Universidade do Minho

Trabalho Prático II

Mestrado Integrado em Engenharia Informática

Sistemas de Representação de Conhecimento e Raciocínio.

2º Semestre — 2018/2019

a77523 - Alexandre Martins

a74814 - João Bernardo Freitas

a74806 - João Amorim

ae4560 - Tiago Gomes

.

22 de Abril 2019

Resumo

O presente trabalho tem como propósito o desenvolvimento de um sistema de representação de conhecimento e raciocínio com capacidade de caracterizar um universo de discurso na área de prestação de cuidados de saúde, no âmbito da representação de conhecimento imperfeito com a utilização de valores nulos e da criação de mecanismos de raciocínio adequados, recorrendo à programação lógica em *Prolog*.

De forma a abordar os desafios propostos, desenvolvemos uma base de conhecimento com os componentes que determinamos apropriados, tendo em especial atenção algumas alterações que ocorreram no panorama geral do conhecimento tendo em conta o enunciado do primeiro trabalho prático. Será realizada uma introdução aos objetivos que pretendemos cumprir, com uma posterior caracterização de todo o sistema desenvolvido.

No final, será feita uma reflexão crítica sobre todo o trabalho desenvolvido.

Contents

1 Introdução

A linguagem de programação PROLOG é baseada na lógica matemática, nesse sentido, permite expressar programas na forma lógica e usar processos de inferência para produzir resultados. De facto, é uma linguagem de programação declarativa que difere da semântica imperativa e funcional das habituais linguagens. Para além deste facto, a escolha de representação de conhecimento é uma tarefa bastante árdua, uma vez que, é necessário garantir a construção de uma base de conhecimento, estável e consistente. Em suma, o principal objetivo é a introdução da programação em lógica aos alunos, através de um exercício que consiste na estruturação de conhecimento dinâmico envolvendo uma área de prestação de cuidados de saúde.

Neste segundo desafio prático, é solicitado o desenvolvimento de um sistema de representação de conhecimento e raciocínio capaz de caracterizar o discurso na área da prestação de cuidados de saúde, no âmbito da representação de conhecimento imperfeito com a utilização de valores nulos e da criação de mecanismos de raciocínio adequados. Fazendo uso das características da linguagem PROLOG, são implementadas funcionalidades que dão resposta ao pedido no enunciado, assim como a algum conteúdo adicional.

2 Preliminares

Para o desenvolvimento do problema proposto, a aprendizagem efectuada pelo grupo no decorrer das aulas, assim como a realização do primeiro trabalho prático foi crucial para o desenvolvimento de todo o projecto. Para além disso, uma análise cuidada e prévia do enunciado proposto, tal como algumas pesquisas de material na web permitiram um desenvolvimento mais completo do trabalho.

3 Descrição do Trabalho e Análise dos Resultados

À semelhança do primeiro exercício prático, o objetivo do trabalho é o de desenvolver um sistema de representação de raciocínio sobre a prestação de cuidados de saúde. No primeiro trabalho tinhamos como conhecimento obrigatório o utente, o serviço e a consulta, tendo sido posteriormente adicionados os conhecimentos sobre instituição e médico. Neste caso, o conhecimento sobre "serviços" foi trocado pelo de "prestadores" o que levou à automática eliminação de tudo

aquilo que no primeiro exercício envolvia os "médicos", visto que ambos se referiam ao mesmo, e as "consultas" passaram a ser chamadas de "cuidados". No caso da "instituição", consideramos que deveria ser tratado como qualquer outra entidade referida no enunciado, tal como no primeiro exercício. Desta vez não será feita qualquer distinção entre trabalho extra desenvolvido, visto que tudo o que diz respeito a instituições corresponde ao mesmo.

3.1 Inserção de Conhecimento Positivo e Negativo

Neste trabalho é necessário fazer o tratamento de conhecimento positivo, negativo e imperfeito. Nesta secção serão apresentadas todas as características do conhecimento positivo e negativo referentes a este enunciado, sem grande aprofundamento nas características do conhecimento positivo, uma vez que já foi apresentado no trabalho anterior.

3.1.1 Conhecimento Positivo

Passemos então à representação do conhecimento positivo:

Utente:

```
% Extensao do predicado utente: #IdUt, Nome, Idade, Morada -> {V,F}

utente( 0, 'Elias',   28, 'Rua Jose Joaquim').
utente( 1, 'Sebastiao', 18, 'Rua Pinto Salgado').
utente( 2, 'Martim',  32, 'Rua da Luz').
utente( 3, 'Lucia',   55, 'Rua Vasconcelos').
utente( 4, 'Alexandre', 76, 'Rua da Feira').
utente( 5, 'Juliana', 22, 'Rua da Nazare').
utente( 6, 'Joao',    8,  'Rua Albertino Sousa').
utente( 7, 'Antonio', 16, 'Rua Estreita').
utente( 8, 'Isabela', 36, 'Rua Larga').
utente( 9, 'Artur',   51, 'Rua Comprida').
utente(10, 'Alice',   86, 'Rua Curta').
```

Prestador:

```
% Extensao do predicado prestador: #IdPrest, Nome, Especialidade,
    Instituicao -> {V, F}
```

```
prestador( 0, 'Andre', 'Dermatologia', 2).
prestador( 1, 'Maria', 'Psiquiatria', 1).
prestador( 2, 'Josefina', 'Neurologia', 0).
prestador( 3, 'Jose', 'Ortopedia', 3).
prestador( 4, 'Helena', 'Podologia', 4).
prestador( 5, 'Bernardo', 'Cardiologia', 5).
prestador( 6, 'Tiago', 'Oftalmologia', 6).
prestador( 7, 'Nuno', 'Pediatria', 7).
prestador( 8, 'Vasco', 'Psicologia', 7).
```

Cuidado:

```
% Extensao do predicado cuidado: Data, #IdUt, #IdPrest, Descricao,
    IdMed, Custo -'Consulta de' {V, F}

cuidado(data(20, 2, 2019), 0, 4, 'Consulta de Podologia', 20).
cuidado(data(20, 2, 2019), 1, 0, 'Consulta de Dermatologia', 70).
cuidado(data(21, 3, 2019), 1, 7, 'Consulta de Pediatria', 35).
cuidado(data(21, 3, 2019), 2, 7, 'Consulta de Pediatria', 50).
cuidado(data(07, 4, 2019), 3, 5, 'Consulta de Cardiologia', 95).
cuidado(data(07, 4, 2019), 4, 3, 'Consulta de Ortopedia', 45).
cuidado(data(10, 4, 2019), 4, 6, 'Consulta de Oftalmologia', 25).
cuidado(data(10, 4, 2019), 5, 1, 'Consulta de Psiquiatria', 40).
cuidado(data(10, 4, 2019), 6, 2, 'Consulta de Neurologia', 50).
cuidado(data(13, 4, 2019), 6, 1, 'Consulta de Psiquiatria', 40).
cuidado(data(13, 4, 2019), 7, 6, 'Consulta de Oftalmologia', 25).
cuidado(data(19, 4, 2019), 8, 7, 'Consulta de Pediatria', 35).
cuidado(data(01, 5, 2019), 8, 0, 'Consulta de Dermatologia', 70).
cuidado(data(01, 5, 2019), 9, 3, 'Consulta de Ortopedia', 45).
cuidado(data(06, 6, 2019), 10, 2, 'Consulta de Neurologia', 50).
cuidado(data(12, 5, 2019), 10, 0, 'Consulta de Dermatologia', 70).
cuidado(data(12, 5, 2019), 10, 4, 'Consulta de Podologia', 20).
cuidado(data(13, 5, 2019), 2, 8, 'Consulta de Psicologia', 50).
```

Instituição:

```
% Extensao do predicado instituicao: #Id, Nome, Cidade -> {V, F}
instituicao(0, 'Hospital de Braga',      'Braga').
instituicao(1, 'Hospital Sao Joao',      'Porto').
instituicao(2, 'Hospital de Santa Maria', 'Porto').
instituicao(3, 'Hospital de Santa Maria', 'Lisboa').
instituicao(4, 'Hospital Santa Maria Maior', 'Barcelos').
instituicao(5, 'Hospital Escala',        'Braga').
instituicao(6, 'Hospital de Faro',       'Faro').
```

```
instituicao(7, 'Hospital de Famalicao', 'Famalicao').
```

3.1.2 Conhecimento Negativo

Tal como já foi mencionado, é necessário criar um novo tipo de conhecimento ao qual chamamos de conhecimento negativo. O conhecimento negativo pode ser visto como o contrário do conhecimento positivo, visto que o objetivo do negativo é o de informar que, por exemplo, um determinado utente não existe na base do conhecimento ao representá-lo através de uma negação forte, enquanto que o do positivo é o de informar que o utente realmente existe. De notar também que foi assumido o Pressuposto do Mundo Fechado para utente, prestador, cuidado e instituição, assumindo assim a não existência de qualquer um dos mesmos que não esteja definido no universo de discurso. De notar que o predicado nao utilizado de seguida será fornecido em Anexo.

Utente:

```
% Pressuposto do Mundo Fechado para utente.

-utente(Id, Nome, Idade, Morada) :-
    nao(utente(Id, Nome, Idade, Morada)),
    nao(excecao(utente(Id, Nome, Idade, Morada))).

% Negacoes fortes para utente.

-utente( 21, 'Diogo',  12, 'Rua Benjamim Salgado').
-utente( 22, 'Helena', 46, 'Rua Humberto Salgado').
-utente( 23, 'Jose',  45, 'Rua Antonio Salgado').
```

Prestador:

```
% Pressuposto do Mundo Fechado para prestador.

-prestador(Id, Nome, Esp, IdInst) :-
    nao(prestador(Id, Nome, Esp, IdInst)),
    nao(excecao(prestador(Id, Nome, Esp, IdInst))).

% Negacoes fortes para prestador.

-prestador( 21, 'Antonio', 'Dermatologia', 1).
-prestador( 22, 'Joao', 'Psiquiatria', 3).
-prestador( 23, 'Joaquim', 'Neurologia', 4).
```

Cuidado:

```
% Pressuposto do Mundo Fechado para cuidado.

-cuidado(Data, IdUt, IdPrest, Descricao, Custo) :-
    nao(cuidado(Data, IdUt, IdPrest, Descricao, Custo)),
    nao(excecao(cuidado(Data, IdUt, IdPrest, Descricao, Custo))).

% Negacoes fortes para cuidado.

-cuidado(data(25, 12, 2019), 3, 5, 'Consulta de Podologia', 20).
-cuidado(data(26, 12, 2019), 6, 2, 'Consulta de Dermatologia', 70).
-cuidado(data(27, 12, 2019), 7, 1, 'Consulta de Pediatria', 35).
```

Instituição:

```
% Pressuposto do Mundo Fechado para instituicao.

-instituicao(Id, Nome, Cidade) :-
   nao(instituicao(Id, Nome, Cidade)),
   nao(excecao(instituicao(Id, Nome, Cidade))).

% Negacoes fortes para instituicao.

-instituicao(20, 'Hospital de Braga',      'Braga').
-instituicao(21, 'Hospital Sao Joao',      'Porto').
-instituicao(22, 'Hospital de Santa Maria', 'Porto').
```

3.2 Sistema de Inferência

Antes de partir-mos para a representação do conhecimento imperfeito, vamos apresentar o sistema de inferência criado, o que permitirá apresentar alguns exemplos mais à frente com o mesmo.

Para que fosse possível demonstrar a veracidade de um determinado termo, ou de um conjunto de termos, foi necessária a criação de um sistema de inferência, que dado esses mesmos termos, nos dissesse se estes são verdadeiros ou falsos. De acordo com o que foi mencionado até agora, se o sistema é informado através de conhecimento positivo sobre um termo, então o valor de veracidade deste mesmo termo será verdadeiro, enquanto que se o sistema é informado através de conhecimento negativo sobre um termo, este termo será falso. No entanto, com a introdução do conhecimento imperfeito, iremos ter conhecimento que não está presente na base do conhecimento, mas cuja veracidade não pode ser vista como falsa, uma vez que que o sistema não foi informado do mesmo através de conhecimento negativo. Isto leva à introdução de um novo valor de veracidade, o desconhecido. De acordo com este novo valor, foi criado o sistema de inferência que se segue:

```
% Extenso  do Meta-Predicado demo: Questo, Resposta -> {V,F}

demo(Questao, verdadeiro) :- Questao.
demo(Questao, falso) :- -Questao.
demo(Questao, desconhecido) :- nao(Questao), nao(-Questao).


% Extensao do Meta-Predicado demo: Questao1, Questao2, Operacao,
    Resposta -> {V, F, D}
% e - conjuncao; ou - disjuncao; imp - implicacao; equiv - equivalencia;

demo(Q1, Q2, OP, R) :-
   OP == e,
   demo(Q1, R1),
   demo(Q2, R2),
   conj(R1, R2, R).

demo(Q1, Q2, OP, R) :-
   OP == ou
   demo(Q1, R1),
   demo(Q2, R2),
   disj(R1, R2, R).

demo(Q1, Q2, OP, R) :-
   OP == imp,
   demo(Q1, R1),
   demo(Q2, R2),
```

```
   implic(R1, R2, R).

demo(Q1, Q2, OP, R) :-
   OP == equiv,
   demo(Q1, R1),
   demo(Q2, R2),
   equiv(R1, R2, R).


% Extensao do Meta-Predicado demoLista: [Questao], [Resposta] -> {V,F,D}

demoLista( [],[] ).
demoLista( [A|H],[B|T] ) :- demo( A,B ),
                            demoLista( H,T ).
```

A extensão base do Meta-Predicado demo, permite avaliar uma determinada questão, devolvendo o seu valor de veracidade, seja verdadeiro, falso ou desconhecido. Sendo esta base demasiado simples, foi criada mais uma extensão de demo que corresponde à avaliação de um conjunto de questões, que dadas duas questões e uma operação lógica, podendo esta ser de conjunção, disjunção, implicação e equivalência, devolve o valor de verdade da operação entre estes dois termos. As tabelas de verdade pela qual estas operações se regem serão apresentadas em Anexo. Por último, é criado o Meta-Predicado demoLista que permite avaliar um conjunto qualquer de questões, devolvendo um conjunto de resultados obtidos.

3.3 Conhecimento Imperfeito

Antes de começarmos a descrever a implementação da evolução e involução do sistema e os invariantes utilizados pelo mesmo para o correto desenvolvimento do sistema, vamos abordar o conhecimento imperfeito para que haja uma melhor perceção das razões escolhidas para o porquê das escolhas que foram feitas mais para a frente. Primeiramente, é importante saber que este conhecimento é dividido em três categorias, interdito, impreciso e incerto, sendo o conhecimento imperfeito impreciso de dois tipos, conjunto de hipóteses conhecidas e intervalo de valores. A titulo de exemplo, apenas serão apresentados os casos para o utente, com o objetivo de reduzir a quantidade de informação desnecessária, uma vez que os outros exemplos seriam semelhantes. Serão também apresentadas as demonstrações do conhecimento, utilizando o predicado demo, o que permitirá concluír que de facto, o valor de veracidade da informação correspondente a conhecimento imperfeito é de facto desconhecido. Um facto muito importante em relação a este último ponto é que este conhecimento imperfeito apenas é considerado como desconhecido se toda a informação restante que é de facto conhecida estiver correcta, caso contrário o resultado da demonstração será falso em vez de desconhecido.

3.3.1 Interdito

O conhecimento imperfeito interdito diz respeito a situações em que se desconhece alguma informação, sabendo-se que essa informação permanecerá sempre como uma incógnita.

```
% Nao e possivel saber qual a idade do utente com id 13

utente(14, 'Azevedo', int001, 'Rua dos Padres').

excecao(utente(Id, Nome, Idade, Morada)) :-
   utente(Id, Nome, int001, Morada).
nulo(int001).
```

Como se pode observar, é criada uma variável int00X no campo que corresponde à informação desconhecida, em conjunto com a criação da exceção correspondente, que utiliza essa mesma variável para informar qual dos campos corresponde de facto à informação desconhecida. Posteriormente, esta variável é definida através do predicado nulo, atribuindo-lhe obviamente o valor nulo.

```
| ?- demo(utente(14,'Azevedo',20,'Rua dos Padres'),L).
L = desconhecido ?
yes
```

Figure 1: Exemplo da demonstração do conhecimento interdito para Utente

É importante ter em atenção que este tipo de conhecimento nunca poderá ser evoluído, sendo necessária a criação de invariantes que impeçam a inserção de conhecimento relativo ao conhecimento interdito, fazendo uso do predicado nulo apresentado em cima. Na secção dos invariantes este tema será abordado com mais atenção.

3.3.2 Impreciso

O conhecimento imperfeito impreciso tem que ser primeiramente dividido em duas situações:

Conjunto de Hipóteses Conhecidas: Este ramo do conhecimento imperfeito diz respeito a situações em que não se sabe, por exemplo, se o utente com id 12, idade 10 e morada 'Rua do Ceu' se chama 'Alberto' ou 'Albertino'. Este tipo de conhecimento é apresentado com a criação de exceções que representam as várias hipóteses. De notar que é também criado um predicado impreciso cujo campo corresponde ao id, neste caso de utente, para posterior tratamento dos casos deste tipo de conhecimento na evolução do sistema.

```
excecao(utente(12, 'Albertino', 10, 'Rua do Ceu')).
excecao(utente(12, 'Alberto', 10, 'Rua do Ceu')).

impreciso(utente(12)).
```

```
| ?- demo(utente(12,'Alberto', 10, 'Rua do Ceu'),L).
L = desconhecido ?
yes
```

Figure 2: Exemplo da demonstração do conhecimento impreciso para um conjunto de hipóteses conhecidas, para Utente

Intervalo de Valores: Este ramo do conhecimento imperfeito diz respeito a situações em que não se sabe, por exemplo, a idade do utente 13,

com nome 'Telmo' e que mora na 'Rua dos Padres', sabendo-se apenas que está entre os valores 45 e 50 inclusive.

```
excecao(utente(13, 'Telmo', Idade, 'Rua dos Padres')) :-
   Idade >= 45,
   Idade =< 50.
excecao(utente(13, 'Telmo', Idade, 'Rua dos Padres')) :-
   Idade >= 45,
   Idade =< 50.

imprecisoValues(utente(13)).
```

Tal como no tipo de conhecimento impreciso anterior, são criadas exceções que apresentam as hipóteses correspondentes aos valors que, neste caso, a idade pode tomar, sendo também criado um predicado adicional com o id de utente, chamado imprecisoValues.

```
| ?- demo(utente(13,'Telmo', 46, 'Rua dos Padres'),L).
L = desconhecido ?
yes
```

Figure 3: Exemplo da demonstração do conhecimento impreciso para um intervalo de valores, para Utente

3.3.3 Incerto

Este último tipo de conhecimento imperfeito diz respeito à existência de conhecimento desconhecido, em que não se sabe quais são os valores que correspondem a um valor de verdade de um determinado campo. Ao contrário do conhecimento imperfeito interdito, é possível alterar o valor do campo correspondente à informação desconhecida, passando assim o valor da informação sobre o termo a ser verdadeiro.

```
% No  se sabe o nome do utente com id 11.

utente(11, inc001, 24, 'Rua Curta').

excecao(utente(Id, Nome, Idade, Morada)) :-
   utente(Id, inc001, Idade, Morada).

incerto(utente(11)).
```

Para a representação do conhecimento incerto, é criado, neste caso, um utente com uma variável inc00X no campo que corresponde à informação desconhecida, em conjunto com a criação da exceção correspondente, que utiliza essa mesma variável para informar qual dos campos corresponde de facto à informação desconhecida, neste caso, o nome do Utente. Tal como no conhecimento impreciso, é criado um predicado incerto com um campo correspondente ao id de utente que será utilizado para o controlo da evolução do sistema.

```
| ?- demo(utente(13,'Telmo', 46, 'Rua dos Padres'),L).
L = desconhecido ?
yes
```

Figure 4: Exemplo da demonstração do conhecimento incerto para um nome qualquer de Utente.

3.4 Evolução e Involução

Para o correcto funcionamento do sistema, é necessário implementar ferramentas que permitam o aumento e a diminuição de conhecimento. A estas ferramentas, tal como no primeiro exercicio, chamamos de evolução e involução. Neste exercicio, esta implementação teria que dar resposta não só à evolução de conhecimento positivo perfeito como ao conhecimento negativo tendo em atenção o conhecimento imperfeito já existente no sistema de conhecimento, mantendo o mesmo com a devida consistência. Esta evolução e involução são regidas por um conjunto de predicados que têm que ter um valor de verdade, dado o termo que se pretende evoluir ou involuir no sistema, e que cumprir um conjunto de restrições, os invariantes, dependendo do tipo de conhecimento a inserir, positivo ou negativo.

3.4.1 Evolução

A evolução do sistema, apesar de ser simples, requer alguma atenção no que diz respeito ao funcionamento de cada predicado, principalmente no tratamento do conhecimento imperfeito. De notar que em todos os predicados evolução é requisito que os invariantes sejam cumpridos. Em primeiro lugar, é feita a distinção entre a evolução de conhecimento positivo e de conhecimento negativo e em segundo lugar é feita a distinção, dentro do positivo e negativo, entre a inserção de conhecimento perfeito e de inserção de conhecimento perfeito que altere conhecimento imperfeito.
Em relação ao conhecimento perfeito positivo, o predicado evolução é o seguinte:

```
% Extensao do predicado evolucao: Termo, Flag -> {V, F}
% Insercao de conhecimento positivo sem existencia de conhecimento
    imperfeito.

evolucao(T) :-
   nao(isImperf(T)),
   solucoes(I, +T::I, Li),
    teste(Li),
   assert(T).
```

Este predicado é muito semelhante ao desenvolvido no primeiro exercicio, com a diferença de que desta vez, é utilizado o predicado "isImperf" em conjunto com o predicado "nao" para garantir que o conhecimento evoluído por este predicado em especifico de evolução não está a efetuar nenhuma alteração a conhecimento imperfeito mas sim a inserir conhecimento positivo perfeito.

```
| ?- evolucao(utente(50, 'Jose', 25, 'Rua Joaquim Ribeiro')).
yes
```

Figure 5: Evolução de conhecimento positivo perfeito referente a um Utente sem existência de conhecimento imperfeito.

De seguida, temos a evolução de conhecimento positivo perfeito, com existência de conhecimento imperfeito sobre o termo que se pretende evoluir.

```
% Extensao do predicado evolucao: Termo, Flag -> {V, F}
% Insercao de conhecimento positivo existindo conhecimento imperfeito.

evolucao(T) :-
   isImperf(T),
   testConhecimento(T),
   delIncerto(T),
   solucoes(I, +T::I, Li),
   teste(Li),
   assert(T),
   delImperf(T).
```

Como mencionado, este predicado evolução apenas é utilizado se existir conhecimento imperfeito relativamente ao termo que se pretende evoluir, o que é verificado pelo predicado isImperf. O predicado testConhecimento diz-nos se o valor do termo que se pretende evoluir tem como valor de veracidade desconhecido. O predicado delIncerto garante que não existe já um termo com o mesmo Id ao remover conhecimento incerto que já existisse, o que caso acontecesse, impediria a evolução do conhecimento, visto que nos invariantes não é permitida a evolução de conhecimento com Id's repetidos. No final, se o predicado evolução for cumprido, o conhecimento imperfeito é removido.

```
| ?- listing(incerto).
incerto(utente(11)).
incerto(prestador(9)).
incerto(instituicao(8)).

yes
| ?- evolucao(utente(11,'Andre',24,'Rua Curta')).
yes
| ?- listing(incerto).
incerto(prestador(9)).
incerto(instituicao(8)).

yes
```

Figure 6: Evolução de conhecimento referente à existência de conhecimento incerto, com respetiva remoção do conhecimento imperfeito incerto.

```
| ?- evolucao(utente(12,'Alberto',10,'Rua do Ceu')).
yes
| ?- listing(excecao).
excecao(utente(A,B,_,C)) :-
        utente(A, B, int001, C).
excecao(cuidado(A,B,C,_,D)) :-
        cuidado(A, B, C, int002, D).
excecao(utente(13,'Telmo',A,'Rua dos Padres')) :-
        A>=45,
        A=<50.
excecao(utente(13,'Telmo',A,'Rua dos Padres')) :-
        A>=45,
        A=<50.
excecao(prestador(10,'Luis','Podologia',5)).
excecao(prestador(10,'Luis','Ortopedia',5)).
excecao(prestador(10,'Luis','Dermatologia',5)).
excecao(cuidado(data(25,6,2019),2,3,'Consulta de Ortopedia',A)) :-
        A>=25,
        A=<50.
excecao(instituicao(9,'Hospital da Trofa','Trofa')).
excecao(instituicao(9,'Hospital da Treta','Trofa')).
excecao(utente(A,_,B,C)) :-
        utente(A, inc001, B, C).
excecao(prestador(A,B,_,C)) :-
        prestador(A, B, inc001, C).
excecao(instituicao(A,B,_)) :-
        instituicao(A, B, inc001).

yes
| ?- listing(impreciso).
impreciso(prestador(10)).
impreciso(instituicao(9)).
```

Figure 7: Evolução de conhecimento referente à existência de conhecimento impreciso, com respetiva remoção do conhecimento imperfeito impreciso.

Como mencionado, temos por outro lado a evolução de conhecimento negativo. Em primeiro lugar e de modo análogo à inserção de conhecimento positivo sem existência de conhecimento impreciso, temos a evolução de conhecimento negativo sem existência de conhecimento imperfeito. Apenas precisamos de verificar a existência de conhecimento impreciso uma vez que a inserção de conhecimento negativo sobre conhecimento imperfeito incerto não causaria nenhuma alteração no sistema.

```
| ?- evolucao(-utente(40,'Ana',35,'Rua Curta')).
yes
```

Figure 8: Evolução de conhecimento negativo sem existência de conhecimento impreciso.

Para concluir o processo de evolução, temos a evolução de conhecimento negativo, cujos termos dizem respeito a conhecimento impreciso já existente. De notar, que ao ser inserido conhecimento negativo referente a conhecimento impreciso em que os valores possíveis sejam reduzidos a apenas um, o conhecimento impreciso passa a conhecimento perfeito positivo, uma vez que passa apenas a existir um valor possível de verdade. Mais à frente este ponto será explicado e demonstrado. De seguida são mostrados os predicados de evolução referentes à inserção de conhecimento negativo com existência de conhecimento impreciso.

```
% Extensao do predicado evolucao: Termo, Flag -> {V, F}
% Insercao de conhecimento negativo com existencia de conhecimento
    imperfeito.

evolucao(-Te) :-
   isImprec(Te),
   delImprec(Te, Lista),
   comprimento(Lista, S),
   S > 1,
   solucoes(I, +(-Te)::I, Li),
   teste(Li),
    assert(-Te).



% Extensao do predicado evolucao: Termo, Flag -> {V, F}
% Insercao de conhecimento negativo com existencia de conhecimento
    imperfeito.

evolucao(-Te) :-
   isImprec(Te),
```

```
  delImprec(Te, [H|[]]),
  delImperf(Te),
   assert(H).
```

Nesta situação em especifico, sendo o termo a inserir referente a conhecimento impreciso, é chamado o predicado delImprec que recebe como argumento um termo e produz uma lista, devolvendo a lista com todas as exceções relativas ao conhecimento impreciso que dizem respeito ao termo em questão, menos a exceção que diz de facto respeito ao conhecimento negativo inserido.

```
| ?- delImprec(utente(12,'Alberto',10,'Rua do Ceu'),L).
L = [utente(12,'Albertino',10,'Rua do Ceu')] ?
```

Figure 9: Exemplo do funcionamento do predicado delImprec.

Neste caso, após inserção do conhecimento negativo, a única exceção que sobraria seria a de que o utente teria o nome de 'Alberto'. Tendo o primeiro predicado de evolução uma restrição que indica que a lista devolvida pelo predicado delImprec tem que ter tamanho maior que 1, este predicado falharia, entrando então no outro. O outro predicado, vai chamar novamente o predicado delImprec, que devolverá a mesma lista, removendo o conhecimento imperfeito relativo ao termo que se pretende inserir, fazendo a inserção no sistema do único termo devolvido na lista pelo predicado delImprec, correspondendo este termo então ao conhecimento perfeito positivo que deverá de facto ser inserido no sistema. Caso o tamanho da lista devolvido pelo predicado delImprec seja maior que um, o conhecimento negativo será simplesmente adicionado ao sistema e o seu conhecimento impreciso removido.

3.4.2 Involução

No caso da involução, comparando-a com o primeiro exercício, o invariante para a remoção de conhecimento perfeito positivo é não sofreu alterações, continuando a ter que respeitar os invariantes precedidos por "-". Quanto à remoção de conhecimento negativo, o predicado involução é análogo ao do predicado involução para a remoção de conhecimento positivo, com a diferença de que o termo é precedido pelo símbolo "-".

```
% Extensao do predicado involucao: Termo -> {V, F}

involucao(Te) :-
  Te,
```

```
   solucoes(I, -Te::I, Li),
   teste(Li),
    retract(Te).

%------------------------------ - - - - - - - - - - - - - - -
% Extensao do predicado involucao: -Termo -> {V, F}
% -Termo representa a negacao forte de um termo

involucao(-Te) :-
    -Te,
   solucoes(I, -(-Te::Inv), Li),
   teste(Li),
    retract(-Te).
```

```
yes
| ?- involucao(utente(10,'Alice',86,'Rua Curta')).
no
```

Figure 10: Remoção de utente não permitida devido a existência de cuidado associado a Utente

```
| ?- involucao(-utente(21, 'Diogo', 12, 'Rua Benjamim Salgado')).
yes
```

Figure 11: Remoção de conhecimento negativo relativo a Utente.

3.5 Invariantes

Como vimos na secção anterior, a evolução e involução do sistema são feitas apenas se forem cumpridas certas restrições e se se verificarem como verdadeiros certos predicados. Tal como no primeiro exercício, o conjunto destas restrições compõem os invariantes necessários para o correto funcionamento do sistema. De seguida serão demonstrados estes mesmos invariantes, com a devida explicação. De notar que serão apenas explicados os invariantes para o utente, uma vez que para o resto do conhecimento, os invariantes seguem a mesma linha de raciocínio.

3.5.1 Invariantes de Utente

```
% Invariante Estrutural que nao permite a insercao de utentes repetidos
    (com o mesmo ID).
% Id e Idade inteiros.
% Idade entre 0 e 120.

+utente(Id, _, Idade, _) :: (
   integer(Id), integer(Idade), Idade >= 0, Idade =< 120,
   solucoes(Id, utente(Id, _, _, _), S),
    comprimento(S, N),
    N == 0
).


% Invariante que impede a insercao de conhecimento positivo
    contraditorio.
% Invariante que impede a insercao de conhecimento positivo que altere
    conhecimento interdito sobre
% o nome, a idade ou a morada de utente.

+utente(Id, Nome, Idade, Morada) :: (
   solucoes(Id, -utente(Id, Nome, Idade, Morada), S1),
   solucoes(N, (utente(Id, N, Idade, Morada), nulo(N)), S2),
   solucoes(I, (utente(Id, Nome, I, Morada), nulo(I)), S3),
   solucoes(M, (utente(Id, Nome, Idade, M), nulo(M)), S4),
   comprimento(S1, N1),
   comprimento(S2, N2),
   comprimento(S3, N3),
   comprimento(S4, N4),
    N1 >= 0,
    N1 =< 1,
    N2 == 0,
   N3 == 0,
   N4 == 0
```

```
).


% Invariante que impede a insercao de conhecimento negativo contraditorio
% Invariante que impede a insercao de conhecimento negativo que ja exista
% Invariante que impede a insercao de conhecimento negativo que altere
    conhecimento interdito sobre
% o nome, a idade ou a morada de utente.

+(-utente(Id, Nome, Idade, Morada)) :: (
   solucoes(Id, utente(Id, Nome, Idade, Morada), S1),
   solucoes(Id, -utente(Id, Nome, Idade, Morada), S2),
   solucoes(N, (utente(Id, N, Idade, Morada), nulo(N)), S3),
   solucoes(I, (utente(Id, Nome, I, Morada), nulo(I)), S4),
   solucoes(M, (utente(Id, Nome, Idade, M), nulo(M)), S5),
   comprimento(S1, N1),
   comprimento(S2, N2),
   comprimento(S3, N3),
   comprimento(S4, N4),
   comprimento(S5, N5),
   N1 == 0,
   N2 >= 0,
   N2 =< 1,
   N3 == 0,
   N4 == 0,
   N5 == 0
).


% Invariante Referencial que nao permite a remocao de um utente enquanto
    existirem cuidados
% associadas ao mesmo
-utente(IdUt, _, _, _) :: (
   solucoes(IdUt, cuidado(_,IdUt, _, _, _), S1),
   comprimento(S1, N1),
   N1 == 0
).
```

Para cada tipo de conhecimento base, os invariantes dividem-se em quatros partes diferentes. A primeira e a última dizem respeito aos invariantes desenvolvidos no primeiro exercício, que controlam a inserção e remoção de conhecimento positivo, nomeadamente o Invariante Estrutural e o Invariante Referencial, com particular atenção para restrições que impedem a inserção de prestadores cuja instituição não exista e de cuidados cujo id de utente ou de prestador também não exista. Quanto às outras duas partes, os invariantes garantem que nunca haverá inserção de conhecimento positivo ou negativo que altere conhecimento imperfeito interdito. Tal como tinhamos visto, o processo

de evolução do sistema não deverá em qualquer situação alterar este mesmo tipo de conhecimento. Por fim, estes invariantes garantem que não é permitido inserir conhecimento negativo repetido e que não existe inserção de conhecimento contraditório, ou seja, que não é inserido no sistema conhecimento positivo se já existir conhecimento negativo que contradiga o conhecimento positivo e vice-versa.

3.5.2 Invariantes de Prestador

```
% Invariante Estrutural que nao permite a insercao de prestadores que ja
    existam (com o mesmo ID).
% Nao permitir a insercao de prestadores cuja instituicao nao exista.
% Id deve ser inteiro

+prestador(Id, Nome, Esp, IdInst) :: (
   integer(Id), integer(IdInst),
    solucoes(Id, prestador(Id, _, _, _), S1),
    solucoes(IdInst, instituicao(IdInst, _, _), S2),
    comprimento(S1, N1),
   comprimento(S2, N2),
    N1 == 0,
   N2 == 1
).


% Invariante que impede a insercao de conhecimento positivo
    contraditorio.
% Invariante que impede a insercao de conhecimento positivo que altere
    conhecimento interdito sobre
% o nome, a especialidade ou a instituicao de prestador.

+prestador(Id, Nome, Esp, IdInst) :: (
   solucoes(Id, -prestador(Id, Nome, Esp, IdInst), S1),
   solucoes(N, (prestador(Id, N, Esp, IdInst), nulo(N)), S2),
   solucoes(E, (prestador(Id, Nome, E, IdInst), nulo(E)), S3),
   solucoes(I, (prestador(Id, Nome, Esp, I), nulo(I)), S4),
   comprimento(S1, N1),
   comprimento(S2, N2),
   comprimento(S3, N3),
   comprimento(S4, N4),
   N1 >= 0,
   N1 =< 1,
   N2 == 0,
   N3 == 0,
   N4 == 0

).
```

```
% Invariante que impede a insercao de conhecimento negativo
    contraditorio.
% Invariante que impede a insercao de conhecimento negativo que ja
    exista.
% Invariante que impede a insercao de conhecimento negativo que altere
    conhecimento interdito sobre
% o nome, a especialidade ou a instituicao de prestador.

+(-prestador(Id, Nome, Esp, IdIns)) :: (
   solucoes(Id, prestador(Id, Nome, Esp, IdIns), S1),
   solucoes(Id, -prestador(Id, Nome, Esp, IdIns), S2),
   solucoes(N, (prestador(Id, N, Esp, IdInst), nulo(N)), S3),
   solucoes(E, (prestador(Id, Nome, E, IdInst), nulo(E)), S4),
   solucoes(I, (prestador(Id, Nome, Esp, I), nulo(I)), S5),
   comprimento(S1, N1),
   comprimento(S2, N2),
   comprimento(S3, N3),
   comprimento(S4, N4),
   comprimento(S5, N5),
   N1 == 0,
   N2 >= 0,
   N2 =< 1,
   N3 == 0,
   N4 == 0,
   N5 == 0
).


% Invariante Referencial que nao permite a remocao de servico enquanto
    existirem cuidados
% associados ao mesmo

-prestador(Id, _, _, _) :: (
   solucoes(Id, cuidado(_, _, Id, _, _), S1),
   comprimento(S1, N1),
   N1 == 0
).
```

3.5.3 Invariantes de Cuidado

```
% Invariante que impede a insercao de um cuidado ja existente ou cujo
% utente ou prestador nao existam.
% IdUt e IdPrest inteiros; Custo number e maior que 0.
```

```
+cuidado(Data, IdUt, IdPrest, Desc, Custo) :: (
   integer(IdUt), integer(IdPrest),
   number(Custo), Custo >= 0,
   solucoes(IdUt, utente(IdUt, _, _, _), S1),
   solucoes(IdPrest, prestador(IdPrest, _, _, _), S2),
   solucoes((Data, IdUt, IdPrest, Desc, Custo),
            cuidado(Data, IdUt, IdPrest, Desc, Custo), S3),
   comprimento(S1, N1),
   comprimento(S2, N2),
    comprimento(S3, N3),
   N1 == 1,
   N2 == 1,
   N3 == 0

).


% Invariante que impede a insercao de conhecimento positivo
    contraditorio.
% Invariante que impede a insercao de conhecimento positivo que altere
    conhecimento interdito sobre
% o custo do cuidado

+cuidado(Data, IdUt, IdPrest, Desc, Custo) :: (
   solucoes((Data, IdUt, IdPrest, Desc, Custo),
          -cuidado(Data, IdUt, IdPrest, Desc, Custo), S1),
   solucoes(C, (cuidado(Data, IdUt, IdPrest, Desc, C), nulo(C)), S2),
   comprimento(S1, N1),
   comprimento(S2, N2),
   N1 >= 0,
   N1 =< 1,
   N2 == 0
).


% Invariante que impede a insercao de conhecimento negativo
    contraditorio.
% Invariante que impede a insercao de conhecimento negativo que ja
    exista.
% Invariante que impede a insercao de conhecimento negativo que altere
    conhecimento interdito sobre
% o custo de um cuidado.

+(-cuidado(Data, IdUt, IdPrest, Desc, Custo)) :: (
   solucoes((Data, IdUt, IdPrest, Desc, Custo),
            cuidado(Data, IdUt, IdPrest, Desc, Custo), S1),
   solucoes((Data, IdUt, IdPrest, Desc, Custo),
             -cuidado(Data, IdUt, IdPrest, Desc, Custo), S2),
   solucoes(C, (cuidado(Data, IdUt, IdPrest, Desc, C), nulo(C)), S3),
   comprimento(S1, N1),
```

```
    comprimento(S2, N2),
    comprimento(S3, N3),
   N1 == 1,
   N2 >= 0,
   N2 =< 1,
   N3 == 0
).


% Invariante que remove um cuidado se este existir.

-cuidado(Data, IdUt, IdPrest, Desc, Custo) :: (
   solucoes((Data, IdUt, IdPrest, Desc, Custo),
   consulta(Data, IdServ, IdPrest, Desc, Custo), S1),
   comprimento(S1, N1),
   N1 = 1
).
```

3.5.4 Invariantes de Instituição

```
% Invariante Estrutural que nao permite a insercao de instituicoes
    repetidas (com o mesmo ID)
% Id inteiro

+instituicao(IdInst, _, _) :: (
   integer(IdInst),
   solucoes(IdInst, instituicao(IdInst, _, _), S),
    comprimento(S, N),
    N == 0
).


% Invariante que impede a insercao de conhecimento positivo
    contraditorio.
% Invariante que impede a insercao de conhecimento positivo que altere
    conhecimento interdito sobre
% o nome ou a ci

+instituicao(IdInst, Nome, Cidade) :: (
   solucoes(Id, -instituicao(Id, Nome, Cidade), S1),
   solucoes(N, (instituicao(IdInst, N, Cidade), nulo(N)), S2),
   solucoes(Cidade, (instituicao(IdInst, Nome, C), nulo(C)), S3),
   comprimento(S1, N1),
   comprimento(S2, N2),
   comprimento(S3, N3),
   N1 >= 0,
```

```
   N1 =< 1,
   N2 == 0,
   N3 == 0
).


% Invariante que impede a insercao de conhecimento negativo
    contraditorio.
% Invariante que impede a insercao de conhecimento negativo que ja
    exista.
% Invariante que impede a insercao de conhecimento negativo que altere
    conhecimento interdito sobre
% o nome ou a cidade de uma instituicao.

+(-instituicao(IdInst, Nome, Cidade)) :: (
   solucoes(Id, instituicao(Id, Nome, Cidade), S1),
   solucoes(Id, -instituicao(Id, Nome, Cidade), S2),
   solucoes(N, (instituicao(IdInst, N, Cidade), nulo(N)), S3),
   solucoes(Cidade, (instituicao(IdInst, Nome, C), nulo(C)), S4),
   comprimento(S1, N1),
   comprimento(S2, N2),
   comprimento(S3, N3),
   comprimento(S4, N4),
   N1 == 0,
   N2 >= 0,
   N2 =< 1,
   N3 == 0,
   N4 == 0
).


% Invariante Referencial que nao permite a remocao de uma instituicao se
    existirem
% prestadores associados a mesma

-instituicao(IdInst, _, _) :: (
   solucoes(IdInst, prestador(_, _, _, IdInst), S1),
   comprimento(S1, N1),
   N1 == 0
).
```

Conclusão e Sugestões

A nosso ver, o objetivo fundamental do trabalho foram cumpridos. Achamos que conseguimos expor o conhecimento que adquirimos ao longo do semestre de forma correta. Neste sentido, achamos que este tanto este trabalho como o anterior foram importantes para a consolidação dos conceitos relativos ao *PROLOG* e às variantes que este apresenta, bem como da relevância do mesmo na abordagem de problemas/desafios.

Posto isto, achamos que este novo conhecimento adquirido é crucial para abordagem de situações que requerem rigor e restrição no âmbito da criação e desnvolvimento de sistemas de representação de conhecimento e raciocínio, sendo que nos permite restringir e otimizar o funcionamento do mesmo sistema de forma objectiva e consistente.

Referências

- Ivan Bratko, "PROLOG: Programming for Artificial Intelligence", 3rd Edition, Addison-Wesley Longman Publishing Co., Inc., 2000;
- Hélder Coelho, "A Inteligência Artificial em 25 lições", Fundação Calouste Gulbenkian, 1995;

Anexos

Tabelas de Verdade:

Questão1	Questão 2	Conjunção	Disjunção
Verdadeiro	Verdadeiro	Verdadeiro	Verdadeiro
Verdadeiro	Falso	Falso	Verdadeiro
Verdadeiro	Desconhecido	Desconhecido	Verdadeiro
Desconhecido	Desconhecido	Desconhecido	Desconhecido
Desconhecido	Verdadeiro	Desconhecido	Verdadeiro
Desconhecido	Falso	Falso	Desconhecido
Falso	Verdadeiro	Falso	Verdadeiro
Falso	Desconhecido	Falso	Desconhecido
Falso	Falso	Falso	Falso

Questão1	Questão 2	Equivalência	Implicação
Verdadeiro	Verdadeiro	Verdadeiro	Verdadeiro
Verdadeiro	Falso	Falso	Falso
Verdadeiro	Desconhecido	Desconhecido	Desconhecido
Desconhecido	Desconhecido	Desconhecido	Desconhecido
Desconhecido	Verdadeiro	Desconhecido	Desconhecido
Desconhecido	Falso	Desconhecido	Desconhecido
Falso	Verdadeiro	Falso	Verdadeiro
Falso	Desconhecido	Desconhecido	Verdadeiro
Falso	Falso	Verdadeiro	Verdadeiro

Predicados Auxiliares:

```
% Extensao do predicado nao: Questao -> {V, F}

nao(Questao) :- Questao, !, fail.
nao(Questao).


% Extensao do predicado solucoes: F, Q, S -> {V, F}

solucoes(F, Q, S) :- findall(F, Q, S).


% Extensao do predicado comprimento: S, N -> {V, F}

comprimento(S, N) :- length(S, N).
```

```
% Extensao do predicado teste: Lista -> {V, F}

teste([]).
teste([X | L]) :- X, teste(L).

% Extensao do predicado testeImperfeito: Termo -> {V, F}
testConhecimento(T) :- demo(T, desconhecido).


% Extensao do predicado isImperf: T -> {V, F}
% Indica se o conhecimento sobre o termo e imperfeito.

isImperf(utente(Id, _, _, _)) :- incerto(utente(Id)).
isImperf(utente(Id, _, _, _)) :- impreciso(utente(Id)).
isImperf(utente(Id, _, _, _)) :- imprecisoValues(utente(Id)).
isImperf(prestador(Id, _, _, _)) :- incerto(prestador(Id)).
isImperf(prestador(Id, _, _, _)) :- impreciso(prestador(Id)).
isImperf(prestador(Id, _, _, _)) :- imprecisoValues(prestador(Id)).
isImperf(cuidado(Data, IdUt, IdPrest, _, _)) :- incerto(cuidado(Data,
    IdUt, IdPrest)).
isImperf(cuidado(Data, IdUt, IdPrest, _, _)) :- impreciso(cuidado(Data,
    IdUt, IdPrest)).
isImperf(cuidado(Data, IdUt, IdPrest, _, _)) :-
    imprecisoValues(cuidado(Data, IdUt, IdPrest)).
isImperf(instituicao(Id, _, _)) :- incerto(instituicao(Id)).
isImperf(instituicao(Id, _, _)) :- impreciso(instituicao(Id)).
isImperf(instituicao(Id, _, _)) :- imprecisoValues(instituicao(Id)).


% Extensao do predicado isImperf: T -> {V, F}
% Indica se o conhecimento sobre o termo e impreciso.

isImprec(utente(Id, _, _, _)) :- impreciso(utente(Id)).
isImprec(prestador(Id, _, _, _)) :- impreciso(prestador(Id)).
isImprec(instituicao(Id, _, _)) :- impreciso(instituicao(Id)).
isImprec(cuidado(Data, IdUt, IdPrest, _, _)) :- impreciso(cuidado(Data,
    IdUt, IdPrest)).



% ----------------- Remover Conhecimento Impreciso
    ------------------------

% Extensao do predicado delImprec: Utente -> {V, F}

delImprec(utente(Id, Nome, Idade, Morada), L) :-
   retract(excecao(utente(Id, Nome, Idade, Morada))),
   solucoes((utente(Id, N, I, M)), excecao(utente(Id, N, I, M)), L).
```

```
delImprec(utente(Id, Nome, Idade, Morada), L) :-
   solucoes((utente(Id, N, I, M)), excecao(utente(Id, N, I, M)), L).


% Extensao do predicado delImprec: Prestador -> {V, F}

delImprec(prestador(Id, Nome, Esp, IdInst), L) :-
   retract(excecao(prestador(Id, Nome, Esp, IdInst))),
   solucoes((prestador(Id, N, E, I)), excecao(prestador(Id, N, E, I)),
       L).
delImprec(prestador(Id, Nome, Esp, IdInst), L) :-
   solucoes((prestador(Id, N, E, I)), excecao(prestador(Id, N, E, I)),
       L).


% Extensao do predicado delImprec: Cuidado -> {V, F}

delImprec(cuidado(Data, IdUt, IdPrest, Desc, C), L) :-
   retract(excecao(cuidado(Data, IdUt, IdPrest, Desc, C))),
   solucoes((cuidado(Data, IdUt, IdPrest, D, C)), excecao(cuidado(Data,
       IdUt, IdPrest, D, C)), L).
delImprec(cuidado(Data, IdUt, IdPrest, Desc, C), L) :-
   solucoes((cuidado(Data, IdUt, IdPrest, D, C)), excecao(cuidado(Data,
       IdUt, IdPrest, D, C)), L).


% Extensao do predicado delImprec: Instituicao -> {V, F}

delImprec(instituicao(Id, Nome, Cidade), L) :-
   retract(excecao(instituicao(Id, Nome, Cidade))),
   solucoes((instituicao(Id, N, C)), excecao(instituicao(Id, N, C)), L).
delImprec(instituicao(Id, Nome, Cidade), L) :-
   solucoes((instituicao(Id, N, C)), excecao(instituicao(Id, N, C)), L).



% ----------------- Remover Conhecimento Incerto -----------------------

% Extensao do predicado delIncerto: Utente -> {V, F}

delIncerto(utente(Id, Nome, Idade, Morada)) :-
   incerto(utente(Id)),
   retract(utente(Id, inc001, Idade, Morada)).
delIncerto(utente(Id, _, _, _)) :-
   impreciso(utente(Id)).
delIncerto(utente(Id, _, _, _)) :-
   imprecisoValues(utente(Id)).


% Extensao do predicado delIncerto: Prestador -> {V, F}
```

```
delIncerto(prestador(Id, Nome, Esp, IdInst)) :-
   incerto(prestador(Id)),
   retract(prestador(Id, Nome, inc001, IdInst)).
delIncerto(prestador(Id, _, _, _)) :-
   impreciso(prestador(Id)).
delIncerto(prestador(Id, _, _, _)) :-
   imprecisoValues(prestador(Id)).


% Extensao do predicado delIncerto: Cuidado -> {V, F}

delIncerto(cuidado(Data, IdUt, IdPrest, Desc, C)) :-
   incerto(cuidado(Data, IdUt, IdPrest)).
delIncerto(cuidado(Data, IdUt, IdPrest, _, _)) :-
   impreciso(cuidado(Data, IdUt, IdPrest)).
delIncerto(cuidado(Data, IdUt, IdPrest, _, _)) :-
   imprecisoValues(cuidado(Data, IdUt, IdPrest)).


% Extensao do predicado delIncerto: Instituicao -> {V, F}

delIncerto(instituicao(Id, Nome, Cidade)) :-
   incerto(instituicao(Id)),
   retract(instituicao(Id, inc01, Cidade)).
delIncerto(instituicao(Id, _, _)) :-
   impreciso(instituicao(Id)).
delIncerto(instituicao(Id, _, _)) :-
   imprecisoValues(instituicao(Id)).



% ----------------- Remover Conhecimento Imperfeito
    ------------------------

% Extensao do predicado delImperf: Utente -> {V, F}

delImperf(utente(Id, Nome, Idade, Morada)) :-
   incerto(utente(Id)),
   retract(incerto(utente(Id))).
delImperf(utente(Id, _, _, _)) :-
   imprecisoValues(utente(Id)),
   retract(imprecisoValues(utente(Id))),
   retract(excecao(utente(Id, _, _, _))).
delImperf(utente(Id, Nome, Idade, Morada)) :-
   impreciso(utente(Id)),
   retract(excecao(utente(Id, N, Idade, Morada))),
   delImperf(utente(Id, Nome, Idade, Morada)).
delImperf(utente(Id, _, _, _)) :-
   impreciso(utente(Id)),
```

```
   retract(impreciso(utente(Id))).


% Extensao do predicado delImperf: Prestador -> {V, F}
% Remove conhecimento imperfeito relativamente a um prestador
delImperf(prestador(Id, _, _, _)) :-
   incerto(prestador(Id)),
   retract(incerto(prestador(Id))).
delImperf(prestador(Id, _, _, _)) :-
   imprecisoValues(prestador(Id)),
   retract(imprecisoValues(prestador(Id))).
delImperf(prestador(Id, Nome, Esp, Ins)) :-
   impreciso(prestador(Id)),
   retract(excecao(prestador(Id, Nome, E, Ins))),
   delImperf(prestador(Id, Nome, Esp, Ins)).
delImperf(prestador(Id, _, _, _)) :-
   impreciso(prestador(Id)),
   retract(impreciso(prestador(Id))).


% Extensao do predicado delImperf: Cuidado -> {V, F}

delImperf(cuidado(Data, IdUt, IdPrest, _, _)) :-
   incerto(cuidado(Data, IdUt, IdPrest)),
   retract(incerto(cuidado(Data, IdUt, IdPrest))).
delImperf(cuidado(Data, IdUt, IdPrest, _, _)) :-
   imprecisoValues(cuidado(Data, IdUt, IdPrest)),
   retract(imprecisoValues(cuidado(Data, IdUt, IdPrest))).
delImperf(cuidado(Data, IdUt, IdPrest, Desc, Custo)) :-
   impreciso(cuidado(Data, IdUt, IdPrest)),
   retract(excecao(cuidado(Data, IdUt, IdPrest, Desc, C))),
   delImperf(cuidado(Data, IdUt, IdPrest)).
delImperf(cuidado(Data, IdUt, IdPrest, _, _)) :-
   impreciso(cuidado(Data, IdUt, IdPrest)),
   retract(impreciso(cuidado(Data, IdUt, IdPrest))).


% Extensao do predicado delImperf: Instituicao -> {V, F}

delImperf(instituicao(Id, Nome, Cidade)) :-
   incerto(instituicao(Id)),
   retract(incerto(instituicao(Id))).
delImperf(instituicao(Id, Nome, Cidade)) :-
   imprecisoValues(instituicao(Id)),
   retract(imprecisoValues(instituicao(Id))).
delImperf(instituicao(Id, Nome, Cidade)) :-
   impreciso(instituicao(Id)),
   retract(excecao(instituicao(Id, N, Cidade))),
   delImperf(instituicao(Id, Nome, Cidade)).
delImperf(instituicao(Id, Nome, Cidade)) :-
```

```
  impreciso(instituicao(Id)),
  retract(impreciso(instituicao(Id))).
```