Interpreter języka Prolog
zaimplementowany w języku OCaml

Karol Stosiek* Szymon Fogiel†

Wrocław, 12 lutego 2008

E-mail: karol.stosiek@gmail.com
†*E-mail*: szymek.fogiel@gmail.com

Spis treści

1 Wstęp

2 Ogólny zarys

Napisany przez nas interpreter składnią odpowiada ogólnie przyjętej składni Prologa; staraliśmy się także zachować semantykę operatorów. W obecnej implementacji interpreter zawiera obsługę predykatów; nie można korzystać z dyrektyw ani gramatyk bezkontekstowych.

2.1 Ładowanie bazy wiedzy

Interpreter przyjmuje na wejściu bazy wiedzy, według których będzie przeprowadzał wnioskowanie. Każda baza wiedzy jest reprezentowana przez osobny plik, którego nazwa jest podawana jako argument wywołania interpretera.

Tym samym, uruchomienie intepretera z bazami zawartymi w plikach db1.opl, db2.opl, db3.opl będzie wyglądało następująco:

```
opl.exe db1.opl db2.opl db3.opl
```

Interpreter wczytuje bazy wiedzy w kolejności podanej przy wywołaniu i dołącza listę predykatów zawartych w danej bazie do listy dotychczas wczytanych predykatów, tworząc tym samym jednolitą bazę wiedzy.

2.2 Formułowanie zapytań

Po wczytaniu baz wiedzy, interpreter przechodzi do trybu zapytań. Szczegółowa składnia zapytań znajduje się w dodatku B. 6; symbolem startowym gramatyki w tym wypadku jest `<query>`.

Uwaga:

1. Interpreter zezwala na zadawanie zapytan postaci $X = f(X)$. W tym wypadku zostanie wypisane podstawienie $X = f(X)$.
2. Interpreter nie zezwala na uzycie funktorów bezargumentowych.

3 System wnioskowania

System wnioskowania działa w dość prosty sposób. Interpreter zapytany o jakąś relację zaczyna przeglądanie po kolei bazy wiedzy. Stara się zunifikować zapytanie z faktem lub następnikiem implikacji w bazie danych.

Jeśli zapytanie zostało zunifikowane z faktem to zwrócona zostaje odpowiedź („Yes”) oraz podstawienie które zostało użyte do unifikacji.

Jeśli zapytanie zostało zunifikowane z następnikiem implikacji to program stara się udowodnić poprzednik implikacji, korzystając już z podstawienia które zostało wygenerowane przy unifikacji z następnikiem. Proces jest rekurencyjny, tzn. podtermy są interpretowane w taki sam sposób jak termy do których należą.

W trakcie wnioskowania zapamiętywane są inne możliwe ścieżki dowodu. Po zwróceniu wyniku użytkownik jest pytany, czy chce szukać innych rozwiązań.

W wypadku natrafienia na operator odcięcia program nie zapamiętuje dalszych ścieżek obliczeń, w ten sposób można przerwać obliczenia, które mogą trwać w nieskończoność (do przepełnienia stosu).

4 Budowa interpretera

4.1 Podział logiczny

Projekt składa się z kilku modułów oraz jednego pliku "'głównego"', w którym znajduje się główna pętla programu.

- **Parser:**
  Moduł ten zawiera implementację parsera języka *Prolog* oraz udostępnia funkcje służące do parsowania zarówno zapytań, jak i plików z bazami danych.
- **Lexer:**
  Wykorzystywany przez moduł parsujący do dzielenia zadanego wejścia na tokeny.
- **Unificator:**
  Jednostka ta udostępnia zestaw funkcji pozwalających na unifikowanie termów oraz udostępnia szereg funkcji pomocniczych do pracy na termach.
- **Evaluator:**
  Moduł `Evaluator` jest podstawową jednostką interpretującą programy (zapytania) w programie. Udostępnia funkcję służącą do interpretacji abstrakcyjnych drzew rozbioru, zwróconych przez funkcje z modułu `Parser`.
- **Types:**
  Moduł zawierający deklaracje typów potrzebnych do konstrukcji abstrakcyjnych drzew rozbioru.
- **Opl:**
  Jednostka kompilacji, będąca zarazem interpreterem samym w sobie.

Część z modułów nie ma osobno zdefiniowanego interfejsu (sygnatury); jest tak dlatego, iż upubliczniany jest pełen zakres funkcji zaimplementowanych w tych modułach.

4.2 Podział fizyczny

- `opl.ml`:
  Plik z główną pętlą programu.
- `evaluator.ml`:
  Plik z implementacją modułu `Evaluator`.
- `evaluator.mli`:
  Plik z interfejsem modułu `Evaluator`.
- `unificator.ml`:
  Plik z implementacją modułu `Unificator`.

- `unificator.mli`:
  Plik z interfejsem modułu `Unificator`.

- `lexer.mll`:
  Plik z wejściem dla generatora lekserów `ocamllex`.

- `parser.mly`:
  Plik z wejściem dla generatora parserów `ocamlyacc`.

- `types.ml`:
  Plik z implementacją modułu `Types`.

W plikach `*.bat` znajdują się proste pliki wsadowe systemu *Windows*, dzięki którym można skompilować projekt zarówno w trybie *debugowania* (pliki `debug.bat`, `undebug.bat`), jak i w zwykłym (plik `make.bat`). Jednocześnie w skład projektu wchodzą także skrypty kompilujące dla systemu *Linux* (pliki `makefile`, `OcamlMakefile`).

5 Dodatek A - operatory

Operatory infixowe:

Operator	Działanie
=	unifikacja termów, unifikuje termy jeśli to możliwe i zwraca podstawienie
\=	odpowiada na pytanie, czy termy się nie unifikują
is	ewaluuje prawą stronę i podstawia pod zmienną z lewej strony
==	sprawdza czy termy są identyczne
\==	sprawdza czy termy nie są identyczne
=:=	sprawdza czy wyrażenia arytmetyczne po obu stronach dają ten sam wynik
=\=	sprawdza czy wyrażenia arytmetyczne po obu stronach dają różny wynik
<	sprawdza czy wyrażenie arytmetyczne po lewej stronie daje mniejszy wynik
>	sprawdza czy wyrażenie arytmetyczne po lewej stronie daje większy wynik
<=	sprawdza czy wyr. arytmetyczne po lewej stronie daje mniejszy lub równy wynik
>=	sprawdza czy wyr. arytmetyczne po lewej stronie daje większy lub równy wynik
+	sumowanie wyrażeń arytmetycznych
-	odejmowanie wyrażeń arytmetycznych
*	mnożenie wyrażeń arytmetycznych
/	dzielenie wyrażeń arytmetycznych
//	dzielenie całkowite wyrażeń arytmetycznych
,	koniunkcja
;	alternatywa

Operatory bezargumentowe:

Operator	Działanie
! alternatywne	obcięcie, jest zawsze prawdziwy i powoduje, że nie są obliczane ścieżki wnioskowania
rel($a_1, a_2, ..., a_k$) "rel" na	zapytanie do bazy danych o prawdziwość relacji argumentach $a_1, a_2, ..., a_k$

6 Dodatek B - gramatyka

⟨sentence list⟩ → ⟨clause⟩ `.` ⟨sentence list⟩ | ⟨clause⟩ `.`
⟨query⟩ → ⟨clause⟩ `.`
⟨clause⟩ → ⟨head⟩ `:-` ⟨body⟩ | ⟨head⟩
⟨head⟩ → ⟨goal⟩
⟨body⟩ → ⟨goal⟩ `;` ⟨body⟩ | ⟨goal⟩ `,` ⟨body⟩ | ⟨goal⟩
⟨goal⟩ → ⟨term⟩

⟨term⟩ → ⟨term0⟩
⟨term0⟩ → ⟨term1⟩
⟨term1⟩ → ⟨term2⟩
⟨term2⟩ → ⟨term2⟩ `->` ⟨term3⟩ | ⟨term2⟩ `->` ⟨term3⟩ `:` ⟨term3⟩ | ⟨term3⟩
⟨term3⟩ → ⟨term4⟩
⟨term4⟩ → `not` ⟨term5⟩ | ⟨term5⟩
⟨term5⟩ → ⟨term5⟩ `=:=` ⟨term5⟩
| ⟨term5⟩ `==` ⟨term5⟩
| ⟨term5⟩ `=` ⟨term5⟩
| ⟨term5⟩ `\=` ⟨term5⟩
| ⟨term5⟩ `==` ⟨term5⟩
| ⟨term5⟩ `\==` ⟨term5⟩
| ⟨term5⟩ `is` ⟨term5⟩
| ⟨term5⟩ `=..` ⟨term5⟩
| ⟨term5⟩ `>=` ⟨term5⟩
| ⟨term5⟩ `<=` ⟨term5⟩
| ⟨term5⟩ `<` ⟨term5⟩
| ⟨term5⟩ `>` ⟨term5⟩
| ⟨term5⟩ `@=` ⟨term5⟩
| ⟨term5⟩ `@\=` ⟨term5⟩
| ⟨term5⟩ `@<` ⟨term5⟩
| ⟨term5⟩ `@>` ⟨term5⟩
| ⟨term5⟩ `@>=` ⟨term5⟩
| ⟨term5⟩ `@=<` ⟨term5⟩
| ⟨term6⟩
⟨term6⟩ → ⟨term6⟩ `::` ⟨term6⟩ | ⟨term7⟩
⟨term7⟩ → ⟨term7⟩ `+` ⟨term7⟩ | ⟨term7⟩ `-` ⟨term7⟩ | ⟨term8⟩
⟨term8⟩ → ⟨term8⟩ `rem` ⟨term8⟩
| ⟨term8⟩ `mod` ⟨term8⟩
| ⟨term8⟩ `divs` ⟨term8⟩
| ⟨term8⟩ `mods` ⟨term8⟩
| ⟨term8⟩ `divu` ⟨term8⟩
| ⟨term8⟩ `modu` ⟨term8⟩
| ⟨term8⟩ `/` ⟨term8⟩
| ⟨term8⟩ `//` ⟨term8⟩
| ⟨term8⟩ `*` ⟨term8⟩
| ⟨term8⟩ `>>` ⟨term8⟩

| ⟨term8⟩ `<<` ⟨term8⟩
| ⟨term8⟩ `**` ⟨term8⟩
| `^` ⟨term8⟩
| ⟨term8⟩ `/\` ⟨term8⟩
| ⟨term8⟩ `\/` ⟨term8⟩
| ⟨term8⟩ `\\` ⟨term8⟩
| ⟨term9⟩
⟨term9⟩ → `!` | ⟨term10⟩
⟨term10⟩ → ⟨term list⟩ | `(` ⟨term0⟩ `)` | ⟨string⟩ | ⟨variable⟩ | ⟨constant⟩
| ⟨functor name⟩ `(` ⟨arguments⟩ `)`

⟨term list⟩ → `[ ]` | `[` ⟨arguments⟩ `]` | `[` ⟨arguments⟩ `|` ⟨arguments⟩ `]`
⟨functor name⟩ → ⟨name⟩
⟨arguments⟩ → ⟨term0⟩ `,` ⟨arguments⟩ | ⟨term0⟩
⟨constant⟩ → ⟨name⟩ | ⟨number⟩
⟨name⟩ → ⟨name⟩
⟨number⟩ → ⟨float number⟩ | ⟨integer number⟩

⟨capital⟩ → [`A` − `Z`]
⟨small⟩ → [`a` − `z`]
⟨digit⟩ → [`0` − `9`]
⟨underline⟩ → `_`
⟨alpha⟩ → ⟨capital⟩ | ⟨small⟩ | ⟨digit⟩ | ⟨underline⟩
⟨word⟩ → ⟨small⟩ ⟨alpha⟩∗
⟨quoted name⟩ → `'` | [`^'`]+ | `'`
⟨symbol⟩ → `+` | `-` | `*` | `/ \` | `^` | `<` | `> =` | `~` | `:` | `? @` | `#` | `$` | `&`
⟨solo char⟩ → `!: ;` | `; ; [ ; ] ; ( ; ) ; , ; |`

⟨name⟩ → ⟨quoted name⟩ | ⟨word⟩ | ⟨symbol⟩+ | ⟨solo char⟩
⟨variable⟩ → (⟨capital⟩ | ⟨underline⟩) ⟨alpha⟩ ∗
⟨string⟩ → ⟨any printable character in double qoutes⟩

⟨sign⟩ → `+` | `-`
⟨exp⟩ → (`e` | `E`) ⟨sign⟩`?` digit`+`
⟨simple float⟩ → ⟨digit⟩∗ `.` ⟨digit⟩+
⟨simple integer⟩ → ⟨digit⟩+
⟨float number⟩ → ⟨sign⟩? ⟨simple float⟩ ⟨exp⟩?
⟨integer number⟩ → ⟨sign⟩? ⟨simple integer⟩ ⟨exp⟩?