Transformações de Programas

- Noções básicas
- Computação Parcial
- Unfolding
- Folding

Optimização de programas com transformações *source-to-source*. Uso da chamada tradução (compilação) horizontal de programas.

Transformação de programas

Em programação lógica a ideia base da transformação de programas é a de usar resolução para a partir dos programas originais gerar outros programas. Neste sentido dizemos que a transformação é horizontal pois tanto o programa fonte como o programa alvo são escritos na mesma linguagem.

Esta noção de transformação de programas sugere que a partir de especificações possamos gerar os "verdadeiros" programas. Assim, temos uma linguagem (a lógica) com alguns graus de liberdade e suficientemente expressiva para descrever (especificar) os nossos problemas. A questão de eficiência é resolvida por análise e transformação das especificações em programas eficientes.

Uma das técnicas de transformação de programa consiste em, usando resolução, especializar um programa para um determinado padrão de queries. Outra resume-se à identificação de padrões de conjunções de literais no corpo de regras e substituição destes por uma invocação a um novo predicado. Finalmente, incluída nestas duas, existe a ideia de que em "compile-time" (tempo de compilação) podemos fazer computações (i.e. passos de resolução) que normalmente se fazem em "run-time" (tempo de execução).

Como a linguagem usada é a lógica de Horn e as transformações são feitas à custa de passos de resolução, podemos usar a noção de consequência lógica para provar correcção das nossas transformações.

Especialização de Programas (*Partial Evaluation*)

A técnica da computação parcial tem como objectivo especializar programas por forma a que estes sejam mais eficientes para uma classe específica de queries.

Exemplo: testar se uma lista está ordenada.

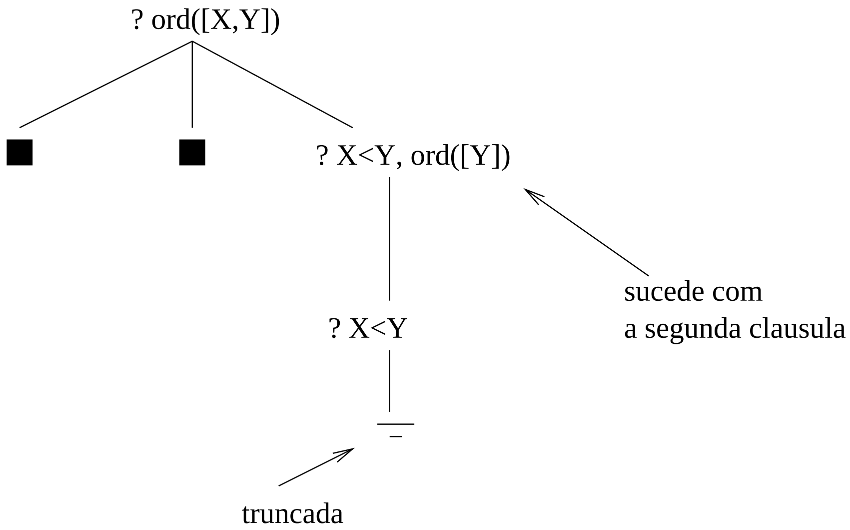
```
ord([]).
ord([U]).
ord([U,V|W]) :- U < V, ord([V|W]).
```

Supor a classe de queries que determinam se as listas com dois elementos estão ordenadas. Esta classe pode ser representada pela query ?ord([X, Y]).

O método de especialização consiste em gerar computações usando esta query e este programa. Cada computação (ou seja ramo da árvore SLD) é efectuada completamente ou truncada prematuramente. O resultado destas computações é uma (parcialmente gerada) árvore SLD de profundidade finita.

Para cada computação (ramo) desta árvore extraímos uma nova cláusula. O conjunto destas cláusulas forma o nosso novo programa já especializado.

Especialização de Programas (exemplos)



As computações parcialmente geradas são denominadas *resultantes*.

Para cada ramo gera-se uma nova cláusula seguindo as seguintes regras: (sendo ?q a query e ?calls a subquery truncada)

- se o resultante termina em sucesso infere-se $\forall \mathsf{q}\theta$
- se o resultante termina em falhanço não inferimos nenhuma cláusula
- se termina truncado infere-se a cláusula $\forall(\mathsf{q} \leftarrow \mathsf{calls})\theta$

No nosso exemplo especializamos o programa original num programa com a única cláusula:

```
ord([X,Y]) :- X < Y.
```

Outro Exemplo

Com o predicado append.

? append([X,Y],Zs,W)

W/[X|W1]

? append([Y],Zs,W1)

W1/[Y|W2]

? append([],Zs,W2)

W2/Zs

□

Para a query generalizada em questão infere-se a cláusula:

```
append([X,Y],Zs,[X,Y|Zs]).
```

Assim para o novo programa append, responder à query em questão exige menos passos de resolução do que usando o programa original.

Unfolding

A técnica de *unfolding* faz transformação de programas usando especialização de calls no corpo das cláusulas de um dado programa.

Do programa:

```
A :- ..., C , ...

C :- corpo_1
.
.
C :- corpo_n
```

Obtém-se o programa

```
A :- ...,  corpo_1 , ...
.
.
A :- ...,  corpo_n , ...

C :- corpo_1
.
.
C :- corpo_n
```

O novo programa é obtido pelas sucessivas aplicações do processo de *unfolding* na cláusula que define A com a subquery C contida no corpo dessa cláusula. Cada nova cláusula de A é obtida processando a subquery C com o conjunto de cláusulas que define C.

Assim, o processo de *unfolding* consiste num passo de resolução entre as subqueries contidas no corpo de uma cláusula com o conjunto de cláusulas que define o predicado dessas subqueries.

Unfolding (Exemplos)

Considere o seguinte programa que verifica pares consecutivos numa lista.

```
P1:   cp(X,Y,[X,Y|W]).
P2:   cp(X,Y,[Z|W]) :- cp(X,Y,W)
```

Fazendo unfold com P2 e a sua única subquery com ambas as cláusulas de cp obtemos duas novas cláusulas:

```
C1:   cp(X,Y,[Z,X,Y|W]).
C2:   cp(X,Y,[Z1,Z2|W]) :- cp(X,Y,W)
```

assim, C1, C2 podem substituir P2 ficando o novo programa:

```
P1:   cp(X,Y,[X,Y|W]).
C1:   cp(X,Y,[Z,X,Y|W]).
C2:   cp(X,Y,[Z1,Z2|W]) :- cp(X,Y,W)
```

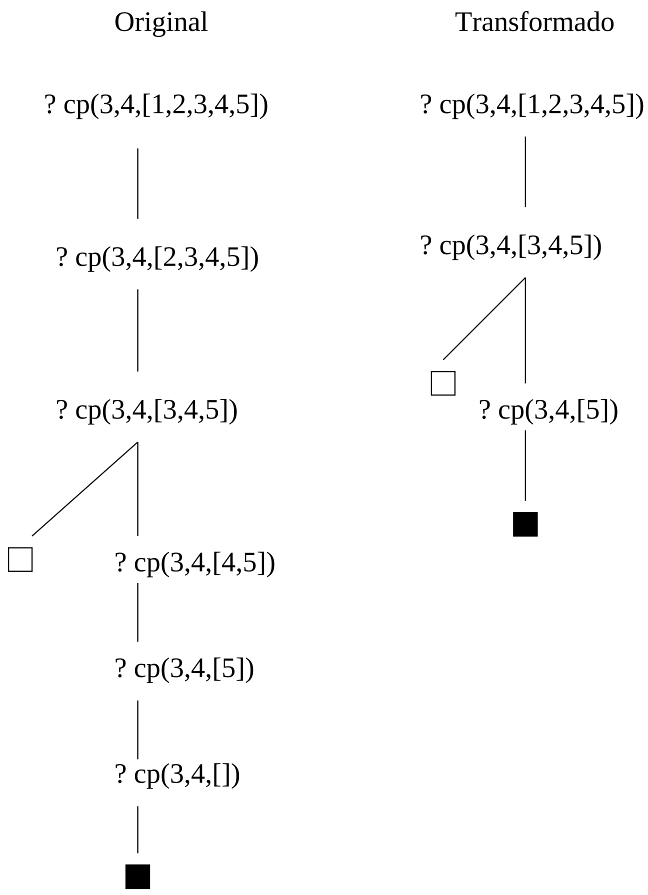
Notar que *unfolding* é um processo correcto (sound) (gera programas correctos). Isto é:

$$\forall \mathsf{q} \in \mathsf{B(P)} \qquad \mathsf{P} \cup \mathsf{U(Cl)} - \mathsf{Cl} \models \mathsf{q} \text{ se e só se } \mathsf{P} \models \mathsf{q}$$

Podemos facilmente provar esta afirmação pois unfolding resume-se ao uso de resolução.

Unfolding (cont)

Como no caso de especialização de programas, o objectivo de unfolding é o de fornecer computações mais curtas. No entanto não é linear que este efeito permita obter maior eficiência. Esta eficiência está intimamente relacionada com a escolha das regras de computação, tanto as usadas durante o processo de unfolding como as usadas durante a própria execução do programa.



Em geral, as transformações de programas fazem uso de vários passos de unfolding. Estes podem ser aplicados tanto às cláusulas do programa fonte como às cláusulas geradas pelo próprio processo de unfolding.

Folding

O processo de transformação *Folding* é essencialmente o inverso do processo de unfolding. No entanto a dedução do programa transformado faz-se a partir da *completion* (algo que já foi referido aquando do estuda da negação em programação em lógica) do programa e não apenas do programa original.

Essencialmente a partir de

$$\mathsf{A} \leftarrow \ldots \ \& \ \mathsf{C1} \ \& \ \mathsf{C2} \ \& \ \ldots$$

e

$$\mathsf{C} \leftarrow \mathsf{C1} \ \& \ \mathsf{C2}$$

obtemos o folding de Cl que é

$$\mathsf{A} \leftarrow \ldots \ \& \ \mathsf{C} \ \& \ \ldots$$

assumindo que esta cláusula é única que define C.

Tipicamente folding é usado conjuntamente com unfolding. Um problema que surge com a conjugação destas duas técnicas é que gera uma incompleta. Isto é, podemos ter $P1 \models P2$ mas não possível, por folding e unfolding, obter $P1$ usando $P2$ ou vice versa.

Exemplo

```
1)   A :- B, C.
2)   C :- D1.
3)   C :- D2.
4)   E :- B, D2.
5)   D1.
6)   D2.
```

Fazendo unfold de 1) e 2) obtém-se

```
U1)  A :- B, D1.
```

Unfold de U1) com 5) obtém-se

```
U2)  A :- B.
```

Unfold de 1) com 3) obtém-se

```
U3)  A :- B, D2.
```

Folding de U3) com 4) obtém-se

```
F1)  A :- E.
```

O programa final transformado é $\{\mathsf{U2}, \mathsf{F1}, 4)\}$

Conclusão

- Transformação de programas permite derivar programas mais eficientes a partir de programas originais, por simples aplicação de métodos baseados em resolução,
- Especialização de programas em relação a uma query abstracta é um exemplo destas técnicas,
- Unfolding pode ser visto como um caso particular de partial evaluation mas em que as queries a especializar estão no corpo das cláusulas,
- Folding é feito depois de sucessivos passos de Unfolding,
- Unfolding e Folding conjuntamente dá origem a um novo programa que calcula as mesmas soluções que o programa original, mas (por vezes!) de uma forma mais eficiente.
- Folding constrói cláusulas que herdam a recursividade dos seus predicados auxiliares.