 22 862–22 873, 2020.

[13] E. Zeydan e Y. Turk, “Avanços recentes em rede baseada em intenção: uma pesquisa”, em 2020 IEEE 91ª Conferência de Tecnologia Veicular (VTC2020-Primavera), 2020.

[14] AS Jacobs, RJ Pfitscher, RA Ferreira e LZ Granville, “Refining network intents for self-driving networks”, em Proceedings of the Afternoon Workshop on Self-Driving Networks, ser. AutoDN 2018. Nova York, NY, EUA: Association for Computing Machinery, 2018, p. 15–21. [On-line]. Disponível: https://doi.org/10.1145/3229584.3229590 [15] M. Bezahaf, E. Davies, C. Rotsos e N. Race, “Para todas as intenções e propósitos: Rumo a uma expressão de intenção flexível”, em 2021 IEEE 7º Conferência Internacional sobre Software de Rede (NetSoft), 2021, pp. 31–37.

[16] B. Fong e DI Spivak, “Sete esboços em composicionalidade: um convite à teoria de categoria aplicada”, 2018.

[17] Categoria hask. [On-line]. Disponível: https://wiki.haskell.org/Hask [18] D. Spivak. Lentes: aplicações e generalizações. [On-line]. Disponível: http://math.ucr.edu/home/baez/ACTUCR2019/ACTUCR2019 spivak.pdf