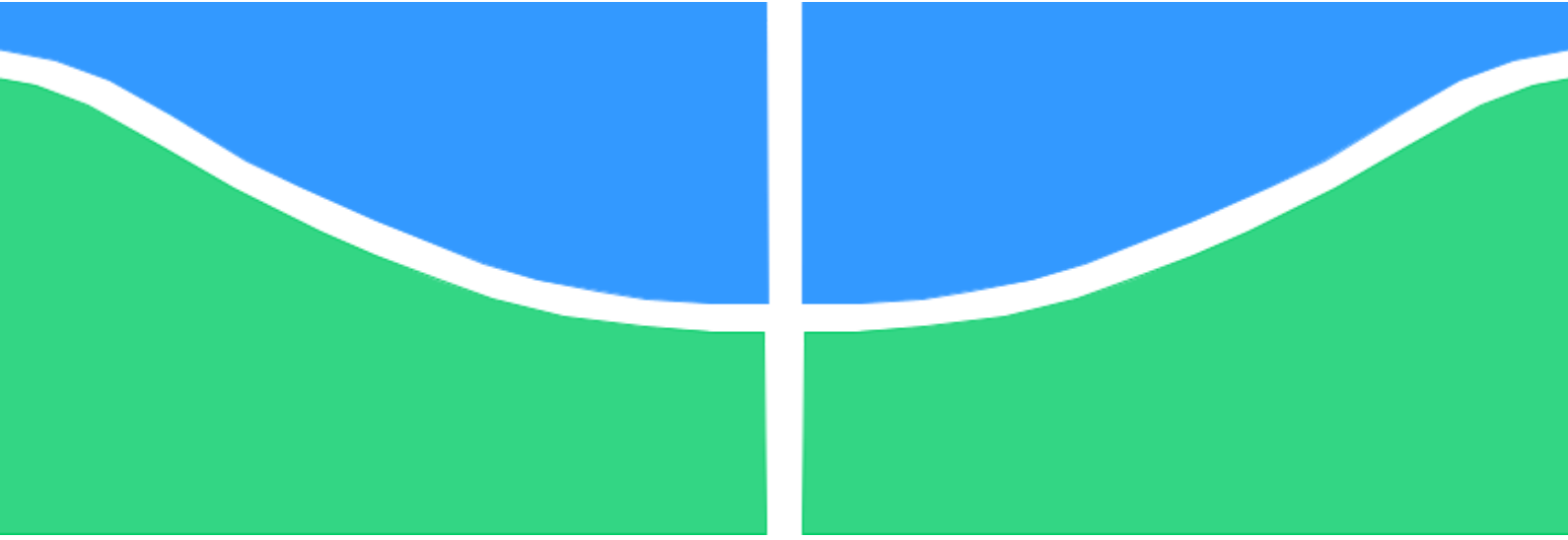

Universidade de Brasília - UnB
Faculdade UnB Gama - FGA
Curso de Engenharia de Software

Paradigma Lógico

Paradigmas de Programação
Professora: Milene Serrano

Autores:
Jackes Tiago Ferreira da Fonseca - 190030291

Brasília, DF

Sumário

1. Introdução

O **Problema de Einstein**, como é conhecido popularmente o **enigma da zebra**, é um problema lógico famoso frequentemente atribuído ao físico Albert Einstein. Como segue a lenda, Einstein teria afirmado que somente 2% da população do mundo conseguiria resolvê-lo. Apesar de não existir evidências concretas de que Einstein realmente tenha criado o enigma, ele permanece um desafio popular devido à sua complexidade lógica e estrutural.

No problema existem cinco casas de diferentes cores e habitadas por pessoas de nacionalidades distintas. Cada morador possui uma bebida, um animal de estimação específico e, na nossa versão, navegam em um site específico, sendo que nenhum desses itens se repete entre as casas. A questão central geralmente é descobrir quem possui um determinado animal, como uma zebra ou peixe, com base em um conjunto de pistas.

Para a resolução do problema espera-se habilidades como dedução lógica, análise combinatória e organização sistemática da informação. Esse tipo de problema ilustra como a lógica formal pode ser aplicada para resolver situações complexas, destacando a importância do raciocínio estruturado e da paciência. O enigma é amplamente utilizado em contextos educacionais e recreativos para treinar habilidades de pensamento crítico e resolução de problemas.

2. Comparação entre Prolog e Haskell na resolução do problema de Einstein

Haskell (Paradigma Funcional)

- **Características**:
  - Baseado em funções puras, avaliação preguiçosa e imutabilidade.
  - Utiliza expressões matemáticas para descrever transformações e computações.
  - Abordagem declarativa, mas com mais controle explícito sobre o fluxo.
- **Pontos Fortes**:
  - Ideal para manipulação de listas e modelagem de conjuntos de dados.
  - Controle mais explícito sobre otimizações e estratégias de execução.
  - Melhor integração com sistemas modernos e bibliotecas, além de excelente suporte para paralelismo.
- **Pontos Fracos**:
  - Maior complexidade para modelar problemas que requerem busca e *backtracking*.
  - A modelagem explícita das restrições e relações pode levar a um código mais verboso.
  - O desempenho pode ser inferior ao Prolog em problemas de busca lógica pura.

Prolog (Paradigma Lógico)

- **Características**:
  - Baseado em lógica de predicados e unificação.

    - Declara relações e regras, deixando o motor lógico resolver os problemas.
    - O código é escrito de forma declarativa, especificando *o que* deve ser verdadeiro, não *como* resolver.
- **Pontos Fortes**:
    - Natural para problemas de lógica e busca, como o Problema de Einstein.
    - Simplicidade na modelagem de restrições e relações, como `ao_lado` e `a_esquerda`.
    - Estrutura de *backtracking* automatizada para explorar diferentes combinações de forma eficiente.
- **Pontos Fracos**:
    - Escalabilidade limitada para problemas de maior dimensão devido ao custo do *backtracking*.
    - Legibilidade pode ser comprometida em problemas muito complexos.
    - Difícil integração com sistemas modernos devido à natureza nichada da linguagem.

Implementação em Haskell

A seguir tem uma versão simplificada desenvolvida em Haskell

```haskell
import Data.List (permutations)

-- Definição de tipos
data Cor = Vermelha | Verde | Branca | Amarela | Azul deriving (Eq, Show)
data Nacionalidade = Brasileiro | Inglês | Chileno | Argentino | Português deriving (Eq, Show)
data Bebida = Leite | Chá | Café | Cerveja | Água deriving (Eq, Show)
data Linguagem = Prolog | Python | Java | PHP | HaskellLang deriving (Eq, Show)
data Animal = Cachorros | Gatos | Cavalos | Peixes | Pássaros deriving (Eq, Show)

type Casa = (Cor, Nacionalidade, Bebida, Linguagem, Animal)
type Solucao = [Casa]

-- Restrições
valida :: Solucao -> Bool
valida casas = and [
    -- Dicas e restrições aqui
    any (\(c, n, b, l, a) -> n == Brasileiro && c == Vermelha) casas,
    any (\(c, n, b, l, a) -> n == Chileno && a == Cachorros) casas,
    -- Exemplo simplificado, incluir as demais restrições
    True
    ]

-- Solução por força bruta
resolver :: [Solucao]
resolver = filter valida (permutations [(c, n, b, l, a) |
    c <- [Vermelha, Verde, Branca, Amarela, Azul],
    n <- [Brasileiro, Inglês, Chileno, Argentino, Português],
    b <- [Leite, Chá, Café, Cerveja, Água],
    l <- [Prolog, Python, Java, PHP, HaskellLang],
    a <- [Cachorros, Gatos, Cavalos, Peixes, Pássaros]])
```

Implementação em Prolog

A seguir está nossa versão desenvolvida em Prolog usando paradigma

```
% Predicado para definir se a casa A fica a esquerda da casa B
a_esquerda(A, B, [A, B | _]).
a_esquerda(A, B, [_ | T]) :-
    a_esquerda(A, B, T).

% Predicado para definir se a casa A fica ao lado da casa B (esquerda ou direita)
ao_lado(A, B, [A, B | _]).
ao_lado(A, B, [B, A | _]).
ao_lado(A, B, [_ | T]) :-
    ao_lado(A, B, T).

% Predicado para garantir que cada possui informações unicas (ex: não existem duas casas vermelhas)
sao_unicas([]).
sao_unicas([H|T]) :-
    \+ member(H,T), % H não está em T
    sao_unicas(T).

% Predicado para perguntar a cor da casa N
% cor(N,Cor)

% Definicao de casa
% casa(Cor,Nacionalidade,Bebida,Linguagem,Animais).
resolver(Casas) :-
    Casas = [
        casa(Cor1,Nacionalidade1,Bebida1,Linguagem1,Animais1),
        casa(Cor2,Nacionalidade2,Bebida2,Linguagem2,Animais2),
        casa(Cor3,Nacionalidade3,Bebida3,Linguagem3,Animais3),
        casa(Cor4,Nacionalidade4,Bebida4,Linguagem4,Animais4),
        casa(Cor5,Nacionalidade5,Bebida5,Linguagem5,Animais5)
    ],
    % Dicas do problema
    Casas = [casa(_,brasileiro,_,_,_) | _],                    % O Brasileiro vive na primeira casa.
    Casas = [_,_,casa(_,_,leite,_,_),_,_],                     % O homem que vive na casa do meio bebe Leite.
    member(casa(vermelha,ingles,_,_,_), Casas),                % O Inglês vive na casa Vermelha.
    member(casa(_,chileno,_,_,cachorros), Casas),              % O Chileno tem Cachorros como animais de estimação.
    member(casa(_,argentino,cha,_,_), Casas),                  % O Argentino bebe Chá.
    member(casa(verde,_,cafe,_,_), Casas),                     % O homem que vive na casa Verde bebe Café.
    member(casa(_,_,_,haskell,passaros), Casas),               % O homem que programa em Haskell cria Pássaros.
    member(casa(amarela,_,_,prolog,_), Casas),                 % O homem que vive na casa Amarela programa em Prolog.
    member(casa(_,_,cerveja,python,_), Casas),                 % O homem que programa em Python bebe Cerveja.
    member(casa(_,portugues,_,php,_), Casas),                  % O Português programa em PHP.
    ao_lado(casa(_,_,_,java,_), casa(_,_,_,_,gatos), Casas),   % O homem que programa em Java vive ao lado do que tem Gatos.
    ao_lado(casa(_,_,_,_,cavalos), casa(_,_,_,prolog,_), Casas),% O homem que cria Cavalos vive ao lado do que programa em Prolog.
    ao_lado(casa(_,brasileiro,_,_,_), casa(azul,_,_,_,_), Casas),% O Brasileiro vive ao lado da casa Azul.
    ao_lado(casa(_,_,_,java,_), casa(_,_,agua,_,_), Casas),    % O homem que programa em Java é vizinho do que bebe Água.
    ao_lado(casa(_,_,_,_,peixes), casa(_,_,cerveja,_,_), Casas),% O homem que tem Peixes vive ao lado do que bebe Cerveja.
    a_esquerda(casa(verde,_,_,_,_), casa(branca,_,_,_,_), Casas),% A casa Verde fica do lado esquerdo da casa Branca.

    sao_unicas([Cor1,Cor2,Cor3,Cor4,Cor5]),
    sao_unicas([Nacionalidade1,Nacionalidade2,Nacionalidade3,Nacionalidade4,Nacionalidade5]),
    sao_unicas([Bebida1,Bebida2,Bebida3,Bebida4,Bebida5]),
    sao_unicas([Linguagem1,Linguagem2,Linguagem3,Linguagem4,Linguagem5]),
    sao_unicas([Animais1,Animais2,Animais3,Animais4,Animais5]).
```

3. Análise de desempenho

Haskell:

- A implementação funcional exige geração explícita de combinações (como permutations), o que pode ser menos eficiente.
- No entanto, a imutabilidade e o controle explícito do fluxo permitem paralelismo e otimizações específicas.
- Escala melhor para problemas grandes se as restrições forem bem modeladas para evitar combinações redundantes.

Prolog:

- A resolução é otimizada para problemas de lógica, graças ao *backtracking* nativo.
- Para o Problema de Einstein, o desempenho é excelente, pois o motor lógico explora apenas as combinações relevantes.
- Escala mal para problemas maiores devido ao crescimento exponencial do espaço de busca.

4. Conclusão

Prolog

Prolog é uma linguagem fortemente especializada em problemas que envolvem lógica, como o enigma proposto pela nossa equipe. Seu ponto forte consiste principalmente no uso de *backtracking* nativo e unificação para resolver problemas declarativos de forma direta e eficiente. Isso faz com ele seja ideal para resolver situações em que as relações entre variáveis são mais importantes do que as operações explícitas realizadas sobre os dados. Além disso, podemos dizer que o Prolog permite uma modelagem natural das restrições do problema, com uma sintaxe que o aproxima com a lógica formal.

Entretanto, o desempenho de Prolog é limitado em problemas que envolvem um espaço de busca bastante grande, pois o custo do *backtracking* aumenta exponencialmente com a complexidade do problema. Outra limitação existente é a integração com sistemas modernos e tecnologias populares, já que Prolog não possui a mesma versatilidade de linguagens funcionais ou multiparadigma como Haskell.

Haskell

Haskell, por sua vez, é uma linguagem funcional que oferece grande flexibilidade para resolver uma ampla variedade de problemas, incluindo problemas de lógica como o Problema de Einstein. A força do paradigma funcional está, sobretudo, na imutabilidade e no uso de funções puras, que facilitam o raciocínio matemático sobre os dados e permitem um controle explícito do fluxo de execução. Em Haskell, o controle sobre a avaliação preguiçosa e a capacidade de explorar paralelismo tornam a linguagem uma opção mais robusta para problemas de grande escala e que envolvem muitos dados.

Por outro lado, a modelagem de restrições e relações, que é intuitiva em Prolog, exige mais esforço em Haskell. O programador precisa lidar explicitamente com a geração de combinações e a aplicação das restrições, o que pode resultar em um código mais verboso e propenso a erros. No entanto, esse controle explícito também abre espaço para otimizações, permitindo uma abordagem mais eficiente para problemas que envolvem grandes volumes de dados ou múltiplas camadas de restrições.

Aspectos Complementares

Além dessas características técnicas, é importante considerar outros fatores, como por exemplo:

- **Aprendizado e Usabilidade**: Prolog é mais acessível para pessoas iniciantes que desejam resolver problemas lógicos, devido à sua sintaxe declarativa focada em *o que* resolver. Haskell, em contrapartida, possui uma curva de aprendizado mais íngreme, mas oferece mais poder e flexibilidade em longo prazo.
- **Escalabilidade e Reutilização**: Haskell se destaca em problemas que exigem alta escalabilidade e integração com sistemas modernos. Sua versatilidade permite a reutilização de código e integração com bibliotecas contemporâneas, o que é uma limitação de Prolog.
- **Comunidade e Ecossistema**: A comunidade de Prolog é menor, e o número de ferramentas e bibliotecas é limitado em comparação ao vasto ecossistema funcional de Haskell.

Conclusão Geral

O **Prolog** se mostra mais eficiente e direto para resolver o Problema de Einstein devido

à sua especialização em problemas de lógica. Ele é ideal para situações onde a complexidade reside nas relações lógicas entre variáveis e onde o problema está bem delimitado.

Por outro lado, **Haskell** apresenta maior potencial para aplicações que requerem flexibilidade, extensibilidade e integração com outras tecnologias. Apesar de demandar mais esforço inicial para modelar problemas de lógica, ele oferece benefícios significativos em termos de desempenho e escalabilidade quando bem projetado.

Portanto, a escolha entre Prolog e Haskell dependerá do contexto. Se o objetivo é resolver o problema específico de forma simples e eficiente, Prolog é a melhor escolha. Se o objetivo é usar a solução como uma parte de um sistema maior ou explorar questões como escalabilidade e desempenho, Haskell se torna uma alternativa mais vantajosa. Essa análise demonstra como o entendimento dos paradigmas e suas características é essencial para a escolha da ferramenta adequada a cada problema.

5. Comentários de aprendizado

A comparação entre **Prolog** e **Haskell** usando o Problema de Einstein como comparação oferece diversas lições importantes sobre paradigmas de programação e a escolha de ferramentas para resolver problemas. Alguns dos meus principais aprendizados foram:

1. A importância da escolha do paradigma

Cada paradigma de programação possui vantagens e desvantagens intrínsecas, que tornam algumas linguagens mais adequadas para determinadas classes de problemas. Prolog, como paradigma lógico, facilita a modelagem de problemas baseados em restrições e relações. Haskell, por outro lado, como uma linguagem funcional, se destaca em problemas que exigem maior flexibilidade e integração com outras ferramentas.

Lição: A escolha do paradigma deve estar alinhada às características do problema e aos requisitos do sistema maior em que a solução será integrada.

---

2. Modelagem explícita versus abstração nativa

O **backtracking** automático em Prolog reduz o esforço de modelagem, permitindo que o programador foque nas relações lógicas do problema. Em Haskell, por outro lado, o programador precisa modelar explicitamente o espaço de busca e as restrições, o que pode ser mais trabalhoso, mas oferece maior controle sobre o processo de solução.

Lição: Soluções que abstraem detalhes do processo de busca podem ser mais simples, mas soluções explícitas oferecem maior flexibilidade para problemas complexos ou personalizados.

---

3. Otimização e escalabilidade

A abordagem de Prolog é mais otimizada para resolver problemas de lógica pura de forma eficiente, mas sua escalabilidade é limitada. Haskell, ao permitir paralelismo e

otimizações customizadas, pode lidar melhor com problemas de maior escala ou integração com sistemas maiores.

Lição: Abordagens especializadas são ótimas para resolver problemas isolados, mas podem ser insuficientes quando a solução precisa escalar ou interagir com outras partes de um sistema.

---

4. Curva de aprendizado

Prolog tem uma curva de aprendizado mais suave para problemas lógicos específicos, enquanto Haskell exige um maior investimento inicial em conceitos de programação funcional, como imutabilidade, avaliação preguiçosa e funções de ordem superior.

Lição: Linguagens com uma curva de aprendizado maior podem oferecer recompensas em termos de flexibilidade e reutilização de código a longo prazo.

---

5. Aplicação em contextos reais

Embora Prolog seja excelente para resolver problemas de lógica como o **Problema de Einstein**, ele é menos utilizado em aplicações modernas devido à sua falta de integração com tecnologias contemporâneas. Haskell, por outro lado, possui uma comunidade ativa e bibliotecas robustas para integração com sistemas modernos, como servidores, bancos de dados e APIs.

Lição: A relevância prática de uma linguagem vai além de sua capacidade de resolver um problema específico; ela deve ser avaliada também em relação ao ecossistema e às demandas do mercado.

Reflexão Final

Esse exercício ressalta que não existe uma linguagem "melhor" ou "pior" de forma absoluta. A escolha da linguagem ideal é sempre contextual e depende das características do problema, da necessidade de integração, da experiência da equipe e do foco no desempenho ou simplicidade. Além disso, trabalhar com paradigmas diferentes amplia o entendimento sobre abordagens diversas para a resolução de problemas, enriquecendo as habilidades do desenvolvedor.

6. Referências

- “https://www.geeksforgeeks.org/difference-between-functional-and-logical-programming/
- https://arxiv.org/pdf/cs/9301109
- https://staff.fmi.uvt.ro/~mircea.marin/lectures/LFP/Curs-01A.pdf
- https://www.researchgate.net/publication/1880293_Logic_Programming_Functional_Programming_and_Inductive_Definitions
- https://blogit.michelin.io/logic-programing-loves-data/
- https://www.geeksforgeeks.org/functional-programming-paradigm/
- https://www.linode.com/docs/guides/logic-programming-languages/