Terceiro trabalho - Datalog

Linguagens e Ambientes de Programação
NOVA FCT

Versão de 24 de Maio de 2024

- 17 de Maio de 2024 - Versão inicial do enunciado.
- 23 de Maio de 2024 - Correção de uma gralha no enunciado.
- 23 de Maio de 2024 - GitHub Classroom link adicionado.
- 24 de Maio de 2024 - Corrigido GitHub Classroom link.

O objectivo deste trabalho é implementar um interpretador de uma linguagem de programação, Datalog, em OCaml e usando um algoritmo de ponto fixo.

1 Introdução

Datalog[1] é uma linguagem de programação lógica cuja sintaxe é derivada do Prolog. Em Datalog, os programas são compostos por regras de produção, que são usadas para inferir novos factos a partir de factos existentes. Os programas Datalog são normalmente usados para resolver problemas de lógica, como a gestão de bases de dados, representação de conhecimento, verificação de propriedades de programas, síntese de programas, e análise de dados. Um sistema baseado em regras pode muitas vezes ser resolvido de forma simples usando Datalog. Um exemplo de um interpretador Datalog é o Datomic[2], e uma extensão do Datalog pode ser encontrada na ferramenta *Souffle*[3].

Um programa Datalog é composto por um conjunto de regras de factos, por exemplo

```
1 parent(john, mary).
2 parent(mary, ann).
```

e regras de produção, por exemplo

```
1 ancestor(X, Y) :- parent(X, Y).
2 ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
```

Neste caso, o programa Datalog define a relação `ancestor` que é verdadeira se `X` é um antepassado de `Y` e permite deduzir os seguintes factos:

[1] `https://en.wikipedia.org/wiki/Datalog`
[2] `https://www.datomic.com/`
[3] `https://souffle-lang.github.io/`

```
parent(john, mary).
parent(mary, ann).
ancestor(john, mary).
ancestor(john, ann).
ancestor(mary, ann).
```

O que acontece em algumas implementações, é que em vez de deduzir todos os factos possíveis, o interpretador responde a queries, do género:

```
?- ancestor(john, ann).
```

Uma regra de produção é composta por uma cabeça com um termo (H) e um corpo com uma lista de termos ($B_1, B_2, \ldots B_n$), separados pelo símbolo :−. A regra determina que o predicado na cabeça é verdadeiro se todos os predicados no corpo da regra forem verdadeiros (conjunção). Temos portanto implicações da forma:

$$B_1 \wedge B_2 \wedge \ldots \wedge B_n \Longrightarrow H$$

com

$$\mathsf{true} \Longrightarrow H$$

no caso dos factos.

2 Descrição do Trabalho

O trabalho consiste em implementar um motor de inferência para a linguagem Datalog. O trabalho está dividido em dois patamares de dificuldade, o primeiro é considerar apenas regras de produção para predicados sem parâmetros e o segundo é considerar regras de produção para predicados com parâmetros. O primeiro patamar é obrigatório e avaliado até 16 valores e o segundo é opcional e avaliado para 20 valores. Neste enunciado irá encontrar instruções detalhadas e ajudas que permitem implementar o trabalho. Haverá ainda aulas teóricas e práticas dedicadas a explicar partes do trabalho e a responder a dúvidas.

2.1 Linguagem Datalog: parte 1

A primeira parte do trabalho é desenvolver um algoritmo de inferência para a linguagem Datalog que suporta apenas regras de produção para predicados sem parâmetros. Esta parte é classificada para 16 valores.

O programa

```
A :−.
B :− A.
C :− B, A.
D :− C, E.
```

permite deduzir os seguintes factos

```
A.
B.
C.
```

mas não permite deduzir o facto D porque a regra de produção para D depende de um predicado E que não está definido.

Note que a primeira regra, para o predicado A, não tem premissas, ou condições, pelo que o predicado A é um facto. Esta representação permite que o interpretador só lide com regras e não com a declaração de factos e regras.

2.1.1 Algoritmo de Inferência

Para implementar esta dedução pretende-se usar o algoritmo mais simples para a semântica da linguagem Datalog, o algoritmo naive, o algoritmo de ponto fixo. Este algoritmo, que calcula o conjunto de factos F^∞ para um programa P, a partir de um conjunto vazio de dados e acrescentando os factos verdadeiros passo a passo, até que não seja possível deduzir mais factos.

Seja P o conjunto de regras de produção de um programa Datalog, o conjunto de factos F^∞ é o conjunto de factos que são deduzidos a partir de P. O algoritmo consiste nos seguintes passos:

$F^0 := \emptyset$
$i := 0$
repetir
  $F^{i+1} := \emptyset$
  para cada regra H :- $B_1, B_2, \ldots, B_n \in P$:
    se $B_1, B_2, \ldots, B_n \subseteq F^i$ então $F^{i+1} := F^{i+1} \cup \{H\}$
    $i := i + 1$
enquanto $F^{i+1} = F^i$

De uma forma mais declarativa podemos dizer que:

$F^0 := \emptyset$
$F^{i+1} = \{H \mid H$:- $B_1, B_2, \ldots, B_n \in P$ e, tal que, $B_1, B_2, \ldots, B_n \subseteq F^i\}$

Como em cada passo será necessário iterar todas as regras e verificar repetidamente a validade de todos os termos no corpo da regra, é necessário optimizar estas operações. Para implementar este algoritmo é conveniente guardar os factos numa estrutura de dados que optimize estas consultas e que armazene os factos deduzidos.

2.1.2 Especificação da linguagem

A representação de programas Datalog (parte 1) em OCaml pode ser feita com os seguintes tipos de dados:

```
1 type term = string
2 type rule = term * term list
3 type program = rule list
4 type query = term
```

sendo que o exemplo acima seria representado por:

```
1 let program = [
2   ("A", []);
3   ("B", ["A"]);
4   ("C", ["B"; "A"]);
5   ("D", ["C"; "E"])
6 ]
```

2.1.3 Especificação do motor de inferência

Pretende-se implementar uma função que aceita um programa e uma query e devolve um booleano que significa que a query é satisfeita pelo programa. A função deve ser implementada em OCaml e ter a seguinte assinatura:

```
val solve: program -> query -> bool
```

Os resultados para alguns testes são o seguintes:

```
assert (solve program "A");
assert (solve program "B");
assert (solve program "C");
assert (not (solve program "D"));
```

Note-se que estamos a ignorar o parsing dos programas a partir de texto e estamos a partir logo da representação abstrata dos programas.

2.2 Linguagem Datalog: parte 2

A segunda parte do trabalho inclui a implementação de um motor de inferência para a linguagem Datalog que suporta regras de produção para predicados com parâmetros. Esta parte é classificada para 20 valores.

Considere o programa Datalog,

```
A(0) :-.
B(X) :-.
C(Y) :- A(Y), B(Y).
D(Y) :- B(Y).
```

que permite deduzir os seguintes factos

```
A(0).
B(X).
C(0).
D(X).
```

onde os predicados B(X) e D(X) são válidos para qualquer valor de X e os predicados A(0) e C(0) são apenas válidos para o valor 0.

2.2.1 Algoritmo de Inferência

O algoritmo de ponto fixo deve ser agora estendido para lidar com predicados com parâmetros. O algoritmo é o mesmo, mas a comparação dos termos deve agora ter em conta a unificação entre variáveis, entre variáveis e constantes, e entre constantes. Para isso é preciso encontrar substituições de variáveis por outras variáveis que unificam os termos que estamos a comparar.

Seja P o conjunto de regras de produção de um programa Datalog, o conjunto de factos F^{∞} é o conjunto de factos que são deduzidos a partir de P. O algoritmo consiste nos seguintes passos:

$F^0 := \emptyset$
$i := 0$
repetir
  $F^{i+1} := \emptyset$
  para cada regra H :- $B_1, B_2, \ldots, B_n \in P$:
    para cada substituição θ unificando $B_1, B_2, \ldots, B_n$ em F^i :
      $F^{i+1} := F^{i+1} \cup \{H\theta\}$
    $i := i + 1$
enquanto $F^{i+1} = F^i$

Este exemplo inclui a aplicação de uma substituição θ a um termo H, que é escrita $H\theta$. Quer a formação de substituições, quer a aplicação de substituições a termos, são operações que são explicadas nas secções seguintes.

De uma forma mais declarativa podemos também dizer que:

$$F^0 := \emptyset$$
$$F^{i+1} = \{H\theta \mid H \text{ :- } B_1, B_2, \ldots, B_n \in P, \\ \theta \text{ unifica } B_1, B_2, \ldots, B_n, \text{ tal que, } B_1\theta, B_2\theta, \ldots, B_n\theta \subseteq F^i\}$$

Esta definição pode ler-se, todos os factos que unificam com os factos da iteração anterior (que já estão deduzidos) e que são utilizados numa regra de produção, dão origem a uma substituição (instanciação de parâmetros) que é aplicada à cabeça da regra de produção, que é adicionada ao conjunto de factos deduzidos.

2.2.2 Especificação da linguagem em OCaml

A diferença para a linguagem da primeira parte é que os termos passam a ser uma lista de parâmetros, que podem ser ou variáveis ou constantes (como prova de conceito vamos apenas utilizar valores inteiros). A representação do programa Datalog (parte 2) em OCaml pode ser feita com os seguintes tipos de dados:

```
1 type parameter = Var of string | Value of int
2 type term = string * parameter list
3 type rule = term * term list
4 type program = rule list
5 type query = term
```

sendo que o exemplo acima seria representado por:

```
1 let program =
2 [ (("A", [Value 0]), [])
3 ; (("B", [Var "X"]), [])
4 ; (("C", [Var "Y"]), [( "A", [Var "Y"]); ( "B", [Var "Y"])])
5 ; (("D", [Var "Y"]), [( "B", [Var "Y"])])
6 ]
```

2.3 Especificação do motor de inferência

Pretende-se implementar uma função que aceita um programa, em que os predicados têm parâmetros, e uma query (com parâmetros) e devolve um booleano que significa que a query é satisfeita pelo programa. A função deve ser implementada em OCaml e ter a seguinte assinatura:

```
solve : program -> query -> bool
```

Alguns testes que devolvem **true** são o seguintes:

```
    solve program ( "A", [Value 0] )
    solve program ( "C", [Value 0] )
not(solve program ( "C", [Value 1] ))
    solve program ( "D", [Var "X"] )
    solve program ( "B", [Value 99] )
```

Note-se (novamente) que estamos a ignorar o parsing dos programas a partir de texto e estamos a partir logo da representação abstrata dos programas.

Na implementação deste motor de inferência é necessário implementar funções auxiliares para tratar termos com variáveis. Estas funções são fornecidas como ponto de partida para a implementação do projeto, são explicadas nas secções seguintes.

2.3.1 Substituições

Uma substituição θ é um mapeamento de variáveis para termos e é representada por uma lista de pares de variáveis e de termos. Por exemplo, a substituição que mapeia a variável `X` para o termo `Value 0` é representada por `[("X", Value 0)]`. A aplicação de uma substituição a um termo $H\theta$ é feita através da função `apply` que substitui todas as ocorrências de variáveis por termos na substituição.

```
let apply sigma = function
  | Var v -> (try List.assoc v sigma with Not_found -> Var v)
  | Value v -> Value v

let apply_term (f, ps) sigma = (f, List.map (apply sigma) ps)
```

Por exemplo, a aplicação individual de uma substituição a um parâmetro, tem o seguinte resultado:

```
apply [("X", Value 0)] (Var "X") = Value 0
```

No caso dos termos, em que têm um nome e uma lista de parâmetros, a substituição é aplicada a todos os parâmetros, preservando a estrutura do termo. Os resultados são os seguintes, para os exemplos:

```
apply_term ("A", [Var "X"]) [("X", Value 0)] = ("A", [Value 0])
apply_term ("A", [Var "X"; Var "Y"; Var "Z"]) [("X", Value 0); ("Y", Value 1)] =
           ("A", [Value 0; Value 1; Var "Z"])
```

2.3.2 Comparação de termos fechados

Comparar dois termos fechados, onde os seus parâmetros são constantes, é uma operação simples. A comparação destes termos resulta da operação de comparação nativa do OCaml que verifica se os termos têm o mesmo nome e se os parâmetros são iguais.

2.3.3 Comparação de termos abertos

O caso de termos abertos é diferente. Termos abertos são aqueles que contém variáveis nos seus parâmetros. Para comparar termos e encontrar substituições que unificam termos, é necessário normalizar os termos. Comparar A(X) com A(Y) deveria ser possível, mas se a comparação for feita de forma literal dará sempre resultado negativo. A normalização de um termo é a substituição de todas as variáveis por novas variáveis que não dependam da sintaxe dada pelo utilizador. Quando guardamos um termo $A(X)$ queremos que ele seja comparável a um termo $A(Y)$ porque estas variáveis representam qualquer valor. A normalização de um termo é feita através da função normalize que substitui todas as variáveis por novas variáveis cujo nome representa a sua posição no predicado.

```
1  let subst_of_head (f,ps) =
2    List.fold_left
3      (fun (i,sigma) -> function
4        | (Var v) ->
5          begin try
6            ignore(List.assoc v sigma); (i,sigma)
7          with Not_found -> (i+1,(v, Var ("_"^ string_of_int  i ))::sigma)
8          end
9        | p -> (i,sigma))
10     (0,[]) ps
11
12 let normalize_variables (head, body) =
13   let (_,sigma) = subst_of_head head in
14   (apply_term head sigma, List.map (fun t -> apply_term t sigma) body)
```

A função normalize_variables começa por invocar a função subst_of_head que devolve uma substituição que mapeia as variáveis do termo da cabeça para novas variáveis cujos nomes dependem da sua posição. Depois, normalize_variables aplica a substituição a todos os termos da regra de produção. Sendo assim, alguns exemplos de normalização de termos são os seguintes:

```
1 subst_of_head ("A", [Var "X"]) = (0, [("X", Var "_0")])
2 normalize_variables (("A", [Var "X"]), []) = (("A", [Var "_0"]), [])
3 normalize_variables (("A", [Var "X"]), [("B", [Var "X"])]) =
4             (("A", [Var "_0"]), [("B", [Var "_0"])])
5 normalize_variables (("A", [Var "X"; Var "Y"]), [("B", [Var "Y"])]) =
6             (("A", [Var "_0"; Var "_1"]), [("B", [Var "_1"])])
```

Desta forma quando se verificar se A(_0) unifica com outra instância do predicado A com uma variável, ela vai também ter a variável _0 como parâmetro. Não confundir com o número/valor 0.

2.3.4 Unificação

A comparação de termos abertos, para responder a uma query, para saber se um termo numa regra é válido, é feita através de um processo de unificação. A unificação é o processo de encontrar uma substituição de variáveis por outras variáveis ou valores que torne dois termos iguais. As funções seguintes implementam a unificação de parâmetros (variáveis e valores), de termos, e de listas de termos.

A função unify_param definida a seguir que compara dois parâmetros e devolve uma substituição que os unifica. A função complementa uma substituição já existente com a necessária adenda para unificar o parâmetro corrente. Se não houver substituição possível, a função devolve None.

```
let unify_param p p' sigma =
  let p = apply sigma p in
  let p' = apply sigma p' in
  match p, p' with
  | Value v, Value v' when v = v' -> Some sigma
  | Var v, Var v' when v = v' -> Some sigma
  | Var v, Var v' -> Some ((v, Var v')::sigma)
  | Var v, Value v' -> Some ((v, Value v')::sigma)
  | Value v, Var v' -> Some ((v', Value v)::sigma)
  | _ -> None
```

Com base nesta função podemos implementar a unificação de termos e de listas de termos. A função unify_term compara dois termos e devolve uma substituição que os unifica.

```
let unify_term (f, ps) (f', ps') sigma =
  if f = f' then begin
    let rec unify_params ps ps' sigma =
      match ps, ps' with
      | [], [] -> Some sigma
      | p::ps, p'::ps' ->
        let sigma = unify_params ps ps' sigma in
        bind_maybe (unify_param p p') sigma
      | _ -> None
      in
    unify_params ps ps' sigma
  end
  else None
```

Esta função faz uso de uma função auxiliar para tratar o facto de trabalhar com o tipo option para representar o não haver uma substituição.

```
let bind_maybe f x = match x with
  | None -> None
  | Some x -> f x
```

3 Critérios de avaliação

Para além da correção do programa, obtida através da execução de testes automáticos, a avaliação do trabalho terá em conta a qualidade do código, a clareza da implementação e a documentação do código.

O trabalho é avaliado em 2 vertentes, com dois projetos presentes no repositório. A primeira parte é avaliada para 16 valores e a segunda é avaliada para 20 valores. A avaliação da segunda parte é feita apenas se a primeira parte estiver completa e funcional.

4 Testes e Entrega

No repositório (criado em: https://classroom.github.com/a/CizYSU2T) estão um conjunto de testes que exercitam as funções de resolução para ambas as partes (em ficheiro diferentes). Para executar os testes, basta correr o comando `dune runtest`. É importante usarem o repositório *git* como meio de trabalho de forma a garantirem que todos os testes passam antes da data de submissão do trabalho.

5 Data de entrega

O prazo de entrega é o dia 31 de Maio às 23:59. A entrega deve ser feita através do github classroom, com um push no repositório que for criado para o efeito quando aceitar o *assignment* disponível online em link a disponibilizar brevemente.