Universidade do Minho

Trabalho Prático 1

Programação em Lógica e Invariantes

Disciplina de Programação em Lógica, Conhecimento e Raciocínio

Mestrado Integrado em Engenharia Biomédica

Ano Letivo de 2018/2019

Docente

Professor César Analide

Realizado por

Ana Duarte, A74407

Ângela Gonçalves, A76542

Diogo Bessa, A75544

Entregue em

24 de outubro de 2018

Sumário

A utilização do software de programação PROLOG permite desenvolver as capacidades de representação do conhecimento e raciocínio segundo o conceito da lógica matemática. O objetivo da elaboração do presente trabalho é, assim, demonstrar a viabilidade da programação lógica para a criação de uma base de conhecimento de um universo na área da prestação de cuidados de saúde. Para tal, é elaborado um programa em PROLOG que permita a manipulação de uma base de conhecimento, que inclui o registo e remoção de utentes, prestadores e cuidados de saúde prestados.

O presente trabalho desenvolveu-se em três etapas principais, onde, na etapa inicial se incluíram nas declarações do povoamento inicial, na segunda etapa procedeu-se à elaboração dos predicados funcionais e auxiliares necessários à gestão da base de conhecimento e na última fase foram construídos os invariantes que definem as regras que a manipulação tem de respeitar para garantir a coerência dos dados.

Do conjunto das simulações realizadas, constatou-se que o software criado resolveu de forma eficaz as diferentes situações analisadas e comprovou a sua adequação para a resolução do tipo de problemas inerentes a uma base de conhecimento.

Palavras-chave: PROLOG, Invariante, Cláusula, Predicado, Lógica, Base de Conhecimento.

Conteúdo

1.Introdução

Os conceitos matemáticos de lógica podem ser aplicados em sistemas de programação que, por sua vez, poderão ser utilizados para a criação e gestão de bases de conhecimento.

Com o presente trabalho pretende-se desenvolver um programa de gestão de bases de conhecimento, para uma instituição de saúde, através da utilização da linguagem de programação em lógica (PROLOG), assim como conhecer as suas potencialidades com o intuito de se construir um sistema de representação de conhecimento e raciocínio. Esta representação será baseada nos seguintes pressupostos:

- **Pressuposto dos Nomes Únicos**: indica que dois nomes distintos correspondem a 2 objetos diferentes. Assim, por exemplo, se dois utentes tivessem o mesmo nome teria de se incluir uma forma de os distinguir para o programa ter um funcionamento correto.
- **Pressuposto do Mundo Fechado:** tudo o que é declarado é tudo o que existe. Assim, só se considera como verdade o que for explicitamente declarado como verdade. O que não tiver sido declarado é considerado não existente e, em consequência, falso.
- **Pressuposto do Domínio Fechado:** os únicos objetos que existem são que estão declarados. Quaisquer outros objetos são considerados não existentes.

O programa deverá permitir a evolução do conhecimento através da inserção e/ou remoção de novos dados e deverá possibilitar uma consulta rápida e eficaz dos registos.

A base de conhecimento desenvolvida foi decomposta em áreas elementares para uma melhor manipulação dos dados, tendo sido consideradas as bases de conhecimento utente, prestador, instituição, especialidade, tipo de cuidado, cuidado prestado, instituição-especialidade e instituição-cuidado.

Estas bases de conhecimentos disporão de elementos-chave que permitam a sua interligação e terão associados invariantes que funcionam como mecanismos de controlo, de forma a terem um funcionamento isento de erros.

Para que esta base de conhecimento fosse passível de construção, foi necessária a criação de predicados lógicos de vários tipos. Assim, e tendo em atenção os pré-requisitos descritos no enunciado muito além da própria caraterização do panorama de conhecimento (povoamento), deve ser assegurada a manutenção e manipulação da respetiva base de conhecimento.

2.Preliminares

A programação em lógica, através da linguagem de programação PROLOG (***PRO****grammation en* ***LOG****ique*), foi criada em 1972 em França, e tem tido uma forte aplicação ao nível da inteligência artificial e de tradutores linguísticos. O PROLOG é uma linguagem de programação para computação simbólica e não numérica, sendo particularmente adequado para resolver problemas que envolvam objetos e relações entre eles. Nesta linguagem, as relações e os objetos iniciam-se com letra minúscula, sendo escritos sob a forma de predicados do tipo relação(argumentos), em que os argumentos representam os objetos.

Ao contrário de outras linguagens, as estruturas de controlo não são realizadas por comandos específicos, como *while* ou *if*, mas sim através de mecanismos lógicos de imposição sobre como o programa deverá ser executado.

O PROLOG é uma linguagem declarativa que fornece uma descrição do problema com recurso a cláusulas que se podem apresentar sob a forma de factos ou de regras.

Uma cláusula termina sempre com o carater "." e é uma forma de se declarar um facto ou regra acerca do problema que se quer resolver. Um exemplo de um facto é declarar uma relação de parentesco entre duas pessoas, em que o João é filho do António e que, em Prolog, seria realizado através da cláusula `filho(joao, antonio).`

Por outro lado, as regras especificam acontecimentos que são verdade se uma determinada condição for satisfeita. A sintaxe de uma regra é definida do seguinte modo e conterá, obrigatoriamente os carateres ":-": `declaracao_1(X) :- declaracao_2(Y).`

Esta regra indica que a declaração_1 é verdade para X, se for verdade para Y. Um exemplo mais intuitivo será definir-se a regra de que o António é pai do João se o João for filho do António. Em PROLOG ficaria: `pai(antonio, joao):-filho(joao, antonio).`

Outra das características deste *software* é a convenção do uso de maiúsculas apenas para as variáveis. Nos exemplos acima, X e Y são variáveis enquanto que *joao* e *antonio* não o são.

Outro modo de se estabelecerem relações é através da conjugação dos operadores lógicos "e" e "ou", que se representam por uma vírgula e por um ponto e vírgula, respetivamente. Por exemplo, poderá definir-se que o Paulo é avô do João se o Paulo for pai do António e se o António for pai do João. Em PROLOG esta regra seria:

```
avo(paulo, joao):-pai(paulo, antonio), pai(antonio, joao).
```

Em PROLOG é ainda possível criarem-se regras recursivas, isto é, regras cuja verificação da condição "se" está ela própria dependente da verificação da declaração inicial. Por exemplo, a verificação de Paulo ser antepassado de X pode ser verificada pela condição de X ser filho de Y e, por sua vez, de Paulo ser antepassado de Y. Assim, dito de outro modo, ter-se-ia a seguinte regra recursiva:

```
antepassado(paulo, X):-pai(X, Y), antepassado(paulo, Y).
```

Para se saber se Paulo é antepassado de João declarava-se `antepassado(paulo, joao).`, obtendo-se a resposta `yes.` Note-se que, em PROLOG, uma resposta de *no* não significa que o facto seja falso, mas antes que não se pode concluir que seja verdadeiro.

Esta forma interrogativa de se declarar designa-se por *query* e representa uma forma de questionar o programa para se apurar se uma determinada afirmação é verdadeira ou falsa.

Para a execução das cláusulas é importante também se descreverem a forma de se representarem em PROLOG alguns dos operadores básicos. A tabela seguinte mostra a algumas operações usuais e a sua correspondência com este *software*.

Operação	Prolog
X = Y (X unifica com Y)	X = Y
X ≠ Y	X \= Y
X < Y	X < Y
X > Y	X > Y
X = Y (literal)	X == Y
X assume o valor de Y	X is Y

Nota: Diz-se que X unifica com Y se X e Y forem iguais ou se um deles for indefinido e passar a assumir o valor do outro.

Muitas vezes para se definir uma funcionalidade em PROLOG são necessárias várias cláusulas, passando esse conjunto de cláusulas a designar-se por *extensão do predicado*. Por exemplo, a extensão do predicado soma_lista pode ser escrita da forma indicada na tabela que se segue.

PROLOG	*Observações*
%Extensão do predicado soma_lista: L, S -> {V,F}	*O carater "%" em Prolog indica que o texto que lhe sucede é um comentário (não influencia a execução do programa).*
soma_lista([ ], 0).	*Como temos a cláusula de baixo é recursiva esta instrução é o chamado critério de paragem. Quando a lista a ser somada estiver vazia assume-se que o resultado da soma dessa lista é zero.*
soma_lista([X|L], S) :-soma_lista(L, S1), S is X + S1.	*Dada uma lista cujo primeiro elemento é X e os restantes elementos são representados por L a soma dessa lista é dada pela*

	soma da lista L mais a soma de X. Como se remete novamente para a soma de uma lista esta cláusula é recursiva e só termina quando a lista estiver vazia e a primeira cláusula for ativada.

No exemplo apresentado, L é um dado de entrada e S um resultado, isto é, L é concretizável e S pode ser concretizável quando se efetua uma *query* ou pode ser uma variável e será um dado de saída.

3.Descrição do Trabalho e Análise de Resultados

3.1. Declarações Iniciais

Em primeiro lugar, antes de se definir a base de conhecimento e as funcionalidades do sistema, é necessária a explicitação de declarações iniciais, que têm por finalidade a aplicação, a todo o programa, de regras gerais que visam obstar a erros e garantir um eficiente funcionamento do sistema.

O primeiro conjunto de declarações tem como intuito a desativação de diversos avisos internos como, por exemplo, os alertas de inserção de predicados de forma não contígua e a existência de variáveis que são utilizadas apenas uma vez. Este primeiro conjunto de declarações apresenta-se do seguinte modo[Carlsson, 2014]:

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

No segundo conjunto de declarações, é preciso declarar *op(900,xfy,'::')*., que representa a indicação da simbologia que será utilizada para a indicação dos invariantes. Além disso, importa ainda definir-se que os predicados são dinâmicos, ou seja, que podem ser alterados durante a execução do programa. Para isso, tendo em conta a base de conhecimento em estudo, são efetuadas declarações do tipo *dynamic p/n*, em que *p* representa o predicado e *n* o número de argumentos que o compõe [Shoham, 1994]:

```
:- op(900,xfy,'::').
:- dynamic utente/5.
:- dynamic prestador/5.
:- dynamic instituicao/1.
:- dynamic especialidade/1.
:- dynamic tipo_cuidado/1.
:- dynamic cuidado/5.
:- dynamic instituicao_especialidade/2.
:- dynamic instituição_cuidado/2.
```

Note-se ainda que a inclusão do número de elementos pelos quais é composto cada predicado é importante para se prevenirem eventuais erros de escrita, evitando-se, assim, um consequente mau funcionamento do programa.

3.2. Base de Conhecimento

O desenvolvimento do sistema de representação de conhecimento e raciocínio que se pretende implementar, capaz de caracterizar um universo na área da prestação de cuidados de saúde, requer o estabelecimento prévio do conhecimento existente. Desta forma, é necessário definir-se a base de conhecimento do sistema, ou seja, indicar-se toda

a informação que é conhecida e relativa a cada uma das estruturas que armazenam os dados.

Para a configuração da base de conhecimento do programa, optou-se por uma disposição interligada entre os predicados, tal como ilustra a Figura 1. Esta interligação entre os pares de predicados que se relacionam entre si é efetuada através de uma chave comum a cada par, que se encontra também assinalada na figura.

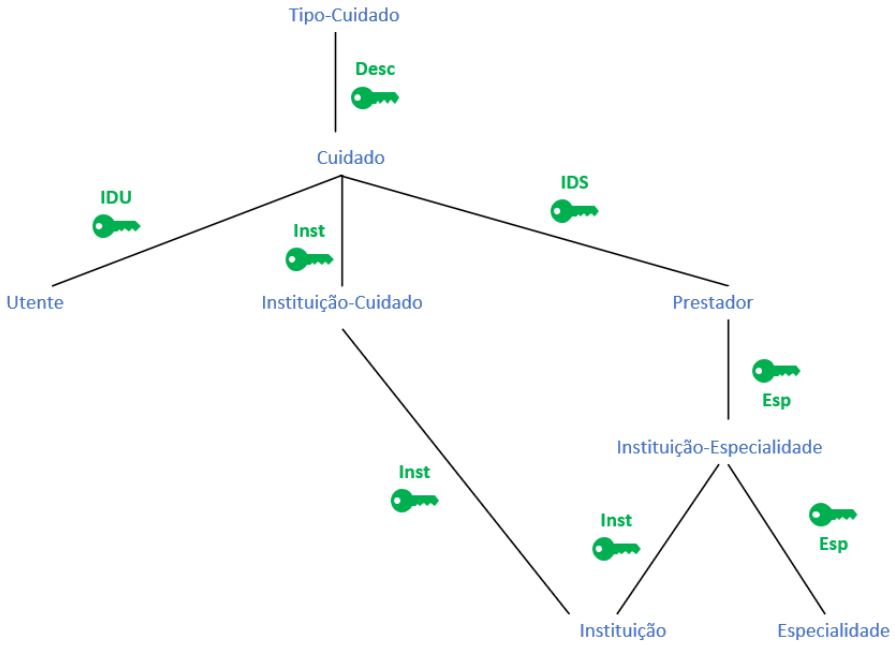


Figura 1 *- Esquema representativo da Base de Conhecimento e dos seus relacionamentos.*

Assim, na Figura 1 apresenta-se, esquematicamente, um mapeamento genérico que relaciona as diferentes estruturas e de onde se pressupõe a necessidade de criação dos predicados *Utente, Tipo-Cuidado, Cuidado, Instituição, Prestador, Especialidade, Instituição-Especialidade e Instituição-Cuidado*.

3.2.1. Predicado *Utente*

O predicado *Utente* é um dos predicados básicos da base de conhecimento e nele registam-se os dados relativos a cada um dos utentes. Esta informação a inserir é composta por uma chave única que o identifica (IdU) e pelo seu nome, idade, género e morada. A extensão deste predicado é do tipo:

```
%Extensão do predicado Utente: #idUt, Nome, Idade, Genero, Morada -> {V, F}
```

A título de exemplo, o comando para se declarar a utente Rita Pinto, cujo ID é u1, e que é do sexo feminino, tem 41 anos e vive na Rua do Sol (Braga), é o seguinte:

```
utente(u1, rita_pinto, 41, feminino, braga).
```

3.2.2. Predicado *Prestador*

Uma vez definidas as especialidades e as instituições, é fundamental caracterizarem-se os prestadores, registando-se, para cada um deles, a chave única que permite a sua identificação, assim como o seu nome, especialidade, instituição e tipo (médico, enfermeiro ou técnico).

Note-se que um prestador apenas pode prestar uma única especialidade e estar associado a uma única instituição e que, quer a especialidade quer a instituição, têm de existir na base de conhecimento respetiva. Deste modo, este predicado é declarado através de uma estrutura do tipo:

```
%Extensão do predicado Prestador: #idPrest, Nome, Especialidade, Instituição, Tipo -> {V, F}
```

Como exemplo para se registar o médico otorrinolaringologista Manuel Sousa, pode proceder-se do seguinte modo:

```
prestador(p2, manuel_sousa, otorrinolaringologia, hospital_braga, medico).
```

3.2.3. Predicado *Instituição*

É ainda necessário efetuar-se o registo de todas as instituições que pertencem à base de conhecimento. Desta forma, o predicado *Instituição* é representado por:

```
%Extensão do predicado Instituição: Designacao -> {V, F}
```

Assim, para se inserir na base de conhecimento o Hospital da Luz, a declaração deve ser do tipo:

```
instituicao(hospital_luz).
```

3.2.4. Predicado *Especialidade*

O predicado *Especialidade* é também um dos predicados básicos do sistema, onde se enumeram todas as especialidades existentes e foi construído da forma que se representa de seguida:

```
%Extensão do predicado Especialidade: Designacao -> {V, F}
```

Assim, para se introduzir, por exemplo, a especialidade de Medicina Dentária na base de conhecimento, a cláusula a inserir é:

```
especialidade(medicina_dentaria).
```

3.2.5. Predicado *Tipo_Cuidado*

O registo dos diferentes tipos de cuidados que podem ser prestados é efetuado através do seguinte predicado:

%Extensão do predicado Tipo-Cuidado: Designação -> {V, F}

Assim, para se inserir, por exemplo, na base de conhecimento o tipo de cuidado "tomografia", recorre-se à seguinte instrução:

tipo_cuidado(tomografia).

3.2.6. Predicado *Cuidado*

O predicado *Cuidado* deve reunir todos os cuidados que foram prestados aos utentes. Desta forma, este predicado é constituído pelo registo da data, do id do utente atendido, do id do prestador do cuidado médico, da descrição do tipo cuidado e do custo suportado pelo utente. A estrutura de inserção dos parâmetros iniciais é do tipo:

%Extensão do predicado Cuidado-Prestado: Data, #idUt, #idP, Descricao, Custo -> {V, F}

De uma forma exemplificativa, para se declarar que no dia 20 de maio de 2018 foi prestado um cuidado pelo prestador p1 ao utente u1, do tipo raio-x, que teve um custo de 30, a cláusula a introduzir é:

cuidado(200518, u1, p1, raio_x, 30).

Importa ainda referir que os serviços rinectomia e densitometria óssea, devem ser prestados apenas por hospitais em que existam prestadores das especialidades de otorrinolaringologia e ortopedia, respetivamente.

3.2.7. Predicado *Instituição-Especialidade*

Uma determinada especialidade apenas é apta para prestar determinadas especialidades. Desta forma, torna-se necessária a criação de um predicado *Instituição-Especialidade*, que relacione cada instituição às especialidades que pode prestar. A estrutura deste predicado é do tipo:

%Extensão do predicado instituição_especialidade: Inst, Esp -> {V, F}

A forma de se colocar os dados iniciais pode ser exemplificada do seguinte modo:

instituicao_especialidade(hospital_braga, medicina_geral).

3.2.8. Predicado *Instituição_Cuidado*

Para que um dado hospital possa prestar um determinado cuidado, é requerido que possua as instalações e os equipamentos necessários para a sua realização. De modo a relacionarem-se os cuidados que cada instituição é capaz de prestar, é criado o predicado *Instituição-Cuidado*. A estrutura deste predicado é do tipo:

%Extensão do predicado Instituição-Cuidado: Inst, Desc -> {V, F}

A forma de se colocar os dados iniciais pode ser exemplificada do seguinte modo:

instituicao_cuidado(hospital_braga, raio_x).

3.2.9. Povoamento Inicial

Para o desenvolvimento da base de conhecimento foi necessária a simulação de dados iniciais para cada um dos predicados. Estes dados, que representam o povoamento inicial, foram inseridos através de um editor de texto para serem posteriormente analisados e manipulados pelo *software* SICStus PROLOG.

Para uma melhor compreensão das relações entre os pares de predicados *Prestador* e *Instituição*, *Prestador* e *Especialidade*, *Cuidado* e *Especialidade e Instituição* e *Especialidade*, elaborou-se uma tabela (Tabela 1) que resume e compila os dados inseridos no povoamento inicial.

Tabela 2 – Correspondência entre os predicados que se encontram associados na base de conhecimento

		Instituição				Especialidade								
		H1 - Hospital de Braga	H2 - Hospital Priv Braga	H3 - Hospital de Santa Luzia	H4 - Hospital da Luz	E1 - Medicina Dentária	E2 - Medicina Geral	E3 - Cardiologia	E4 - Neurologia	E5 - Ortopedia	E6 - Estomatologia	E7 - Oncologia	E8 - Otorrinolaringologia	E9 - Nutrição
Prestador	Médico - P1	✓						✓						
	Médico - P2	✓											✓	
	Técnico - P3				✓	✓								
	Médico - P4	✓									✓			
	Enfermeiro - P5			✓				✓						
	Técnico - P6	✓					✓							
	Médico - P7				✓				✓					
	Técnico - P8		✓									✓		
	Enfermeiro - P9			✓						✓				
	Técnico - P10	✓						✓						
	Enfermeiro - P11	✓											✓	
	Médico - P12				✓									✓
	Técnico - P13		✓							✓				
	Médico - P14		✓							✓				
Cuidado	Raio X - C1	✓	✓	✓	✓									
	Análises Clinicas - C2	✓												
	Tomografia - C3	✓			✓									
	Ultrassonografia - C4	✓												
	Rinectomia - C5	✓				← Apenas Otorrinolaringologia								
	Fisioterapia - C6				✓									
	Densitometria Óssea - C7		✓			← Apenas Ortopedia								
Especialidade	Medicina Dentária - E1	✓	✓	✓	✓									
	Medicina Geral - E2	✓	✓	✓	✓									
	Cardiologia - E3	✓		✓										
	Neurologia - E4			✓	✓									
	Ortopedia - E5		✓	✓										
	Estomatologia - E6	✓												
	Oncologia - E7	✓	✓	✓										
	Otorrinolaringologia - E8	✓												
	Nutrição - E9				✓									

3.3. Predicados Auxiliares

A construção dos predicados funcionais necessários para a implementação da base de conhecimento requer a definição prévia de predicados auxiliares. Estes predicados têm como principal função simplificar as cláusulas dos predicados principais e dos invariantes, tendo em conta a necessidade de se estruturar o programa de uma forma simples e que permita a sua rápida verificação. Considerando-se a projeção pretendida para a base de conhecimento em estudo, os predicados auxiliares que servem de base à sua construção são: *Comprimento*, *Máximo*, *Soluções*, *Não*, *Pertence*, *Sem Repetidos*, *Apagar Tudo*, *Apagar Tudo Lista*, *Soma Lista*, *Inserir*, *Remover e Testar*.

3.3.1. Comprimento

Quando se dispõe de listas, uma das funções auxiliares é a que calcula o seu tamanho, ou seja, a que indica o número de elementos pelos quais é constituída. O predicado *Comprimento* foi estruturado do seguinte modo:

```
%Extensão do predicado Comprimento: Lista, Tamanho -> {V, F}

comprimento([ ], 0).

comprimento([X|L], N) :- comprimento(L,N1), N is N1+1.
```

Justificação: Na primeira cláusula define-se o critério de paragem "comprimento ([],0).". O objetivo deste critério é o de parar a execução da segunda cláusula, que é recursiva. Assim, quando o primeiro elemento do predicado Comprimento é um conjunto vazio, o programa termina a sua execução. A segunda cláusula "comprimento([X|L],N) :- comprimento(L,N1), N is N1+1." calcula o comprimento de uma lista cujo primeiro elemento é X e os restantes formam uma lista L, e adiciona uma unidade ao comprimento dessa lista L. Deste modo, trata-se de uma cláusula recursiva, que vai retirando um elemento à lista em cada iteração e, ao mesmo tempo, adiciona uma unidade ao comprimento. Quando se atinge o critério de paragem, ou seja, quando se decompõe a lista até que não tenha elementos, o valor devolvido N1 corresponde ao seu número de elementos.

Exemplo: Dada uma lista L, por exemplo L= [1, 4, 8, 1, 3, 2], o predicado Comprimento calcula o seu tamanho e devolve o valor 6.

3.3.2. Máximo

Quando se pretende determinar o elemento que corresponde ao valor máximo de uma dada lista, recorre-se a um predicado do tipo:

```
%Extensão do predicado Máximo: Lista, Máximo -> {V, F}

maximo([X], X).

maximo([X|L], X) :- maximo(L, M), (X>M).
```

```
maximo(X|L], M) :- maximo(L, M), (X=<M).
```

Justificação: Como as segunda e terceira cláusulas são recursivas, a primeira cláusula define o critério de paragem. Este critério indica que, quando a lista for composta por apenas um elemento X, o valor retornado (o valor máximo de uma lista de um elemento) é o de X. A segunda cláusula é processada quando o valor do primeiro elemento da lista for superior ao valor máximo existente. Neste caso, o máximo passa a ser o valor desse primeiro elemento e, de forma recursiva, continua-se com o cálculo do máximo para os restantes elementos da lista. A terceira cláusula é ativada quando o primeiro elemento da lista não é superior ao máximo. Nesta situação, o valor máximo mantém-se e, recursivamente, calcula-se o máximo para o resto da lista.

Exemplo: Dada uma lista L, por exemplo L= [7, 15, 32, 6, 12], o predicado Máximo calcula o elemento de maior valor e devolve 32.

3.3.3. Soluções

Outro dos predicados auxiliares, quando se trabalha com listas, é o predicado *Soluções*. Este predicado requer a introdução de um elemento X sobre o qual incide a pesquisa. Deste modo, é percorrida uma determinada lista e são retornados todos os elementos X contidos nessa lista. Para a sua definição, utiliza-se o predicado *findall*, tal como se apresenta de seguida:

```
%Extensão do predicado Soluções: Termo, Objetivo, Lista -> {V, F}

solucoes(X, T, S) :- findall(X, T, S).
```

Justificação: Este predicado utiliza o findall para encontrar todos os elementos X que pertençam a T, colocando-os na lista S. O findall é um predicado que tem a sintaxe findall(termo, objetivo, lista), em que termo representa a expressão a ser procurada, objetivo refere-se ao local onde a pesquisa é efetuada e lista corresponde à lista que contém os termos encontrados. Este predicado nunca falha, ou seja, caso não encontre nenhum X em T, o valor de S é o conjunto vazio [ ]. Além disso, importa ainda notar que a lista S não é apresentada de forma ordenada e que pode conter elementos repetidos.

Exemplo: Definindo-se T, em Prolog, do seguinte modo "T(3, azul).", "T(5, azul).", "T(3, verde)." e "T(3, azul).", ao efetuar-se solucoes(X, T(3,X), S). a lista S retornará: S=[azul, verde, azul].

3.3.4. Não

A negação é um dos constituintes mais elementares e fundamentais para a execução dos predicados funcionais e dos invariantes. Este predicado altera o valor de uma dada regra de verdadeiro para falso, ou vice-versa, sendo organizado pelo seguinte conjunto de cláusulas[Metakides, 1996]:

Extensão do predicado Não: Q -> {V, F}

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

nao(Q).

Justificação: Na primeira cláusula deste predicado, quando existe prova de que Q existe, é combinado o procedimento do cut (!) e do fail, de forma a retornar "insucesso". O insucesso, neste caso, representa o contrário (não) da afirmação inicial Q. Quando não existe prova de que Q existe, afirma-se que Q não existe, tal como representado na segunda cláusula. Esta segunda cláusula baseia-se no Pressuposto do Mundo Fechado, onde a não existência de prova de Q pressupõe a não existência de Q.

Exemplo: Existindo um Q que seja verdade, o predicado não(Q) assume o valor de falso e vice-versa.

3.3.5. Pertence

Quando é necessário saber se um dado elemento é parte integrante de uma lista, recorre-se ao predicado *Pertence*. Este predicado requer a introdução do elemento X a procurar e da lista L, onde será efetuada a pesquisa, e devolve *yes* ou *no*, consoante o elemento X exista ou não, respetivamente, na lista L. As instruções que compõem este predicado são as que se apresentam de seguida:

Extensão do predicado Pertence: X, Lista -> {V, F}

pertence(X, [X|L]).

pertence(X, [Y|L]) :- pertence(X,L).

Justificação: A primeira cláusula deste predicado é "pertence(X, [X|L]).", que assume o valor de verdade caso se pesquise por um termo X que existe numa lista cujo primeiro elemento é X. Assim, se a lista começa por X, então X pertence a essa lista. Por outro lado, caso se pesquise por X numa lista que não é iniciada por esse elemento, é executada a cláusula "pertence(X,[Y|L]) :- pertence(X,L).". Esta segunda cláusula irá, de forma recursiva, eliminar o primeiro elemento da lista e verificar se X pertence à restante lista L. Este procedimento será executado assim, sucessivamente, até o primeiro valor de uma lista ser X ou até não existirem mais elementos. No primeiro caso, conclui-se que X pertence a L e, na segunda situação, o valor do predicado é falso.

Exemplo: Se L=[1, 2, 3, 4, 5] e fazendo-se pertence(6, L)., o programa devolverá o valor "no".

3.3.6. Sem Repetidos

Quando se manipulam listas torna-se, muitas vezes, necessária a supressão dos elementos repetidos que nelas existem. O predicado *Sem Repetidos* remove os elementos duplicados e devolve uma lista composta por todos os elementos diferentes que existem na lista original. A forma de se efetuar este procedimento é apresentada de seguida:

```
%Extensão do predicado Sem Repetidos: Lista Original, Lista Sem Repetidos -> {V, F}

sem_repetidos([X], [X]).

sem_repetidos([X|L], S) :- pertence(X, L), sem_repetidos(L, S).

sem_repetidos([X|L], [X|S]) :- nao(pertence(X, L)), sem_repetidos(L, S).
```

Justificação: *A primeira cláusula corresponde ao critério de paragem, em que a expressão "sem_repetidos([X], [X])." define que, caso exista apenas um elemento numa lista, é devolvida uma lista apenas com esse próprio elemento. Na segunda e terceira cláusulas, verifica-se se o primeiro elemento X da lista pertence ou não ao conjunto L que contém os restantes elementos, respetivamente. Caso este elemento também esteja contido no conjunto L, é descartado e invoca-se novamente o predicado Sem Repetidos, tendo como ponto de partida esse conjunto L. Caso X não pertença ao conjunto L, o resultado será uma lista começada por X e, de forma análoga à cláusula anterior, invoca-se novamente o predicado, tendo como ponto de partida o conjunto L. Estas duas últimas cláusulas são efetuadas sucessivamente até ao critério de paragem e o resultado é expresso na forma de uma lista S.*

Exemplo: *Tendo-se L=[5, 2, 1, 3, 1, 4, 4] e efetuando-se sem_repetidos(L, S). o programa devolverá S=[5, 2, 1, 3, 4].*

3.3.7. Apagar Tudo

Existem situações em que é preciso retirarem-se determinados elementos a uma lista. O predicado *Apagar Tudo* elimina todas as ocorrências de um elemento X que se verificam numa lista L. Uma das formas de poder ser declarado é a que se indica de seguida:

```
%Extensão do predicado Apagar Tudo: X, Lista, Solução -> {V, F}

apagar_tudo(X, [ ], [ ]).

apagar_tudo(X, [X|L], S) :- apagar_tudo(X, L, S).

apagar_tudo(X, [Y|L], [Y|L1]) :- Y\=X, apagar_tudo(X, L, L1).
```

Justificação: *Como critério de paragem deste predicado, definiu-se que o resultado de eliminar um elemento X a uma lista vazia resulta numa lista também vazia. A segunda cláusula aplica-se quando a lista é encabeçada pelo elemento X que se pretende apagar e, neste caso, a lista S será obtida a partir da aplicação recursiva da eliminação de X a uma lista menor em que é retirado o primeiro elemento X. Quando o primeiro elemento da lista não é X, verificam-se as condições de aplicabilidade da terceira cláusula. O seu funcionamento é similar ao da cláusula anterior, sendo que a única diferença reside no facto de o primeiro elemento da lista (que, nesta situação, é diferente de X) não ser eliminado da lista final.*

Exemplo: Supondo L=[5, 3, 4, 1, 3, 1, 8, 4] e fazendo-se apagar_tudo(3, L, S)., o programa devolverá S=[5, 4, 1, 1, 8, 4].

3.3.8. Apagar Tudo Lista

Em problemas mais complexos, poderá ainda surgir a necessidade de se apagar mais do que um determinado elemento a uma lista. No fundo, trata-se de retirar um conjunto de elementos, dispostos na forma de uma lista L1, a uma Lista L2. Um dos modos possíveis de se implementar esta funcionalidade é:

```
Extensão do predicado Apagar Tudo Lista: L1, L2, R -> {V, F}

apagar_tudo_lista ([ ], L, L).

apagar_tudo_lista ([X|Y], L, R) :- apagar_tudo(X, L, S), apagar_tudo_lista (Y, S, R).
```

Justificação: O critério de paragem ocorre quando se retira a uma lista L o conjunto vazio, o que resulta na própria lista L. A segunda cláusula deste predicado seleciona o primeiro elemento (X) do conjunto de elementos a retirar e utiliza o predicado Apagar Tudo para eliminar X da lista L. De forma recursiva, na etapa seguinte será executado o mesmo procedimento, agora com o conjunto de elementos a retirar reduzido do elemento X.

Exemplo: Supondo L1 = [1, 2, 5, 3], L2 = [8, 5, 6, 2, 1, 6, 9] e fazendo-se apagar_tudo_lista (L1, L2, S)., o programa retornará S=[8, 6, 6, 9].

3.3.9. Soma de Lista

Algumas operações matemáticas são importantes no processamento de listas numéricas. No caso da soma, para se efetuar esta operação nos valores numéricos contidos numa determinada lista, é preciso construir o predicado do seguinte modo:

```
%Extensão do predicado Soma Lista: Lista, Soma -> {V, F}

soma_lista([ ], 0).

soma_lista([X|L], T) :- soma_lista(L, S), T is X + S.
```

Justificação: A cláusula soma_lista([ ], 0). define o critério de paragem e considera que a soma de uma lista vazia é zero. A segunda expressão, soma_lista([X|L], T) :- soma_lista(L, S)., T is X + S., adiciona o valor do primeiro elemento X de uma lista a uma variável S e efetua recursivamente o mesmo procedimento para o restante conjunto de elementos L, até ser atingido o critério de paragem. No final, o valor de S corresponderá à soma total dos valores dos elementos contidos na lista.

Exemplo: Dada a lista L= [3, 7, 8, 5], o predicado soma_lista(L, S). devolve o valor S=23.

3.3.10. Inserir

Uma das operações básicas efetuadas em bases de conhecimento é a possibilidade de se adicionarem novos registos, possibilitando a evolução do conhecimento nelas contido. Para auxiliar nessa função, é criado o predicado *Inserir*, que utiliza o meta-predicado *assert*, composto por um único argumento (tanto pode ser um facto ou uma regra). Desta forma, caso sejam descobertas novas regras ou novos factos, eles podem ser inseridos na base de conhecimento, tornando-a dinâmica. O predicado *Inserir* apresenta o seguinte esqueleto[Trappl, 1986]:

```
Extensão do predicado inserir: T -> {V, F}

inserir(T) :- assert(T).

inserir(T) :- retract(T), !, fail.
```

Justificação: De cada vez que o termo T é armazenado na base de dados, uma alternativa na árvore de prova que é capaz de retirar T é deixada em aberto. Esta alternativa será selecionada em caso de falha na inserção.

3.3.11. Remoção

Outra das operações básicas efetuadas envolve a remoção de elementos existentes, que é necessária para a implementação de bases de conhecimento dinâmicas. O predicado *Remoção* foi elaborado nos seguintes termos[Trappl, 1986]:

```
%Extensão do predicado Remoção: T -> {V, F}

remocao(T) :- retract(T).

remocao(T) :- assert(T), !, fail.
```

Justificação: Este é um predicado idêntico ao anterior e onde o termo T é removido da base de conhecimento e, em simultâneo é deixada em aberto uma alternativa na árvore de prova que é capaz de voltar a armazenar T. Esta alternativa será selecionada em caso de falha na remoção.

3.3.12. Testar

Durante a execução de invariantes deverá ser sempre efetuado um teste para averiguar se é possível a inserção ou remoção de um determinado facto ou regra, tendo em conta as restrições dos invariantes a que lhes está associado. Uma das possíveis formas de se efetuar esse teste é através do estabelecimento das seguintes cláusulas:

```
Extensão do predicado teste: ListaInvariantes -> {V, F}

testar([ ]).
```

```
testar([R|LR]) :- R, testar(LR).
```

Justificação: Neste predicado percorre-se a lista que contém as restrições associadas aos invariantes num determinado contexto. Deste modo, é feito o teste a cada elemento contido nessa lista, de uma forma recursiva, até se atingir o critério de paragem, que ocorre quando se efetua testa uma lista vazia.

3.4. Predicados Funcionais

A gestão de uma base de conhecimento pressupõe ainda a execução de um conjunto de funcionalidades que permita a obtenção do conhecimento de acordo com os critérios pretendidos. No caso em estudo, considerou-se que o sistema deveria ter uma capacidade de resposta mínima às seguintes funcionalidades:

- Registar utentes, instituições, especialidades, tipo de cuidados, prestadores, cuidados de saúde e especialidades e tipo de cuidados associados a uma determinada instituição;
- Remover utentes, instituições, especialidades, tipo de cuidados, prestadores, cuidados de saúde e especialidades e tipo de cuidados associados a uma determinada instituição;
- Identificar utentes por critério de seleção;
- Identificar as instituições prestadoras de cuidados de saúde;
- Identificar os cuidados de saúde prestados por instituição/morada/datas;
- Identificar os utentes de um prestador/especialidade/instituição;
- Determinar todas as instituições/prestadores a que um utente já recorreu;
- Calcular o custo total dos cuidados de saúde por utente/especialidade/prestador/datas.

3.4.1. Registar na Base de Conhecimento

Esta funcionalidade possibilita a adição de novos registos à base de conhecimento, sejam eles relativos a utentes, prestadores e cuidados de saúde, ou referentes às instituições, especialidades, tipos de cuidados assim como às especialidades e tipo de cuidados associados a uma dada instituição. Deste modo, a inserção de novos dados é permitida em todos os predicados existentes na base de conhecimento. Para a implementação desta funcionalidade recorre-se à seguinte sintaxe programática:

```
%Extensão do predicado Registar: T -> {V, F}

registar(T) :- solucoes(I, +T::I, L), inserir(T), testar(L).
```

3.4.2. Remover da Base de Conhecimento

O predicado associado à remoção de registos permite que seja eliminada informação existente e pode aplicar-se a qualquer predicado da base de conhecimento. As instruções que permitem esta remoção de registos são:

```
%Extensão do predicado Remover: T -> {V, F}

remover(T) :- T, solucoes(I, -T::I, L), remocao(T), testar(L).
```

3.4.3. Identificar Utentes por Critério de seleção

Com esta funcionalidade possibilita-se a identificação de utentes através da inserção de um dos critérios de pesquisa: id, nome, idade, género e morada. As regras que permitem a pesquisa de utentes por critério de seleção são as que se apresentam:

```
%Extensão do predicado Cs_Utente: CS, X, Solução -> {V, F}

cs_utente(CS, IdU, S) :- CS=id, solucoes((IdU, N, I, G, M), utente(IdU, N, I, G, M), S).

cs_utente(CS, N, S) :- CS=nome, solucoes((IdU, N, I, G, M), utente(IdU, N, I, G, M), S).

cs_utente(CS, I, S):-CS=idade, solucoes((IdU, N, I, G, M), utente(IdU, N, I, G, M), S).

cs_utente(CS, G, S):-CS=genero, solucoes((IdU, N, I, G, M), utente(IdU, N, I, G, M), S).

cs_utente(CS, M, S):-CS=morada, solucoes((IdU, N, I, G, M), utente(IdU, N, I, G, M), S).
```

Justificação: Para a aplicação desta funcionalidade definiu-se uma variável CS que se refere ao critério que se pretende pesquisar. O predicado CS=id (ou outro) impõe que só é considerada essa cláusula no caso de ter sido definido o respetivo valor de CS. No caso de o critério de seleção ser o id, a pesquisa do utente faz-se através do predicado soluções((IdU, N, I, G, M), utente(IdU, N, I, G, M), S)., em que se pesquisam todos os elementos IdU, N, I, G, M da base de conhecimento Utente, considerando que a variável IdU tem de ser igual à introduzida inicialmente pelo utilizador. O resultado é expresso através da lista S.

Nota: Caso se pretenda pesquisar um utente através do seu id, por exemplo, para procurar-se o utente u2, deverá colocar-se cs_utente(id, u2, S), obtendo-se o resultado (u2, carlos_moreira, 12, masculino, vilareal). Note-se ainda que, no caso de o critério de seleção escolhido ser outro que não o id, como a idade, seria possível obter-se mais do que um utente.

3.4.4. Identificar as Instituições Prestadoras de Cuidados de Saúde

Através desta funcionalidade pretende-se identificar as instituições que forneceram pelo menos um cuidado de saúde. A estrutura do predicado que permite esta pesquisa é:

```
%Extensão do predicado Inst_Cuidados: Solução -> {V, F}

inst_cuidados(S) :- solucoes((Inst), (cuidado(_, _, IdP, _, _), prestador(IdP, _, _, Inst, _)), L),
sem_repetidos(L, S).
```

Justificação: O predicado Inst_Cuidados coloca na lista S os elementos que simultaneamente respeitam as expressões solucoes((Inst), (cuidado(_, _, IdP, _), prestador(IdP, _, _, Inst, _)), L). e sem_repetidos(L,S). A primeira expressão remete para o predicado auxiliar Soluções, onde se relacionam as bases de conhecimento Cuidado e Prestador pelo IdP, de forma a terem-se os prestadores que prestaram cuidados de saúde. Uma vez conhecidos estes prestadores e como se pretendem conhecer as instituições que já prestaram cuidados, basta associar, em simultâneo, o IdP a uma variável Inst no predicado Prestador e, do Predicado Soluções, retiram-se todas estas variáveis Inst na forma de uma lista L. No entanto, é preciso ter em atenção que a lista L pode conter instituições repetidas, quer porque o mesmo prestador pode prestar vários cuidados de saúde, quer porque prestadores diferentes podem estar vinculados a uma mesma instituição. Desta forma, o predicado Sem Repetidos permite que se eliminem todas as instituições que aparecem de forma repetida na lista L.

Exemplo: Efetuando-se inst_cuidados(S), obtém-se S = [hospital_sta_luzia, hospital_luz, hospital_priv_braga), hospital_braga].

3.4.5. Identificar os Cuidados de Saúde Prestados por Instituição/Morada/Data

Caso se pretenda retirar quais são os cuidados de saúde prestados por instituição, por morada ou por datas, poderá recorrer-se a um conjunto de instruções, tal como as que se apresentam de seguida, em que o filtro F identifica o tipo de pesquisa pretendida (por instituição, por morada ou por data).

```
%Extensão do predicado Filtro_Cuidado: Filtro, X, Solução -> {V, F}

filtro_cuidado(F, Inst, S) :- F=instituicao, solucoes(Desc, (cuidado(_, _, IdP, Desc, _),
        prestador(IdP, _, _, Inst, _)), L), sem_repetidos(L,S).

filtro_cuidado(F, Mor, S) :-F =morada, solucoes(Desc, (cuidado(_, IdU, _, Desc, _),
        utente(IdU, _, _, _, Mor)), L), sem_repetidos(L,S).

filtro_cuidado(F, Data, S) :- F=data, solucoes(Desc, cuidado(Data, _, _, Desc, _), L),
        sem_repetidos(L,S).
```

Justificação: O predicado Filtro Cuidado requer a introdução de um filtro como argumento inicial (instituição, morada ou data) de modo a selecionar os cuidados prestados de acordo com a categoria pretendida. A título de exemplo, caso se pretendam os cuidados prestados por instituição, a variável F deverá ser substituída por "instituição" e o segundo argumento do predicado deverá conter a indicação da instituição pela qual se deseja aplicar o filtro (por exemplo, hospital_braga). Desta forma, através do predicado Soluções, podem relacionar-se os predicados Cuidado e Prestador, através do IdP, e fixando-se a Inst introduzida pelo utilizador, retirar-se as Desc relativas a essa instituição. Mais uma vez, é usado o predicado Sem Repetidos para eliminar os cuidados duplicados. No final, a lista S conterá o conjunto de cuidados que a instituição pretendida já prestou. Para a

pesquisa por morada ou data é efetuado um procedimento similar. Note-se que, no caso da data, não é necessária a interligação de duas bases de conhecimento através de uma chave comum.

Exemplo: Inserindo-se filtro_cuidado(morada, braga, S)., obtém-se a lista S = [raio_x, tomografia, rinectomia, analises_clinicas, ultrassonografia].

3.4.6. Identificar os Utentes de um Prestador/Especialidade/Instituição

Esta funcionalidade pressupõe que seja fixado um prestador, uma especialidade ou uma instituição, para se proceder à pesquisa de todos os utentes que já recorreram a esse prestador/especialidade/instituição, respetivamente. De uma forma análoga ao predicado *Filtro Cuidado* é requerido que o primeiro argumento seja o tipo de filtro pretendido (prestador, especialidade ou instituição) e o segundo seja o nome do prestador/especialidade/instituição que se pretende fixar. O resultado final corresponderá a uma lista com todos os utentes que verifiquem a condição de procura, ou seja, com os utentes que já recorreram a um determinado prestador/especialidade/instituição. Este predicado *Filtro Utente* pode ser definido do seguinte modo:

```
%Extensão do predicado Filtro Utente: Filtro, X, Solução -> {V, F}

filtro_utente(F, P, S) :- F=prestador, solucoes((IdU, N, I, G, M), (cuidado(_, IdU, P, _, _),
        utente(IdU, N, I, G, M)), L), sem_repetidos(L,S).

filtro_utente(F, E, S) :- F=especialidade, solucoes((IdU, N, I, G, M),
        (cuidado(_, IdU, IdP, _, _), prestador(IdP, _, E, _, _), utente(IdU, N, I, G, M)), L),
        sem_repetidos(L,S).

filtro_utente(F, Inst, S) :- F=instituicao, solucoes((IdU, N, I, G, M), (cuidado(_, IdU, IdP, _,
        _), prestador(IdP, _, _, Inst, _), utente(IdU, N, I, G, M)), L), sem_repetidos(L,S).
```

Justificação: Consoante o filtro aplicado, a pesquisa será efetuada por prestador, especialidade ou instituição. Por exemplo, se F for "especialidade", será utilizado o predicado Soluções para percorrer, em simultâneo, as bases Cuidado, Prestador e Utente e devolver, sob a forma de uma lista L, os registos de utentes (id do utente, nome, idade, género e morada) que verificam a condição de terem tido cuidados da especialidade E indicada. Os IdU e IdP garantem a interligação entre as bases de conhecimento e o predicado Sem Repetidos elimina os registos duplicados, obtendo-se a lista final S.

Exemplo: Caso se se inserisse filtro_utente(instituicao, hospital_sta_luzia, S)., obtinha-se a lista S = [(u1, rita_pinto, 41, feminino, braga), (u4, maria_tavares, 34, feminino, braga), (u7,andre_fernandes, 77, masculino, leiria)].

3.4.7. Determinar Todas as Instituições/Prestadores a que um Utente Já Recorreu

Um dos aspetos também considerados é a possibilidade de se efetuarem consultas às bases de conhecimento, de forma a apurarem-se as instituições ou os prestadores a que um determinado utente (IdU) já recorreu. Para tal, são consideradas as seguintes cláusulas programáticas:

```
%Extensão do predicado Filtro Utente2: Filtro, X, Solução -> {V, F}

filtro_utente2(F, IdU, S) :- F=instituicao, solucoes(Inst, (cuidado(_, IdU, IdP, _, _),
        prestador(IdP, _, _, Inst, _)), L), sem_repetidos(L, S).

filtro_utente2(F, IdU, S) :- F=prestador, solucoes(Nome, (cuidado(_, IdU, IdP, _, _),
        prestador(IdP, Nome, _, _, _)), L), sem_repetidos(L, S).
```

Justificação: O predicado Filtro Utente2 retorna uma lista S com as instituições ou os prestadores que já atenderam um dado utente, consoante F seja "instituição" ou "prestador", respetivamente. No caso de, por exemplo, F ser "prestador", o predicado Soluções encontra os nomes dos prestadores (IdP) que se associam a um dado utente indicado (IdU). Para isso, é necessário fixar-se o IdU introduzido pelo utilizador e considerá-lo no IdU do predicado Cuidado. Posteriormente, para esse id do utente, através da chave IdP associa-se cada cuidado ao respetivo prestador. No final, é obtida uma lista S, que tem de ser previamente tratada através do predicado Sem Repetidos. No caso de F ser "instituição", a pesquisa seria realizada de uma forma análoga, diferindo no facto de se retirar o nome da instituição, ao invés do prestador.

Exemplo: Efetuando-se filtro_utente2(prestador, u3, S)., obtém-se S = [alice_ribeiro, henrique_alves, manuel_sousa, leonor_bastos, beatriz_barbosa].

3.4.8. Calcular o Custo Total dos Cuidados por Utente/Especialidade/Prestador/Data

Outros dos critérios de consulta previstos é o que se refere aos custos. Assim, deve ser criado um predicado que permita efetuar a pesquisa dos custos totais associados aos utentes, às especialidades, aos prestadores e às datas. A estrutura criada para permitir esta pesquisa é:

```
%Extensão do predicado Filtro Custo: Filtro, X, Solução -> {V, F}

filtro_custo(F, IdU, S) :- F=utente, solucoes(Custo, cuidado(_, IdU, _, _, Custo), C),
        soma_lista(C, S).

filtro_custo(F, E, S) :- F=especialidade, solucoes(Custo, (cuidado(_, _, IdP, _, Custo),
        prestador(IdP, _, E, _, _)), C), soma_lista(C, S).

filtro_custo(F, IdP, S) :- F=prestador_id, solucoes(Custo, cuidado(_, _, IdP, _, Custo), C),
        soma_lista(C, S).
```

```
filtro_custo(F, N, S) :- F=prestador_nome, solucoes(Custo, (cuidado(_, _, IdP, _, Custo),
        prestador(IdP, N, _, _, _)), C), soma_lista(C, S).

filtro_custo(F, D, S) :- F=data, solucoes(Custo, cuidado(D, _, _, _, Custo), C), soma_lista(C,
        S).
```

Justificação: O modo de se consultarem os custos foi, de forma idêntica às anteriores, através da especificação do filtro a utilizar (por utente, especialidade ou prestador). No caso do prestador, e para exemplificar as potencialidades, foi permitido identificar o prestador por nome ou por id. Como exemplo, caso F seja "especialidade", o predicado Soluções devolve uma lista C com todos os custos associados a essa especialidade, independentemente do utente ou prestador que a realizou. Para isso, é usada a chave IdP para ligar as bases Cuidado e Prestador. Adicionalmente, é calculada a soma dos elementos que constituem a lista C e apresenta-se o seu resultado através de S.

Exemplo: Efetuando-se filtro_custo(prestador_id, p4, S)., obtém-se S = 35.

3.5. Predicados Adicionais

Além das funcionalidades descritas foram ainda consideradas funcionalidades adicionais que possibilitam uma gestão da base de conhecimento mais versátil e eficaz. Assim, considerou-se que a base de conhecimento deveria dar resposta às seguintes funcionalidades adicionais:

- Identificar os utentes que pagaram mais por um cuidado médico;
- Identificar o género que mais cuidados de uma dada especialidade teve;
- Identificar os prestadores que não prestaram quaisquer cuidados médicos;
- Obter a informação completa e um utente/prestador a partir de um nome;
- Calcular o custo total de uma dada especialidade.

3.5.1. Identificar os Utentes que Pagaram Mais por um Cuidado Médico

A funcionalidade que permite retirar qual ou quais os utentes que pagaram mais, para um determinado cuidado médico que lhes foi prestado, pode ser implementada através da construção de um predicado *Utente Gasto Máx*, da seguinte forma:

```
%Extensão do predicado Utente Gasto Máx: Solução -> {V, F}

utente_gasto_max(S) :- solucoes(Custo, cuidado(_, _, _, _, Custo), L), maximo(L, Max),
        solucoes((IdU, Nome),(cuidado(_,IdU, _, _, Max), utente(IdU, Nome, _, _, _)), R),
        sem_repetidos(R, S).
```

Justificação: Este predicado pesquisa, através do predicado Soluções, o custo de todos os cuidados e devolve uma lista L com os resultados. Em simultâneo, é calculado o valor máximo de L associando-se o valor desse máximo à variável Max. Por outro lado, em paralelo, o predicado Soluções pesquisa os utentes que tenham um custo igual a Max e devolve o seu id e nome numa

lista R, à qual se retiram os registos duplicados pela utilização do predicado Sem Repetidos. O IdU serve para efetuar a ligação entre as bases Cuidado e Utente e o resultado final apresenta-se sob a forma de uma lista S.

Exemplo: Fazendo-se utente_gasto_max(S)., obtém-se S = [(u3, rui_sousa), (u7, andre_fernandes), (u1, rita_pinto), (u6, matilde_neves)].

3.5.2. Identificar o Género Mais Cuidados de uma Dada Especialidade Teve

O predicado *Género Especialidade* é usado quando se quer conhecer qual o género que mais recorreu a uma determinada especialidade. A título de exemplo, pode-se pretender saber se as consultas de ortopedia foram maioritariamente prestadas a indivíduos do sexo feminino ou masculino. Para a determinação dos resultados, a construção do predicado Género Especialidade pode ser definida do seguinte modo:

```
%Extensão do predicado Género Especialidade: Especialidade, Solução -> {V, F}

genero_especialidade(E, S) :- solucoes(_, (cuidado(_, IdU, IdP, _, _),

        prestador(IdP, _, E, _, _), utente(IdU, _, _, G, _), G=masculino), R1),
        comprimento(R1, S1), solucoes(_, (cuidado(_, IdU, IdP, _, _),

        prestador(IdP, _, E, _, _), utente(IdU, _, _, G, _), G=feminino), R2),
        comprimento(R2, S2), S1>S2, S=(masculino, S1).

genero_especialidade(E, S) :- solucoes(_, (cuidado(_, IdU, IdP, _, _),

        prestador(IdP, _, E, _, _), utente(IdU, _, _, G, _), G=masculino), R1),
        comprimento(R1, S1), solucoes(_, (cuidado(_, IdU, IdP, _, _),

        prestador(IdP, _, E, _, _), utente(IdU, _, _, G, _), G=feminino), R2),
        comprimento(R2, S2), S1<S2, S=(feminino, S2).

genero_especialidade(E, S) :- solucoes(_, (cuidado(_, IdU, IdP, _, _),

        prestador(IdP, _, E, _, _), utente(IdU, _, _, G, _), G=masculino), R1),
        comprimento(R1, S1), solucoes(_, (cuidado(_, IdU, IdP, _, _),

        prestador(IdP, _, E, _, _), utente(IdU, _, _, G, _), G=feminino), R2),
        comprimento(R2, S2), S1=S2, S=(igual_prevalencia_de_genero, S2).
```

Justificação: O predicado Género Especialidade utiliza a interseção de duas listas, R1 e R2, que dizem respeito aos cuidados prestados a utentes do sexo masculino e feminino, respetivamente. Para se obterem estas listas, usa-se o predicado Soluções, relacionando-se as bases de conhecimento Cuidado, Prestador e Utente, de forma a terem-se todos os registos de cuidados prestados para todos os utentes. Além disso, impõe-se ainda que, na primeira situação, o género seja "masculino" e, no segundo caso, do tipo "feminino". Desta forma, são obtidas as listas R1 e R2 que têm os registos de todos os homens e mulheres que recorreram a cuidados de saúde, respetivamente. O

comprimento de R1 e R2 é calculado através do predicado comprimento e origina os valores S1 e S2, respetivamente. Note-se que, nesta situação, não se eliminam os registos repetidos porque, mesmo que a um determinado utente tenha sido prestado um cuidado por um mesmo prestador mais do que uma vez, devem-se contabilizar todas essas ocorrências. A primeira cláusula do predicado Género Especialidade é considerada quando o valor de S1 é superior ao de S2 e devolverá S, que contém o género masculino e o valor de S1. De forma similar, a segunda cláusula será tida em conta quando S2>S1 e devolverá o género feminino e o valor de S2, enquanto a terceira cláusula retorna a indicação de que o número de homens e de mulheres que recorreram a essa especialidade é igual, assim como o número de indivíduos de cada género.

Exemplo: Fazendo-se género_especialidade(otorrinolaringologia, S). obtém-se S = (igual_prevalencia_de_genero, 3).

3.5.3. Identificar os Prestadores que Não Prestaram Quaisquer Cuidados Médicos

Para se conhecerem quais os prestadores que não prestaram quaisquer cuidados, pode utilizar-se o predicado *Prestador Zero*.

```
%Extensão do predicado Prestador Zero: Solução -> {V,F}

prestador_zero(S) :- solucoes(Nome, (prestador(IdP, Nome, _, _, _), cuidado(_, _, IdP, _, _)), L), sem_repetidos(L, L1), solucoes(Nome, prestador(IdP, Nome, _, _, _), L2), apaga_lista(L1, L2, S).
```

Justificação: Neste predicado encontram-se inicialmente os nomes de todos os prestadores que prestaram algum cuidado médico e inserem-se esses nomes na lista L. Esta operação é realizada através do predicado Soluções em que o IdP funciona de chave entre as bases Prestador e Cuidado. É preciso notar que os nomes dos prestadores podem aparecer de forma repetida e, por isso, usa-se o predicado Sem Repetidos para eliminar as duplicações, transformando-se L em L1. Em paralelo, também através do predicado Soluções, é construída a lista L2, com todos os nomes dos prestadores contidos na base de conhecimento, tenham eles cuidados médicos associados ou não. Por último, a esta lista L2 retiram-se os elementos de L1, ou seja, retiram-se os prestadores que já realizaram algum cuidado médico, e reduz-se a lista, obtendo-se a lista S. Assim, em S, ficam apenas os prestadores que não realizaram qualquer cuidado médico.

Exemplo: Ao indicar-se prestador_zero(S)., obtém-se S = [ricardo_couto].

3.5.4. Obter Informação Completa de um Utente/Prestador a Partir do Nome

Uma das possibilidades que se também poderá implementar refere-se à possibilidade de, a partir do nome de um utente ou de um prestador, se obter toda a restante informação referente a esse utente ou prestador. Para tal poderá recorrer-se a:

```
%Extensão do predicado informacoes: F, N, S -> {V,F}
```

informacoes(F,N,S):-F=utente, solucoes((IdU, N, I, G, M), utente(IdU, N, I, G, M), S).

informacoes(F,N,S):-F=prestador, solucoes((Id, N, E, I, T), prestador(IdP, N, E, I, T), S).

Justificação: Através deste predicado e a partir de um nome de um utente ou de um prestador é recebida a informação completa desse utente/prestador. Caso se pretenda saber a informação de um utente a primeira cláusula pesquisa através do predicado soluções todos os utentes registados que têm o nome pretendido e insere numa lista S esses registos. Com os nomes são únicos ou a lista S é uma lista vazia ou contém um elemento.

Exemplo: Ao colocar-se informacoes(utente, rita_pinto, S). obtém-se indicar-se prestador_zero(S)., obtém-se S = [(u1, rita_pinto, 41, feminino, braga)].

3.5.5. Calcular o Custo Total de uma dada Especialidade

Para se obter o custo que uma especialidade representa considerando-se todos os cuidados de todas as instituições poderá ser realizado o predicado *Especialidade Custo* do seguinte modo:

%Extensão do predicado especialidade_custo: E, S -> {V,F}

especialidade_custo(E,R):-solucoes(C, (cuidado(_, _, IdP, _, C),

prestador(IdP, _, E, _, _)), S), soma_lista(S,R).

Justificação: Para se obter o custo total dos cuidados de uma determinada especialidade pesquisa-se, através do predicado soluções, os custos dos cuidado de cada um dos IdPs que, por sua vez estão associados à especialidade pretendida e que se encontra descrita na base de conhecimento prestador. Esse IdP faz a correspondência com a especialidade a que se refere o cuidado e insere os resultados numa lista S. Os elementos da lista S são somados e obtém-se o custo total de uma dada especialidade.

Exemplo: Ao solicitar-se especialidade_custo(medicina_geral, S) é devolvido S = 65.

3.6. Invariantes

Os invariantes têm como função garantir a correção e a coerência dos dados, impondo restrições que asseguram que os dados armazenados respeitam os parâmetros de qualidade necessários. Quando é necessário restringir-se a inserção ou remoção de dados em bases de conhecimento, utilizam-se invariantes específicos. Nesta secção, tendo em conta a base de conhecimento em estudo, serão detalhados os invariantes implementados, que validam estas operações.

A estrutura tipo de um invariante é a seguinte:

$$\pm\ C :: R$$

O sinal inicial é "+" quando o invariante se aplicar à inserção de novos dados na base de conhecimento – Invariante de Inserção – e usa-se "-" quando o invariante se refere à remoção de dados existentes – Invariante de Remoção.

A regra R é chamada sempre que se estiver no contexto C, em que o contexto se refere a uma base de conhecimento e a regra é o invariante que garante a consistência dos dados adicionados ou removidos.

É importante ainda notar que os invariantes podem dividir-se em dois tipos distintos: os estruturais – que não permitem a inserção de informação repetida – e os referenciais – que, testam, além da estrutura, o significado da base de conhecimento, garantindo consistência.

3.6.1. Utente

Na manipulação do predicado *Utente* foram adicionados os invariantes de inserção que garantem o cumprimento das seguintes regras:

- Não podem ser registados novos utentes com ids já existentes na base de conhecimento. O id é único e este invariante impede que, acidentalmente, existam dois ids iguais;
- Não podem ser registados novos utentes com nomes já existentes na base de conhecimento. À semelhança do id, o nome é único (de acordo com o Pressuposto dos Nomes Únicos) e não podem existir nomes iguais;
- Não podem ser inseridos utentes com idades negativas ou iguais ou superiores a 130 anos. Este invariante previne erros na introdução da idade, obrigando a que apenas seja possível inserir idades que estejam contidas no intervalo entre 0 e 130 anos.
- Só poderão ser adicionados novos utentes cujo género seja "feminino" ou "masculino". Este invariante tem um papel importante na uniformização da informação, impedindo a colocação de iniciais ambíguas como *h, m, ou f* (em que, por exemplo, a letra m tanto pode representar masculino ou mulher).

Relativamente aos invariantes de remoção associados a este predicado, terão de ser cumpridas as seguintes restrições:

- Não se pode permitir remover utentes que tenham já algum registo associado a um cuidado médico. Este invariante garante a coerência dos dados uma vez que, se se permitisse remover utentes que têm registos de cuidados médicos associados, esses registos ficariam sem a informação do utente que realizou esse cuidado.

A estrutura de cada um dos invariantes referidos e o detalhe do seu funcionamento são apresentados de seguida.

3.6.1.1. Invariantes de Inserção

Invariante 1:

Não permite inserir utentes com ids que já pertençam à base de conhecimento

+utente(IdU, _, _, _, _) :: (solucoes(IdU, utente(IdU, _, _, _, _),S), comprimento(S, N), N==1).

Justificação: No predicado Utente, considera-se o elemento IdU e encontram-se as soluções, na forma de uma lista S, de todos os utentes que contêm esse IdU na base de conhecimento. Simultaneamente impõe-se que S tenha comprimento 1, apenas se permitindo, deste modo, a inserção de IDs únicos. O motivo de o comprimento ser igual a 1 prende-se com o facto de o novo IdU ter sido temporariamente acrescentado à base de conhecimento e, por isso, aparecer na lista S.

Simulação (SICStus Prolog): Para se testar este invariante, inseriram-se duas cláusulas no SICStus Prolog: na primeira simulou-se o registo de um novo utente usando-se um id já existente na base de conhecimento (u7) e na segunda simulou-se esse registo com um novo id (u8).

```
| ?- registar(utente(u7, ana_alves, 15, feminino, braga)).
no
| ?- registar(utente(u8, ana_alves, 15, feminino, braga)).
yes
```

Invariante 2:

Não permite inserir utentes com nomes que já pertençam à base de conhecimento

+utente(_, Nome, _, _, _) :: (solucoes(Nome, utente(_, Nome, _, _, _),S), comprimento(S, N), N==1).

Justificação: O procedimento é o mesmo que o do invariante anterior mas, desta vez, ao invés do id, é garantido que o nome é único.

Invariante 3:

Não permite inserir utentes com idades negativas ou superiores a 130 anos

+utente(_, _, IDD, _, _) :: (IDD>0, IDD<130).

Justificação: No predicado Utente considera-se a variável IDD e aplica-se a restrição de este parâmetro ser obrigatoriamente maior do que 0 e menor do que 130 anos. A vírgula representa o elemento lógico "e". Assim, sempre que se regista um novo utente, associa-se uma idade IDD e a regra do invariante (IDD>0, IDD<130) só assume o valor de verdade se IDD for positivo e menor do que 130, em simultâneo.

Simulação (SICStus Prolog): De forma a validar-se este invariante, simularam-se os registos de um utente com 130 anos e de outro com 19 anos.

```
| ?- registar(utente(u9, ana_alves, 130, feminino, braga)).
no
| ?- registar(utente(u9, ana_alves, 19, feminino, braga)).
yes
```

Invariante 4:

Não permite inserir utentes com género diferente de "feminino" ou de "masculino"

+utente(_, _, _, G, _) :: (G=feminino; G=masculino).

Justificação: No predicado utente considera-se o elemento "género" e aplica-se a restrição que força a que os valores deste campo sejam obrigatoriamente "feminino" ou "masculino". O ponto e vírgula corresponde ao elemento lógico "ou". Desta forma, quando se insere um novo utente, está a associar-se um género GEN e a regra do invariante (GEN=feminino; GEN=masculino) só assume o valor de verdade se o GEN for igual ou a "feminino" ou a "masculino".

Simulação (SICStus Prolog): Testou-se a inserção de duas cláusulas para o registo de um utente: na primeira, o seu género é indicado como "f" e, na segunda, toma o valor "feminino".

```
| ?- registar(utente(u10, ana_alves, 19, f, braga)).
no
| ?- registar(utente(u10, ana_alves, 19, feminino, braga)).
yes
```

3.6.1.2. Invariantes de Remoção

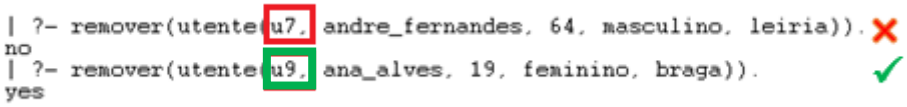
Invariante 1:

Não permite remover utentes que já tenham algum registo em cuidado médico

-utente(IdU, _, _, _, _) :: (solucoes(IdU, cuidado(_, IdU, _, _, _), S), S==[ ]).

Justificação: Considera-se o elemento IdU do predicado "utente" e constrói-se uma lista S que contém a totalidade dos ids do utente, presentes no predicado "cuidado", e que são iguais ao id do utente que se pretende remover. Para que se possa remover esse utente, essa lista S não pode conter quaisquer registos, isto é, S tem de ser uma lista vazia.

Simulação (SICStus Prolog): De modo a testar-se este invariante, foi inserida uma primeira cláusula, com o intuito de se remover um utente com registos em cuidados médicos e uma segunda para remover um utente sem quaisquer registos em cuidados médicos associados.



3.6.2. Prestador

Na base de conhecimento dos prestadores foram também adicionados invariantes de inserção que, por sua vez:

- Impedem a inserção de prestadores com ids já existentes;
- Impedem a inserção de prestadores com nomes já existentes;
- Impossibilitam a inserção de novos prestadores cujo par instituição-especialidade não esteja declarado.

Do ponto de vista das restrições de remoção, foi considerado o invariante que:

- Impede a remoção de prestadores que já tenham cuidados médicos associados.

3.6.2.1. Invariantes de Inserção

Invariante 1:

Não permite inserir prestadores com ids que já pertençam à base de conhecimento

+prestador(IdP, _, _, _, _) :: (solucoes(IdP, prestador(IdP, _, _, _, _), S), comprimento(S,N), N==1).

Justificação: Tal como na base de conhecimento anterior, também nesta terá de se salvaguardar a unicidade do ID do prestador, impedindo que sejam adicionados novos registos com IDs já anteriormente registados. Utiliza-se o predicado auxiliar Soluções para encontrar todos os IDs de prestadores iguais aos que se quer adicionar. O resultado é colocado na lista S e, se o comprimento desta lista for 1, aceita-se o novo registo. O motivo de o comprimento ser igual a 1 tem a ver com o facto de o novo ID ter sido previamente registado e de, por isso, já aparecer na lista S.

Invariante 2:

Não permite inserir prestadores com nomes que já pertençam à base de conhecimento

+prestador(_, Nome, _, _, _) :: (solucoes(Nome, prestador(_, Nome, _, _, _), S), comprimento(S,N), N==1).

Justificação: *O procedimento é o mesmo que o do invariante anterior mas, desta vez, ao invés do id, é garantido que o nome é único.*

Invariante 3:

Não permite inserir prestadores com pares instituição/especialidade que não existam na base de conhecimento

+prestador(_, _, E, Inst, _) :: instituicao_especialidade(Inst, E).

Justificação: *Os novos registos de prestadores apenas são possíveis se o par instituição-especialidade estiver declarado e existir na base de conhecimento. Desta forma, quer o E quer o Inst associados ao novo prestador, têm de corresponder aos argumentos do predicado "instituicao_especialidade". Note-se que, se o par instituição-especialidade estiver declarado na base de conhecimento, quer a instituição, quer a especialidade também existem, isoladamente, nas respetivas bases de conhecimento, uma vez que a construção do predicado "instituicao_especialidade" está associada à construção de invariantes que garantem que a instituição e a especialidade estão declaradas.*

Simulação (SICStus Prolog): *O teste foi novamente simulando com recurso a duas cláusulas: na primeira usa-se o "registar" para se inserir um prestador da especialidade de ortopedia, no hospital_braga, sendo que esta instituição não dispõe desta especialidade; e na segunda, o "registar" é usado para a inserção de um prestador de cardiologia, também nos hospital_braga, sendo que o hospital já pode prestar esta especialidade.*

```
| ?- registar(prestador(p16, paulo_campos, ortopedia, hospital_braga, medico)).
no
| ?- registar(prestador(p16, paulo_campos, cardiologia, hospital_braga, medico)).
yes
```

Invariante 4:

Não permite inserir prestadores com tipo diferente de "medico", "enfermeiro" ou "tecnico"

+prestador(_, _, _, _, Tipo) :: (Tipo=medico; Tipo=enfermeiro; Tipo=tecnico).

Justificação: *Para a inserção de um novo prestador, também é necessário que o seu tipo seja um dos três: médico, enfermeiro ou técnico. Todos os prestadores cujo tipo seja diferente destes, serão considerados inválidos e não poderão ser adicionados à base de conhecimento.*

Simulação (SICStus Prolog): *Na primeira cláusula, é feita uma simulação a um novo prestador cujo tipo é "auxiliar" e, no segundo caso, a simulação é testada com um prestador do tipo "tecnico".*

```
| ?- registar(prestador(p17, paulo_campos, cardiologia, hospital_braga, auxiliar)).
no
| ?- registar(prestador(p17, paulo_campos, cardiologia, hospital_braga, tecnico)).
yes
```

3.6.2.2. Invariantes de Remoção

Invariante 1:

Não permite remover prestadores que já tenham realizado cuidados médicos

-prestador(IdP, _, _, _, _) :: (solucoes(IdP, cuidado(_, _, IdP, _, _), S), S==[ ]).

Justificação: *Na base de conhecimento relativa ao prestador, considera-se o elemento IdP do prestador que se pretende remover e verifica-se se existe algum cuidado médico que esteja associado a esse IdP. Essa pesquisa é efetuada através do predicado Soluções, e os registos encontrados são colocados numa lista S. A regra S ==[ ] obriga a que a lista S esteja vazia, impedindo que se retirem médicos que já tenham realizado cuidados médicos.*

3.6.3. Especialidade

Em relação à base de conhecimento "especialidade", foram também definidos invariantes de inserção, de forma a que os novos registos nessa base de conhecimento respeitem as restrições indicadas:

- Não permitir a inserção de novas especialidades cuja designação seja igual a outra já existente;

Do ponto de vista das regras para a remoção de especialidades, impôs-se como necessário:

- Impedir a remoção de especialidades que estejam associadas a prestadores existentes.

3.6.3.1. Invariantes de Inserção

Invariante 1:

Não permite inserir especialidades com designação já existente

+especialidade(E) :: (solucoes(E, especialidade(E), S), comprimento(S,N), N==1).

Justificação: A designação de uma especialidade é única sendo, por isso, necessário impedir-se que se adicionem novos registos com uma designação igual a outra já existente. Para este invariante, recorre-se ao predicado Soluções," que irá pesquisar a designação "E" na base de conhecimento especialidade, devolvendo na lista S todos os registos encontrados. Se a lista S tiver comprimento igual a 1, aceita-se o novo registo. O motivo de o comprimento ser igual a 1 tem a ver com o facto de a nova especialidade, com a respetiva designação, já ter sido registada na base de conhecimento "especialidade", antes de ser validada pelo invariante.

Simulação (SICStus Prolog): No primeiro caso efetua-se o "registar" de uma especialidade já existente na base de conhecimento e, na segunda situação, o procedimento efetuado é o mesmo, neste caso com uma especialidade nova.

```
| ?- registar(especialidade(oncologia)).
no
| ?- registar(especialidade(pediatria)).
yes
```

3.6.3.2. Invariantes de Remoção

Invariante 1:

Não permite remover uma especialidade que esteja associada a um prestador

-especialidade(E) :: (solucoes(E, prestador(IdP, _, E, _, _), S), S==[ ]).

Justificação: Sempre que uma especialidade já faz parte de algum registo de um prestador, deve ser proibida a sua remoção, de modo a impedir que os registos que a contenham fiquem sem elementos para a sua correta manipulação. Impede-se essa remoção através da utilização do predicado Soluções que irá devolver, numa lista S, os IDs dos prestadores que tenham essa especialidade. Se a lista S estiver vazia, a especialidade poderá ser removida.

3.6.4. Instituição

Relativamente aos novos registos de instituições, a sua adição só é possível a caso sejam respeitados os invariantes que:

- Não permitem a inserção de novas instituições com designações iguais a outras já registadas;

Por outro lado, considerou-se adequada a inclusão de um invariante de remoção que:

- Impeça a remoção de uma instituição que já tenha, pelo menos, um prestador associado.

3.6.4.1. Invariantes de Inserção

Invariante 1:

Não permite inserir instituições com designação já existente

+instituicao(Inst) :: (solucoes(Inst, instituicao(Inst), S), comprimento(S,N), N==1).

Justificação: As instituições têm designações que são únicas e, por isso, serão impedidos os novos registos que adicionem instituições com uma designação já utilizada. Isto é feito com o recurso ao predicado soluções que adiciona, numa lista S, todas as instituições com a mesma designação que a do novo registo. Se o comprimento da lista S, calculada através do predicado "comprimento", for igual a 1, permite-se a adição do novo registo. O motivo de o comprimento ser igual a 1 tem a ver com o facto de a nova especialidade, com a respetiva designação, já ter sido registado na base de conhecimento especialidade.

3.6.4.2. Invariantes de Remoção

Invariante 1:

Não permite remover uma instituição que esteja associada a, pelo menos, um prestador

-instituicao(Inst) :: (solucoes(Inst, prestador(_, _, _, Inst, _), S), S==[ ]).

Justificação: Quando uma instituição está associada a pelo menos um prestador, não deverá ser possível removê-la. Para garantir esta condição, é utilizado um invariante que recorre ao predicado Soluções, para encontrar todas as instituições que já estão associadas a um prestador, e coloca-as na lista S, que deverá estar vazia para permitir a sua remoção. Note-se que a condição S==[ ] impõe que a lista S esteja vazia.

3.6.5. Tipo de Cuidado

A base de conhecimento *Tipo de Cuidado* necessita de ter invariantes que restrinjam os novos registos e as remoções dos registos existentes. Em relação à inserção de novos registos é preciso criarem-se invariantes para:

- Impedir a colocação de tipos de cuidados repetidos.

Ao nível da remoção de registos, foi definido o invariante que assegura que:

- Não é possível remover registos que já tenham um cuidado associado.

3.6.5.1. Invariantes de Inserção

Invariante 1:

Não permite inserir tipos de cuidados já existentes

+tipo_cuidado(Desc) :: (solucoes(Desc, tipo_cuidado(Desc), S), comprimento(S,N), N==1).

Justificação: A descrição de cada cuidado é única e, por isso, não podem ser inseridos dois tipos de cuidados com igual designação. Para isso e de forma a garantir que estes nomes são únicos, é preciso criar um invariante que, recorrendo ao predicado Soluções, assegure que o comprimento de uma lista S que contenha todos os cuidados com a mesma designação seja igual a 1.

3.6.5.2. Invariantes de Remoção

Invariante 1:

Não permite remover registos de tipos de cuidado que estejam associados a um cuidado

-tipo_cuidado(Desc) :: (solucoes(Desc, cuidado(_, _, _, Desc, _), S), S==[]).

Justificação: Para se removerem tipos de cuidados, é preciso garantir que nenhum dos cuidados já prestados seja desse tipo. Assim, usa-se um invariante que recorre ao predicado Soluções, para encontrar todas os tipos de cuidado já associados a um cuidado médico e, apenas no caso em que não exista nenhum tipo de cuidado associado a um cuidado, é que é possível removê-lo.

3.6.6. Cuidado

Para a inserção de novos cuidados médicos foram criados invariantes que seguem as seguintes regras:

- Não se permite a inserção de novos cuidados médicos que incluam um utente ou um prestador que não pertença à base de conhecimento respetiva;
- Não se podem inserir cuidados com custos negativos;
- Os idosos com mais de 65 anos estão isentos de pagamento caso lhe seja prestado um cuidado de análises clínicas e, deste modo, não se podem inserir cuidados, nesta situação, que tenham um custo diferente de 0;
- O cuidado de rinectomia só pode ser realizado por um prestador da especialidade de otorrinolaringologia;
- O cuidado de densitometria óssea só pode ser efetuado por um prestador da especialidade de ortopedia.

No caso deste predicado não existem quaisquer invariantes de remoção associados.

3.6.6.1. Invariantes de Inserção

Invariante 1:

Não permite inserir cuidados de utentes nem de prestadores inexistentes na base de conhecimento nem cuidados que não sejam prestados numa dada instituição

+cuidado(_, IdU, IdP, Desc, _) :: (prestador(IdP, _, _, _, _), utente(IdU, _, _, _, _), instituicao_cuidado(Inst, Desc)).

Justificação: *Um novo cuidado prestado só pode ser adicionado se estiver associado a um utente e a um prestador que exista na base de conhecimento respetiva. Além disso, esse cuidado também tem de se encontrar registado na base de conhecimento Instituição-Cuidado. Para tal, é necessário garantir-se a existência desse IdP na base "prestador", em simultâneo, com a obrigação de existência do IdU na base Utente e da existência da descrição do cuidado na base Instituição-Cuidado. Se alguma destas condições falhar, o novo registo não se efetiva.*

Invariante 2:

Não permite inserir cuidados com custos negativos

+cuidado(_, _, _, _, C) :: (C>=0).

Justificação: *No campo custo relativo a um cuidado médico só é possível inserir valores positivos. O invariante que garante esta condição é simples e apenas impõe que o custo C inserido seja maior ou igual do que zero.*

Simulação (SICStus Prolog): *Como simulação, foram testadas duas questões no PROLOG, em que na primeira se testa o registo de um cuidado médico com custo negativo e, na segunda situação, o teste é feito à inserção de um cuidado de custo 10.*

```
| ?- registar(cuidado(230217,u4, p11, raio_x, -10)).
no
| ?- registar(cuidado(230217,u4, p11, raio_x, 10)).
yes
```

Invariante 3:

Apenas permite inserir cuidados de análises clínicas a utentes com mais de 65 anos, se o seu custo for 0

+cuidado(_, IdU, _, analises_clinicas, C) :: (utente(IdU, _, I, _, _), ((C==0, I>=65); I<65)).

Justificação: Foi definido que os utentes idosos, com idade superior a 65 anos, não pagam se o cuidado prestado for de análises clínicas. Desta forma, para utentes que realizem análises clínicas, o invariante verifica se o custo é zero quando a idade do utente é maior ou igual a 65 e possibilita que, para utentes com menos de 65 anos, o custo seja de qualquer valor. Para isso, incluiu-se a condição "C==0, I>=65); I<65", que adequa a restrição associada ao custo, consoante a idade seja maior (ou igual) ou menor do que 65 anos.

Simulação (SICStus Prolog): Para testar o funcionamento deste invariante, foram criadas três cláusulas: nas duas primeiras simulou-se a inserção de cuidado de análises clínicas com custos de 10 e 0, respetivamente, aplicados a um utente u4, que tem mais de 65 anos; na última cláusula foi simulação a inserção deste mesmo cuidado, com um custo de 10, a um utente u1, que tem 41 anos.

```
| ?- registar(cuidado(230217, u4, p11, analises_clinicas, 10)).
no
| ?- registar(cuidado(230217, u4, p11, analises_clinicas, 0)).
yes
| ?- registar(cuidado(230217, u1, p11, analises_clinicas, 10)).
yes
```

Invariante 4:

Apenas permite inserir cuidados de rinectomia associados a prestadores de otorrinolaringologia

+cuidado(_, _, _, rinectomia, _) :: (cuidado(_, _, IdP, rinectomia, _),

prestador(IdP, _, otorrinolaringologia, _, _)).

Justificação: Definiu-se que apenas os prestadores da especialidade de otorrinolaringologia podem realizar cuidados médicos de rinectomia. Assim, o invariante obrigará a que um novo cuidado de rinectomia inserido esteja associado a um prestador de otorrinolaringologia. Para isso, torna-se necessário relacionarem-se os predicados "cuidado" e "prestador", sendo a conexão entre as estas bases realizada através do IdP.

Simulação (SICStus Prolog): O prestador p1 está associado a uma especialidade de cardiologia e o prestador p2 tem uma especialidade de otorrinolaringologia. Assim, foram efetuados dois testes idênticos para registar um cuidado de rinectomia a um utente u1: um primeiro prestado pelo prestador p1 e um segundo prestado pelo prestador p2.

```
| ?- registar(cuidado(230217,u1, p1, rinectomia, 10)).
no
| ?- registar(cuidado(230217,u1, p2, rinectomia, 10)).
yes
```

Invariante 5:

Apenas permite inserir cuidados de densitometria óssea a prestadores de ortopedia

+cuidado(_, _, _, densitometria_ossea, _) :: (cuidado(_, _, IdP, densitometria_ossea, _), prestador(IdP, _, ortopedia, _, _)).

Justificação: De um modo semelhante ao do invariante 4, só é possível inserir novos cuidados prestados de densitometria óssea, se a especialidade correspondente for ortopedia. Desta forma, todos os novos cuidados que estejam associados a densitometria óssea, têm de ter um prestador de ortopedia.

Simulação (SICStus Prolog): O prestador p3 está associado a uma especialidade de medicina dentária e o prestador p13 à especialidade de ortopedia. Neste caso, os testes foram simulados para se registar um cuidado de densitometria óssea, efetuados por pelo prestador p3 e p13, respetivamente.

```
| ?- registar(cuidado(230217, u1, p3, densitometria_ossea, 10)).
no
| ?- registar(cuidado(230217,u1, p13, densitometria_ossea, 10)).
yes
```

3.6.7. Instituição-Especialidade

A base de conhecimento *Instituição-Especialidade* necessita de manter a coerência dos seus dados, de acordo com as bases de conhecimento *Instituição* e *Especialidade*. Assim, os invariantes de inserção têm de impedir a existência de incoerências entre as diversas bases de conhecimento, ao nível de:

- Só permitir a adição de instituições e de especialidades que já existam nas respetivas bases de conhecimento;
- Impedir a colocação de um par instituição-especialidade repetido;

Ao nível das necessidades de coerência na remoção de registos foi criado o invariante que garante que:

- Não é possível remover registos que já tenham um cuidado médico associado.

3.6.7.1. Invariantes de Inserção

Invariante 1:

Não permite inserir instituições nem especialidades que não pertençam à base de conhecimento respetiva

+instituicao_especialidade(Inst, E) :: (instituicao(Inst), especialidade(E)).

Justificação: Os novos registos de pares instituição-especialidade terão de ter cada um dos seus elementos, Inst (descrição da instituição) e E (designação da especialidade), registados nas respetivas bases de conhecimento. Esta condição é garantida através do invariante que verifica se Inst existe na base de conhecimento Instituição e se E existe na base Especialidade. Se algum deles não pertencer à respetiva base de conhecimento, o invariante impede a adição do novo registo.

Invariante 2:

Não permite inserir um par instituição-especialidade repetido

```
+instituicao_especialidade(Inst, E) :: (solucoes((Inst, E), instituicao_especialidade(Inst, E), S),
comprimento(S,N), N==1).
```

Justificação: Cada novo par instituição-especialidade tem de ser diferente dos que já existem registados. Utilizando-se o predicado Soluções verificam-se se existem registos, na base Instituição-Especialidade, que tenham o mesmo par de variáveis Inst e E. Caso existam, esses registos são colocados na lista S, que terá de ter um comprimento igual a 1 para se poder inserir o novo registo. O motivo de o comprimento ser igual a 1 tem a ver com o facto de o novo registo ter sido previamente inserido na base de conhecimento.

3.6.7.2. Invariantes de Remoção

Invariante 1:

Não permite remover registos que já tenham cuidados médicos associados

```
-instituicao_especialidade(Inst, E) :: (solucoes((Inst, E), (cuidado(_, _, IdP, _, _),
prestador(IdP, _, E, Inst, _)), S), S==[ ]).
```

Justificação: Qualquer registo instituição-especialidade, só pode ser removido, se não existir nenhum cuidado associado, simultaneamente, a essa especialidade e instituição. Para verificar se isto acontece, através do predicado Soluções constrói-se a lista S, colocando-se nela os pares instituição-especialidade que tenham o mesmo valor do registo a remover e que respeitem, ao mesmo tempo, as condições: existam prestadores, identificados pelo seu IdP, com a especialidade E e a instituição Inst, e existam cuidados prestados cujo IdP seja igual ao encontrado na base Prestador. Se a lista S estiver vazia, então, poderá ser adicionado o novo registo.

3.6.8. Instituição-Cuidado

A base de conhecimento *Instituição-Cuidado* tem de estar coerente com as bases *Instituição* e *Cuidado* e necessita que sejam considerados os seguintes invariantes de inserção:

- Só se permite a adição de um novo registo Instituição-Cuidado no caso de os tipos de cuidados e instituições já existirem nas respetivas bases de conhecimento;
- É preciso impedir a colocação de um par instituição-cuidado repetido.

Ao nível das necessidades de coerência na remoção de registos, foi incluído o invariante que assegura que:

- Não é possível remover registos que já tenham um cuidado associado.

3.6.8.1. Invariantes de Inserção

Invariante 1:

Não permite inserir nem instituições nem tipos de cuidados que não pertençam à base de conhecimento respetiva

```
+instituicao_cuidado(Inst, Desc) :: (instituicao(Inst), tipo_cuidado(Desc)).
```

Justificação: Os novos registos de pares instituição-cuidado terão de ter cada um dos seus elementos, Inst (descrição da instituição) e Desc (designação do tipo de cuidado), registados nas respetivas bases de conhecimento. Garante-se esta condição através do invariante que verifica se Inst existe na base de conhecimento Instituição e se Desc existe na base Especialidade. Caso algum destes elementos não esteja declarado na respetiva base de conhecimento, o invariante não permite a adição do novo registo.

Invariante 2:

Não permite inserir um par instituição-cuidado repetido

```
+instituicao_cuidado(Inst, Desc) :: (solucoes((Inst,Desc), instituicao_cuidado(Inst, Desc), S), comprimento(S,N), N==1).
```

Justificação: Para se assegurar que cada novo par instituição-cuidado é diferente dos que já existem registados, usa-se o predicado Soluções, que permite obter uma lista S com os registos existentes que são iguais aos que se pretendem inserir. Uma vez construída esta lista S, é necessário garantir que o seu comprimento é 1.

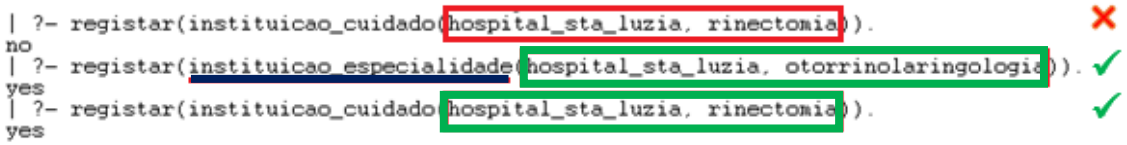
Invariante 3:

Apenas permite inserir um cuidado de rinectomia se a instituição tiver a especialidade de otorrinolaringologia

+instituicao_cuidado(Inst, rinectomia) :: (instituicao_especialidade(Inst, otorrinolaringologia)).

Justificação: *Para se declarar que uma determinada instituição pode prestar o cuidado rinectomia, é preciso assegurar-se que essa mesma instituição possibilita disponibiliza a especialidade de otorrinolaringologia, uma vez que este cuidado apenas pode ser realizado por um prestador desta especialidade. Para isso, é preciso garantir que a variável Inst do novo registo, está declarada também no predicado Instituição-Especialidade associada a uma especialidade de otorrinolaringologia.*

Simulação (SICStus Prolog): *Na base de conhecimento inicial, não está declarado que o hospital_sta_luzia presta a especialidade de otorrinolaringologia. De forma a simular-se o uso deste invariante, tentou-se, numa primeira etapa, registar o cuidado de rinectomia neste hospital. Posteriormente, registou-se a especialidade de otorrinolaringologia neste hospital e, num último passo, voltou-se a simular o registo do cuidado de rinectomia associado a esta instituição.*



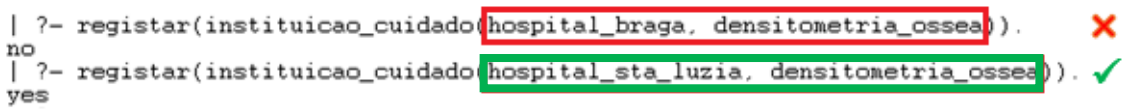
Invariante 4:

Apenas permite inserir um cuidado de densitometria óssea se a instituição tiver a especialidade de ortopedia

+instituicao_cuidado(Inst, densitometria_ossea) :: (instituicao_especialidade(Inst, ortopedia)).

Justificação: *A justificação da inclusão deste invariante é análoga à do anterior. Neste caso, apenas é possível declarar que uma dada instituição pode prestar densitometria óssea, se essa mesma especial for prestadora da especialidade de ortopedia.*

Simulação (SICStus Prolog): *O teste a este invariante foi efetuado através de duas questões, na primeira tentando inserir-se o cuidado de densitometria óssea num hospital que não dispõe de ortopedia (hospital_braga) e, na segunda, efetua-se o mesmo procedimento mas aplicado a uma instituição que já dispõe desta especialidade (hospital_sta_luzia)..*



3.6.8.1. Invariantes de Remoção

Invariante 1:

Não permite remover registos que já tenham cuidados médicos associados

-instituicao_cuidado(Inst, Desc) :: (solucoes((Inst, Desc), (cuidado(_, _, IdP, Desc, _), prestador(IdP, _, _, Inst, _)), S), S==[ ]).

Justificação: *Qualquer registo instituição-cuidado só pode ser removido se não tiver nenhum cuidado associado a esse registo. Para isso, com o predicado Soluções, constrói-se a lista S, colocando-se nela os pares instituição-cuidado que tenham o mesmo valor do registo que se quer remover (Inst e Desc) e que respeitem em simultâneo as condições: existam prestadores cuja instituição seja Inst e existam cuidados cuja descrição seja Desc, sendo a interligação entre as bases prestador e cuidado feita através do IdP correspondente. Se a lista S estiver vazia, então, pode ser adicionado o novo registo.*

4. Conclusão

O presente trabalho demonstra que a programação baseada no conceito de lógica matemática é um modo rápido, intuitivo e eficaz de responder às funcionalidades mais importantes que são necessárias para uma manipulação eficiente uma base de conhecimento. Além disso, possibilita o entendimento dos pressupostos que lhe estão associados – pressuposto dos nomes únicos, pressuposto do mundo fechado e pressuposto do domínio fechado.

Como forma de, no futuro, poder-se melhorar o desempenho de uma base de conhecimento como a que foi realizada deveriam ser incluídas outras funcionalidades como, por exemplo, impossibilidade de maiores de idade se inscreverem em consultas de pediatria e utentes do sexo masculino serem impedidos de se registarem em cuidados da especialidade de ginecologia. Também poderiam ser incluídas funcionalidades mais elaboradas como, por exemplo, a listagem dos cuidados que são prestados por todas as instituições presentes na base de conhecimento. De forma complementar deveriam existir mecanismos de gravação e leitura de ficheiros para garantira a preservação das alterações efetuadas.

Em conclusão, verificou-se que, com um pequeno conjunto de instruções, é possível manipular uma base de conhecimento e revela as algumas das potencialidades no uso da programação baseada em lógica.

5. Referências Bibliográficas

[Carlsson, 2014] Carlsson, Mats

"SICStus Prolog – User's Manual 4.3"

SICS Swedish, Suécia, Abril 2014

[Shoham, 1994] Shoham, Yoav

"Artificial intelligence techniques in prolog"

Morgan Kaufmann Publishers, EUA, 1994

[Metakides, 1996] Metakides, G.; Nerode, A.

"Principles of Logic and Logic Programming"

Elsevier Science, EUA, 1996

[Trappl, 1986] Trappl, R.

"Cybernetics and Systems '86"

R. Trappl, Austria, 1986