FLP Project 2 – Prolog

Deadline: 18/12/2012 18:00

1 Inleiding

Dit is de *vanilla* Prolog interpreter:

```
eval((G1,G2)) :- !,
  eval(G1),          % <<<
  eval(G2).
eval(true) :- !,
   true.
eval(G) :-
  clause(G,NG),
  eval(NG).
```

De gemarkeerde recursieve goal in de eerste clause staat niet in tail-positie. Dit betekent dat de vanilla meta-interpreter niet in constante stack space uitgevoerd kan worden. Dat is jammer en eigenlijk ook niet nodig.

We kunnen namelijk stack space ruilen voor heap space als volgt:

```
eval(G) :-
  eval(G,[]).

eval((G1,G2),Conts) :- !,
  eval(G1,[G2|Conts]).
eval(true, Conts) :- !,
   continue(Conts).
eval(G,Conts) :-
  clause(G,NG),
  eval(NG,Conts).

continue([]).
```

```
continue([G|Conts]) :-
  eval(G,Conts).
```

Nu hebben we een volledig tail-recursieve definitie. De extra recursieve call in de eerste clause vatten we in een datastructuur (een lijst) die de conjuncties voorstelt die we moeten uitvoeren als we klaar zijn met de huidige goal. Omdat we hiermee moeten verdergaan (Engels: *to continue*) wordt deze datastructuur ook de *continuatie* genoemd. Deze versie van de meta-interpreter noemen we de CPS (continuation-passing style)[1] versie van de vanilla meta-interpreter.

Merk op dat we stack space voor heap space geruild hebben omdat we nu voor de niet-tail call in plaats van een stack frame te alloceren op de call stack een term alloceren op de heap.

2 Het Project

In het project bouwen we in een aantal fases verder op de CPS meta-interpreter. In fases 0 tot en met 3, breid je de vorige fase telkens uit met nieuwe functionaliteit. De functionaliteit uit de vorige fases moet daarbij blijven werken!

De fasen sommen op tot meer dan 5 punten, maar er zijn maximaal 5 punten te verdienen. Je eindscore is dus $\min(\sum_i f_i, 5)$, waarbij f_i je score is op fase i ($i \in 0..4$). Je moet dus niet alle fasen maken om het maximum te halen.

2.1 Fase 0: Voeg built-ins toe (1/5 pt.)

Voeg ondersteuning toe voor een 10-tal built-ins, inclusief `=/2`, `is/1`, `(;)/2`, `>/2`, `writeln/1`, `read/1`, `call/1`.

Hierdoor moet het o.a. mogelijk worden om het fibonacci predikaat `fib/2` (zie definitie verderop) uit te voeren in je meta-interpreter.

2.2 Fase 1: `end/0` (1/5 pt.)

Volgens de Mayas eindigt de wereld op 21 december. Ondersteun een nieuwe operatie `end/0` in je meta-interpreter waarmee het einde nog sneller komt: zodra die wordt uitgevoerd eindigt de uitvoering van de query meteen met succes.

Bijvoorbeeld:

```
?- eval((writeln(a),end,writeln(b))).
a
true.

?- eval(p).
start(p)
start(q)
true.
```

[1] Niet te verwarren met Gangnam style!

voor dit programma:

```
p :-
  writeln(start(p)),
  q,
  writeln(end(p)).
q :-
  writeln(start(q)),
  end,
  writeln(end(q)).
```

2.3 Fase 2: `mark/1` en `unwind/0` (2/5 pt.)

De `mark(G)` built-in voert de goal `G` uit. De `unwind/0` built-in moet genest voorkomen binnen een `mark/1` en gooit het deel van de continuatie weg tot aan de binnenste `mark/1`.

Bijvoorbeeld:

```
?- eval(ex1).
a
b
d
true.

?- eval(ex2).
a
c
e
d
b
true.
```

voor dit programma:

```
ex1 :-
  writeln(a),
  mark(ex1_1),
  writeln(d).

ex1_1 :-
  writeln(b),
  unwind,
  writeln(c).
```

```
ex2 :-
  writeln(a),
  mark(ex2_1),
  writeln(b).

ex2_1 :-
  writeln(c),
  mark(ex2_2),
  writeln(d).

ex2_2 :-
  writeln(e),
  unwind,
  writeln(f).
```

Een `unwind/0` die niet voorkomt binnen een `mark/1` beeindigt het programma met een foutboodschap en failure:

```
?- eval(unwind).
ERROR: uncaught ‘unwind’
false.
```

2.4 Fase 3: `handle/2` en `raise/1` (2/5 pt.)

De nieuwe built-ins `handle/2` en `raise/1` breiden respectievelijk `mark/1` en `unwind/0` uit met een extra argument. Dit extra argument laat `raise/1` toe om een term door te geven aan `handle/2`. Als er geen `raise/1` optreedt binnen een `handle(G,X)`, dan wordt `X` niet gebonden bij het verlaten van `handle/2` (maar mogelijks kan `X` wel elders gebonden worden, b.v., in `G`).

Bijvoorbeeld:

```
?- eval(handle(raise(42),X)).
X = 42.

?- eval(handle(true,X)).
true.

?- eval(product([1,2,0,42],P)).
P = 0.
```

voor dit programma:

```
product(L,P) :-
  handle(product_(L,P),P).
```

```
product_([],1).
product_([X|Xs],P) :-
  X \== 0,
  product_(Xs,P1),
  P is X * P1.
product_([X|Xs],P) :-
  X == 0,
  raise(0).
```

Een `raise/1` die niet voorkomt binnen een `handle/1` beeindigt het programma met een foutboodschap en failure:

```
?- eval(raise(0)).
ERROR: uncaught ‘raise(0)’
false.
```

Een `raise/1` wordt niet opgevangen door een `mark/1`. Noch wordt een `unwind/0` opgevangen door een `handle/1`.

2.5 Fase 4: CPS Transformatie (2/5 pt.)

De *eerste Futamura projectie* stelt dat de specialisatie van een meta-interpreter voor een gegeven invoerprogramma een “executable” oplevert. In ons geval levert de specialisatie van de vanilla meta-interpreter voor een gegeven Prolog programma P dat programma P terug op. De specialisatie van de CPS interpreter voor P levert echter P' op, een variante van P in continuation-passing style.

Specialiseer manueel onderstaande programma’s naar CPS vorm.

```
% Te specializeren top-level query:
%   ?- eval(fib(N,F)).
% kan je na specialisatie oproepen als:
%   ?- eval_fib(N,F).

fib(0,1).
fib(1,1).
fib(N,F) :-
  N > 1,
  N1 is N - 1,
  fib(N1,F1),
  N2 is N1 - 1,
  fib(N2,F2),
  F is F1 + F2.

%-------------------------------------------------
```

```
% Te specializeren top-level query:
%   ?- eval(main).
% kan je na specialisatie oproepen als:
%   ?- eval_main.

main :-
  loop(work).

loop(G) :-
  call(G),
  loop(G).

work :-
  read(Number),
  work_(Number).

work_(0) :-
  writeln('I've seen enough'),
  end.
work_(N) :-
  N > 0,
  writeln('That's a nice number.').

%-------------------------------------------------

% Te specializeren top-level query:
%   ?- eval(product(L,P)).
% kan je na specialisatie oproepen als:
%   ?- eval_product(L,P).

product(L,P) :-
  handle(product_(L,P),P).

product_([],1).
product_([X|Xs],P) :-
  X \== 0,
  product_(Xs,P1),
  P is X * P1.
product_([X|Xs],P) :-
  X == 0,
  raise(0).
```

3 Evaluatie

Je project wordt beoordeeld op een aantal verschillende criteria:

- correcte werking
- verzorgdheid van de code
- gebruik van de meest geschikte Prolog-features

4 Indienen

Je werkt het project individueel uit en dient individueel in op Indianio. Je inzending bestaat uit een `zip`-bestand met volgende inhoud:

- `inzending.pl`: Een file met een enkel feit van vorm `inzending(LoginNaam,Fasen)` met `LoginNaam` je UGent login naam en `Fasen` de lijst van fases die je opgelost hebt.

  Bijvoorbeeld:

  ```
  inzending(tschrijv,[0,1,4]).
  ```

- `fase0123.pl`: je oplossing voor de fases 0 tot en met 3.
- `fase4.pl`: je oplossing voor de fase 4.

5 Vragen, Opmerkingen en Voorbeeldprogramma's

Met al je inhoudelijke vragen en opmerkingen kan je terecht op het Minerva-forum.

Als je testprogramma's voor je meta-interpreters schrijft, wees dan solidair en deel ze met je medestudenten via de studentenpublicatie op Minerva.