# Sistemas de Representação de Conhecimento e Raciocínio: Exercício Prático Nº2

Universidade do Minho

*Mestrado Integrado em Engenharia Informática*
Grupo 14


Duarte Freitas
A63129

Afonso Sousa
A74196

José Lopes Ramos
A73855

Rafael Silva
A74264


22 de Abril de 2019

## Resumo

Este documento é um relatório técnico sobre o trabalho desenvolvido para a segunda fase do trabalho prático da Unidade Curricular de Sistemas de Representação de Conhecimento e Raciocínio, no qual nos foi pedido para desenvolver um sistema para caracterizar um universo de discurso na área da prestação de cuidados de saúde.
Durante a primeira de três fases desenvolvemos um sistema elementar e numa forma muito próxima da linguagem (de acordo com as aulas práticas até então), sem grandes extensões. Nesta segunda fase vamos estender a linguagem para melhor aproximar a nossa representação do sistema, tanto no conhecimento como no raciocínio sobre este, tomando agora em conta que o conhecimento pode ser imperfeito.
Sobre o desenvolvimento apresentaremos primeiro uma secção de contextualização com a fase anterior, seguida por uma sobre o conhecimento imperfeito e as alterações feitas ao sistema, acabando depois com uma análise ao programa desenvolvido.
Primeiro é feita uma recapitulação da fase anterior, onde foi usada programação em lógica sem extensões, com pressupostos de mundo fechado, nomes únicos e domínio fechado. Foram também usados invariantes estruturais para incutir significado.
De seguida apresenta-se o problema da representação de conhecimento imperfeito, a necessidade de estender a lógica usada e adoptar novos pressupostos. Aborda-se também a alteração dos predicados e invariantes para lidar com conhecimento negativo durante evolução/involução da base de conhecimento.
Finalmente apresentamos o sistema de inferência adaptado para implementar mecanismos de raciocinio sobre a nova base de conhecimento, agora com informação incompleta. A apresentação está redigida por forma a dar ênfase ao raciocínio que está por detrás de blocos de código semelhante, não como um ditado linha a linha do que está feito.

# Conteúdo

# Lista de Figuras

# Introdução

Uma das limitações da lógica clássica é a incapacidade de expressar situações de senso comum, e como a programação em lógica tradicional não permite representar directamente informação incompleta (apenas completa positiva ou negativa), não nos resta outra opção que não estender a lógica de forma a permitir conhecimento imperfeito.
Para mostrar como tal pode ser feito vamos desenvolver o caso de estudo da fase anterior, um universo de discurso na área da prestação de cuidados de saúde, usando-o para mostrar o conhecimento adquirido durante esta fase, destacando-se o porquê e como da extensão feita, a representação de conhecimento imperfeito e formas de lidar com a nova representação, tanto no caso dos predicados como no caso dos invariantes.

## Preliminares

Este trabalho prático foi resolvido com recurso ao conhecimento adquirido nas aulas da UC, material bibliográfico digital sobre *PROLOG* e aos nossos apontamentos. *PROLOG* é uma linguagem declarativa de lógica de primeira-ordem, com uma sintaxe base semelhante a outras linguagens de programação populares, mas tem algumas diferenças chave que a diferenciam. As mais relevantes para este trabalho podem ser descritas sumariamente como:

| | | |
|---|---|---|
| Objetos: | Facto | Constata algo que se reconhece e se sabe verdadeiro; |
| | Predicado | Implementa uma relação; |
| | Regra | Utilizada para definir um novo predicado; |
| | Invariante | Regra inviolável ao longo da execução do programa. |
| Pontuação: | **.** | Utilizado para terminar uma declaração; |
| | **,** | Utilizado para representar a conjunção lógica; |
| | **:-** | Utilizado para representar a implicação conversa lógica (← ); |
| | **;** | Utilizado para representar a disjunção inclusiva lógica; |
| | // | Utilizado para representar a unificação de dois conjuntos. |

Por indicação do docente foram usadas três *flags* específicas da linguagem *PROLOG*, que evitam *warnings* ou erros de carregamento/consulta do ficheiro usado para desenvolver o projeto, podendo ser assim executado corretamente e de acordo com os parâmetros definidos para o conhecimento.

```
1 :- set_prolog_flag( discontiguous_warnings,off ).
2 :- set_prolog_flag( single_var_warnings,off ).
3 :- set_prolog_flag( unknown,fail ).
```

### Fase anterior

Até agora toda a programação foi feita em lógica não estendida, pelo que todo o conhecimento que nos é possível representar é efetivamente conhecimento perfeito positivo, tomando a ausência na base de conhecimento como sinal de conhecimento perfeito negativo. Tal acontece devido aos pressupostos assumidos:

| Pressuposto | Descrição |
|---|---|
| 1. Pressuposto do mundo fechado | Toda a informação que não existe na base de conhecimento é considerada falsa; |
| 2. Pressuposto dos nomes únicos | Duas constantes diferentes (que definam valores atómicos ou objectos) designam, necessariamente, duas entidades diferentes do universo de discurso; |
| 3. Pressuposto do domínio fechado | Não existem mais entidades para além das já representadas por constantes. |

Para incorporar significado nas fórmulas, algo que a linguagem *PROLOG* não faz naturalmente, recorremos a invariantes estruturais para a inserção (+) e remoção (-) de conhecimento, já usados

para representar evolução e involução. Com eles garantimos consistência e protegêmo-nos de coisas como inserir conhecimento com identificadores repetidos, pois violaria o pressuposto de nomes únicos. Recordemos o formato da base de conhecimento:

```
% Utente: #idUt, Nome, Idade, Cidade -> {V,F}
utente(1, jose, 22, braga).
% Serviço: #idServ, Descrição, Instituição, Cidade -> {V,F}
servico(1, ortopedia, hospitalBraga, braga).
% Consulta: Data, #idUt, #idServ, Custo -> {V,F}
consulta(23-02-2016, 1, 3, 23).
```

Tal como dito anteriormente, esta representação escolhida não permite exprimir conhecimento imprefeito e por causa disto torna-se evidente a necessidade de alterar os pressupostos para ganhar capacidade expressiva.

### Conhecimento Imperfeito

Todo o conhecimento, positivo ou negativo, sobre o qual há algum grau de incerteza é chamado de imperfeito e pode ser de três formas: incerto, impreciso ou interdito, caracterizados genericamente por:

- Incerto, desconhece-se valores de um conjunto indeterminado de hipóteses;
- Impreciso, desconhece-se valores de um conjunto determinado de hipóteses;
- Interdito, proibido conhecer valores ou conjuntos de valores.

Para que seja permitido representar este tipo de conhecimento temos de alterar os pressupostos usados na fase anterior, em particular o do mundo fechado e domínio fechado. Com as alterações passamos agora a usar:

| Pressuposto | Descrição |
|---|---|
| 1. Pressuposto do mundo aberto | Podem existir outros factos ou conclusões verdadeiros para além daqueles representados na base de conhecimento; |
| 2. Pressuposto dos nomes únicos | Duas constantes diferentes (que definam valores atómicos ou objectos) designam, necessariamente, duas entidades diferentes do universo de discurso; |
| 3. Pressuposto do domínio aberto | Podem existir mais objectos do universo de discurso para além daqueles designados pelas constantes da base de conhecimento. |

Agora que foram alterados os pressupostos para ser possível obter conclusões sem recorrer à negação por falha, representar explicitamente informação negativa e informação desconhecida, estamos a lidar com programação em lógica estendida e não com lógica tradicional.

# Descrição do Trabalho e Análise de Resultado

## Funcionalidades

1. Representar conhecimento positivo e negativo;
2. Representar casos de conhecimento imperfeito, pela utilização de valores nulos de todos os tipos estudados;
3. Manipular invariantes que designem restrições à inserção e à remoção de conhecimento do sistema;
4. Lidar com a problemática da evolução do conhecimento, criando os procedimentos adequados;
5. Desenvolver um sistema de inferência capaz de implementar os mecanismos de raciocínio inerentes a estes sistemas.

### 1. Representar conhecimento positivo e negativo

Conhecimento (perfeito) positivo ou negativo permite-nos saber tudo em relação a uma certa entidade, sendo representado por um predicado na forma normal para o caso positivo (P), e um predicado com um símbolo de negação a preceder caso seja negativo (-P)

```
% Utentes -> Conhecimento Positivo
%-------------------------------- - - - - - - - - - - - - - - - -
utente( 1 , jose , 22 , braga ).
utente( 2 , rafael , 22 , chaves ).
utente( 3 , afonso , 22 , braga ).
% Utentes -> Conhecimento Negativo
%-------------------------------- - - - - - - - - - - - - - - - -
-utente( 4 , duarte , 23 , braga ).
-utente( 5 , luis , 34 , porto ).
-utente( 6 , manuela , 25 ,faro ).
% Prestador -> Conhecimento Positivo
%-------------------------------- - - - - - - - - - - - - - - - -
prestador( 1 , manuel , medicinafamiliar , hospitalBraga ).
prestador( 2 , andre , cardiologia , hospitalBraga ).
prestador( 3 , francisco , cardiologia , hospitalBraga ).
% Prestador -> Conhecimento Negativo
%-------------------------------- - - - - - - - - - - - - - - - -
-prestador( 4 , afonso , psiquiatria , hospitalBraga ).
-prestador( 5 , duarte , oftalmologia , hospitalPorto ).
-prestador( 6 , manuela , cardiologia , hospitalPorto ).
% Cuidado -> Conhecimento Positivo
%-------------------------------- - - - - - - - - - - - - - - - -
cuidado( 31-01-2012 , 10 , 9 , consulta , 34).
cuidado( 23-02-2016 , 1 , 1 , consulta_de_avaliacao ,19).
cuidado( 23-02-2016 , 2 , 4 , consulta_Rotina  , 42).
% Cuidado-> Conhecimento Negativo
```

```
%------------------------------ - - - - - - - - - - - - - - - - -
-cuidado( 20-01-2017 , 4 , 5 , consulta_Rotina , 54).
-cuidado( 02-09-2014 , 9 , 2 , consulta_Rotina ,  21).
-cuidado( 23-02-2016 , 2 , 1 , eletrocardiograna ,29).
```

## 2. Representar casos de conhecimento imperfeito, pela utilização de valores nulos de todos os tipos estudados

Ao contrário do conhecimento perfeito, o imperfeito tem mais que se lhe diga, sendo preciso recorrer a exceções explícitas para demarcar este tipo de conhecimento. De entre os três tipos possíveis, todos eles apresentam um padrão na sua representação.
No caso de ser incerto (recorde-se, conjundo interminado de hipóteses) todas as hipóteses são parte de um conhecimento sobre o qual nada se sabe, e por isso assinala-se com um valor que representa esse desocnhecimento, pois todas as instâncias são excecionais. O valor desconhecido é marcado como exceção para qualquer que seja o conhecimento que lhe esteja associado (e portanto todos os valores possíveis).
Se for impreciso (conjundo determinado de hipóteses) já sabemos quais são os valores possíveis, ou pelo menos qual a gama de valores em que se encontram as respostas. Caso sejam valores discretos (ex: ou é A ou é B), esses conhecimentos são simples predicados declarados como excecionais. Caso haja uma gama de valores (ex: entre A e B), o conhecimento é marcado como excecional mas com os parâmetros associados (ex: excecao(predicado(x,y,Impreciso)):- Impreciso>=A, Impreciso=<B).
Para os que são interditos (proibido conhecer) somos obrigados a usar um valor especial de proibição (declarado como nulo) para o conhecimento excecional, o qual vamos impedir de tomar valor com um invariante a proibir a sua evolução, verificando que o número de predicados que tem valores atribuído 'naquele conhecimento' (tudo que não seja o símbolo de proibido) é sempre zero. Sumariamente:

| Conhecimento Imperfeito | Implementação |
|---|---|
| Incerto | O conhecimento é declarado como exceção. |
| Impreciso | O conhecimento é declarado como exceção, instanciado ou condicionado. |
| Interdito | O conhecimento é declarado com um valor proibido, valor que é tornado impossível ser conhecido com um invariante. |

```
%------------------------------ - - - - - - - - - - - - - - - - -
% Utentes -> Conhecimento Imperfeito Incerto
%------------------------------ - - - - - - - - - - - - - - - - -
utente( 7 , desconhecido01 , 34 , algarve ).
excecao( utente( ID , N , I , M ) ) :-
  utente( ID , desconhecido01 , I , M ).
% Prestadores -> Conhecimento Imperfeito Incerto
%------------------------------ - - - - - - - - - - - - - - - - -
prestador( 7 , desconhecido02 , ortopedia , hospitalPorto ).
excecao( prestador( ID , N , E , I ) ) :-
  prestador( ID , desconhecido02 , E , I ).
```

```
prestador( 8,sofia,desconhecido03,hospitalPorto ).
excecao( prestador( ID , N , E , I ) ) :-
  prestador( ID , sofia , desconhecido03 , hospitalPorto ).
% Cuidado -> Conhecimento Imperfeito Incerto
%-------------------------------- - - - - - - - - - - - - - - - - -
cuidado( 10-5-2018 , desconhecido03 , 2 , eletrocardiograma , 47 ).
excecao( cuidado( Data , IdUT, IdPrest ,Descri , Custo ) ) :-
    cuidado( Data , desconhecido03 , IdPrest , Descri , Custo ).

%-------------------------------- - - - - - - - - - - - - - - - - -
% Utentes ->  Conhecimento Imperfeito Impreciso
%-------------------------------- - - - - - - - - - - - - - - - - -
excecao( utente( 8 , maria , 43 , lisboa ) ).
excecao( utente( 8 , sofia , 43 , lisboa ) ).
% Prestadores -> Conhecimento Imperfeito Impreciso
%-------------------------------- - - - - - - - - - - - - - - - - -
excecao( prestador( 9 , roberto , cirugia , hospitalLisboa ) ).
excecao( prestador( 10 , paulo , podologia , hospitalLisboa ) ).
% Cuidado -> Conhecimento Imperfeito Impreciso
%-------------------------------- - - - - - - - - - - - - - - - - -
excecao( cuidado( 2018 , 1 , 3 , chek-Up-completo , 150 ) ).
excecao( cuidado( 2018 , 2 , 3 , chek-Up-completo , 150 ) ).

%-------------------------------- - - - - - - - - - - - - - - - - -
% Utentes ->  Conhecimento Imperfeito Interdito
%-------------------------------- - - - - - - - - - - - - - - - - -
utente( 9, proibido01 , 56 , aveiro ).
excecao( utente( ID,N,I,M ) ) :-
  utente( ID,proibido01,I,M ).
nulo( proibido01 ).
+utente( ID,N,I,M ) ::
    (solucoes( (  ID , Nome , I , M ),
    (utente( 9 , Nome , 56 , aveiro ),
    nao( nulo( Nome ) ) ), S ),
    comprimento( S , N ),
    N==0).

utente( 10,rui,21,proibido02 ).
excecao( utente( ID , N , I, M ) ) :-
  utente( ID , N , I , proibido02 ).
nulo( proibido02 ).
+utente( ID , N , I ,M ) ::
    (solucoes( ( ID , N , I , Morada ),
    (utente( 10 , rui , 21 , Morada),
    nao( nulo( Morada ) )) , S ),
    comprimento( S , N ),
    N==0 ).
% Prestadores -> Conhecimento Imperfeito Interdito
%-------------------------------- - - - - - - - - - - - - - - - - -
```

```
prestador( 11 , proibido03 , ortopedia , hospitalLisboa ).
excecao( prestador( ID , N , E , I ) ) :-
  prestador( ID , proibido03 , E , I ).
nulo( proibido03 ).
+prestador( ID , Nome , E ,I ) ::
    (solucoes(( ID , Nome , E , I ),
    (prestador( 11 , Nome , ortopedia , hospitalLisboa),
    nao( nulo( Nome ) )), S ),
    comprimento( S , N ),
    N==0 ).

prestador( 12 , luisa , proibido04 , hospitalLisboa ).
excecao( prestador( ID , N , E , I ) ) :-
  prestador( ID , proibido04 , E , I ).
nulo( proibido04 ).
+prestador( ID , N , E , I ) ::
    (solucoes(( ID , N , Especialidade , I ),
    ( prestador( 12 , luisa , Especialidade , hospitalLisboa ),
    nao( nulo( Especialidade ) )), S ),
    comprimento( S , N ),
    N==0 ).
% Cuidado -> Conhecimento Imperfeito Interdito
%-------------------------------- - - - - - - - - - -  -  -  -  -   -
cuidado( 2018 , proibido03 , 1 , gripe , 500 ).
excecao( cuidado( Data , IdUT , IdPrest , Descri , Custo ) ):-
    cuidado( Data , proibido03 , IdPrest , Descri , Custo ).
nulo(proibido03).
+cuidado( Data , IdUT , IdPrest , Descri , Custo ) ::
    (solucoes(( Data , IdUT , IdPrest , Descri , Custo ),( cuidado(2018 , IdUT , 1 ,
     ↪  gripe , 500),
    nao( nulo( IdUt ) ) ) , S ),
    comprimento( S , N ),
    N==0 ).

cuidado( 2018,3,1,gripe,proibido06).
excecao( cuidado( Data,IdUT,IdPrest,Descri,Custo)) :-
    cuidado( Data , IdUT , IdPrest , Descri , proibido06 ).
nulo( proibido06 ).
+cuidado( Data , IdUT , IdPrest , Descri , Custo ) ::
    (solucoes(( Data , IdUT , IdPrest , Descri , Custo ),
    (cuidado( 2018 , 3 , 1 , gripe , Custo ),
    nao( nulo( Custo ) ) ), S ),
    comprimento( S , N ),
    N==0 ).
```

## 3. Manipular invariantes que designem restrições à inserção e à remoção de conhecimento do sistema

Um dos grandes problemas das bases de conhecimento é a garantia de consistência, e esse problema complica-se ainda mais com a extensão agora feita. Para lidar com o conhecimento imperfeito temos de forçar essa mesma imperfeição ao longo das evoluções e involuções feitas, e também passar conhecimento imperfeito a perfeito caso tal seja necessário, mas nunca o oposto.
Quando se evolui com uma adição de conhecimento perfeito relativo a conhecimento que até agora era imperfeito, é preciso determinar se a evolução é permitida, isto é, se não proibida e dentro do conjunto de hipóteses (caso impreciso). Se tal se confirmar é também necessário remover o que até agora era conhecimento imperfeito para que se mantenha a consistência.

```
%-------------------------------- - - - - - - - - - -  -  -  -  -   -
%-------------------------------- - - - - - - - - - -  -  -  -  -   -
%---------------------- Invariantes  - - - - - - - - -  -  -  -  -   -
%-------------------------------- - - - - - - - - - -  -  -  -  -   -
%-------------------------------- - - - - - - - - - -  -  -  -  -   -
%-------------------------------- - - - - - - - - - -  -  -  -  -   -
%---------- Conhecimento Perfeito Positivo e Desconhecido  -  -  -   -
%-------------------------------- - - - - - - - - - -  -  -  -  -   -
% Não permitir adicionar quando se tem o conhecimento perfeito negativo oposto
+utente( ID , N , I , M ) :: nao( -utente( ID , N , I , M ) ).
+prestador( ID , N , E , I ) :: nao( -prestador( ID , N , E , I ) ).
+cuidado( D , IDU , IDP , D , C ) :: nao( -cuidado( D , IDU , IDP , D , C ) ).
%-------------------------------- - - - - - - - - - -  -  -  -  -   -
%---------- Conhecimento Perfeito Negativo - - - - - -  -  -  -  -   -
%-------------------------------- - - - - - - - - - -  -  -  -  -   -
%-------------------------------- - - - - - - - - - -  -  -  -  -   -
% Não permitir adicionar se houver o conhecimento positivo perfeito oposto
+(-T) :: nao( T ).
%-------------------------------- - - - - - - - - - -  -  -  -  -   -
% Não permitir adicionar conhecimento negativo repetido
+(-T) :: (solucoes( T, (-T) , S ),
        comprimento( S , N ),
        N < 2 ).
%-------------------------------- - - - - - - - - - -  -  -  -  -   -
% Impossível adicionar excecoes a conhecimento perfeito positivo
+excecao( Termo ) :: nao( Termo ).
%-------------------------------- - - - - - - - - - -  -  -  -  -   -
% Nao permitir adicionar conhecimento perfeito positivo repetido,
% removendo conhecimento desconhecido, se este existir
%-------------------------------- - - - - - - - - - -  -  -  -  -   -
%-------- Utente
+utente( ID , N , I , M ) :: verificaPerfeitaEvoUtente( ID , N , I , M ).
verificaPerfeitaEvoUtente( ID , N , I , M ) :-
    solucoes( utente( ID , B , I , M ),
    utente( ID , B , I , M) , S ),
    removeDesconhecidoUtente( S ).
```

```
37 verificaPerfeitaEvoUtente( ID , N , I , M ) :-
38     solucoes( ( ID , N , I , M ),
39     utente( ID , N , I , M ) , S ),
40     comprimento( S , N ),
41     N==1.
42 removeDesconhecidoUtente( [ utente( ID , N , I ,M )] ) :-
43     demo(utente( ID , e ,I , M ) , desconhecido ),
44     removeTermos( [utente( ID , N , I , M ),
45     (excecao( utente( A , B , C , D )) :-
46             utente( A , N , C , D ))] ).
47 removeDesconhecidoUtente( [utente( ID , N , I , M)|L] ) :-
48     demo(utente( ID , e , I , M ) , desconhecido ),
49     removeTermos( [utente( ID , N , I , M ),
50     (excecao( utente( A , B , C , D )) :-
51             utente( A , N , C , D ))] ).
52 removeDesconhecidoUtente( [X|L] ) :-
53     removeDesconhecidoUtente( L ).
54 %-------------------------------- - - - - - - - - - -  -  -  -  -   -
55 %-------- Prestador
56 +prestador( ID , N , E , I ) :: verificaPerfeitaEvoPrestador( ID , N , E , I ).
57 verificaPerfeitaEvoPrestador( ID , N , E , I ) :-
58     solucoes( prestador( ID , B , E , I ),
59     prestador( ID , B , E , I ) , S),
60     removeDesconhecidoPrestador( S ).
61 verificaPerfeitaEvoPrestador( ID , N , E , I ) :-
62     solucoes( ( ID , N , E , I ) , prestador( ID , N , E , I ) , S ),
63     comprimento( S , N ),
64     N==1.
65 removeDesconhecidoPrestador( [prestador( ID , N , E , I )] ) :-
66     demo(prestador( ID , e , E , I ) , desconhecido ),
67     removeTermos( [prestador( ID , N , E , I ),
68     (excecao( prestador( A , B , C ,D )):-
69                 prestador( A , N , C , D ))] ).
70 removeDesconhecidoPrestador( [prestador( ID , N , E , I )|L] ) :-
71     demo( prestador( ID , e , E , I ) , desconhecido ),
72     removeTermos( [prestador( ID , N , E , I ) , ( excecao( prestador( A , B ,C ,  D
       ↪  )):-
73                 prestador( A , N , C , D ))] ).
74 removeDesconhecidoPrestador( [X|L] ) :-
75     removeDesconhecidoPrestador( L ).
76 %-------------------------------- - - - - - - - - - -  -  -  -  -   -
77 %-------- Cuidado
78 +cuidado( D , IDU , IDP , D , C ) :: verificaPerfeitaEvoCuidado( D , IDU , IDP , D , C
    ↪  ).
79 verificaPerfeitaEvoCuidado( D , IDU , IDP , D , C ) :-
80     solucoes( cuidado( D , B , IDP , D , C), cuidado( D , B , IDP , D , C ) , S ),
81     removeDesconhecidoCuidado( S ).
82 verificaPerfeitaEvoCuidado( D , IDU , IDP , D , C) :-
83     solucoes( (D , IDU , IDP , D , C ), cuidado( D , IDU , IDP , D , C ) , S ),
```

```
84      comprimento( S , N ),
85      N==1.
86  removeDesconhecidoCuidado( [cuidado( D , IDU , IDP , D , C )] ) :-
87      demo( cuidado( D , e , IDP , D , C ) , desconhecido ),
88      removeTermos( [cuidado( D , IDU , IDP , D , C ) , ( excecao( cuidado( D , IDU ,
          ↪  IDP , D , C )):-
89                  cuidado( A , IDU , C , D , E ))] ).
90  removeDesconhecidoCuidado( [cuidado( D , IDU , IDP , D , C )|L] ) :-
91      demo( cuidado( D , e , IDP , D , C ) , desconhecido ),
92      removeTermos( [cuidado( D , IDU , IDP , D , C ) , (excecao( cuidado( A , B , C , D
          ↪  , E )):-
93                  cuidado( A , IDU , C , D , E ))] ).
94  removeDesconhecidoCuidado( [X|L] ) :-
95      removeDesconhecidoCuidado( L ).
96  %--------------------------------- - - - - - - - - - - - - - - - - -
97  %---------- Conhecimento Imperfeito Impreciso - - - - - - - - - - -
98  %--------------------------------- - - - - - - - - - - - - - - - - -
99  %Apenas deixar adicionar conhecimento positivo se este pertence ao
100 %conjunto de conhecimento impreciso e remover o conhecimento impreciso
101 %--------------------------------- - - - - - - - - - - - - - - - - -
102 %-------- Utente
103 +utente( ID , N , I , M ) :: ( solucoes( B, excecao( utente( ID , B , I , M )) , S ),
104     contem( N , S ),
105     solucoes( excecao( utente( ID , B , I , M )), excecao( utente( ID , B , I , M)) ,
          ↪  S2 ),
106     removeTermos( S2 )
107     ).
108 %--------------------------------- - - - - - - - - - - - - - - - - -
109 %-------- Prestador
110 +prestador( ID , N , E , I ) :: ( solucoes( B , excecao( prestador( ID , B , E ,I )) ,
      ↪  S ),
111     contem( N , S ),
112     solucoes( excecao( prestador( ID , B , E , I )), excecao( prestador( ID , B , E ,
          ↪  I )) ,S2),
113     removeTermos( S2 )
114     ).
115 %--------------------------------- - - - - - - - - - - - - - - - - -
116 %-------- Cuidado
117 +cuidado( D , IDU , IDP , D , C ) :: ( solucoes( B, excecao( cuidado( D , B , IDP , D
      ↪  , C )) , S ),
118     contem( IDU , S) ,
119     solucoes( excecao( cuidado( D , B , IDP , D , C )), excecao( cuidado( D , B ,IDP,
          ↪  D , C )) , S2 ),
120     removeTermos( S2 )
121     ).
122 %--------------------------------- - - - - - - - - - - - - - - - - -
123 %-------------- Conhecimento Desconhecido - - - - - - - - - - - - -
124 %--------------------------------- - - - - - - - - - - - - - - - - -
125 +(excecao(T)) :: (solucoes( excecao( T ) , excecao( T ) , S ),
```

```
126     comprimento( S , N ),
127     N < 2).
128 +(excecao(-T)) :: (solucoes( excecao( T ),excecao( -T ) , S ),
129     comprimento( S , N ),
130     N < 2).
```

### 4. Lidar com a problemática da evolução do conhecimento, criando os procedimentos adequados

Juntamente com os invariantes já apresentados que respondem ao ponto anterior, mais predicados auxiliares foram usados para garantir consistência durante as evoluções. Um dos problemas mais importantes de se resolver é o de saber se o conhecimento imperfeito deve ou não ser evoluído em primeiro lugar. Isto obriga a que se determine o tipo de conhecimento a ser evoluído para depois saber, com base no que já temos na base de conhecimento, se a evolução é legal ou não.
Entre os casos mais óbvios temos a inserção de conhecimento perfeito quando já há uma versão perfeita, a inserção de conhecimento relativo a algo imperfeito interdito, adicionar conhecimento positivo/negativo quando já existe o oposto (quebraria consistência), ou ainda a adição de exceções sobre conhecimento perfeito (recorde-se que as exceções são usadas para demarcar conhecimento imperfeito).

### 5. Desenvolver um sistema de inferência capaz de implementar os mecanismos de raciocínio inerentes a estes sistemas

Como a linguagem *PROLOG* não usa a nossa lógica estendida, é necessário criar um interpretador para questões colocadas no novo formato, e suporte as três respostas agora possíveis: respostas do tipo verdadeira, falsa ou desconhecida. Seguindo com a convenção de nomes usada pelo docente e tendo por base o demonstrador usado nas aulas práticas, o nosso demonstrador (demo) difere de um interpretador de lógica normal apenas nas respostas de tipo desconhecido, sendo preciso não haver prova da questão nem prova da sua negação na base de conhecimento. Adicionamos também duas outras funcionalidades extra, uma para manipular listas de questões aplicando o demo a cada questão (demoCada), e outras duas para lidar com operações lógicas de questões (demoConj e demoDisj), que aplicam os operadores de conjunção e disjunção lógica sobre o conjunto de respostas, que na prática implementam os quantificadores ∀ e ∃, respectivamente.

```
1  %------------------------------ - - - - - - - - - - - -  -  -  -  -  -
2  %------------------------------ - - - - - - - - - - - -  -  -  -  -  -
3  %----------------------- Demonstradores - - - - - - -  -  -  -  -  -
4  %------------------------------ - - - - - - - - - - - -  -  -  -  -  -
5  %------------------------------ - - - - - - - - - - - -  -  -  -  -  -
6  %------------------------------ - - - - - - - - - - - -  -  -  -  -  -
7  % Extensao do meta-predicado demo: Questao,Resposta -> {verdadeiro, falso,
     ↪  desconhecido}
8  demo( Questao,verdadeiro ) :-
9      Questao.
10 demo( Questao, falso ) :-
11     -Questao.
```

```
12 demo( Questao,desconhecido ) :-
13     nao( Questao ),
14     nao( -Questao ).
15 %---------------------------------- - - - - - - - - - -  -  -  -  -   -
16 % Extensao do meta-predicado demoCada: [Questao],[Resposta] -> {verdadeiro, falso,
    ↪  desconhecido}
17 demoCada([], []).
18 demoCada([Questao|T], R) :-
19     demo(Questao, X),
20     demoCada(T, B),
21     R = [X|B].
22 %---------------------------------- - - - - - - - - - -  -  -  -  -   -
23 % Extensao do meta-predicado demoConj: [Questao | T], Resposta -> {verdadeiro, falso,
    ↪  desconhecido}
24 demoConj([], verdadeiro).                          % lista vazia -> verdadeiro
25 demoConj([Questao|T], verdadeiro) :-    % todos verdadeiros -> verdadeiro
26     demo(Questao, verdadeiro),
27     demoConj(T, verdadeiro).
28 demoConj([Questao|T], desconhecido) :-  % se cabeça desconhecido e nao há falsos na
    ↪  tail -> desconhecido
29     demo(Questao, desconhecido),
30     nao( demoConj(T, falso)).
31 demoConj([Questao|T], desconhecido) :-  % se cabeça nao é falso e na tail o resultado
    ↪  é desconhecido -> desconhecido
32     nao( demo(Questao, falso)),
33     demoConj(T, desconhecido).
34 demoConj([Questao|T], falso) :-                  % se cabeça for falso -> falso
35     demo(Questao, falso).
36 demoConj([Questao|T], falso) :-                % se tail tiver falsos -> falso
37     demoConj(T, falso).
38 %---------------------------------- - - - - - - - - - -  -  -  -  -   -
39 % Extensao do meta-predicado demoDisj: [Questao | T], Resposta -> {verdadeiro, falso,
    ↪  desconhecido}
40 demoDisj([], falso).                            % lista vazia -> falso
41 demoDisj([Questao|T], verdadeiro) :-         % se a cabeça for verdadeira ->
    ↪  verdadeiro
42     demo(Questao, verdadeiro).
43 demoDisj([Questao|T], verdadeiro) :-        % se a tail tiver resultado verdadeiro ->
    ↪  verdadeiro
44     demoDisj(T, verdadeiro).
45 demoDisj([Questao|T], desconhecido) :-        % se a cabeça for desconhecido e a tail
    ↪  tiver resultado nao verdadeiro -> desconhecido
46     demo(Questao, desconhecido),
47     nao( demoDisj(T, verdadeiro)).
48 demoDisj([Questao|T], desconhecido) :-        % se a cabeça nao for verdadeiro e a
    ↪  tail tiver resultado desconhecido -> desconhecido
49     nao( demo(Questao, verdadeiro)),
50     demoDisj(T, desconhecido).
```

```
demoDisj([Questao|T], falso) :-                % se a cabeça for falso e a tail tiver
 ↪  resultado falso -> falso
    demo(Questao, falso),
    demoDisj(T, falso).
```

### Predicados e meta-predicados auxiliares

Durante a resolução foram usados predicados e meta-predicados que, apesar de não responderem diretamente a qualquer problema proposto, serviram de suporte para os predicados que os resolvem, ou são usados como sinais para a linguagem de programação em uso.

```
:- op( 900,xfy,'::' ).
:- dynamic utente/4.
:- dynamic prestador/4.
:- dynamic cuidado/4.
:- dynamic excecao/1.
:- dynamic nulo/1.
:- dynamic '-'/1.
% Extensao do meta-predicado utente: ID,N,I,M -> {V,F,D}
% Utente: #idUt, Nome, Idade, Morada -> {V,F,D}
-utente( ID,N,I,M ) :-
    nao( utente( ID,N,I,M ) ),
    nao( excecao( utente( ID,N,I,M ) ) ).
% Extensao do meta-predicado prestador: ID,N,E,I -> {V,F,D}
% Prestador: #idPrest, Nome, Especialidade, Instituição -> {V,F,D}
-prestador( ID,N,E,I ) :-
    nao( prestador( ID,N,E,I ) ),
    nao( excecao( prestador( ID,N,E,I ) ) ).
% Extensao do meta-predicado cuidado: D,IDU,IDP,D,C -> {V,F,D}
% Cuidado: Data, #idUt, #idPrest, Descrição, Custo -> {V,F}
-cuidado( D,IDU,IDP,D,C ) :-
    nao( cuidado( D,IDU,IDP,D,C ) ),
    nao( excecao( cuidado( D,IDU,IDP,D,C ) ) ).
% Extensão do predicado solucoes
solucoes( X,Y,Z ) :-
  solucoes( X,Y,Z ).
% Extensão do predicado removeTermos: [X|L] -> {V, F}
removeTermos( [] ).
removeTermos( [X] ) :-
    retract(X).
removeTermos( [X|L] ) :-
    retract(X),
    removeTermos( L ).
% Extensão do predicado insercao: Termo -> {V, F}
insercao( Termo ) :-
    assert( Termo ).
insercao( Termo ) :-
    retract( Termo ),!,fail.
```

```
% Calcula o comprimento de uma lista
% Extensao do predicado comprimento: L,R -> {V,F}
comprimento([],0).
comprimento([H|T],R) :- comprimento(T,N), R is N+1.
% Testa todos os elementos da lista
% Extensão do predicado teste: [R|LR] -> {V,F}
teste([]).
teste([I|L]) :- I, teste(L).
% Verifica se contem um elemento numa dada lista
% Extensão do predicado contem: H,[H|T] -> {V, F}
contem(X,[]).
contem(H, [H|T]).
contem(X, [H|T]) :-
    contem(X, T).
% Extensao do meta-predicado nao: Questao -> {V,F}
nao( Questao ) :-
    Questao, !, fail.
nao( Questao ).
```

## Conclusão e Trabalho Futuro

Agora que temos a lógica estendida foi-nos possível desenvolver uma melhor e mais fiel representação do sistema quando comparado com a da fase anterior. Esta foi claramente mais difícil devido a toda a nova forma de pensar à volta do conhecimento imperfeito, mas acabamos por conseguir atingir as metas estabelecidas.
Apesar da notável evolução temos a certeza de que será possível melhorar a representação do sistema atual com conhecimentos mais avançados.

## Referências

[1] Cesar Analide e José Neves. *Representação de Informação Incompleta*. Departamento de Informática, Universidade do Minho.