Conjecturas sobre Dificuldades do Uso da Linguagem CHR na Programação de Raciocínios Abdutivos por não Especialistas em Programação Lógica

Ricardo José Martins[1,2], Osvaldo Luiz de Oliveira[1]

[1]Faculdade Campo Limpo Paulista (FACCAMP)
Rua Guatemala, 167, Jd. América – Campo Limpo Paulista – SP – Brasil

[2]IFSULDEMINAS – Campus Muzambinho
Caixa Postal 02 – 37890-000 – Muzambinho – MG – Brasil
ricardo.martins@muz.ifsuldeminas.edu.br, osvaldo@faccamp.br

Abstract. *Abductive reasoning formulate hypotheses to explain observed facts using as basis a theory. Numerous intellectual tasks, including medical diagnosis, fault diagnosis, scientific discovery, legal reasoning and interpretation make use of abductive reasoning. This article describes how the CHR (Constraint Handling Rules) language can be used in abductive reasoning programming and conjectures the main difficulties faced by non-specialists in logic programming when using CHR.*

Resumo. *Raciocínios abdutivos formulam hipóteses para explicar fatos observados, considerando uma teoria como base. Inúmeras tarefas intelectuais, incluindo diagnóstico médico, diagnóstico de falhas, descoberta científica, argumentação jurídica e interpretação fazem uso de raciocínio abdutivo. Este artigo descreve como a linguagem CHR (Constraint Handling Rules) pode ser empregada na programação de raciocínios abdutivos e conjectura as principais dificuldades do uso desta linguagem por não especialistas em programação lógica.*

1. Introdução

Abdução é um tipo de inferência lógica que objetiva formular possíveis hipóteses para explicar fatos observados considerando como fundamento uma teoria. O campo de aplicação das técnicas para realizar raciocínio abdutivo por meio de computadores é extenso e inclui diagnóstico médico, descoberta de falhas em sistemas, interpretação de linguagem natural e planejamento, entre outras. Uma definição formal de raciocínio abdutivo e exemplos de emprego deste tipo de raciocínio são apresentados em Rodrigues, Oliveira & Oliveira (2014) e Rodrigues (2015).

Entre as diversas propostas para programação de raciocínio abdutivo destaca-se hoje em dia a linguagem CHR (*Constraint Handling Rules*). O trabalho em desenvolvimento descrito neste artigo se insere dentro de um contexto mais amplo do desenvolvimento de uma linguagem para programação de raciocínios abdutivos dirigida especialmente a não especialistas em programação lógica. Particularmente, este artigo descreve como a linguagem CHR pode ser utilizada na programação de raciocínios

adbtivos e conjectura as principais dificuldades do uso desta linguagem por não especialistas em programação lógica.

O restante deste artigo está organizado da seguinte maneira. A Seção 2 apresenta a linguagem CHR e discute o funcionamento de um programa em CHR. A Seção 3 mostra como raciocínios abdutivos podem ser programados em CHR. Como considerações finais, a Seção 4 levanta hipóteses sobre dificuldades do uso de CHR por não especialistas em programação lógica.

2. A Linguagem CHR

A linguagem CHR foi desenvolvida na década de 1990 e foi apresentada pela primeira vez em Frühwirth (1998). Hoje CHR está presente como extensão linguística dos principais sistemas que implementam Prolog (e.g., SWI-Prolog, SICStus Prolog). Programas nestes sistemas podem conter uma mistura de sentenças das linguagens Prolog e CHR ou apenas sentenças da linguagem CHR, uma vez que CHR é Turing completa. CHR herda a nomenclatura básica da linguagem Prolog. Aplicam-se a CHR as mesmas noções e definições atribuídas pela literatura de Prolog (e.g., Bramer (2013)) a constantes, variáveis, átomos, termos, predicados, aridade de predicado, regra, cláusula, cabeça e corpo de cláusula, questões, objetivo, uso de letras maiúsculas para variáveis etc.. Um programa CHR é composto por três tipos regras cujos formatos são:

- Regras de simplificação: $h_1 , h_2 , \ldots , h_n$ <=> *Guarda* | $b_1 , b_2 , \ldots , b_m$.
- Regras de propagação: $h_1 , h_2 , \ldots , h_n$ ==> *Guarda* | $b_1 , b_2 , \ldots , b_m$.
- Regras de "Simpagação": $h_1 , h_2 , \ldots , h_n \setminus h_{n+1} , h_{n+2} , \ldots , h_p$ ==> *Guarda* | $b_1 , b_2 , \ldots , b_m$.

Cada h_i e cada b_i é um predicado especialmente denominado predicado de restrição ou, simplesmente, restrição ($n \geq 1$, $m \geq 1$ e $p \geq 2$). Os predicados de restrição h_i e b_i formam o que, de maneira correlata às regras de Prolog, chamam-se, respectivamente, de cabeça e corpo da regra. A "," (virgula) é o operador de sequenciamento e significa conjunção. No corpo de uma regra é possível usar o operador ";" no lugar do operador "," para significar disjunção entre predicados. O *Guarda* é um conjunto, possivelmente vazio, de predicados separados pelo operador ",". Um *Guarda* vazio é interpretado como verdade (true) e pode deixar de ser descrito.

As regras de um programa CHR podem ser entendidas com regras de reescrita sobre estados, sendo um estado definido pelo conjunto de restrições presentes em um certo instante em um banco de restrições, a partir daqui chamado apenas de banco. Inicialmente o banco é preenchido com a questão a ser avaliada. Uma regra é aplicada se (1) a cabeça da regra coincide com a cabeça de restrições presentes no banco e (2) o guarda da regra é satisfeito. Regras de simplificação implementam uma bi-implicação, trocando um conjunto de restrições no banco por outro conjunto de restrições equivalente. Regras de propagação implementam uma implicação por adicionar novas restrições sem remover seus "antecedentes". Regras de "simpagação" são uma mistura das outras duas, sendo que as restrições na cabeça que aparecem antes da barra invertida permanecem no banco e aquelas que estão após a barra invertida são removidas. CHR poderia funcionar apenas com regras de "simpagação", uma vez que ela pode ser usada para simular as outras duas. A unificação de átomos ocorre como em Prolog. Uma descrição formal da semântica operacional de CHR é apresentada em Duck *et al*. (2004).

Exemplo 1. O programa CHR ilustrado na Figura 1-a estabelece regras para a igualdade de duas variáveis e está escrito de acordo com a sintaxe da biblioteca CHR disponível no sistema SWI-Prolog. A linha 1 declara que o programa usará o predicado de restrição *menorIgual* que possui aridade 2. As linhas 2, 3, 4 e 5 definem regras que descrevem, respectivamente, as propriedades reflexiva, antissimétrica, idempotente e transitiva da igualdade entre duas variáveis. Todas as regras possuem cabeça, corpo e guarda vazio, o que significa que o guarda é sempre true. As linhas 2 e 3 são exemplos de regras de simplificação. Ou seja, a ocorrência de *menorIgual(X,X)* no banco de restrições deve ser substituída por *true* e, a ocorrência simultânea de *menorIgual(X,Y)* e *menorIgual(Y,X)* deve ser substituída por *X=Y*. A linha 4 é um exemplo de uma regra de "simpagação" e descreve que na ocorrência de duas restrições *menorIgual(X,Y)* uma delas será removida do banco de restrições. A linha 5 é um exemplo de uma regra de propagação e descreve que, na ocorrência das restrições *menorIgual(X,Y)* e *menorIgual(Y,Z)* elas devem ser mantidas e uma nova restrição, *menorIgual(X,Z)*, deve ser escrita no banco de restrições. ■

```
1.  :- chr_constraint menorIgual/2.
2.  menorIgual(X,X) <=> true.   % Regra para a propriedade reflexiva.
3.  menorIgual(X,Y) , menorIgual(Y,X) <=> X = Y.  % Regra para a propriedade antissimétrica.
4.  menorIgual(X,Y) \ menorIgual(X,Y) <=> true.  % Regra para a propriedade idempotente.
5.  menorIgual(X,Y) , menorIgual(Y,Z) ==> menorIgual(X,Z).  % Regra para a propriedade transitiva.
```

(a)

```
1.  ?- menorIgual(X,Y), menorIgual(Y,Z), menorIgual(Z,X).
2.  X = Y, X = Z ;
3.  false.
```

(b)

Figura 1. Exemplo de um programa CHR.

Operacionalmente um programa CHR usa regras para, passo a passo, modificar um banco B, inicialmente em um estado B^1 preenchido com uma questão Q. Em cada passo uma regra é escolhida para ser aplicada. A regra a ser escolhida é a primeira da ordem em que são escritas a casar suas cabeças com restrições em B. A aplicação de uma regra é chamada de derivação. Quando a quantidade de derivações é finita e termina com B em um estado B^n ($n \geq 1$), se B^n for uma contradição então diz-se que as derivações falharam, caso contrário, as derivações foram bem sucedidas e B^n é o conjunto de restrições que respondem à questão Q.

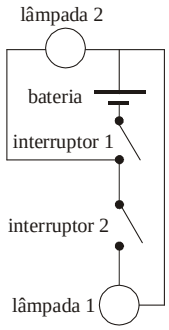
Exemplo 2. A Figura 1-b mostra a execução do programa da Figura 1-a para uma questão *Q = menorIgual(X,Y), menorIgual(Y,Z), menorIgual(Z,X)*, descrita na linha 1. O sistema executa *Q* e dá como um resultado *X = Y, X = Z* (linha 2). O ";" digitado ao final da linha 2 solicita ao sistema que apresente outros resultados. Como neste caso não há, o sistema responde com *false* (linha 3). Operacionalmente, o sistema começa inicializando um banco, digamos B, com as restrições da questão, assim *B* em seu estado 1 é B^1 = *{ menorIgual(X,Y), menorIgual(Y,Z), menorIgual(Z,X) }*. A regra de transitividade (linha 1, Figura 1-a) é aplicada à primeira e segunda restrição de B, produzindo a inserção de *menorIgual(X,Z)* a *B*, que no estado 2 passa a ser B^2 = *{ menorIgual(X,Y), menorIgual(Y,Z), menorIgual(Z,X), menorIgual(X,Z) }*. No próximo passo, a regra de antissimetria (linha 3, Figura 1-a) é aplicada à terceira e à quarta restrição de B, para produzir a ligação (*binding*) entre as variáveis *X* e *Z*, derivando B^3 = *{ menorIgual(X,Y), menorIgual(Y,Z), X=Z }*. Como *X=Z* então B^3 = *{ menorIgual(X,Y), menorIgual(Y,X), X=Z }* também. Em seguida, novamente a regra de antissimetria é aplicada à primeira e segunda restrições de *B*, produzindo a ligação *X=Y*, para derivar B^4 = *{ X=Y, X=Z }*. Como nenhuma regra pode ser aplicada a B^4 então o conjunto de derivações tem fim e *{ X=Y, X=Z }* é o conjunto de restrições que responde à questão *Q*. ■

3. Raciocínios Abdutivos com CHR

A ideia de usar CHR para implementar abdução foi proposta pela primeira vez em Abdennadher & Christiansen (2000). De forma geral, um programa CHR para abdução possui a seguinte estrutura: (1) a teoria é descrita por meio de regras; (2) as hipóteses

(h_1, h_2, ..., h_n) que possivelmente possam ser respostas ao raciocínio são declaradas como predicados de restrição (e.g., :- chr_constraint h_1, h_2, ..., h_n), neste contexto também chamados de predicados abdutíveis; (3) os fatos (f_1, f_2, ..., f_m) são inseridos como uma questão (e.g., ?- f_1, f_2, ..., f_m).

Exemplo 3. A Figura 2-a apresenta o diagrama elétrico de um circuito constituído por duas lâmpadas, dois interruptores, uma bateria e fios. Um programa CHR modelando possíveis falhas e funcionalidades que levam as lâmpadas do circuito estarem apagas ou acesas é apresentado na Figura 2-b. Na linha 1 estão declarados os possíveis predicados que podem ocorrer como resposta de um raciocínio abdutivo. As linhas de 2 a 5 contêm regras para modelar falhas e funcionalidades do circuito. As linhas de 6 a 8 definem regras para manipular inconsistências como, por exemplo, não é possível ao mesmo tempo o interruptor 1 estar ligado e desligado (linha 6). A Figura 2-c ilustra a execução do programa para o fato *lâmpada_1_apagada*. A série de pontos e virgula que separam o resultado indica que o raciocínio realizado conduziu a três hipóteses que são *interruptor_1_desligado* ***ou*** *bateria_sem_carga* ***ou*** *interruptor_2_desligado*. A Figura 2-d ilustra a execução do programa para os fatos *lâmpada_1_apagada* , *lâmpada_2_acesa*. Neste caso, o resultado conduziu a uma hipótese que deve ser lida como *interruptor_2_desligado* ***e*** *bateria_com_carga* ***e*** *interruptor_1_ligado*. ■



(a)

```
1. :- chr_constraint bateria_sem_carga, interruptor_1_desligado, interruptor_2_desligado,
     lampada_1_apagada, lampada_2_apagada, bateria_com_carga, interruptor_1_ligado,
     interruptor_2_ligado,lampada_1_acesa, lampada_2_acesa.
2. lampada_1_apagada <=> true | interruptor_1_desligado ; bateria_sem_carga ; interruptor_2_desligado.
3. lampada_2_apagada <=> true | interruptor_1_desligado ; bateria_sem_carga.
4. lampada_1_acesa <=> true | bateria_com_carga , interruptor_1_ligado, interruptor_2_ligado.
5. lampada_2_acesa <=> true | bateria_com_carga , interruptor_1_ligado.
6. interruptor_1_ligado , interruptor_1_desligado <=> fail.
7. interruptor_2_ligado , interruptor_2_desligado <=> fail.
8. bateria_sem_carga , bateria_com_carga <=> fail.
```

(b)

```
1. ?- lampada_1_apagada.
2.    interruptor_1_desligado
3.    true ;
4.    bateria_sem_carga
5.    true ;
6.    interruptor_2_desligado
7.    true ;
8.    false.
```

(c)

```
1. ?- lampada_1_apagada , lampada_2_acesa.
2.    interruptor_2_desligado
3.    bateria_com_carga
4.    interruptor_1_ligado
5.    true ;
6.    false.
```

(d)

Figura 2. Um programa CHR para abdução sobre um circuito elétrico.

4. Considerações Finais: conjecturas sobre dificuldades do uso da linguagem CHR por não especialistas em programação lógica

CHR permite a descrição de raciocínios abdutivos e o seu uso conjuntamente com Prolog tem uma série de vantagens. Entretanto, é possível conjecturar que o uso de CHR por não especialistas em programação lógica engloba as seguintes dificuldades.

Em primeiro lugar, há uma grande diferença sintática entre sentenças descrevendo regras em língua natural e as respectivas sentenças expressas em CHR. Por exemplo, é natural a expressão da sentença “se a bateria está sem carga ou o interruptor 1 está desligado então a lâmpada 2 está apagada”. No entanto, a estrutura sintática da correspondente sentença em CHR para descrever este mesmo conteúdo é muito diferente (Figura 2-b, linha 3), impondo ao programador esforço cognitivo de tradução. De forma similar, a leitura e interpretação dos resultados também tem seus desafios. Por exemplo, o resultado apresentado na Figura 2-c é “interruptor 1 desligado ou bateria sem carga ou interruptor 2 desligado” e o resultado da Figura 2-d é “interruptor 2 desligado e bateria

sem carga e interruptor 1 ligado". No entanto, o que é apresentado nestas figuras é sintaticamente muito diferente disto. Adicionalmente, não tendo sido proposta originalmente para a realização de raciocínios abdutivos, mas para resolver diferentes tipos de restrições: (1) CHR não incorpora, diretamente na sua terminologia, estruturas linguísticas do domínio dos raciocínios abdutivos (e.g., teoria, hipóteses, fatos etc.); (2) oferece fraco suporte linguístico para a descrição de sentenças que contêm negação; (3) obriga determinar por antecipação os predicados abdutíveis e a descrevê-los na seção chr_constraint (e.g., linha 1, Figura 2-b).

CHR, assim como Prolog, são linguagens declarativas. No entanto, a beleza da programação lógica em permitir ao programador pensar apenas nas declarações para compor um programa, há muito tempo foi perdida pelas implementações possíveis destes sistemas. Hoje, para programar em Prolog, e isto também vale para CHR, há a necessidade do programador conhecer profundamente a semântica operacional destas linguagens (e.g., *backtracking*, negação como falha, derivação de regra CHR).

Por fim, CHR não oferece facilidades especiais para: (1) o desenvolvimento de raciocínios mais complexos a partir da associação dos resultados de dois ou mais raciocínios, seja pela união, intersecção ou diferença que existe entre eles; (2) administração da complexidade da base de conhecimento à medida em que ela cresce em quantidade de regras. A impossibilidade de segmentar a base de conhecimento, por exemplo conferindo nomes a diferentes conjuntos de regras, pode levar o não especialista a se perder diante de uma grande base.

Os próximos passos desse trabalho envolvem o projeto de uma linguagem que seja uma resposta aos problemas aqui identificados.

Referências

Abdennadher, S., Christiansen, H. (2000) "An experimental CLP platform for integrity constraints and abduction", in *Proceedings of the Flexible Query Answering Systems: Advances in Soft Computing Series*, FQAS 2000, pp. 141–152.

Bramer, M. (2013) "Logic programming with Prolog", 2nd ed., Springer, London.

Duck,G. J., Stuckey, P. J., Banda, M. J. G., Holzbaur, C. (2004) "The refined operational semantics of Constraint Handling Rules", in *Proceeedings of the 20th International Conference on Logic Programming*, ICLP 2004, pp. 90–104.

Frühwirth, T. (1998) "Theory and practice of constraint handling rules", *Journal of Logic Programming*, 37 (1–3), pp. 95-138.

Rodrigues, F., Oliveira, C. E. A. & Oliveira, O. L. (2014) "Peirce: an algorithm for abductive reasoning operating with a quaternary reasoning framework", *Research in Computer Science*, v. 82, pp. 53-66.

Rodrigues, F. (2015) "Um algoritmo para abdução peirceana baseado em uma estrutura de raciocínio quaternária em Lógica", Dissertação de Mestrado, Faccamp, [on-line], Disponível em http://www.cc.faccamp.br/Dissertacoes/Felipe_2015.pdf.