Universidade do Minho
Departamento de Informática

2º Exercício do Trabalho de Grupo: Programação em Lógica Estendida e Conhecimento Imperfeito

Diogo Braga
Diogo Rocha
João Silva
Ricardo Caçador
Ricardo Veloso

Universidade do Minho

Departamento de Informática

2º Exercício do Trabalho de Grupo: Programação em Lógica Estendida e Conhecimento Imperfeito

Diogo Braga

Diogo Rocha

João Silva

Ricardo Caçador

Ricardo Veloso

Resumo

O seguinte relatório consiste na apresentação do trabalho realizado no âmbito da Unidade Curricular de Sistemas de Representação de Conhecimento e Raciocínio relativo ao segundo exercício requerido, e tem como objetivo a continuação do desenvolvimento de competências na linguagem de programação em lógica PROLOG.

O exercício referido acima consiste num sistema de representação de conhecimento imperfeito com capacidade para caracterizar um universo de discurso na área da prestação de cuidados de saúde pela realização de serviços de atos médicos, recorrendo à temática de valores nulos.

Ao longo do presente relatório irá ser explicado todo o processo e todos os mecanismos necessário para dar resposta aos desafios propostos no enunciado desta segunda fase.

Tabela de Conteúdos

Tabela de Figuras

1 - Introdução

Muito à semelhança da primeira fase, para esta segunda fase foi-nos proposta a criação de um sistema representação de conhecimento e raciocínio que caracterize a estrutura de uma área de prestação de cuidados de saúde. A criação deste sistema é feita através da utilização da linguagem de programação PROLOG.

Para o efeito, foi-nos apresentado um panorama possível para caracterizar o conhecimento bem como um conjunto de funcionalidades que o sistema deve respeitar tendo uma atenção redobrada no que conta ao tema da representação de conhecimento imperfeito.

De seguida, iremos apresentar todas as soluções realizadas pelo grupo para a realização do exercício proposto bem como as extensões de conhecimento implementadas no sistema.

2 - Descrição do Trabalho

Devido à ligação que existe com a primeira fase do projeto, é importante relembrar os predicados que foram criados:

- utente: IdUt, Nome, Idade, Cidade, Seguro -> {V,F,D}
- serviço: IdServ, Descrição, Instituição, Cidade -> {V,F,D}
- consulta: Data, IdUt, IdServ, Custo, IdMed -> {V,F,D}
- data: Dia, Mes, Ano -> {V,F}
- medico: IdMed, Nome, Idade, IdServ -> {V,F,D}
- seguro: IdSeg, Descrição, Taxa -> {V,F,D}

Tendo em conta a Programação em Lógica Estendida abordada nesta fase, é importante diferenciar que, em complemento com um sistema de inferência, agora os predicados podem resultar num valor de Verdadeiro, Falso ou Desconhecido.

2.1 - Representação de Conhecimento Positivo

A representação de conhecimento positivo foi replicada da primeira fase, tendo em conta que já nesse momento tinha existido este tipo de representação de conhecimento.

Desta forma, a seguir apresentamos um exemplo de conhecimento positivo, para cada um dos predicados estabelecidos no projeto:

```
utente(1, joao, 31, guimaraes, 1).
servico(1, cardiologia, hospitaldaluz, guimaraes).
consulta(data(7,3,2019), 1, 1, 10, 3).
medico(3, rui, 56, 1).
seguro(1, adse, 0.4).
```

Figura 1 – Exemplos de Conhecimento Positivo

2.2 – Representação de Conhecimento Negativo

A representação de conhecimento negativo foi algo que, devido aos novos requisitos provenientes da Lógica Estendida, fez sentido criar no trabalho, pois desta forma tornamos possível representar algo que é falso. Tal é possível representar através da negação explícita, como através da negação forte.

Negação Explícita

De seguida, são apresentados alguns exemplos de negação explícita dos predicados envolvidos no trabalho, associados à explicação dos mesmos.

```
-utente(11,rafa,50,guimaraes,0).
-servico(8, neurologia, hsog, guimaraes).
-consulta(data(7,3,2019), 10, 3, 10, 8).
-medico(8, roberto, 48, 5).
-seguro(4, secur, 0.2).
```

Figura 2 – Exemplos de Negação Explícita

Exemplos:

1. Não existe um utente de 50 anos com id 11 e nome rafa, que viva em guimaraes sem seguro.
2. Não existe um serviço de neurologia, com id 8, no hsog em guimaraes.
3. Não existe uma consulta do dia 07/03/2019, do utente com o id 10 para o serviço com o id 3, no qual o custo monetário é 10 e o id do médico é 8.
4. Não existe um médico de ginecologia com o id 8, chamado roberto, e com 48 anos de idade.
5. Não existe um seguro com id 4 e nome secur que possuía uma taxa de retorno de 0.2.

Negação Forte

Este género de negação assenta na aplicação do Pressuposto do Mundo Fechado na programação em Lógica Estendida. Com tal aplicação garantimos que algo que não tem prova de ser verdadeiro, é implicitamente falso.

De seguida, é apresentada a extensão do predicado que define a negação forte do predicado utente. As extensões dos restantes predicados são realizadas da mesma forma, só que com o predicado respetivo.

```
-utente(IU,N,I,C,IdS) :-
    nao(utente(IU,N,I,C,IdS)),
    nao(excecao(utente(IU,N,I,C,IdS))).
```

Figura 3- Negação Forte do predicado utente

Esta negação do utente parametrizado na figura é validada caso não exista prova que o utente seja conhecimento verdadeiro, e caso não exista prova duma exceção associada a esse mesmo utente.

2.3 – Representação Conhecimento Imperfeito

2.3.1 – Conhecimento Imperfeito Incerto

Para representar conhecimento imperfeito incerto vamos recorrer a um exemplo prático definido pelo grupo:

```
servico(9,fisiatria,xpto021,braga).
excecao( servico(IDS,D,I,C) ) :-
        servico(IDS,D,xpto021,C).
```

Figura 4 - Exemplo de conhecimento imperfeito incerto

Podemos verificar que, de facto, existe um serviço com id nº9 correspondente à área da fisiatria e foi realizado em Braga mas, relativamente à instituição onde este foi realizado a informação é desconhecida. Neste caso, foi escolhido "xpto021" no lugar da instituição prestadora do serviço para representar um caso de exceção.

Resumidamente, o conhecimento imperfeito incerto representa algo desconhecido de um conjunto indeterminado de hipóteses.

2.3.2 – Conhecimento Imperfeito Impreciso

Para representar conhecimento imperfeito impreciso vamos, tal e qual como no ponto antecedente, recorrer a um exemplo:

```
excecao( seguro(5,allianz,X) ) :-
        X >= 0.2,
        X =< 0.25.
```

Figura 5 - Exemplo de conhecimento imperfeito impreciso

É possível averiguar que existe um seguro 5 denominado de "Allianz" mas não sabemos a taxa de dedução do valor das consultas, apenas sabemos que se encontra entre 0.2 (20%) e 0.25 (25%).

Portanto, o conhecimento imperfeito impreciso representa algo desconhecido de um conjunto limitado de soluções.

2.3.3 – Conhecimento Imperfeito Interdito

Por fim, para representar conhecimento imperfeito interdito vamos também recorrer a um exemplo construído pelo grupo:

```
consulta(data(10,4,2019),2,7,xpto024,1).
excecao( consulta(DA,IU,IS,C,IM) ) :-
         consulta(DA,IU,IS,xpto024,IM).
nulo(xpto024).
+consulta( DA,IU,IS,C,IM ) :: (solucoes((CS), (consulta(data(10,4,2019),2,7,CS,1), nao(nulo(CS))), S),
                               comprimento( S,N ), N == 0 ).
```

Figura 6 - Exemplo de conhecimento imperfeito interdito

Temos portanto uma consulta realizada no dia 10 de Abril de 2019, pelo médico com id igual a 1 (Ricardo) que presta o serviço do tipo 7 (Oftalmologia) e requisitada pelo utente com id igual a 2 (André). Fora tudo isto, nunca será possível determinar o valor dela.

Para representar este tipo de conhecimento é necessário definir uma exceção (“xpto024”) e atribuir-lhe um valor nulo. Além disso, é também preciso impedir a adição de conhecimento a esta consulta em que o custo seja não nulo.

Finalmente, o conhecimento imperfeito interdito representa algo desconhecido e não permitido conhecer.

2.4 – Invariantes de inserção e remoção de conhecimento

Para que o sistema funcione corretamente, foi necessário manipular alguns invariantes que controlam a inserção e remoção de conhecimento.

Não deve ser, por exemplo, permitido inserir um utente com um ID que já esteja associado a outro utente já presente na base do conhecimento.

```
% Invariante Estrutural:  nao permitir a insercao de conhecimento positivo repetido
+utente(IU,_,_,_,_) :: (solucoes(IU, (utente(IU,_,_,_,_)), S),
                        comprimento(S,1)).
```

Figura 7 - Invariante que não permite a inserção de conhecimento repetido

São então procurados utentes com o ID do utente que pretendemos inserir e criámos uma lista com estes. Caso a lista tenha o comprimento igual a 1, podemos inserir o utente na base do conhecimento.

Também não deve ser possível inserir consultas relativas a clientes não existentes.

```
% Invariante Referencial:  nao permitir a insercao de consultas relativas a utentes inexistentes.
+consulta(_,U,_,_,_) :: (utente(U,_,_,_,_)).
```

Figura 8 - Invariante que não permite a inserção de consultas relativas a um utente inexistente

Para que a consulta seja adicionada à base do conhecimento é imperativo que exista um utente com ID respetivo ao ID do utente da consulta.

Outro exemplo de invariante é a impossibilidade de inserir conhecimento negativo repetido.

```
% Invariante Estrutural: nao permitir a insercao de conhecimento negativo repetido
+(-medico(IM,N,I,IS)) :: (solucoes(IM, -medico(IM,N,I,IS), S),
                          comprimento(S, 1)).
```

Figura 9 - Invariante que não permite a inserção de conhecimento negativo repetido

Este invariante tem o mesmo método de processo que o primeiro invariante apresentado.

Relativamente às remoções os invariantes criados foram:

```
%----------------UTENTE----------------%

% Invariante Referencial: um utente so pode ser removido se nao existir consultas associadas a este.
-utente(ID,_,_,_,_) :: nao(consulta(_,ID,IDS,_,_)).

%----------------SERVICO----------------%

% Invariante Referencial:  nao permitir a remoção dum serviço se existirem consultas associadas a este.
-servico(ID,_,_,_) :: nao(consulta(_,_,ID,_,_)).

%----------------MEDICO----------------%

% Invariante Referencial:  nao permitir a remoção dum medico se existirem consultas associadas a este.
-medico(ID,_,_,_) :: nao(consulta(_,_,_,_,ID)).
```

Figura 10 - Invariantes para a remoção de conhecimento

Todos estes predicados não permitem remover conhecimento que esteja associado, de qualquer forma, a outro tipo de conhecimento. Por exemplo, se um utente tiver consultas associadas a ele não podemos removê-lo.

2.5 – Evolução e regressão do conhecimento

Uma vez que nesta fase estamos a lidar com conhecimento perfeito e imperfeito, os predicados de inserção e regressão da fase anterior não conseguem evoluir ou regredir a base de conhecimento da forma que se esperaria.

Desta forma, foi imperativo que criássemos predicados capazes de lidar com este novo problema. A nossa proposta de solução para esta problemática passou por dividir cada tipo de conhecimento e criar um predicado para cada um deles.

A explicação que se segue contempla apenas o predicado Utente. A evolução e regressão para os restantes predicados foram assentes nas mesmas ideias.

2.5.1 – Evolução de Conhecimento Perfeito

Quando inserimos conhecimento perfeito, precisamos de ter em conta se este é positivo ou negativo, e se existe alguma imprecisão ou incerteza na base de conhecimento associado a este. No caso de ser algum conhecimento interdito a estar associado ao novo conhecimento apenas é necessário verificar os invariantes definidos.

O predicado **evolucaoPerfeito: Utente -> {V,F,D}**, trata da inserção de conhecimento perfeito, tanto positivo como negativo, na base de conhecimento. Este predicado é apresentado a seguir.

```
evolucaoPerfeito(utente(Id,Nome,Idade,Morada,Seguro)):-
    si(utente(Id,Nome,Idade,Morada,Seguro), desconhecido),
    removerConhecimentoImpreciso(utente(Id,Nome,Idade,Morada,Seguro)),
    evolucao(utente(Id,Nome,Idade,Morada,Seguro)),
    evolucao(conhecimentoPerfeito(Id)).

evolucaoPerfeito(utente(Id,Nome,Idade,Morada,Seguro)):-
    si(utente(Id,Nome,Idade,Morada,Seguro), falso),
    removerConhecimentoImpreciso(utente(Id,Nome,Idade,Morada,Seguro)),
    evolucao(utente(Id,Nome,Idade,Morada,Seguro)),
    evolucao(conhecimentoPerfeito(Id)).

evolucaoPerfeito(-utente(Id,Nome,Idade,Morada,Seguro)):-
    si(-utente(Id,Nome,Idade,Morada,Seguro), verdadeiro),
    evolucao(-utente(Id,Nome,Idade,Morada,Seguro)).
```

Figura 11 - Definição do predicado evolucaoPerfeito

Como se pode reparar na imagem acima, o caso em que o conhecimento a adicionar for **verdadeiro**, não é tratado nem admissível uma vez que não é permitido ter conhecimento repetido na base de conhecimento.

Para o caso em que o conhecimento a ser inserido é **desconhecido** ou **falso**, este é tratado da mesma forma. Primeiro é removido todo o conhecimento impreciso e incerto associado este, caso exista, e de seguida é usado o predicado **evolução** proveniente da primeira fase que insere ou não o novo facto. Por último é inserido na base de conhecimento um facto **conhecimentoPerfeito**, que marca através do **ID** do utente o tipo de conhecimento associado a si.

Os predicados auxiliares para contruir o **evolucaoPerfeito** são apresentados a seguir:

- removerConhecimentoImpreciso: Utente -> {V,F,D}

```
removerConhecimentoImpreciso(utente(Id,Nome,Idade,Morada,Seguro)) :-
    retract(excecao(utente(Id,_,_,_,_))),
    removerConhecimentoImpreciso(utente(Id,Nome,Idade,Morada,Seguro)).
removerConhecimentoImpreciso(utente(Id,Nome,Idade,Morada,Seguro)) :-
    retract(conhecimentoImpreciso(utente(Id))),
    removerImpreciso(utente(Id,Nome,Idade,Morada)).
removerConhecimentoImpreciso(utente(Id,Nome,Idade,Morada,Seguro)) :-
    removerConhecimentoIncerto(utente(Id,Nome,Idade,Morada,Seguro)).
```

Figura 12 - Definição do predicado auxiliar removerConhecimentoImpreciso

- removerConhecimentoIncerto: Utente -> {V,F,D}

```
removerConhecimentoIncerto(utente(IdUt,Nome,Idade,Morada,Seguro)) :-
    conhecimentoIncertoIdade(utente(IdUt,I)),
    regressaoQuery((excecao(utente(Id,N,Ida,M,S)) :- utente(Id,N,I,M,S))),
    regressao(utente(IdUt, _, _ , _, _)),
    regressao(conhecimentoIncertoIdade(utente(IdUt, _))).
removerConhecimentoIncerto(utente(IdUt,Nome,Idade,Morada,Seguro)) :-
    conhecimentoIncertoMorada(utente(IdUt,M)),
    regressaoQuery((excecao(utente(Id,N,I,Mora,S)) :- utente(Id,N,I,M,S))),
    regressao(utente(IdUt, _, _ , _, _)),
    regressao(conhecimentoIncertoMorada(utente(IdUt, _))).
removerConhecimentoIncerto(utente(IdUt,Nome,Idade,Morada,Seguro)).
```

Figura 13 - Definição do predicado auxiliar removerConhecimentoIncerto

Os predicados acima definidos removem da base de conhecimento situações de incerteza e imprecisão associado ao facto a ser inserido. Usamos ainda um predicado já apresentado no exercício anterior que é **evolucao**.

2.5.2 – Regressão de Conhecimento Perfeito

Para remover conhecimento perfeito da base de conhecimento, não é necessário verificar nenhuma restrição especial, apenas verificar que existe na base de conhecimento, pelo que utilizamos o predicado:

- **regressaoPerfeito: Utente -> {V,F,D}**,

com auxílio do predicado **regressao** já apresentado no exercício anterior.

```
regressaoPerfeito(Termo) :-
    si(Termo,verdadeiro),
    regressao(Termo).

regressaoPerfeito(-Termo) :-
    si(-Termo,verdadeiro),
    regressao(-Termo).
```

Figura 14 - Definição do predicado regressaoPerfeito

2.5.3 – Evolução de Conhecimento Incerto

No caso do utente, para evoluir conhecimento incerto é necessário dizer qual dos seus parâmetros será incerto. Para resolver tal questão, decidimos que para cada parâmetro criar-se-ia um predicado. Um exemplo da evolução deste tipo de conhecimento é apresentado a seguir **evolucaoIncertoMorada**: **Utente -> {V,F,D}**. Este predicado adiciona conhecimento incerto sobre a morada dum dado utente.

```
evolucaoIncertoMorada(utente(IdUt, Nome, Idade, Morada,Seguro)) :-
    si(utente(IdUt, Nome, Idade, Morada,Seguro),falso),
    assert((excecao(utente(Id,N,I,M,S)) :- utente(Id,N,I,Morada,S))),
    assert(utente(IdUt, Nome, Idade, Morada, Seguro)),
    assert(conhecimentoIncertoMorada(utente(IdUt,Morada))).
```

Figura 15 - Definição do predicado evolucaoIncertoMorada

Para adicionar este tipo de conhecimento inicialmente temos que verificar que este não existe na base de conhecimento, e só depois inserir a exceção relativa a conhecimento incerto, o utente e por último marcar o **ID** e a **Morada** do utente como sendo conhecimento incerto através do predicado **conhecimentoIncertoMorada: Utente -> {V,F,D}**.

Ao adicionar este tipo de conhecimento não temos que ter em atenção em remover outro tipo de conhecimento imperfeito uma vez que o conhecimento incerto é mais abstrato (considera mais casos desconhecidos) do que o conhecimento impreciso e do que o conhecimento interdito.

2.5.4 - Evolução de Conhecimento Impreciso

Para evoluir conhecimento impreciso é necessário que se receba uma lista de utentes, e que esta lista represente o mesmo utente. Temos ainda que verificar que o que está a ser inserido não é conhecimento perfeito. Depois de concretizadas estas verificações basta apenas inserir as exceções na base de conhecimento.

Para tal tarefa utilizamos o predicado **evolucaoImpreciso: [Utente] -> {V,F,D}**.

```
evolucaoImpreciso([utente(IdUt, Nome, Idade, Morada, Seguro)|T]) :-
    T \= [],
    mesmoId(T, IdUt),
    nenhumPerfeito([utente(IdUt, Nome, Idade, Morada, Seguro)|T]),
    removerConhecimentoIncerto(utente(IdUt, Nome, Idade, Morada, Seguro)),
    insereExcecoes([utente(IdUt, Nome, Idade, Morada, Seguro)|T]).
```

Figura 16 - Definição do predicado evolucaoImpreciso

Para concretizar este predicados necessitamos de definir alguns predicados auxiliares:

- **mesmoId: [Utente], Id -> {V,F,D}**

Pretende-se verificar com este predicado se todos os utentes da lista representam o mesmo utente.

```
mesmoId([], _).
mesmoId([utente(Id1, _, _, _, _) | T], Id2) :-
    Id1 == Id2,
mesmoId(T, Id2).
```

Figura 17 - Definição do predicado mesmoId

- **nenhumPerfeito: [Utente] -> {V,F,D}**

Pretende-se verificar com este predicado se todos os utentes da lista não representam o conhecimento perfeito.

```
nenhumPerfeito([]).
nenhumPerfeito([H|T]) :-
    si(H,falso),
    nenhumPerfeito(T).
nenhumPerfeito([H|T]) :-
    si(H,desconhecido),
    nenhumPerfeito(T).
```

Figura 18 - Definição do predicado nenhumPerfeito

- **insereExcecoes: [Utente] -> {V,F,D}**

Pretende-se com este predicado inserir todas as exceções na base de conhecimento e ainda marcar o utente como detentor de conhecimento impreciso através do predicado **conhecimentoImpreciso: Id -> {V,F,D}**

```
insereExcecoes([]).
insereExcecoes([utente(IdUt, Nome, Idade, Morada, Seguro)|T]) :-
    assert(excecao(utente(IdUt, Nome, Idade, Morada, Seguro))),
    insereExcecoes(T),
    assert(conhecimentoImpreciso(utente(IdUt))).
```

Figura 19 - Definição do predicado insereExcecoes

O predicado **removerConhecimentoIncerto** já foi apresentado nas subsecções seguintes, mas é extremamente necessário pois se quisermos adicionar conhecimento impreciso, mas já existir conhecimento incerto, temos que remover

o incerto e adicionar o impreciso. Esta troca é feita pois o conhecimento impreciso pode ser considerado uma especialização do conhecimento incerto, onde os valores para uma determinada incerteza são limitados apenas pelos valores possíveis.

2.5.4 - Evolução de Conhecimento Interdito

Tal como para o conhecimento incerto, para representar um parâmetro do utente que seja interdito é necessário dizer qual dos seus parâmetros será interdito. É ainda fundamental que não exista nenhum conhecimento na base associado ao utente a ser inserido.

Para resolver tal questão, por exemplo para o parâmetro **Morada** dum utente utilizamos o predicado:

- **evolucaoInterditoMorada: Utente -> {V,F,D}**

```
evolucaoInterditoMorada(utente(IdUt, Nome, Idade, Morada, Seguro)) :-
    testaConhecimento(IdUt),
    assert(nulo(Morada)),
    assert((excecao(utente(Id,N,I,M,S)) :- utente(Id,N,I,Morada,S))),
    assert((+utente(Id,N,I,M,S) :: ( solucoes(Id,(utente(Id,_,_,Morada,_), nulo(Morada)),Lista),
                                     comprimento(Lista,0) ))),
    assert(utente(IdUt,Nome,Idade,Morada,Seguro)).
```

Figura 20 - Definição do predicado evolucaoInterditoMorada

Necessitamos primeiramente de verificar se já existia algum conhecimento associado ao utente em questão, para tal utilizamos o predicado:

- **testaConhecimento: Id -> {V,F,D}**

```
testaConhecimento(IdUt) :-
    si(conhecimentoPerfeito(IdUt),desconhecido),
    si(conhecimentoImpreciso(IdUt),desconhecido),
    si(conhecimentoIncertoIdade(IdUt,_),desconhecido),
    si(conhecimentoIncertoMorada(IdUt,_),desconhecido).
```

Figura 21 - Definição do predicado testaConhecimento

Depois de realizer tal teste é necessário introduzir na base o valor nulo associado à **Morada**, a exceção correspondente, o utente e ainda um invariante que dita que nenhum conhecimento acerca deste utente pode ser inserido. Com isto proíbem-se as violações ao valor interdito.

2.6 – Sistema de Inferência

Tendo em conta a diferente abordagem desta fase, que tem por base a Programação em Lógica Estendida, foi conclusivo que os predicados agora podem resultar num valor de Verdadeiro, Falso ou Desconhecido, demonstrando assim que os contradomínios dos novos predicados são mais abrangentes.

Devido a tal, foi necessário criar um Sistema de Inferência diferente que fosse capaz de analisar os três tipos de respostas que os novos predicados podem efetuar.

- Verdadeiro, caso a prova seja verdadeira
- Falso, caso a prova seja falsa
- Desconhecido, caso não exista prova

O sistema de inferência analisa, portanto, todas as questões da seguinte forma:

```
si(Questao,verdadeiro) :-
    Questao.
si(Questao,falso) :-
    -Questao.
si(Questao,desconhecido) :-
    nao(Questao),
    nao(-Questao).
```

Sendo que o predicado não é definido da seguinte forma:

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

Figura 22 - Predicado nao

Este predicado tenta procurar uma prova para a questão parametrizada, e caso este falhe, é concluído que não existe prova.

De forma a estarmos perante um Sistema de Inferência mais capacitado, foram desenvolvidas extensões para ser possível responder a mais do que uma questão de cada vez.

As operações lógicas abordadas foram:

- Conjunção
- Disjunção
- Implicação
- Equivalência

Desta forma é possível realizar duas ou mais questões de uma só vez no Sistema de Inferência, caso estas possam ser conjugadas com os operadores lógicos referidos.

Para tal, foi construído o seguinte meta-predicado que :

```
si(Questao && X, V) :- si(Questao, V1), si(X, V2), conjuncao(V1, V2, V), !.
si(Questao $$ X, V) :- si(Questao, V1), si(X, V2), disjuncao(V1, V2, V), !.
si(Questao => X, V) :- si(Questao, V1), si(X, V2), implicacao(V1, V2, V), !.
si(Questao <=> X, V) :- si(Questao, V1), si(X, V2), equivalencia(V1, V2, V), !.
```

Figura 23 - Meta-predicado Operações Lógicas

Em complemento com este meta-predicado foi estabelecido, para cada um dos valores lógicos, os resultados que estes podem adquirir consoante os valores resultantes dos predicados parametrizados.

Primeiro, no caso da conjunção, definimos os seguintes casos:

```
conjuncao(verdadeiro, verdadeiro, verdadeiro).
conjuncao(falso, _, falso).
conjuncao(_, falso, falso).
conjuncao(desconhecido, verdadeiro, desconhecido).
conjuncao(verdadeiro, desconhecido, desconhecido).
```

Figura 24 - SI Conjunção

No caso da disjunção, definimos os seguintes casos:

```
disjuncao(verdadeiro, X, verdadeiro).
disjuncao(X, verdadeiro, verdadeiro).
disjuncao(desconhecido, Y, desconhecido) :- Y \= verdadeiro.
disjuncao(Y, desconhecido, desconhecido) :- Y \= verdadeiro.
disjuncao(falso, falso, falso).
```

Figura 25 - SI Disjunção

No caso da implicação, definimos os seguintes casos:

```
implicacao(falso, X, verdadeiro).
implicacao(X, verdadeiro, verdadeiro).
implicacao(verdadeiro, desconhecido, desconhecido).
implicacao(desconhecido, X, desconhecido) :- X \= verdadeiro.
implicacao(verdadeiro, falso, falso).
```

Figura 26 - SI Implicação

Por último, no caso da equivalência, definimos os seguintes casos:

```
equivalencia(X, X, verdadeiro) :- X \= desconhecido.
equivalencia(desconhecido, Y, desconhecido).
equivalencia(X, desconhecido, desconhecido).
equivalencia(verdadeiro, falso, falso).
equivalencia(verdadeiro, falso, falso).
```

Figura 27 - SI Equivalência

3 – Extras

Nesta fase decidimos incluir algumas funcionalidades extra no sistema de modo a que este fique mais evoluído.

Um dos extras foi a inclusão de novos predicados relacionados com a base do conhecimento. Estes predicados, já introduzidos no início do relatório, foram os seguintes:

Médico

Um médico é caracterizado pelo seu ID, Nome, Idade e Id do serviço ao qual pertence. O predicado associado é o seguinte:

- medico: IdMed, Nome, Idade, IdServ -> {V,F,D}

Seguro

Um seguro é caracterizado pelo seu ID, Descrição e taxa de retorno associada a ele.

- seguro: IdSeg, Descrição, Taxa -> {V,F,D}

O outro extra que decidimos incluir foi o guardamento e carregamento de factos através da utilização de um ficheiro de texto.

Aquando da realização de testes com o *sicstus Prolog* reparámos que todo o conhecimento que retirávamos ou colocávamos era apenas conhecimento que se mantinha em tempo de execução. No fim da execução do interpretador todas as alterações que tínhamos feito eram apagadas.

Desta forma achamos que podia ser conveniente guardar de algum modo todos os factos referentes aos diversos predicados, para que conseguíssemos realizar testes com uma maior quantidade de dados, e de facto a nossa base de conhecimento evoluir sem que fosse perdida nenhuma informação.

Apoiado nesta ideia o grupo desenvolveu dois novos predicados. O primeiro fica encarregue de guardar em ficheiro todos os factos existentes na base de conhecimento até ao momento.

- guardaFactos: Ficheiro -> {V,F}

```
guardaFactos(Ficheiro) :-
    tell(Ficheiro),
    listing,
    told.
```

Figura 28 - Extensão do predicado guardaFactos

Neste predicado usamos dois predicados *built-in* do PROLOG, *tell*/1 e *told*/0. O primeiro predicado abre o Ficheiro e torna-o o atual output. Depois de escrever os dados do predicado *listing* no Ficheiro, realiza-se o predicado *told* que fecha o atual output (Ficheiro).

Ora possuindo um predicado que guarda todos os factos em ficheiro necessitámos doutro predicado que carregue esses factos do ficheiro onde se encontram guardados.

- carregaFactos: Ficheiro -> {V,F}

```
carregaFactos(Ficheiro) :-
    seeing(InputAtual),
    see(Ficheiro),
    repeat,
    read(Termo),
    (Termo == end_of_file -> true ;
    assert(Termo),fail),
    seen,
    see(InputAtual).
```

Figura 29 - Extensão do predicado carregaFactos

Para este predicado utilizamos 5 predicados *built-in* do PROLOG. O primeiro, *seeing/1*, guarda em InputAtual o input atual do interpretador. O segundo, *see/1*, torna o Ficheiro o atual input do interpretador. O terceiro, *repeat/0*, é utilizado frequentemente para reproduzir ciclos de falha, neste caso para ler termos do input atual do interpretador em ciclo até que seja atingido *end_of_file*. O

predicado utilizado para ler termos foi o *read/1*. Por último utilizamos o predicado *seen/0*, que fecha o input atual do interpretador.

Para finalizar é utlizado novamente o predicado *see* para tornar o input atual o input anterior à realização do predicado *carregaFactos*, que ficou guardado em InputAtual.

Para dar uso a estes dois predicados junto com o relatório e com o ficheiro relativo ao exercício enviamos um ficheiro *GRUPO10_FACTOS.txt* onde residem alguns factos que podem ser usados para testar os predicados.

4 - Conclusão

Este trabalho prático teve como objetivo solidificar os conhecimentos relativos à representação de conhecimento imperfeito, recorrendo à utilização de valores nulos. Foi assim possível entender a utilização destes conhecimentos para casos práticos.

Em termos de desafios, o principal terá sido representar corretamente a evolução de conhecimento perfeito e imperfeito, tendo-se relevado algo complicado.

Em suma, o grupo considera que realizou um bom trabalho e que respondeu de uma forma coesa e correta a todas as funcionalidades requisitadas no enunciado do projeto. Estamos, assim, prontos para avançar para a fase seguinte.

5 – Bibliografia

[Analide, 2011] ANALIDE, César, NOVAIS, Paulo, NEVES, José,

"Sugestões para a Redacção de Relatórios Técnicos",

Relatório Técnico, Departamento de Informática, Universidade do Minho, Portugal, 2011

[Analide, 1996] ANALIDE, César, NEVES, José,

"Representação de Informação Incompleta",

Departamento de Informática, Universidade do Minho, Portugal, 2011.