SPIDER vs PROLOG: Механизъм на извод

Емилия Големанова

SPIDER vs PROLOG: Inference Mechanism: *Control Network Programming is a programming paradigm that integrates ideas from imperative programming, declarative programming, rule-based systems, nondeterministic programming and graphical programming. This report continues the comparison of the CNP language SPIDER and the logic programming language PROLOG started in [1] and [2]. The focus here is on the comparative investigation of their interpreters. Both algorithms are presented in a generic frame – reduction of goals. As a result of the juxtaposing their pseudo-codes the methodology for simulation PROLOG programs in CNP is developed [1], and the advantages of CNP are outlined.*

Key words: *Control Network Programming, CNP, SPIDER, Logic Programming, PROLOG, nondeterministic programming, declarative programming, interpreter of PROLOG programs, reduction of goals, Backtracking.*

ВЪВЕДЕНИЕ

Логическото програмиране (**ЛП**) е система за извод, базирана на метода на резолюцията, в подмножеството на Предикатното смятане от първи ред, наречено клаузи на Хорн [3]. В термините на математическата логика, клаузите са аксиоми, а въпросът е теорема, която абстрактният (недетерминиран) интерпретатор, реализиращ вградения в логическия език механизъм на извод, се опитва да докаже. Тъй като процесът на извод е с елементи на недетерминизъм, той всъщност е процес на търсене. Недетерминизмът е основна черта и на **Програмирането чрез Управляващи мрежи** (***Control Network Programming, CNP***), което е сравнително нов програмен стил, разработен от колектив, в който авторът участва. Изчислителният процес в CNP е също „търсене". Следователно, интерес представлява изследването на възможността за използване на „търсенето" в CNP за реализиране на „търсенето" в ЛП. Освен това двете програмни парадигми, макар и много различни, имат и други общи черти – декларативност, търсене в дълбочина чрез връщане при неуспех, потребителски средства за управление на механизма на извод. Би било полезно да се установят различията и да се проектират ефективните решения от една от парадигмите в другата, с цел развитието и усъвършенстването ѝ.

„Търсенето" в ЛП се извършва от интерпретатора на логически програми. Следователно реализирането на „търсенето" в тази област с средствата на CNP се свежда до симулиране поведението на интерпретатора на логически програми. За целта е необходимо да се направи сравнителен анализ на алгоритмите на двата интерпретатора, който е предмет на настоящия доклад, а самата симулация е представена в [1] и [2].

ИНТЕРПРЕТАТОР НА ЛОГИЧЕСКИ ПРОГРАМИ

Абстрактният интерпретатор на логически програми извършва изчисляването им, което може да бъде описано по следния неформален начин. Започва от някаква начална (конюнктивна) цел ***G*** и ако приключи връща резултат: успех или неуспех. Възможно е обаче и незавършващо изчисление, което не връща резултат. Псевдо-кодът на интерпретатора [4] е представен на фиг. 1.

Изчислението се извършва чрез редукция на цели. Текущата (обикновено конюнктивна) цел на всяка стъпка от изчислението се нарича резолвента. Всяка итерация на while-цикъла съответства на прилагане на правилото Modus Ponens и осъществява редукция на една цел от резолвентата. Цел от резолвентата и клауза от логическата програма се избират така, че главата на клаузата да е съпоставима с целта. Изчислението продължава с новата резолвента, получена чрез замяна на избраната цел с тялото на избраната клауза и прилагане на последваща

унификация. Процесът приключва, когато резолвентата е празна и следователно целта е „решена" от програмата.

Output: An instance of *G* that is logical consequence of *P*, or *no* otherwise
Algorithm: Initialize the resolvent to *G.*
while the resolvent is not empty *do*
choose a goal *A* from the resolvent
choose a (renamed) clause $A' \leftarrow B_1, \ldots, B_n$ from *P*
such that *A* and A' unify with mgu θ
(if no such goal and clause exist, exit the *while* loop)
replace *A* by $B_1, \ldots, B_n$ in the resolvent
apply θ to the resolvent and to *G*
if the resolvent is empty, *then* output *G,* else *output* no.

Фиг. 1. *Абстрактен интерпретатор на логически програми [4]*

В кода на интерпретатора от Фиг.1 има два недетерминирани избора. Първият е целта за редуциране от резолвентата, а вторият – клаузата, редуцираща тази цел. Понятието „недетерминиран избор" се използва при много изчислителни модели (краен автомат, Тюринг-машина) и се е доказало като мощна теоретична концепция. Разбира се, няма реална машина, която директно да интерпретира тази концепция. Но тя може да бъде апроксимирана по подходящ начин, така както е направено в конкретните езици за Логическо програмиране. Подходът за разрешаване на недетерминизма разграничава последователните от паралелните логически езици. Алтернативните избори, които трябва да направи абстрактният интерпретатор, имплицитно дефинират дърво на търсене (SLD-дърво), чиито възли са резолвенти. То може да бъде обходено в дълбочина (последователен логически език) или в ширина (паралелен логически език). Механизмът на изпълнение в PROLOG се получава от абстрактния интерпретатор от Фиг.1 чрез избор на най-лявата цел, последователно търсене на унифицируема клауза и *Backtracking*. Следователно резолвентата е организирана като стек (извлича се целта от върха на стека и се редуцира като се записва в началото му нейните подцели), а недетерминираният избор на клауза се реализира чрез последователно търсене и *Backtracking*. В термините на Логическото програмиране, PROLOG-изчислението на целта ***G*** е обхождане в дълбочина на конкретното дърво на търсене, получено чрез избор на най-лявата цел в резолвентата.

Така, интерпретаторът на PROLOG е конкретна реализация на абстрактния интерпретатор от Фиг.1, представен чрез булевата функция ***Search*** от Фиг. 2 за изпълнение на списък от цели (модифициран вариант на процедурата *execute* от [5]).

Необходимо е да се отбележи, че в представения псевдо-код не е експлицитно описан механизмът на унификацията и търсенето на повече от едно решения. При манипулиране със списъка *GoalList*, представляващ резолвентата, се използват стандартните операции *empty*, *head*, *tail* и *append*. Когато ***Search*** на дадено рекурсивно ниво завършва с неуспех, изпълнението продължава на предходното рекурсивно ниво с изпробване на следващата унифицируема клауза.

```
Search(GoalList):boolean;
Var     Goal: goal;
        OtherGoals, SubGoals, NewGoals: list of goals;

begin
 if empty(GoalList) then return true
 else
        begin
         Goal:=head(GoalList);
         OtherGoals:=tail(GoalList);
         while Goal has unifiable clauses
          begin
           SubGoals:=body of unifying clause;
           NewGoals:=append(SubGoals, OtherGoals);
           if Search(NewGoals) then return true;
          end;
         return false;
        end;
end;
```

Фиг. 2. *Интерпретатор на PROLOG-програми*

По този начин PROLOG систематично изследва всички възможни алтернативни пътища в дървото на търсене.

ИНТЕРПРЕТАТОР НА CNP-ПРОГРАМИ

Изчислението в CNP също може да се представи като редукция на цели. Началната цел е „изпълнението/изчислението" на началното състояние на главната подмрежа. Тя се редуцира до последователност от подцели, представляващи етикетните компоненти по текущо изпробваната стрелка от това състояние и състоянието, към което тя сочи, т.е. цел в CNP е или *примитив*, или *системен примитив CALL*, или *състояние* (обикновено или системно). Всъщност потребителският примитив и примитивът CALL могат да се разглеждат като частен случай на *състояние*. По-точно:

- *примитив* е **състояние „с параметри" с една излизаща стрелка**, описваща детерминирано изчисление;
- системният примитив за извикване на подмрежа *CALL* е също **състояние „с параметри"**, тъй като може да се разглежда като преход към състояние, но с друга област на действие на данните.

Базовото поведение на интерпретатора на CNP (без динамично управление на механизма на извод) може да бъде описано (в съответствие с въведената терминология) чрез следната булева функция ***Search*** от Фиг.3.

```
Search(GoalList):boolean;
Var     Goal: goal
        OtherGoals, SubGoals, NewGoals: list of goals

begin
 Goal:=head(GoalList);
 OtherGoals:=tail(GoalList);
 case type of Goal
        state:
                begin
                 while Goal has unexplored arrows
                  begin
                   SubGoals:=next arrow;
                   NewGoals:=append(SubGoals, OtherGoals);
                   if Search(NewGoals) then return true;
                  end;
                 return false;
                end;
        CALL v:
                begin
                 NewGoals:=append(v, OtherGoals);
                 return Search(NewGoals);
                end;
        primitive:      return (execute(Goal) and Search(OtherGoals));
        RETURN:         return Search(OtherGoals);
        STOP:           return false;
        FINISH:         return true;
 end;           // case
end;            // Search
```

Фиг. 3. *Интерпретатор на CNP-програми*

Всяка итерация на *while*-цикъла при редукцията на цел-състояние съответства на изпълнение на излизаща стрелка. В общия случай, стрелките се изпробват по реда на дефинирането им. Изчислението продължава с нова последователност от цели (***NewGoals***), получена чрез замяна на състоянието с подцелите от избраната стрелка.

При редукцията на *цел-CALL*, изчислението продължава с нова последователност от цели (***NewGoals***), получена чрез замяна на системния примитив *CALL* със цел-състояние, което съответства на входната точка на извикваната подмрежа.

Редукцията на *цел-потребителски примитив* се състои в неговото изпълнение, което може да бъде успешно или неуспешно. При успех, изпълнението продължава със следващите цели от стрелката.

RETURN е винаги изпълнена цел и изпълнението продължава с ***OtherGoals***, а *STOP* е неизпълнена цел (локален неуспех).

Процесът приключва, когато е достигнато състояние *FINISH*, което се интерпретира като глобален успех. Връщането на стойност true на текущото рекурсивно ниво води до успешно приключване и на всички по-горни рекурсивни нива на ***Search***.

ИНТЕРПРЕТАТОР НА PROLOG-ПРОГРАМИ VS ИНТЕРПРЕТАТОР НА CNP-ПРОГРАМИ

Сравняването на псевдо-кодовете от Фиг. 2 и Фиг. 3, т.е. съпоставянето на механизмите на извод в PROLOG и CNP, води до следните съответствия:

- кодът, редуциращ *цел* в PROLOG съответства на кода за редуциране на *състояние* в CNP;
- „*клауза*" в PROLOG съответства на „*стрелка*" в CNP.

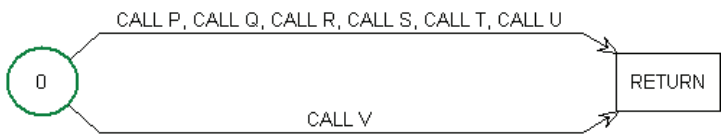
Имайки предвид обаче, че целите в PROLOG представляват извиквания на предикати, т.е. продукции с аргументи [6], то „цел" в PROLOG съответства не на „обикновено" състояние, а на „състояние с аргументи" в CNP. А както вече бе споменато по-горе, изпълнение на „състояние с аргументи" е еквивалентно на извикване на подмрежа с входна точка това състояние. Друга причина за еквивалентност на „цел в PROLOG" и „извикване на подмрежа в CNP" е фактът, че предикатите в една PROLOG-програма могат да бъдат извиквани многократно, но в различен ред (в различни клаузи). Следователно изпълнение на цел в PROLOG съответства на изпълнение на системния примитив CALL в CNP.

Така, **дефиницията на предикат в PROLOG се симулира в CNP с дефиниция на подмрежа с единствено състояние, от което излизат толкова стрелки, колкото са клаузите, описващи предиката и завършващи с преход към RETURN**. Като илюстрация на това правило, на Фиг.4 е моделиран следният фрагмент от PROLOG-програма (модифициран вариант от [5]):

C:-P, Q, R, S, T, U.
C:-V.

където P, Q, …, V са обозначения на предикати.

SUB **C**;



Фиг. 4. *Моделиране на предикат в CNP*

В допълнение на дефинираните по-горе съответствия могат да се отбележат и следните разлики (предимства на CNP):

- Индикатор за успешно изпълнение на една логическа програма е достигането до празна резолвента, докато при CNP – изпълнението на *цел-FINISH*. Това позволява прекратяване на изпълнението от неглавна подмрежа (съдържаща състояние FINISH) и води до оптимизиране на изчислителния процес и гъвкавост при разработване на приложения.
- В CNP целта е три вида – *примитив*, *CALL-примитив* и *състояние*. Това дава възможност за по-детайлно специфициране на решавания проблем и за създаване на подмрежи със сложна структура (граф).
- В CNP има възможност за гъвкаво управление на данните на различни нива – както глобално достъпни данни, така и локални (параметри на подмрежа и локални променливи). При PROLOG данните са само на локално ниво, използвайки механизма на предаване на параметри.
- При CNP „зад" цикъла *while* от Фиг.3 може да се „крие" много по-сложна логика (в случай на обработка на *цел-състояние*). Например CNP разполага със средства (управляващи състояния и системни опции) за

преподреждане или ограничаване на броя на изпробваните алтернативи. При PROLOG единственото управление на механизма на търсене е чрез „аварийно излизане от цикъла while" (при изпълнение на оператор cut).

ЗАКЛЮЧЕНИЕ

Въз основа на направените съответствия между „цел" в PROLOG и „цел" в CNP е разработена методика за симулация на базови PROLOG-програми в CNP, представена в [1]. Идеята е да се покаже, че по принцип тази симулация е възможна, поради сходство на интерпретаторите на двата езика и да се сравни полученият модел на PROLOG-програма със CNP-програма в нейния общ вид. Предложената методика е демонстрирана върху примерна PROLOG-програма и е направен изводът, че съответстващият й CNP-модел е много по-тривиален от общия вид на една CNP-програма. Освен това, тъй като универсалната Тюринг-машина е емулирана в PROLOG, доказвайки неговата Тюринг-пълнота [7][8], симулирайки поведението на PROLOG-програма със средствата на SPIDER, по транзитивен начин се доказва Тюринг-пълнотата и на SPIDER.

ЛИТЕРАТУРА

[1] T. Golemanov, K. Kratchanov, and E. Golemanova, "Spider vs. Prolog: Simulating Prolog in Spider," in 10th Int. Conf. on Computer Systems and Technologies (CompSysTech 2009), Rousse, Bulgaria, 2009, pp. II.9–1–II.9–7.

[2] E. Golemanova, K. Kratchanov, and T. Golemanov, "Spider vs. Prolog: Computation Control," in 10th Int. Conf. on Computer Systems and Technologies (CompSysTech 2009), 2009, pp. II.10–1–II.10–6.

[3] G. Luger, Artificial Intelligence: Structures and Strategies for Complex Problem Solving (6th Edition). Pearson, 2009.

[4] L. Sterling and E. Shapiro, The Art of Prolog: Advanced Programming Techniques (Logic Programming), Second Edi. Cambridge, Massachusetts: The MIT Press, 1994.

[5] I. Bratko, Prolog Programming for Artificial Intelligence, 4th ed. Pearson Education, 2011.

[6] C. Moss, Prolog++: The Power of Object-oriented and Pogic Programming. Addison-Wesley, 1994.

[7] D. Crookes, "Using Prolog to present abstract machines," ACM SIGCSE Bull., vol. 20 (3), pp. 8–12, 1988.

[8] M. Vlada, "Artificial Intelligence and Logic Programming: TUTORIAL: Problem_prolog," 2014. [Online]. Available: http://marinvlada.googlepages.com/Probleme_prolog.pdf.

За контакти:

гл. ас. д-р Емилия Големанова, Катедра "Компютърни системи и технологии", Русенски университет "Ангел Кънчев", тел.: 082-888 681, e-mail: EGolemanova@ecs.uni-ruse.bg

Докладът е рецензиран.