Planejamento Abdutivo no Cálculo de Eventos

Silvio do Lago Pereira, D.Sc.

IME-USP

Setembro/2008

Sumário

Parte I

Abdução

Raciocínio abdutivo

Peirce define *abdução* como um tipo de raciocínio em que se formula hipóteses como possíveis explicações de uma evidência.

Exemplo

- Evidência: *Há goteiras no telhado*
- Hipóteses:
  - *Há telhas quebradas e está chovendo*
  - *Há telhas quebradas e a caixa d'água está vazando*
  - *Há problemas no encanamento*
  - ...

Regra de inferência abdutiva

Dedução vs. Abdução

- **Dedução**: permite inferir conseqüências do que é assumido.

$$\frac{\alpha \rightarrow \beta \quad \alpha}{\beta}$$

- **Abdução**: permite inferir possíveis causas do que é observado.

$$\frac{\alpha \rightarrow \beta \quad \beta}{\alpha}$$

Abdução é um tipo de inferência fraca,
pois apenas garante a plausibilidade das hipóteses

Geração de conhecimento

Segundo *Peirce*:

Dedução apenas revela conhecimento prévio

- Todos os livros dessa prateleira são de lógica.
- Esse livro foi retirado dessa prateleira.
- Então, esse livro é de lógica.

Abdução gera conhecimento novo (que carece de confirmação)

- Todos os livros dessa prateleira são de lógica.
- Esse livro é de lógica.
- Então, (provavelmente) esse livro foi retirado dessa prateleira.

Abdução em lógica

Abdução

Sejam

- $\mathcal{T}$ uma teoria descrevendo um domínio
- $\mathcal{O}$ um conjunto de literais descrevendo uma observação

Abdução consiste em encontrar um conjunto de literais $\mathcal{H}$ tal que:

- $\mathcal{T} \cup \mathcal{H} \models \mathcal{O}$
- $\mathcal{T} \cup \mathcal{H} \not\models \bot$

Em outras palavras, abdução encontra uma explicação $\mathcal{H}$ para $\mathcal{O}$, de acordo com a teoria $\mathcal{T}$, tal que $\mathcal{T} \cup \mathcal{H}$ é consistente. Se $\mathcal{T} \models \mathcal{O}$, a explicação preferencial para $\mathcal{O}$ é $\mathcal{H} = \varnothing$.

Exemplo de abdução em lógica

Teoria do domínio ($\mathcal{T}$)

telha_quebrada $\wedge$ chuva $\rightarrow$ goteira
falha_na_válvula $\rightarrow$ vazamento_na_caixa
vazamento_na_caixa $\rightarrow$ goteira
goteira $\rightarrow$ chão_molhado

Observação ($\mathcal{O}$)

chão_molhado

Explicações ($\mathcal{H}$)

- $\mathcal{H}_1 = \{\text{goteira}\}$
- $\mathcal{H}_2 = \{\text{telha\_quebrada}, \text{chuva}, \text{falha\_na\_válvula}\}$
- $\mathcal{H}_3 = \{\text{telha\_quebrada}, \text{chuva}\}$
- $\mathcal{H}_4 = \{\text{falha\_na\_válvula}\}$
- $\mathcal{H}_5 = \{\text{chão\_molhado}\}$
- $\mathcal{H}_6 = \{\text{telha\_quebrada}, \text{chuva}, \text{vento}\}$

Explicações preferenciais

- Para evitar explicações triviais (e.g., $\mathcal{H}_5$) e arbitrárias (e.g., $\mathcal{H}_6$), em geral, os fatos que compõem uma explicação abdutiva $\mathcal{H}$ são restritos a literais de um conjunto básico de causas primitivas pré-definidas, denominadas *abdutíveis*.
- A existência de múltiplas explicações plausíveis é uma característica do raciocínio abdutivo, sendo a seleção de uma explicação preferencial um importante problema a ser considerado.
- Os critérios para a seleção de uma explicação preferencial é dependente do domínio de aplicação.

Explicações preferenciais

Segundo *Cox & Pietrzykowski*, uma explicação preferencial $\mathcal{H}^\star$, para uma observação $\mathcal{O}$ num domínio $\mathcal{T}$, deve ser:

- **básica**: não deve conter efeitos, mas apenas causas primitivas
- **minimal**: não deve existir $\mathcal{H}' \subset \mathcal{H}^\star$ tal que $\mathcal{T} \cup \mathcal{H}' \models \mathcal{O}$
- **compacta**: deve postular o menor número possível de causas

Exemplo

- $\mathcal{T} = \{q \wedge c \rightarrow g, f \rightarrow v, v \rightarrow g, g \rightarrow m\}$
- $\mathcal{O} = \{m\}$

- $\mathcal{H}_1 = \{g\}$ não é uma explicação básica para $\mathcal{O}$
- $\mathcal{H}_2 = \{q, c, f\}$ é uma explicação básica, mas não é minimal
- $\mathcal{H}_3 = \{q, c\}$ é uma explicação minimal, mas não é compacta
- $\mathcal{H}_4 = \{f\}$ é uma explicação básica, minimal e compacta

Raciocínio não-monotônico

Abdução é uma forma de raciocínio não-monotônico.

Exemplo

- Estado A - explicação $\mathcal{H}$ é consistente
  - $\mathcal{T} = \{q \wedge c \rightarrow g, f \rightarrow v, v \rightarrow g, g \rightarrow m\}$
  - $\mathcal{O} = \{m\}$
  - $\mathcal{H} = \{f\}$
  - $\mathcal{T} \cup \mathcal{H} \models \mathcal{O}$
  - $\mathcal{T} \cup \mathcal{H} \not\models \bot$
- Estado B - explicação $\mathcal{H}$ é inconsistente
  - $\mathcal{T} = \{q \wedge c \rightarrow g, f \rightarrow v, v \rightarrow g, g \rightarrow m\} \cup \{\neg f\}$
  - $\mathcal{O} = \{m\}$
  - $\mathcal{H} = \{f\}$
  - $\mathcal{T} \cup \mathcal{H} \models \mathcal{O}$
  - $\mathcal{T} \cup \mathcal{H} \models \bot$

Algumas aplicações de abdução

- **Diagnóstico**: Dada uma teoria descrevendo o funcionamento de um sistema e um conjunto de falhas observadas, a abdução pode ser usada para encontrar uma explicação para o mal funcionamento do sistema.
- **Revisão de crenças**: O principal problema da revisão de crenças é que uma nova informação pode ser inconsistente com o estado de crença, mas o resultado de sua incorporação não. Dada uma teoria descrevendo um estado de crença e uma nova informação a ser incorporada, abdução pode ser usada para estabelecer uma ordem de preferência entre conjuntos de mundos possíveis.
- **Planejamento**: Dada um teoria descrevendo os efeitos das ações num domínio de planejamento e um estado do mundo desejado, abdução pode ser usada para encontrar um plano de ações cuja execução leva a esse estado.

Considerações

Antes de mostrar como usar abdução em planejamento automatizado de tarefas, vamos apresentar o formalismo necessário para implementação de um mecanismo de prova abdutivo.

Parte II

Programação em Lógica

Programação em Lógica

Programação em lógica é baseada em lógica de predicados de primeira ordem.

Terminologia

- Constante, função ou predicado é um identificador com inicial minúscula
- Variável é um identificador com inicial maiúscula
- Termo é uma constante, variável, ou função seguida de lista de termos
- Átomo é um predicado seguido de uma lista de termos
- Literal é um átomo ou a negação de um átomo
- Cláusula é uma sentença da forma $\varphi_1 \vee \cdots \vee \varphi_m$, onde cada φ_i é um literal
- Cláusula de *Horn* é uma cláusula com no máximo um literal positivo, escrita como $\alpha \leftarrow \beta_1, \ldots, \beta_n$, sendo $n \geq 0$.
  - Fato: $\alpha \leftarrow$
  - Regra: $\alpha \leftarrow \beta_1, \ldots, \beta_n$
  - Consulta: $\leftarrow \beta_1, \ldots, \beta_n$
- Programa lógico é um conjunto de fatos e regras
- A execução de um programa lógico é iniciada com uma consulta

Programação em Lógica

Unificação

- Uma substituição θ é um mapeamento finito de váriáveis em termos
- Se E é uma expressão, $E\theta$ é uma expressão
- Uma cláusula C_1 é uma variante de C_2 se existem θ_1 e θ_2 tais que $C_1\theta_1$ é idêntico a $C_2\theta_2$
- Átomos A_1 e A_2 são unificáveis se existe θ tal que $A_1\theta$ e $A_2\theta$ são idênticos
- θ é u.m.g. para A_1 e A_2 se para todo unificador θ_1 de A_1 e A_2 existe θ_2 tal que $A_1\theta\theta_2$ é idênctico a $A_2\theta_1$.

Resolução

$$\frac{\alpha \leftarrow \beta_1, \ldots, \beta_m \qquad \leftarrow \alpha_1, \ldots, \alpha_n}{(\leftarrow \beta_1, \ldots, \beta_m, \alpha_2, \ldots, \alpha_n)\theta} \; \theta = umg(\alpha, \alpha_1)$$

Programação em Lógica

SLD-refutação

(1) pai(adão,abel) ←
(2) pai(adão,cain) ←
(3) pai(adão,seth) ←
(4) pai(seth,enos) ←
(5) avô(X,Z) ← pai(X,Y), pai(Y,Z)
(6) ← avô(adão,R)
(7) ← pai(adão,Y), pai(Y,R) (6,5)/{X=adão,Z=R}
(8) ← pai(seth,R) (7,3)/{Y=seth}
(9) ← (8,4)/{R=enos}

Considerações

A seguir, mostramos como podemos implementar um algoritmo para raciocínio abdutivo com base nos fundamentos de programação em lógica e na linguagem Prolog.

Parte III

Meta-interpretador Abdutivo

Meta-interpretadores

- Interpretador é um programa que avalia programas
- Meta-linguagem é uma linguagem usada para descrever outra linguagem (i.e., uma linguagem objeto)
- Meta-interpretador é um interpretador para uma linguagem idêntica ou similar a sua própria linguagem de implementação
- Prolog é ideal para a implementação de meta-interpretadores
- Meta-interpretadores em Prolog podem delegar a manipulação do programa interpretado ao próprio núcleo do Prolog

A linguagem Prolog

Prolog (**pro***gramming in* **log***ic*) é uma linguagem de programação declarativa, cujo núcleo implementa o algoritmo de SLD-refutação.

Características de Prolog

- usa uma sintaxe baseada em cláusulas de *Horn*
- usa refutação como método de prova/extração de respostas
- usa resolução/unificação como regra de inferência
- usa busca em profundidade para controlar as inferências

Funcionamento do Prolog

Programa

```
/* 1 */ pai(adão,cain).
/* 2 */ pai(adão,abel).
/* 3 */ pai(adão,seth).
/* 4 */ pai(seth,enos).
/* 5 */ avô(X,Z) :- pai(X,Y), pai(Y,Z).
```

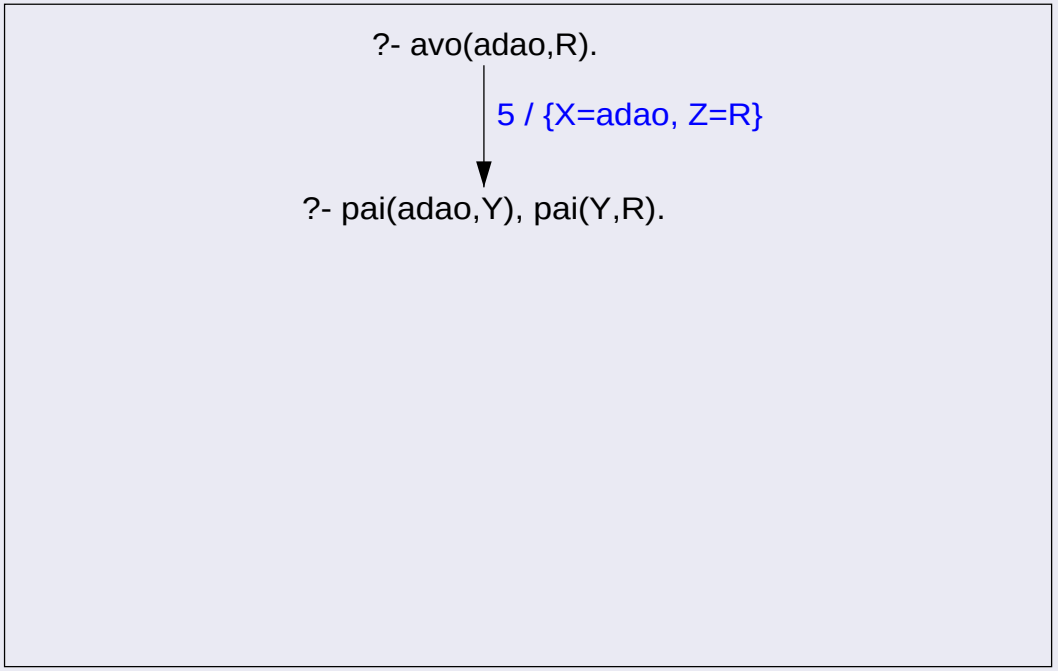
Árvore de refutação: Quem é neto de Adão?

Funcionamento do Prolog

Programa

```
/* 1 */ pai(adão,cain).
/* 2 */ pai(adão,abel).
/* 3 */ pai(adão,seth).
/* 4 */ pai(seth,enos).
/* 5 */ avô(X,Z) :- pai(X,Y), pai(Y,Z).
```

Árvore de refutação: Quem é neto de Adão?

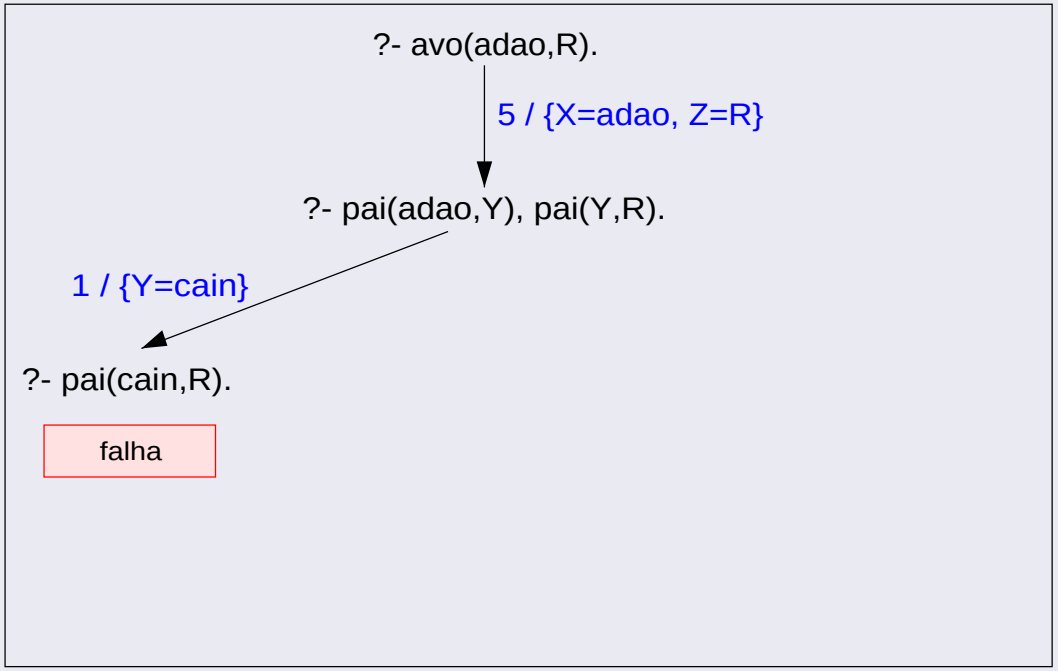


Funcionamento do Prolog

Programa

```
/* 1 */ pai(adão,cain).
/* 2 */ pai(adão,abel).
/* 3 */ pai(adão,seth).
/* 4 */ pai(seth,enos).
/* 5 */ avô(X,Z) :- pai(X,Y), pai(Y,Z).
```

Árvore de refutação: Quem é neto de Adão?

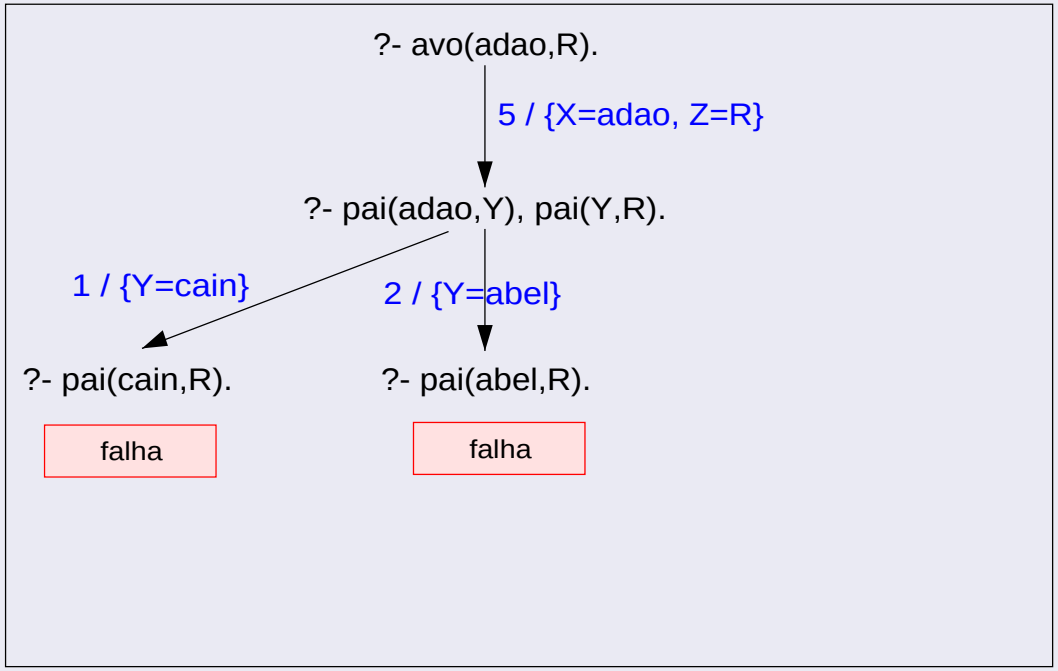


Funcionamento do Prolog

Programa

```
/* 1 */ pai(adão,cain).
/* 2 */ pai(adão,abel).
/* 3 */ pai(adão,seth).
/* 4 */ pai(seth,enos).
/* 5 */ avô(X,Z) :- pai(X,Y), pai(Y,Z).
```

Árvore de refutação: Quem é neto de Adão?

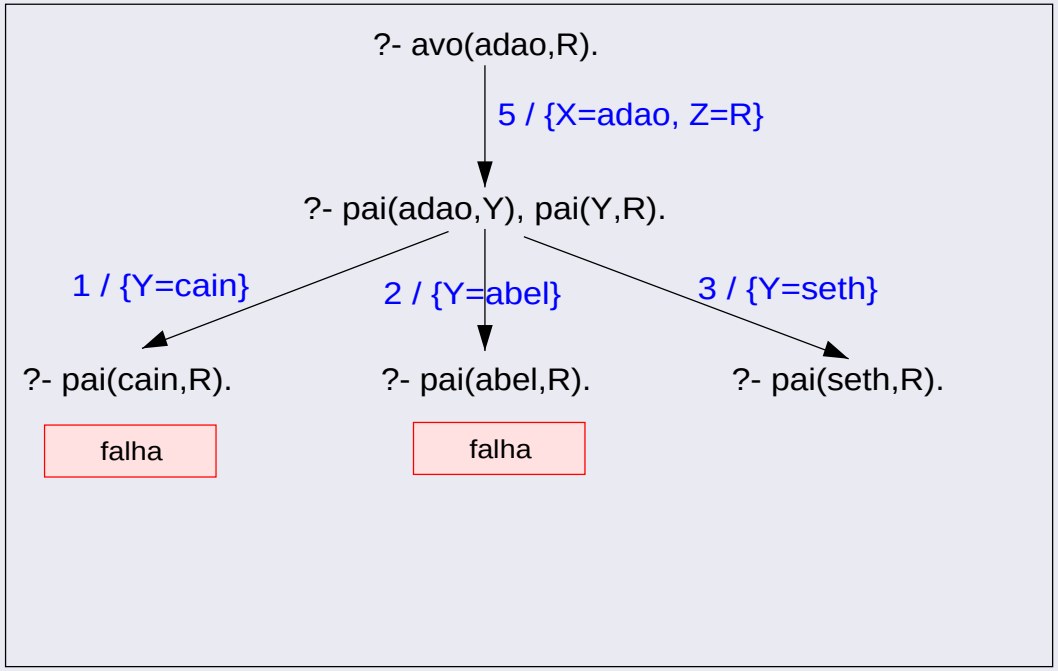


Funcionamento do Prolog

Programa

```
/* 1 */ pai(adão,cain).
/* 2 */ pai(adão,abel).
/* 3 */ pai(adão,seth).
/* 4 */ pai(seth,enos).
/* 5 */ avô(X,Z) :- pai(X,Y), pai(Y,Z).
```

Árvore de refutação: Quem é neto de Adão?

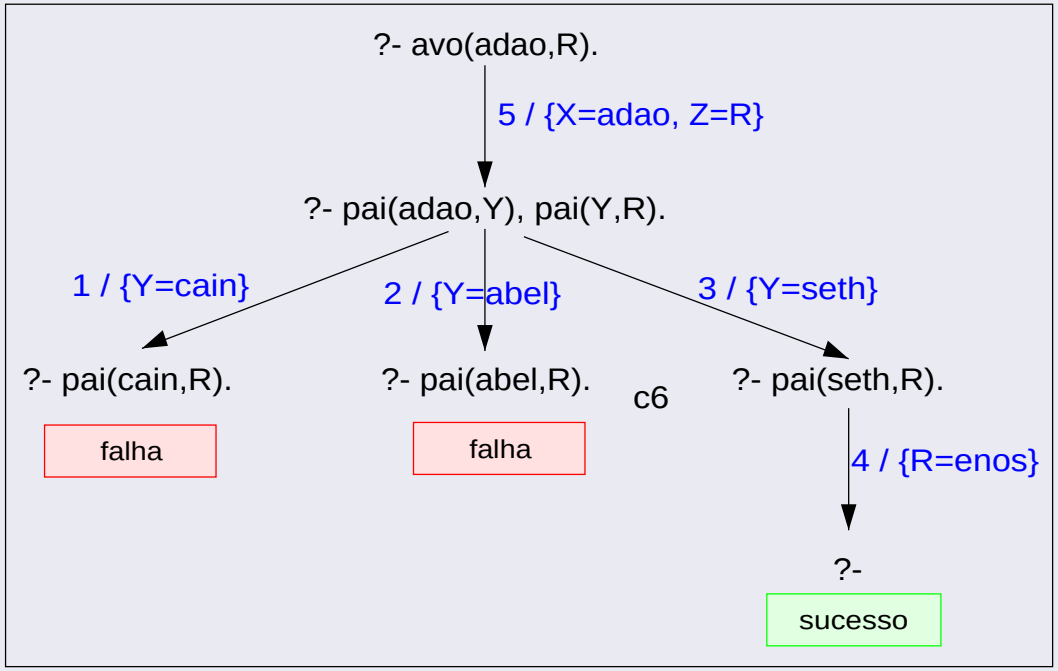


Funcionamento do Prolog

Programa

```
/* 1 */ pai(adão,cain).
/* 2 */ pai(adão,abel).
/* 3 */ pai(adão,seth).
/* 4 */ pai(seth,enos).
/* 5 */ avô(X,Z) :- pai(X,Y), pai(Y,Z).
```

Árvore de refutação: Quem é neto de Adão?



Uso de meta-interpretador

Implementando um meta-interpretador, podemos mudar algumas características de Prolog.

Por exemplo, modemos modificar:

- sintaxe
- estratégia de busca
- regras de inferência

Podemos até mesmo criar "Prolog Abdutivo" !!!

Sintaxe - exemplo 1

Programa objeto

```
axioma(pai(adão,abel),[]).                          (1)
axioma(pai(adão,cain),[]).                          (2)
axioma(pai(adão,seth),[]).                          (3)
axioma(pai(seth,enos),[]).                          (4)
axioma(avô(X,Z),[pai(X,Y),pai(Y,Z)]).               (5)
```

Meta-interpretador (SLD-refutação)

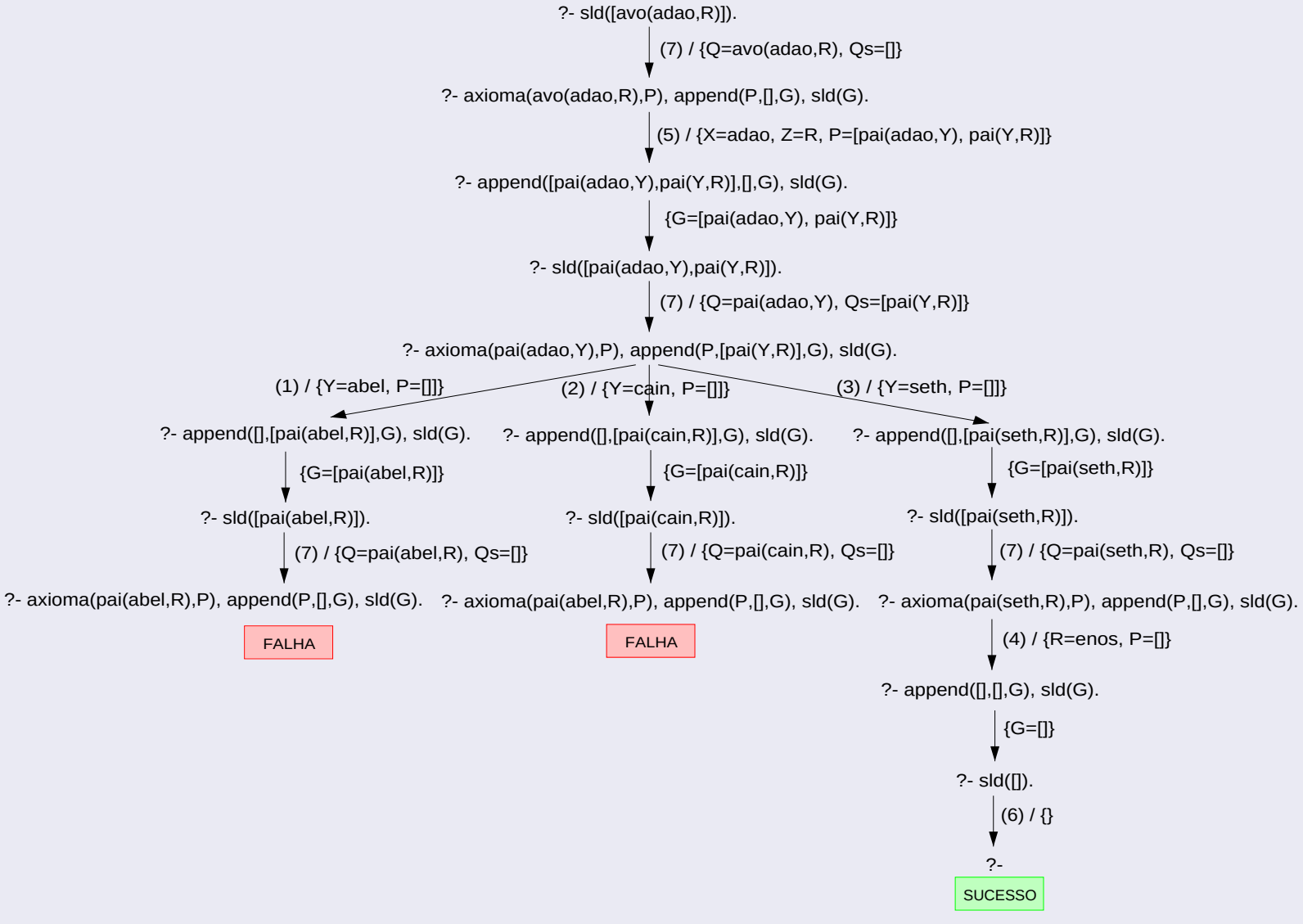
```
sld([]).                                            (6)
sld([Q|Qs]) :- axioma(Q,P), append(P,Qs,G), sld(G).  (7)
```

Consulta

```
?- sld([avô(adão,R)]).                              (8)
```

Sintaxe - exemplo 1

Árvore de refutação



Sintaxe - exemplo 2

Programa objeto

```
true => pai(adão,abel).
true => pai(adão,cain).
true => pai(adão,seth).
true => pai(seth,enos).
pai(X,Y) & pai(Y,Z) => avô(X,Z).
```

Meta-interpretador (SLD-refutação)

```
:- op( 950,xfy,&).
:- op(1150,xfx,=>).

sld(true) :- !.
sld(P & Q) :- sld(P), sld(Q).
sld(Q) :- (P=>Q), sld(P).
```

Consulta

```
?- sld(avô(adão,R)).
```

Estratégia de busca - exemplo 1

O Prolog não é capaz de mostrar que Enos é neto de Adão!

Programa objeto

```
avô(X,Y) => neto(Y,X).
neto(X,Y) => avô(Y,X).
true => avô(adão,enos).
```

Meta-interpretador (SLD-refutação com profundidade limitada)

```
:- op( 950,xfy,&).
:- op(1150,xfx,=>).

sldbs(true,_,_) :- !.
sldbs(P & Q,N,B) :- N<B, succ(N,M), sldbs(P,M,B), sldbs(Q,M,B).
sldbs(Q,N,B) :- N<B, succ(N,M), (P => Q), sldbs(P,M,B).
```

Consulta

```
?- sldbs(neto(enos,adão),0,2).
```

Estratégia de busca - exemplo 2

O que fazer quando não sabemos a profundidade da refutação?

Programa objeto

```
avô(X,Y) => neto(Y,X).
neto(X,Y) => avô(Y,X).
true => avô(adão,enos).
```

Meta-interpretador (SLD-refutação com profundidade iterativa)

```
:- op( 950,xfy,&).
:- op(1150,xfx,=>).

sldbs(true ,N,_) :- write(prof:N), !.
sldbs(P & Q,N,B) :- N<B, succ(N,M), sldbs(P,M,B), sldbs(Q,M,B).
sldbs(Q,N,B) :- N<B, succ(N,M), (P => Q), sldbs(P,M,B).

sldids(P,N) :- sldbs(P,0,N).
sldids(P,N) :- succ(N,M), sldids(P,M).
```

Consulta

```
?- sldids(neto(enos,adão),0).
```

Mecanismo de inferência - exemplo 1

Negação por falha finita

Programa objeto

```
true => pai(adão,abel).
true => pai(adão,cain).
true => pai(adão,seth).
true => pai(seth,enos).
pai(X,Y) & pai(X,Z) & ~igual(Y,Z) => irmão(Y,Z).
true => igual(X,X).
```

Meta-interpretador (SLD-refutação com negação por falha finita)

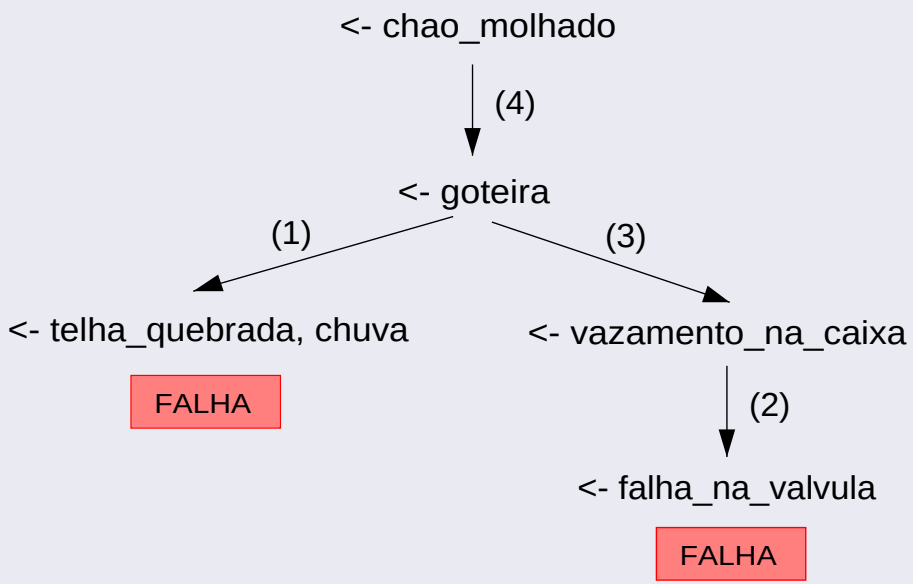
```
:- op( 900,fy,~).
:- op( 950,xfy,&).
:- op(1150,xfx,=>).

sld(true) :- !.
sld(~P) :- not(sld(P)).
sld(P & Q) :- sld(P), sld(Q).
sld(Q) :- (P=>Q), sld(P).
```

O mecanismo abdutivo em programação lógica

Programa lógico

goteira ← telha_quebrada, chuva (1)
vazamento_na_caixa ← falha_na_valvula (2)
goteira ← vazamento_na_caixa (3)
chao_molhado ← goteira (4)

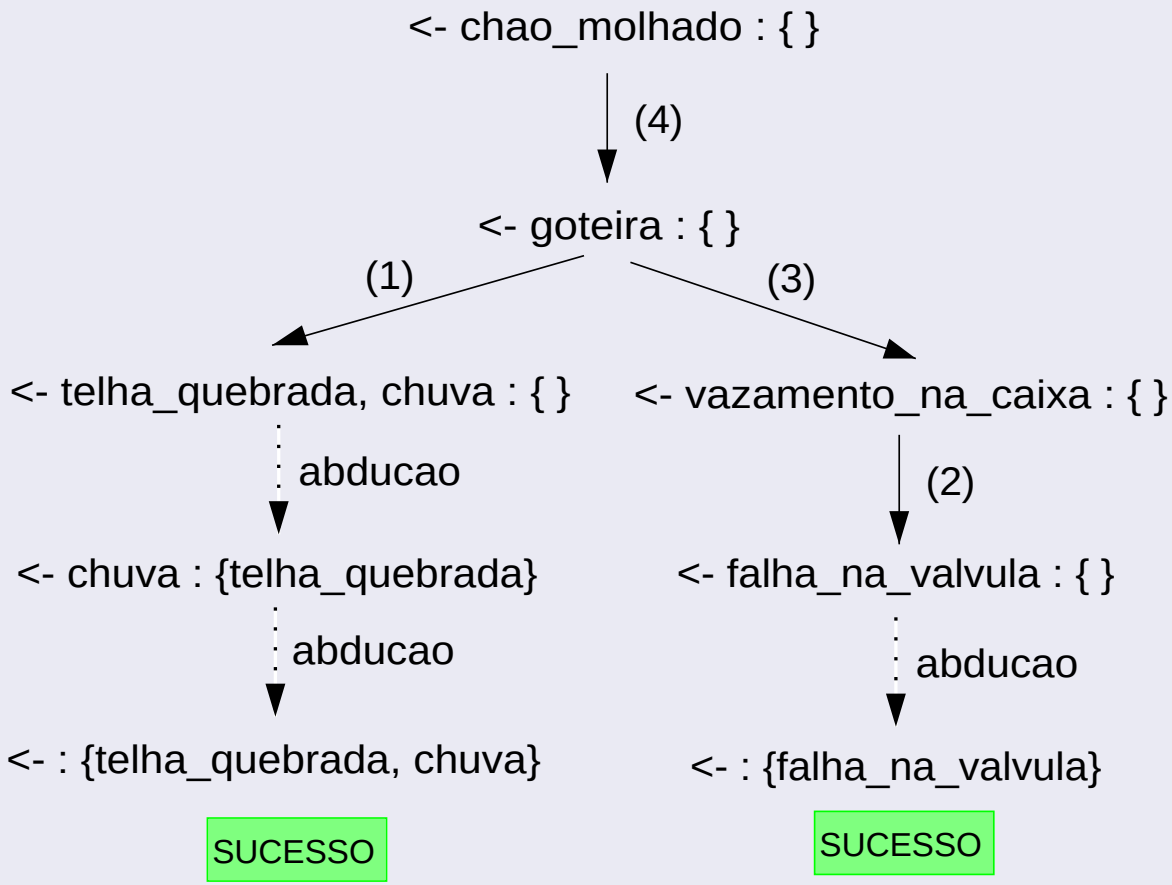
Árvore de SLD-refutação



O mecanismo abdutivo em programação lógica

Idéia de implementação: em vez de falhar, assuma como hipótese o que precisa para prosseguir com o raciocínio.

Árvore de SLDA-refutação



O mecanismo abdutivo em programação lógica

Programa objeto

```
@telha_quebrada & @chuva => goteira.
@falha_na_valvula => vazamento_na_caixa.
vazamento_na_caixa => goteira.
goteira => chao_molhado.
% true => @chuva.
```

Meta-interpretador (SLDA-refutação)

```
:- op( 900, fy, @). % para declarar proposições abdutíveis
:- op( 950,xfy, &).
:- op(1150,xfx,=>).

slda(true, H, H ) :- !.
slda(P & Q, H1, H3) :- slda(P,H1,H2), slda(Q,H2,H3).
slda(Q, H1, H2) :- (P=>Q), slda(P,H1,H2).
slda(@Q, H1, H2) :- not(_P=>(@Q)), union(H1,[Q],H2).
```

Consulta

```
?- slda(chao_molhado,[],H).
```

Abdução e negação por falha

- Abdução e negação por falha interferem mutuamente entre si.
- Na negação por falha, apenas as conseqüências lógicas do programa mais o resíduo abdutivo já computado devem ser consideradas. Portanto, a abdução deve ser desabilitada durante as provas em modo negativo.
- Por outro lado, quando hipóteses são adicionadas ao resíduo abdutivo, conclusões negativas previamente provadas podem se tornar falsas. Portanto, apenas hipóteses consistentes com negações já provadas (resíduo negativo) podem ser adicionadas ao resíduo abdutivo.

SLDNFA-refutação

Sejam

- $\mathcal{T}$ um programa lógico
- $\mathcal{O}$ um conjunto de observações

SLDNFA-refutação encontra

- $\mathcal{H}$ um conjunto de proposições abduzidas
- $\mathcal{N}$ um conjunto de proposições provadas negativas

tal que $Comp(\mathcal{T} \cup \mathcal{H}) \models \mathcal{O} \cup \mathcal{N}$.

O meta-interpretador abdutivo

Meta-predicados

- `abducible`: declara proposições abdutíveis.
- `axiom`: declara cláusulas do programa-objeto.
- `sldnfa`: encontrar explicações abdutivas.
- `check_naf`: verifica a consistência do resíduo negativo `N`, a cada modificação efetuada no resíduo abdutivo.
- `naf`: implementa negação por falha (com base em $\mathcal{T} \cup \mathcal{H}$).
- `resolve`: incorpora fatos abduzidos ao programa (implementa $\mathcal{T} \cup \mathcal{H}$).

Programa objeto

```
abducible(telha_quebrada).
abducible(chuva).
abducible(falha_na_valvula).
axiom(goteira,[telha_quebrada,chuva]).
axiom(vazamento_na_caixa,[falha_na_valvula]).
axiom(goteira,[vazamento_na_caixa]).
axiom(chao_molhado,[goteira]).
```

O meta-interpretador abdutivo

```
sldnfa([],R,R,N,N).
sldnfa([P|Ps],R1,R2,N1,N2) :- % passo de resolução
   axiom(P,Q),
   append(Q,Ps,Qs),
   sldnfa(Qs,R1,R2,N1,N2).
sldnfa([P|Ps],R1,R2,N1,N2) :- % passo de abdução
   abducible(P),
   check_naf(N1,[P|R1]),
   sldnfa(Ps,[P|R1],R2,N1,N2).
sldnfa([neg(P)|Ps],R1,R2,N1,N2) :- % negação por falha
   naf([P],R1),
   sldnfa(Ps,R1,R2,[[P]|N1],N2).
check_naf([],_).
check_naf([N|Ns],R) :- naf(N,R), check_naf(Ns,R).
naf([P|_],R) :- not(resolve(P,R,_)).
naf([P|Ps],R) :- findall(X,(resolve(P,R,Q),append(Q,Ps,X)),Qs),
   check_naf(Qs,R).
resolve(P,_,Q) :- axiom(P,Q).
resolve(P,R,[]) :- member(P,R).
```

Considerações

Eshghi foi o primeiro a mostrar que planejamento é uma tarefa abdutiva.

Sejam β uma meta de planejamento e $\alpha \rightarrow \beta$ uma lei causal, sendo α uma ação e β um efeito. Então, supondo que a ocorrência de α seja um fato abdutível, a explicação abdutiva $\{\alpha\}$ para a observação $\{\beta\}$ é um plano para alcançar β.

A seguir, apresentamos o formalismo que será usado para a especificação lógica de problemas de planejamento abdutivo.

Parte IV

Cálculo de Eventos

Cálculo de Eventos

O cálculo de eventos [Kowaski & Sergot] é um formalismo para raciocínio sobre ações e efeitos, que enfatiza a ocorrência de eventos no mundo, em vez das situações mundo, como faz o cálculo de situações [McCarthy & Hayes].

- Ontologia: eventos, fluentes, tempo
- Linguagem:

$initiates(a, f, t)$	ação a 'inicia' o fluente f no instante t
$terminates(a, f, t)$	ação a 'termina' o fluente f no instante t
$releases(a, f, t)$	ação a 'solta' o fluente f no instante t
$initially_p(f)$	o fluente f vale inicialmente
$initially_n(f)$	o fluente f não vale inicialmente
$happens(a, t_1, t_2)$	a ocorrência da ação a inicia-se em t_1 e termina em t_2
$holdsAt(f, t)$	o fluente f vale no instante t
$clipped(t_1, f, t_2)$	o fluente f deixa de valer entre os instantes t_1 e t_2
$declipped(t_1, f, t_2)$	o fluente f passa a valer entre os instantes t_1 e t_2

Axiomatização do Cálculo de Eventos

$$holdsAt(F, T) \leftarrow \quad (EC1)$$
$$\quad initially_p(F) \wedge \neg clipped(0, F, T)$$

$$holdsAt(F, T) \leftarrow \quad (EC2)$$
$$\quad happens(A, T_1, T_2) \wedge initiates(A, F, T_1) \wedge (T_2 \prec T) \wedge \neg clipped(T_1, F, T)$$

$$\neg holdsAt(F, T) \leftarrow \quad (EC3)$$
$$\quad initially_n(F) \wedge \neg declipped(0, F, T)$$

$$\neg holdsAt(F, T) \leftarrow \quad (EC4)$$
$$\quad happens(A, T_1, T_2) \wedge terminates(A, F, T_1) \wedge (T_2 \prec T) \wedge \neg declipped(T_1, F, T)$$

$$clipped(T_1, F, T_2) \leftrightarrow \quad (EC5)$$
$$\quad \exists A, T_3, T_4[happens(A, T_3, T4) \wedge (T_1 \prec T_3 \prec T_4 \prec T_2) \wedge$$
$$\qquad (terminates(A, F, T_3) \vee releases(A, F, T_3))]$$

$$declipped(T_1, F, T_2) \leftrightarrow \quad (EC6)$$
$$\quad \exists A, T_3, T_4[happens(A, T_3, T4) \wedge (T_1 \prec T_3 \prec T_4 \prec T_2) \wedge$$
$$\qquad (initiates(A, F, T_3) \vee releases(A, F, T_3))]$$

$$happens(A, T_1, T_2) \rightarrow T_1 \preceq T_2 \quad (EC7)$$

Especificações em Cálculo de Eventos

Ações do domínio (robô)

$initiates(walk(X, Y), at(Y), T) \leftarrow holdsAt(at(X), T) \wedge X \neq Y$ (R1)
$terminates(walk(X, Y), at(X), T) \leftarrow holdsAt(at(X), T) \wedge X \neq Y$ (R2)

Estado inicial

$initially_p(at(p_0))$ (R3)
$initially_p(holding(b))$ (R3)

Estado meta

$holdsAt(at(p_2), t_3)$

Plano

$\{happens(walk(p_0, p_1), t_1), happens(walk(p_1, p_2), t_2), t_0 \prec t_1, t_1 \prec t_2, t_2 \prec t_3\}$

Persistência temporal no Cálculo de Eventos

Dado um conjunto de fatos *happens* e $\prec$, representando um plano parcialmente ordenado, a partir dos axiomas (EC1)-(EC7) e (R1)-(R4), podemos determinar a validade dos fluentes do domínio em qualquer instante de tempo após a execução desse plano.

Por exemplo, dado o plano $\{happens(walk(p_0, p_1), t_1), happens(walk(p_1, p_2), t_2), t_0 \prec t_1, t_1 \prec t_2, t_2 \prec t_3\}$ podemos concluir tanto $holdsAt(p_2), t_3)$, que é um efeito da ação $walk(p_1, p_2)$, quanto $holdsAt(holding(b))$, que é uma propriedade que persiste no tempo desde o instante inicial.

A axiomatização do cálculo de eventos captura, essencialmente, a persistência temporal dos fluentes e, portanto, ao contrário do cálculo de situações, o cálculo de eventos não requer o uso de axiomas de quadro (*frame axioms*).

Persistência temporal no Cálculo de Eventos

A persistência temporal embutida na axiomatização do cálculo de eventos é baseada em quatro pressupostos básicos:

- nenhum evento ocorre além daqueles que são conhecidos
- nenhum evento afeta um fluente além daqueles que são conhecidos
- os fluentes persistem até a ocorrência de algum evento que os afete
- todo fluente é efeito de algum evento conhecido

Considerações

A seguir mostramos como podemos usar abdução para sintetizar planos (conjuntos de *happens* e $\prec$) no cálculo de eventos.

Parte V

Planejamento Abdutivo no Cálculo de Eventos

Especificação lógica de planejamento abdutivo

Domínio de planejamento

é uma conjunção finita de fórmulas da forma

$$\mathit{initiates}(\alpha, \phi, \tau) \leftarrow \beta$$
$$\mathit{terminates}(\alpha, \phi, \tau) \leftarrow \beta$$
$$\mathit{releases}(\alpha, \phi, \tau) \leftarrow \beta$$

onde

- β é da forma $(\neg)\mathit{holdsAt}(\phi_1, \tau) \wedge \cdots \wedge (\neg)\mathit{holdsAt}(\phi_n, \tau)$
- α é um termo não-variável denotando uma ação
- ϕ, $\phi_1, \ldots, \phi_n$ são termos não-variáveis denotando fluentes
- τ é um termo denotando um instante de tempo

Especificação lógica de planejamento abdutivo

Situação inicial

é uma conjunção finita de fórmulas da forma

$initially_n(\phi)$
$initially_p(\phi)$

onde

- ϕ é um termo livre de variáveis denotando um fluente, na qual cada fluente ocorre no máximo uma vez.

Especificação lógica de planejamento abdutivo

Meta

é uma conjunção finita de fórmulas da forma

$(\neg)\mathit{holdsAt}(\phi, \tau)$,

onde

- ϕ é um termo livre de variáveis denotando um fluente e τ é um termo constante denotando um instante de tempo.

Especificação lógica de planejamento abdutivo

Narrativa

é uma conjunção finita de fórmulas da forma

$happens(\alpha, \tau)$

$\tau_1 \prec \tau_2$

onde

- α é um termo livre de variáveis denotando uma ação
- τ, τ_1, τ_2 são termos constantes denotando instantes de tempo

Especificação lógica de planejamento abdutivo

Plano

Sejam

- Δ um domínio
- Σ uma situação inicial
- *EC* a conjunção dos axiomas do cálculo de eventos
- Γ uma meta de planejamento.

Um plano para atingir Γ é uma narrativa Π tal que

- $\textsc{Circ}[\Delta; \mathit{initiates}, \mathit{terminates}, \mathit{releases}] \wedge \textsc{Circ}[\Sigma \wedge \Pi; \mathit{happens}] \wedge \mathsf{EC} \models \Gamma$
- $\textsc{Circ}[\Delta; \mathit{initiates}, \mathit{terminates}, \mathit{releases}] \wedge \textsc{Circ}[\Sigma \wedge \Pi; \mathit{happens}] \wedge \mathsf{EC} \not\models \bot$

Especificação lógica de planejamento abdutivo

Circunscrição

A circunscrição de ϕ minimizando ρ, $\textsc{Circ}[\phi; \rho]$, é definida pela fórmula $\phi \wedge \neg \exists Q[\phi(Q) \wedge Q < \rho]$, onde $\phi(Q)$ é a fórmula obtida pela substituição de toda ocorrência de ρ em ϕ por Q.

Segundo *Shanahan*, para garantir a consistência da definição de plano, basta garantir que o domínio Δ seja livre de conflitos.

Consistência

Um domínio é livre de conflitos se, para todo par de fórmulas em Δ da forma $\mathit{initiates}(\alpha, \phi, \tau) \leftarrow \beta_1$ e $\mathit{terminates}(\alpha, \phi, \tau) \leftarrow \beta_2$, temos $\models \neg(\beta_1 \wedge \beta_2)$

Implementação de planejamento abdutivo

Para transformar a especificação lógica de planejamento abdutivo numa implementação prática, *Shanahan* propôs o uso de um meta-interpretador abdutivo especializado para o cálculo de eventos.

Compilação de cláusulas-objetos em meta-cláusulas

Considere a cláusula-objeto $\alpha_0 \leftarrow \alpha_1, \ldots, \alpha_n$

- caso geral

```
axiom(α0,[α1,...,αn]).
dem([G|Gs1]) :- axiom(G,Gs2), append(Gs2,Gs1,Gs3), dem(Gs3).
```

- especialização

```
dem([α0|Gs1]) :- dem([α1,...,αn|Gs1]).
```

Controle no meta-nível

Segundo *Levi & Ramundo*, a compilação preserva a semântica; mas, devido ao grau de controle extra disponível no meta-nível, uma série de manobras podem ser efetuadas durante a compilação das cláusulas-objetos. Por exemplo:

- evitar laços infinitos
- tratar conhecimento incompleto

Controle no meta-nível - exemplo

```
holdsAt(F,T) :-                                          (EC2)
    happens(A,T1,T2),
    initiates(A,F,T1),
    before(T2,T),
    not(clipped(T1,F,T)).
```

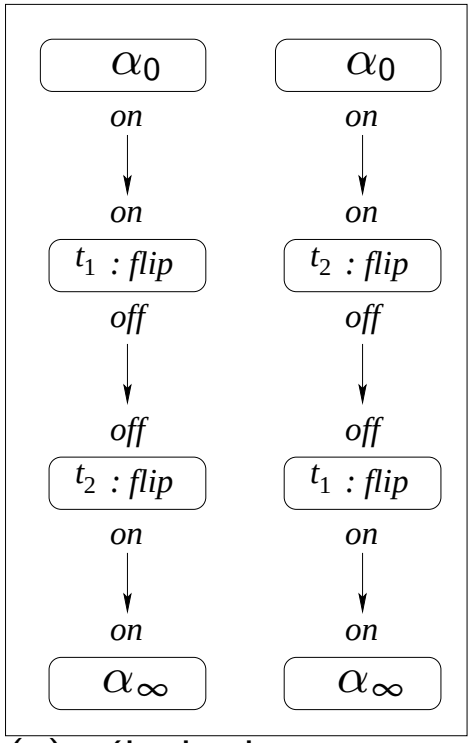
```
dem([holdsAt(F,T)|Gs1]) :-                               (EC2)
    axiom(initiates(A,F,T1),Gs2),
    axiom(happens(A,T1,T2),[]),
    axiom(before(T2,T),[]),
    not(dem([clipped(T1,F,T)])),
    append(Gs2,Gs1,Gs3),
    dem(Gs3).
```

- inversão de `happens` e `initiates` ⇒ redução do espaço de busca
- tratamento das precondições de `initiates` adiado ⇒ evita laços infinitos
- prova da negação de `clipped` antecipada ⇒ evita conflitos de ações

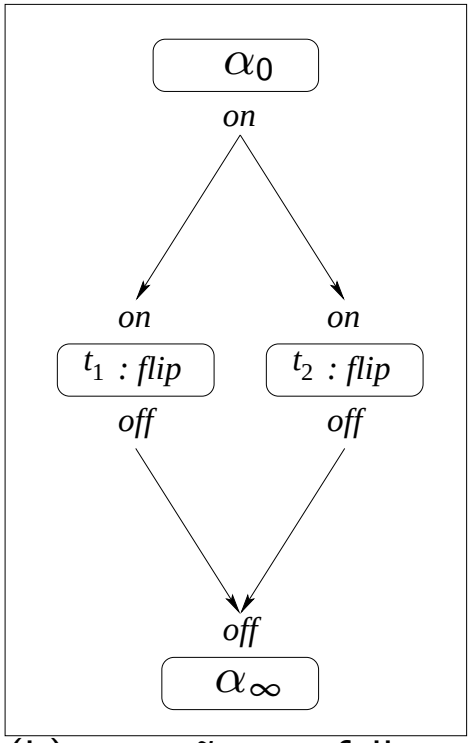
Tratamento de conhecimento incompleto

- A negação por falha é incapaz de tratar conhecimento incompleto, pois todo fato que não é explicitamente estabelecido como verdade é assumido como falso
- No raciocínio temporal com cálculo de eventos, temos informação incompleta sobre *before* e, portanto, usar negação por falha com esse predicado pode resultar em conclusões incorretas.
- Cálculo de eventos considera tempo linear, então, se não foi explicitamente estabelecido $before(t_1, t_2)$, ele assume $before(t_2, t_1)$

Lâmpada acesa ou apagada?



(a) cálculo de eventos



(b) negação por falha

- temos $happens(flip, t_1)$ e $happens(flip, t_2)$
- não temos $before(t_1, t_2)$, nem $before(t_2, t_1)$
- cálculo de eventos: $before(t_1, t_2)$ ou $before(t_2, t_1)$ vale $(t_1 \not< t_2 \rightarrow t_2 < t_1)$
- negação por falha: $\neg before(t_1, t_2)$ e $\neg before(t_2, t_1)$ valem

Cálculo de eventos para planejamento clássico

Conseqüências das suposições do planejamento clássico

- tempo atômico ($happens(a, t_1, t_2)/happens(a, t)$, elimina-se (EC7))
- efeitos determinísticos (*releases* não pode ser usado, altera-se (EC5))
- onisciência ($initially_n$ não é necessário, elimina-se (EC3), (EC4) e (EC6))

Cálculo de eventos simplificado

$$holdsAt(F, T) \leftarrow \quad \text{(SEC1)}$$
$$initially_p(F) \wedge \neg clipped(0, F, T)$$

$$holdsAt(F, T) \leftarrow \quad \text{(SEC2)}$$
$$happens(A, T_1) \wedge initiates(A, F, T_1) \wedge (T_1 \prec T) \wedge \neg clipped(T_1, F, T)$$

$$clipped(T_1, F, T_2) \leftrightarrow \quad \text{(SEC3)}$$
$$\exists A, T[happens(A, T) \wedge (T_1 \prec T \prec T \prec T_2) \wedge terminates(A, F, T)]$$

Considerações finais

- a implementação do meta-interpretador abdutivo especializado para a axiomatização simplificada do cálculo de eventos é apresentada em [Pereira & Barros]
- esse meta-interpretador é capaz de sintetizar planos de ordem parcial para problemas de planejamento clássico especificados em cálculo de eventos.
- planejamento abdutivo no cálculo de eventos é isomorfo a planejamento de ordem parcial no espaço de estados [Pereira & Barros]

Referências

- (Cox & Pietrzykowsky) *Causes for events: their computation and applications*, CADE, 1986.
- (Levi & Ramundo) *A formalization of metaprogramming for real*, ICLP, 1993.
- (Kowalski & Sergot) *A logic-based calculus of events*, New Generating Computing, 1986.
- (McCarthy & Hayes) *Some filosophical problems from the standpoint of Artificial Intelligence*, Machine Intelligence, 1969.
- (Peirce) *Collected papers of Charles Sanders Peirce*, Harvard, 1931-1958.
- (Pereira & Barros) *Planejamento abdutivo no cálculo de eventos*, IME-USP, 2002.
- (Shanahan) *An abductive event calculus planner*, JLP, 2000.