HasKLogiX: Uma DSL embutida em Haskell para programação Prolog

Ari Vitor da Silva Lazzarotto[1], **André Rauber Du Bois**[2]

[1]Centro de Desenvolvimento Tecnológico (CDTec) – Universidade Federal de Pelotas (UFPEL)
Caixa Postal 354 – 96010-900 – Pelotas– RS – Brazil

{avdslazzarotto,dubois}@inf.ufpel.br

Abstract. *The different existing programming paradigms are necessary to solve specific problems, where each paradigm has its main application area and advantages and disadvantages when applied in the specific area. However, when we combine two paradigms in the same program, we obtain greater flexibility and a greater capacity to solve problems, as we can use the advantages of each paradigm together, and not just one of them in isolation. In this work, a Domain-Specific Language (DSL) was developed in Haskell for interpreting Prolog programs, using two programming paradigms, logical and functional, which belong to the same declarative programming style. In this way, we attempted to minimize the disadvantages of using the paradigms together, and to maximize the advantages.*

Resumo. *Os diferentes paradigmas de programação existentes se fazem necessários para resolução de problemas distintos, onde cada paradigma tem sua principal área de aplicação e suas vantagens e desvantagens ao ser aplicado na área em questão. Mas, ao unirmos dois paradigmas em um mesmo programa, obtemos uma maior flexibilidade e uma maior capacidade de resolução de problemas, pois podemos utilizar as vantagens de cada paradigma em conjunto, e não apenas um deles isoladamente. Neste trabalho, foi desenvolvido uma DSL (linguagem de domínio específico) em Haskell, para interpretação de programas Prolog, utilizando dois paradigmas de programação, o lógico e o funcional, mas que pertencem ao mesmo estilo de programação, o declarativo. Deste modo, tentamos minimizar as desvantagens de utilizar os paradigmas em conjunto, e espera-se maximizar as vantagens.*

1. Introdução

Linguagens funcionais, como o Haskell, oferecem uma semântica clara e concisa na interpretação e no comportamento dos programas, avaliação preguiçosa e funções de ordem superior. [Bloss 1989]. Por outro lado, as linguagens lógicas fornecem não-determinismo e predicados com múltiplos modos de entrada/saída, favorecendo a reutilização de código [Antoy and Hanus 2010]. A combinação desses recursos pode trazer benefícios significativos no desenvolvimento de software, abstraindo a ordem de avaliação e possibilitando especificações claras. À avaliação sob demanda da

Trabalho Concluído
[1]Graduando
[2]Prof. Dr.

programação funcional aplicada a operações não determinísticas da programação lógica leva a estratégias de busca mais eficientes [Antoy and Hanus 2010]. Ao longo da evolução das linguagens de programação, tem-se caracterizado a introdução gradual de abstrações, que ocultam detalhes de hardware e execução do programa. As linguagens declarativas, como as funcionais e lógicas, são especialmente adeptas a ocultar a ordem de avaliação ao eliminar atribuições e outras instruções de controle. [Antoy 2005].

O objetivo desse trabalho é implementar uma DSL (domain-specific language) embutida na linguagem Haskell para a escrita de programas Prolog. Permitindo ao programador escrever um programa na linguagem de programação lógica Prolog, mas com uma estrutura de dados semelhante à linguagem de programação funcional Haskell. O programa resultante é executado dentro do ambiente do Haskell, combinando as vantagens dos dois paradigmas em um único programa. A escolha do Haskell como ambiente principal para a DSL foi motivada pelas vantagens que a linguagem oferece, como o polimorfismo de tipos, que permite maior reutilização de código, e funções de alta ordem pré-definidas que facilitam a recursão em listas [Antoy and Hanus 2010].

A linguagem Prolog foi utilizada como *front-end*, sendo uma linguagem declarativa amplamente utilizada em ambientes relacionados à linguística computacional. A abordagem declarativa permite que o programa forneça uma descrição do problema a ser resolvido, através de fatos e regras que representam o domínio relacional do problema, em vez de especificar passo a passo como chegar à solução, como ocorre em linguagens procedurais ou imperativas [Wielemaker et al. 2012].

Esse artigo está organizado da seguinte forma: Na Seção 2, apresenta-se conceitos sobre programação em lógica e Prolog. Na Seção 3, a DSL desenvolvida neste trabalho é descrita. Na Seção 4, explica-se a implementação da DSL. Finalmente, na Seção 5, são apresentadas conclusões e ideias para projetos futuros.

2. Programação lógica e trabalhos relacionados

Nas linguagens de programação lógica, em resumo, um programa consiste em um conjunto de regras de inferência que descrevem um problema. A execução de um programa é um processo de busca de provas, durante o qual são geradas as soluções para o problema. Uma vez que os programas são apenas descrições de soluções a problemas, esse é um estilo de programação baseado em conhecimento, que tem muitas aplicações em áreas como a de processamento de linguagem natural.

A linguagem de programação lógica mais conhecida é o Prolog[3], que é baseada em lógica de primeira ordem e usa o Princípio da Resolução para construir provas. Na realidade, lógica e resolução de primeira ordem são muito gerais para serem usadas diretamente como linguagem [Fernández 2004] mas na década de 1970, Kowalski, Colmerauer e Roussel definiram e implementaram uma restrição adequada, baseada no fragmento clausal clássico de primeira ordem lógica. Isso resultou na primeira versão do Prolog [Roussel 1975].

O Prolog funciona utilizando lógica de predicados [Dinucci et al. 2016], a partir de uma série de fórmulas lógicas chamadas cláusulas de Horn [Monard and Baranauskas 2003]. Estas cláusulas possuem um lado esquerdo

[3]`https://www.swi-prolog.org`

e direito `(h ← p1, p2, ..., pn)`, onde o lado esquerdo `(h)` é verdadeiro se e, somente se, o que há no lado direito `(p1, p2, ..., pn)`, sendo um ou n requisitos, forem simultaneamente verdadeiros. Cláusulas que não possuem um lado direito (algo a ser provado) são denominados átomos, estes são fatos, verdades absolutas, que permitem a definição de predicados, entradas do programa utilizados na prova das cláusulas.

Como trabalhos relacionados, podemos citar alguns interpretadores Prolog, encontrados apenas em repositórios do GitHub, hspl[4] e propella[5], por exemplo. Também foram encontradas implementações em Haskell, utilizando a estrutura de mônadas disponíveis em Erdwolf[6] e hasklog[7]. Na qualidade de publicações, podemos citar o artigo "Embedding Prolog in Haskell" [Spivey and Seres 1999]. Onde é proposta a integração da lógica de programação Prolog com as características de funções preguiçosas da linguagem Haskell. Isso é alcançado ao transformar cada predicado Prolog em uma função Haskell, mantendo a interpretação, tanto declarativa quanto procedimental dos predicados Prolog.

3. HasKLogiX

```
parent(arne,james).
parent(arne,sachiko).
parent(koos,rivka).
parent(sachiko,rivka).
parent(truitje,koos).
```

Figura 1. Programa Prolog composto apenas de fatos (átomos) que expressa uma relação entre pais e filhos.

A Figura 1 é um exemplo simples de um programa Prolog composto apenas por fatos. Esse programa deve ser escrito na sintaxe aceita pela DSL desenvolvida, como exibido na Figura 2. Note como esse exemplo é uma lista de cláusulas simples (um único termo sem um corpo), onde cada uma delas é uma função (`Func`: uma função com um nome e uma lista de termos como argumentos) que possui em sua lista de argumentos dois átomos.

Deve-se primeiro salvar o código em um arquivo com extensão .hs e carregá-lo, utilizando o interpretador GHCI. Podemos então utilizar a função `queryResult` que realizará a consulta, nesse caso, na base de dados Prolog exibida na Figura 2 e retornará a solução encontrada.

A consulta em Prolog exemplificada na Figura 3, realizada no programa da Figura 1, pode ser escrita na sintaxe aceita pela DSL como mostrado na Figura 4. Esta consulta retorna `[("X","arne")]` que representa a substituição realizada `X = arne` como a única solução para a consulta. Se não houver solução para a consulta, a função retorna uma lista vazia.

[4]https://github.com/dcepelik/hspl
[5]https://github.com/propella/prolog
[6]https://github.com/Erdwolf/prolog
[7]https://github.com/cimbul/hasklog

```
        --Modern Compiler Design Example page 613
myExample1 :: Prolog
myExample1 = [
        Simple (Func "parent" [Atom "arne", Atom "james"]),
        Simple (Func "parent" [Atom "arne", Atom "sachiko"]),
        Simple (Func "parent" [Atom "koos", Atom "rivka"]),
        Simple (Func "parent" [Atom "sachiko", Atom "rivka"]),
        Simple (Func "parent" [Atom "truitje", Atom "koos"])]
```

Figura 2. Programa Prolog da Figura 1 escrito na sintaxe aceita pela DSL desenvolvida.

```
?- parent(X,james).
X = arne.
```

Figura 3. Exemplo de consulta no Prolog, utilizando o programa da Figura 1.

Agora, considere o programa da Figura 5 que define uma função através de dois fatos diferentes. O primeiro fato *based* define que `prolog` está ligado a `logic` e `haskell` a `maths`. O fato *likes* define que max gosta de `logic` e que claire gosta de `maths`. A regra likes (que não necessariamente precisa ter o mesmo nome do fato) compõe os dois fatos pré-estabelecidos junto ao uso de variáveis, a fim de obter uma resposta para a consulta de inferência, como exibido na Figura 6.

Esta regra é definida em termos dos fatos based e likes, onde, para satisfazer a consulta, os dois termos existentes no corpo da regra devem ser unificados. Quando consultado se existe algum X que goste de `prolog`, através da base de conhecimento e da regra definida, o algoritmo infere que, como `prolog` é baseado em lógica e que max gosta de lógica, logo, max gosta de `prolog`. O programa Prolog da Figura 5 pode ser implementado na sintaxe aceita pela DSL, como exibido na Figura 7.

4. Implementação

Para criar uma DSL é necessário escolher uma série de primitivas que serão usadas para descrever os programas naquela linguagem. No caso do Prolog, as primitivas necessárias são: 1. Predicados: As unidades básicas da programação em Prolog e que geralmente, representam relações entre entidades do domínio em questão. Por exemplo, um predicado `mãe (X, Y)` pode ser usado para representar a relação ”X é mãe de Y”; 2. Fatos: Fatos são predicados que são verdadeiros em todo o tempo de execução do programa, normalmente usados para descrever informações estáticas do domínio como, por exemplo, ”João é pai de Maria”; 3. Regras: Usadas para definir predicados em termos de outros predicados. Por exemplo: uma regra `avô (X, Y) :- pai (X, Z), pai (Z, Y)` pode ser usada para definir o predicado ”avô”em termos do predicado ”pai”; 4. Variáveis: São usadas para representar valores que podem ser instanciados durante a execução do programa. Por exemplo, na regra acima, as variáveis X, Y e Z são usadas para representar valores que serão instanciados de acordo com as entradas do usuário.

Inicialmente, foram definidos os dois tipos de dados principais do Prolog: Termo

```
queryResult myExample1 (Func "parent" [Var "X", Atom "james"])
```

Figura 4. Exemplo de uso da função `queryResult`, consultando o programa da Figura 2.

```
based(prolog,logic).
based(haskell,maths).
likes(max,logic).
likes(claire,maths).
likes(X,P) :- based(P,Y), likes(X,Y).
```

Figura 5. Exemplo de um programa com uma função em Prolog.

```
?- likes(X,prolog).
X = max ;
```

Figura 6. Consulta Prolog para o programa da Figura 5.

```
        --Maribel example page 171
myExample2 :: Prolog
myExample2 = [
        Simple (Func "based" [Atom "prolog", Atom "logic"]),
        Simple (Func "based" [Atom "haskell", Atom "maths"]),
        Simple (Func "likes" [Atom "max", Atom "logic"]),
        Simple (Func "likes" [Atom "claire", Atom "maths"]),
        Func "likes" [Var "X", Var "P"] :-
        [Func "based" [Var "P", Var "Y"], Func "likes" [Var "X", Var "Y"]]]
```

Figura 7. Programa da Figura 5 implementado na sintaxe da DSL desenvolvida.

e Cláusula, como pode ser observado na Figura 8. O tipo `Term` representa um termo Prolog e pode ser uma variável, um átomo ou uma função. O construtor `Var` representa uma variável, o construtor `Atom` representa um átomo e o construtor `Func` representa uma função que consiste em um nome de função e uma lista de termos como argumentos.

O tipo `Clause` é usado para representar uma cláusula Prolog e pode ser uma cláusula simples ou uma cláusula com um corpo. O construtor `Simple` é usado para representar uma cláusula que consiste em apenas um termo, enquanto o construtor `:-` é usado para representar uma cláusula com um corpo que consiste em uma cabeça (um termo) e o corpo da cláusula (uma lista de termos). O tipo Prolog definido nesse trabalho é simplesmente uma lista de cláusulas que serão unificadas com a consulta realizada. Estas definições são exibidas na Figura 8.

A substituição e a unificação são operações essenciais para o Prolog. Dada uma substituição `[(x1,t1), (x2,t2), ...]`, significa que a variável x1 é substituída pelo termo t1, x2 é substituído por t2 e assim por diante. Quando possível, a unificação entre dois termos gera uma ou mais substituições a serem realizadas pela função `substituteAll`, que substitui uma variável por um termo ou um átomo, em um outro termo passado como argumento, como exibido na Figura 9.

```haskell
data Term = Var String | Atom String | Func String [Term]
      deriving (Eq,Show)
data Clause = Term :- [Term] | Simple Term
      deriving (Eq,Show)

type Prolog = [Clause]
```

Figura 8. Primitivas da DSL desenvolvida.

```haskell
unify :: Term -> Term -> Subst

substituteAll :: [(String, Term)] -> Term -> Term
```

Figura 9. Cabeçalho das funções `unify` e `substituteAll`.

Antes de invocar a função `interpret` e unificar a consulta com o programa, aplicamos a operação denominada conversão alfa que substitui as variáveis utilizadas na consulta por outra nova. Isso garante que não haverá conflitos com outras variáveis já existentes no programa. O tipo `Subst` representa uma substituição que é uma associação entre variáveis e termos, consistindo em uma lista de tuplas `String` e `Term`, onde a String, que representa uma variável, é substituída pelo termo. Se `Nothing` for retornado, significa que não há nenhuma substituição possível.

A Função `interpret` que retorna uma lista de tuplas `(String,Termo)`, simbolizando uma substituição realizada, é responsável por percorrer o programa Prolog unificando as cláusulas que deram match com o termo. Se essa substituição (esse unificador) não existir, `Nothing` será retornado e como consequência, ela implementa o mecanismo de Backtracking, por meio de seu uso de correspondência de padrões e recursão. Se uma consulta tiver várias cláusulas que correspondam ao cabeçalho da consulta realizada, apenas a primeira cláusula correspondente será selecionada para ser interpretada. No entanto, se essa cláusula resultar em uma falha (ou seja, retornar `Nothing`), a função retrocederá e tentará interpretar a próxima cláusula.

```haskell
queryResult :: Prolog -> Term -> [(String, String)]

interpret :: Prolog -> Term -> Subst
```

Figura 10. Cabeçalho das funções `queryResult` e `interpret`.

A função `queryResult` resulta em uma lista de tuplas de Strings, como exibido nas Figuras 11 e 12, sendo a primeira String uma variável e a segunda o átomo unificado na variável, ou seja, a resposta para a consulta, como pode ser visto na Figura 11. Ela recebe a lista de substituições da função `interpret`.

```
GHCi, version 9.0.1: https://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling Main
Ok, one module loaded.
ghci> queryResult myExample2 (Func "likes" [Atom "prolog", Var "Y"])
[("Y","max")]
ghci>
```

Figura 11. Resultado para o exemplo da Figura 7.

```
GHCi, version 9.0.1: https://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling Main
Ok, one module loaded.
ghci> queryResult myExample2 (Func "likes" [Atom "haskell", Var "Y"])
[("Y","claire")]
ghci>
```

Figura 12. Resultado para o mesmo exemplo da Figura 7, passando um argumento diferente.

5. Conclusão

A DSL desenvolvida, disponível no repositório[9] , apresenta uma abordagem para a interpretação de programas escritos em Prolog dentro da linguagem Haskell. O desenvolvimento do interpretador permitiu explorar conceitos fundamentais da linguagem, tais como unificação, cláusulas, termos e a forma como estes são manipulados, até um resolvente ser gerado.

O desenvolvimento desta DSL também salientou a importância de definir tipos de dados claros e funções auxiliares, pois, apesar da programação funcional, nos resultar em um código enxuto e curto, apresenta uma alta complexidade. Isto por causa, por exemplo, das inúmeras chamadas recursivas ou da forma como as funções são valores de primeira categoria, o que pode ser um pouco confuso à primeira vista.

Esta implementação usa listas como estrutura de dados para representar substituições, uma vez que a linguagem de programação Haskell facilita muito sua manipulação através de suas inúmeras bibliotecas. Porém, elas podem ser ineficientes para consultas grandes ou programas com muitas variáveis [Rubinstein-Salzedo 2018]. Um método mais eficiente seria usar HashMap's ou outra estrutura de dados que forneça pesquisas e atualizações mais rápidas. O algoritmo de unificação adotado se encontra no livro "Programming Languages and Operational Semantics"de Maribel Fernandez.

É importante destacar que o interpretador implementado neste trabalho possui funcionalidades limitadas e não é capaz de lidar com todos os recursos oferecidos pelo Prolog. Recursos avançados, como regras recursivas, predicados integrados e estruturas de controle mais avançadas não são suportados pelo interpretador. Note também que nem as cláusulas do programa, nem o termo da consulta são validados, se alguma cláusula do programa tiver sintaxe incorreta, um erro de execução será exibido.

[9] https://github.com/arlvit0r/Prolog_Library

Portanto, esse trabalho pode ser visto como uma introdução à implementação do Prolog e à programação funcional em Haskell e pode servir como base para futuros trabalhos de implementação de interpretadores mais complexos. Outras melhorias no interpretador podem incluir a adição de recursos mais avançados, como o mecanismo de corte, a negação como falha e estruturas de controle, como loops if-then-else e while. Deixo também como sugestão para trabalhos futuros o desenvolvimento de um tradutor do Prolog para a DSL desenvolvida.

Referências

Antoy, S. (2005). Evaluation strategies for functional logic programming. *Journal of Symbolic Computation*, 40(1):875–903.

Antoy, S. and Hanus, M. (2010). Functional logic programming. *Commun. ACM*, 53(4):74–85.

Bloss, A. (1989). Update analysis and the efficient implementation of functional aggregates. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 26–38.

Dinucci, A., Duarte, V., Fontes, L. M., Cabeceiras, A., and de Brito, R. P. (2016). Introdução à lógica proposicional estoica. *Prometheus-Journal of Philosophy*.

Fernández, M. (2004). *Programming Languages and Operational Semantics*, volume 1 of *Texts in computing*. College Publications.

Monard, M. C. and Baranauskas, J. A. (2003). Indução de regras e árvores de decisão. *Sistemas Inteligentes-Fundamentos e Aplicações*, 1:115–139.

Roussel, P. (1975). Prolog: Manuel de reference et d'utilisation groupe d'intelligence artificielle. *UER de Luminy, Universit6 d'Aix-Marseille II*.

Rubinstein-Salzedo, S. (2018). Big o notation and algorithm efficiency. In *Cryptography*, pages 75–83. Springer.

Spivey, J. M. and Seres, S. (1999). Embedding prolog in haskell. In *Proceedings of Haskell*, volume 99, pages 1999–28.

Wielemaker, J., Schrijvers, T., Triska, M., and Lager, T. (2012). Swi-prolog. *Theory and Practice of Logic Programming*.