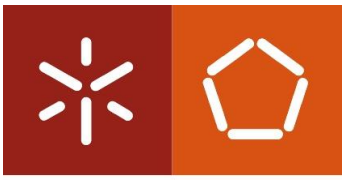

Universidade do Minho

Departamento de Informática

Programação em lógica estendida e Conhecimento imperfeito

Grupo 7

Joao Teixeira (A85504)
Jose Filipe Ferreira (A83683)
Miguel Solino (A86435)

3 de Maio de 2020

Resumo

O objetivo deste projeto é desenvolver um sistema de representação de conhecimento e raciocínio, com recurso à linguagem de programação em lógica *Prolog*.

Conteúdo

Capítulo 1

Introdução

Este projeto tem como objetivo a criação de um sistema de representação de conhecimento e raciocínio que consiga representar informação sobre contratos públicos.

Para a realização deste trabalho utilizamos a linguagem de programação em lógica lecionada nesta UC, *Prolog*.

Em primeiro lugar, iremos apresentar em detalhe o problema proposto, identificar os requisitos e que funcionaliades implementar para responder aos requisitos determinados.

De seguida explicaremos a solução encontrada para este problema, entrando em detalhe sobre as representaçoes de conhecimento escolhidas, diversos invariantes, como é efetuada a evolução e regressão do conhecimento, entre outros, entre outros..

Capítulo 2

Problema

O programa tem que cumprir os seguintes requisitos:

- Representar conhecimento positivo e negativo;
- Representar casos de conhecimento imperfeito, com recurso à utilização de valores nulos impercisos, interditos e incertos;
- Manipulação de invariantes, com o intuito de manter a consistência do conhecimento na base de conhecimento;
- Permitir a evolução e regressão do conhecimento existente;
- Possuir um sistema de inferência capaz de seguir os mecanismos de raciocínio inerente ao sistema.

Capítulo 3

Solução

3.1 Predicados Implementados

Para a implementação deste sistema, foram criados três predicados para representar a informação do sistema

- adjudicante: NifAd, Nome, Morada → {V, F, D};
- adjudicária: NifAda, Nome, Morada → {V, F, D};
- contrato: #Id, NifAd, NifAda, TipoDeContrato, TipoDeProcedimento, Descrição, Valor, Prazo, Local, Dia, Mês, Ano → {V, F, D};

Como estamos perante um contexto de lógica estendida, os predicados podem ter valor de Verdadeiro, Falso, ou Desconhecido.

3.2 Representação do conhecimento

3.2.1 Conhecimento Positivo

O conhecimento positivo é representado através da inserção simples na base de conhecimento como nos exemplos abaixo:

```
adjudicária(500000000, sarilhos, braga).
adjudicária(500000001, limpalimpinho, porto).
adjudicária(500000002, oficina, lisboa).

adjudicante(600000000, bdp, lisboa).
adjudicante(600000001, chmtad, vilareal).

contrato(0, 600000000, 500000000, "Aquisicao de servicos", ad,
    "Assessoria juridica", 4000, 100, porto, 10, 10, 2010).
contrato(1, 600000003, 500000000, "Aquisicao de servicos", cprev,
    "Assessoria juridica", 1000000, 346, porto, 10, 10, 2010).
```

3.2.2 Conhecimento Negativo

Com a presença de conhecimento imperfeito, faz sentido a introdução de conhecimento negativo. Este pode ser obtido representado através de duas formas, negação explícita, e também a negação forte.

A negação explícita é representada através da inserção simples na base de conhecimento como no exemplo abaixo:

```
-adjudicante(600000010, bombeiros, evora).
```

Neste exemplo corresponde a dizer que não existe um adjudicante com o Nif 600000010, com o nome "bombeiros"e sede em Évora.

A inserção deste tipo de conhecimento corresponde a adicionar o prefixo - a um predicado, fazendo-o assim ser a negação deste.

A negação forte deriva da aplicação do Pressuposto do Mundo Fechado, isto é, o conhecimento que não tem prova de ser verdadeiro, é implicitamente falso. Para a implementação deste pressuposto, procedemos à extensão do predicado -T, como representado abaixo, onde T corresponde ao predicado de *adjudicante*:

```
-adjudicante(Nif, Nome, Morada) :-
    not(adjudicante(Nif, Nome, Morada)),
    not(excecao(adjudicante(Nif, Nome, Morada))).
```

Como visto no exemplo, o conhecimento será interpretado como negativo caso não seja positivo, nem seja uma exceção, o como abaixo veremos, será interpretado como desconhecido.

3.2.3 Conhecimento Imperfeito Incerto

O conhecimento imperfeito incerto corresponde a conhecimento ao qual não temos a certeza, nem temos ideia do que poderá ser. Para a representação deste, inserimos o facto na base de conhecimento, com o campo o qual não temos a certeza com um valor arbitrário, uma exceção que será correspondida sempre que se tente aceder ao conhecimento, e inserimos na base de conhecimento a informação do campo nulo.

Abaixo, está presente um exemplo da representação deste tipo de conhecimento, neste caso sendo correspondente à adjudicária com Nif 500000003, com sede em Santarém, da qual não sabemos o nome.

```
adjudicataria(500000003, incerto, santarem).
excecao(adjudicataria(Nif, _, Morada)) :-
    adjudicataria(Nif, incerto, Morada).
incertoNome(adjudicataria(500000003), incerto).
```

3.2.4 Conhecimento Imperfeito Impreciso

O conhecimento imperfeito impreciso consiste no conhecimento desconhecido dentro de um conjunto finito de hipoteses. Para a representação deste fazemos a inserção de exceçoes com as diversas hipoteses, inserimos também o facto do Id corresponder a conhecimento impreciso.

No exemplo abaixo podemos ver a representação do adjudicante com Nif 600000005, e sede em Lisboa, o qual não sabemos se corresponde aos Bombeiros ou ao Exercíto.

```
excecao(adjudicante(600000005, exercito, lisboa)).
excecao(adjudicante(600000005, bombeiros, lisboa)).
impreciso(adjudicante(600000005)).
```

3.2.5 Conhecimento Imperfeito Interdito

O conhecimento interdito corresponde ao conhecimento que nos é desconhecido, mas nunca vai ser possível descobrir.

Para a representação deste conhecimento, procedemos à introdução do predicado com um valor arbitrário no campo desconhecido, da exceção que vai ser correspondida quando quisermos aceder ao campo desconhecido, da informação que o campo corresponde a um valor interdito, e por fim, de um invariante que garante que não é inserido novo conhecimento com que não tenha o campo interdito.

Como exemplo, apresentamos abaixo o contrato com Id 5, o qual não é possível saber o adjudicante.

```
contrato(5, 600000001, interdito, "Aquisicao de servicos", cprev,
    "Assessoria juridica", 2000, 520, porto, 10, 10, 2010).
excecao(contrato(Id, Ad, _, TContrato, TProcedimento, Descricao,
    Custo, Prazo, Local, Dia, Mes, Ano)) :-
        contrato(Id, Ad, interdito, TContrato, TProcedimento,
            Descricao, Custo, Prazo, Local, Dia, Mes, Ano).
interditoAda(contrato(5), interdito).
+contrato(_, _, _, _, _, _, _, _, _, _, _, _) :: (findall(Ada,
        (contrato(5, _, Ada, _, _, _, _, _, _, _, _, _),
        not(interditoAda(contrato(5), Ada))), S),
    length(S, N),
    N == 0).
```

3.3 Evolução do conhecimento

3.3.1 Evolução do conhecimento perfeito

Para a evolução do conhecimento perfeito temos duas hipoteses, quando já existe conhecimento imperfeito referente ao predicado em questão na base de conhecimento, ou quando é a primeira informação que inserimos sobre o predicado.

Na primeira hipotese, para a atualização do conhecimento imperfeito, primeiro é verificado se existe informação sobre conhecimento interdito, que como não é possível saber, não é possível atualizar este predicado. Caso não exista referência a conhecimento interdito, é removido o conhecimento imperfeito e adicionado o conhecimento perfeito. No exemplo abaixo, é possível observar como é efetuado a atualização do conhecimento de um adjudicante.

```
evolucao(adjudicante(Nif, Nome, Morada)) :-
    not(interditoNome(adjudicante(Nif), _)),
    not(interditoMorada(adjudicante(Nif), _)),
    removeIncerto(adjudicante(Nif)),
    inserir(adjudicante(Nif, Nome, Morada)).
```

Caso não haja informação presente para o predicado a adicionar, é apenas feita a inserção na base de conhecimento, como visivel abaixo.

```
evolucao(Termo) :-
    demo(Termo, falso),
    inserir(Termo).
```

3.3.2 Evolução do conhecimento imperfeito Impreciso

A evolução deste tipo de conhecimento é efetuada recebendo uma lista com as hipoteses possíveis, sendo verificado se é fornecido uma lista com mais do que um elemento, se todos os elementos da lista correspondem ao mesmo Id, com o predicado *mesmoNif*, se nenhum representa conhecimento anteriormente declarado como perfeito, com o predicado *nenhumPerfeito*, e só assim são inseridas as exceçoes correspondentes ao conhecimento fornecido, através do predicado *insereExcecoes*.

Como exemplo, incluimos o predicado responsável pela evolução do predicados correspondentes a adjudicatárias.

```
evolucaoImpreciso([adjudicataria(Nif, N, M) | R]) :-
    R \= [],
    mesmoNif(R, Nif),
    nenhumPerfeito([adjudicataria(Nif, N, M) | R]),
    insereExcecoes([adjudicataria(Nif, N, M) | R]).
```

3.3.3 Evolução do conhecimento imperfeito Incerto

A evolução deste tipo de conhecimento é efetuada, primeiro verificando que não existe já conhecimento sobre o campo no qual a informação é incerta, depois passa por verificar que o conhecimento não foi já marcado como interdito, e por fim é inserido o conhecimento, respeitando os invariantes já determinados.

Como exemplo podemos ver o predicado que insere uma adjudicataria com nome incerto.

```
evolucaoIncertoNome(adjudicataria(Nif, N, M)) :-
    demo(adjudicataria(Nif, _, M), falso),
    not(interditoMorada(adjudicataria(Nif), _)),
    assert(excecao(adjudicataria(_, _, _)) :- adjudicataria(_, N, _)),
    inserir(adjudicataria(Nif, N, M)),
    assert(incertoNome(adjudicataria(Nif), N)).
```

3.3.4 Evolução do conhecimento imperfeito Interdito

A evolução deste tipo de conhecimento consiste em primeiro lugar, inserir o predicado com o conhecimento interdito na base de conhecimento, respeitando os invariantes definidos, e se estes forem respeitados, inserir os restantes elementos necessários à representação deste tipo de conhecimento.

Para exemplo, apresentamos abaixo o predicado que permite a evolução de um adjudicante que tem o nome interdito

```
evolucaoInterditoNome(adjudicante(Nif, Nome, Morada)) :-
    inserir(adjudicante(Nif, Nome, Morada)),
    assert(excecao(adjudicante(_, _, _)) :- adjudicante(_, Nome, _)),
    assert(interditoNome(adjudicante(Nif), Nome)),
    assert(+adjudicante(_, _, _) :: (findall(L, (adjudicante(Nif, L, _), not(interditoNome(adjudi
```

3.4 Regressão do conhecimento

3.4.1 Regressão de conhecimento perfeito

Para a regressão do conhecimento perfeito, é verificado se o predicado é verdadeiro e, em seguida, é removido o conhecimento caso sejam respeitados todos os invariantes.

3.4.2 Regressão do conhecimento Imperfeito

A regressão de conhecimento imperfeito é efetuada, verificando se o predicado existe, no caso de conhecimento incerto, ou se a exceção correspondente, no caso de conhecimento impreciso. Depois desta verificação, são encontrados todos os invariantes que se aplicam, e depois removido o conhecimento e testado se os invariantes ainda são respeitados. Caso não sejam, as alteraçoes são revertidas.

Capítulo 4

Sistema de inferência

Para a implementação dos valores desconhecidos, foi preciso desenvolver um novo sistema de inferência, capaz de lidar com valores verdadeiros, falsos e também desconhecidos. Assim chegamos ao seguinte sistema de inferência:

```
demo( Questao,verdadeiro ) :-
    Questao.
demo( Questao, falso ) :-
    -Questao.
demo( Questao,desconhecido ) :-
    not( Questao ),
    not( -Questao ).
```

Aqui, caso o predicado seja verdadeiro, irá responder verdadeiro, se o predicado for falso, ou seja, ser negado explícitamente, ou não haver informação que o prove, a resposta será falsa, e se existir conhecimento imperfeito sobre o predicado a avaliar, retornará desconhecido.

4.1 Invariantes

4.1.1 Inserção e Remoção de conhecimento

Para garantir a consistência do conhecimento presente na base de conhecimento, foram implementados alguns invariantes, como por exemplo:

```
+adjudicante(Nif, _, _) :: (findall(Nif, (adjudicante(Nif, _, _)), S),
                                    length(S, N),
                                    N == 1).
```

O invariante acima garante que apenas há um adjudicante associado a um Nif. De forma análoga, foram implementados invariantes para as adjudicatárias e para contratos e respetivos Ids.

Para garantir que os contratos tem informação válida, foi criado também um invariante que garante que ambos os Nifs do Adjudicante e Adjudicatária existem na base de conhecimento.

```
+contrato(_, Adjudicante, Adjudicataria, _, _, _, _, _, _, _, _, _)
    :: (findall(Adjudicante, (adjudicante(Adjudicante, _, _)), Se),
        length(Se, Ne),
        Ne == 1,
        findall(Adjudicataria, (adjudicataria(Adjudicataria, _, _)), Sa),
        length(Sa, Na),
        Na == 1).
```

Para evitar fraudes e destruição de informação sensível, incluimos um invariante que impede a remoção de contratos.

```
-contrato(_, _, _, _, _, _, _, _, _, _, _, _) :: fail.
```

De forma a garantir que não é possível invalidar informação de contratos já criados, apenas é possível a remoção de adjudicantes e adjudicatárias que não tenham contratos associados. No exemplo seguinte é apresentado o caso dos adjudicantes, sendo o das adjudicatárias análogo.

```
-adjudicante(Nif, _, _) ::
    (findall(Id, contrato(Id, Nif, _, _, _, _, _, _, _, _, _, _), S),
    length(S, 0)).
```

4.1.2 Regra dos três anos

Para implementar a regra dos três anos, foi criado um invariante que faz o somatório do valor de todos os contratos entre um adjudicante e uma adjudicatária, exceto o contrato a adicionar, e verifica se o valor total é inferior a 75000 euros.

```
+contrato(Id, Ad, Ada, TC, TP, D, C, P, L, Dia, Mes, Ano)
    :: (findall(Custo, (contrato(_, Ad, Ada, _, _, _, Custo, _, _, _, _, A),
                        A >= Ano - 3,
                        A =< Ano,
                        not(contrato(Id, Ad, Ada, TC, TP, D, C, P, L, Dia, Mes, Ano))),
                S),
            sum(S, R),
            R =< 75000).
```

4.1.3 Regras gerais de Contratos

De forma a garantir que todos os contratos seguem os termos legais impostos, foi implementado um invariante para validar todas as regras, que possui uma data válida e o tipo de procedimento é válido. Para isso foi implementado o predicado *checkContract*, e o invariante verifica que não há nenhum contrato que não verifique as condiçoes.

```
+contrato(_, _, _, _, _, _, _, _, _, _, _, _)
    :: (findall(Id,
            (not(checkContract(contrato(Id, _, _, _, _, _, _, _, _, _, _, _)))),
            S),
        length(S, N),
        N == 0).
```

Capítulo 5

Conclusão

Com este trabalho prático foi-nos possível consolidar conhecimentos sobre os diversos tipos de conhecimento imperfeito, e por em prática os conceitos de lógica estendida adquiridos no decorrer desta Unidade curricular. Posto isto fazemos um balanço bastante positivo deste trabalho.