Wydział Podstawowych Problemów Techniki
Politechnika Wrocławska

Implementacja abstrakcyjnej maszyny Warrena

Kajetan Bilski
Nr indeksu: 244942

Praca inżynierska napisana
pod kierunkiem
Dr Przemysława Kobylańskiego

Politechnika
Wrocławska
Wrocław 2020

Spis treści

Wstęp

Celem pracy było zaimplementowanie abstrakcyjnej maszyny Warrena dla czystego języka Prolog. Odpowiada on językowi trzeciego poziomu z opisu maszyny. Praca zawiera, poza opisem implementacji, analizę efektywności jej działania na wybranych przykładach programów. Oprócz tego praca zawiera kompilator służący do kompilacji czystego języka Prolog na instrukcje maszyny Warrena.
W wielu językach programowania, np. C/C++, kompilator zamienia kod języka na kod maszynowy, który może być wykonany przez system operacyjny. Prolog różni się pod tym względem, ponieważ jego skompilowany kod nie jest wykonywany przez system operacyjny, a przez maszynę Warrena (WAM). Prolog jest językiem programowania logicznego i jego działanie opiera się na udowodnieniu zapytania zadanego przez programistę za pomocą faktów i reguł (wbudowanych lub również danych przez programistę) i zwraca wynik oznaczający prawdziwość zapytania (czy da się je udowodnić). Prolog próbuje udowodnić zapytanie na wszystkie dostępne sposoby, które umożliwiają dane predykaty. Implementacja abstrakcyjnej maszyny Warrena musi zawierać całą tą funkcjonalność. Moja implementacja dla większej łatwości użytkowania i lepszego porównania z innymi implementacjami zawiera wbudowany kompilator, który w pojedynczym uruchomieniu może być używany razem z maszyną, ale może być też uruchomiony osobno. Oprócz tego sama maszyna zawiera wbudowane predykaty i instrukcje, które nie są częścią czystego Prologa, ale dają lepszy wgląd w jej działanie.

Wykorzystane technologie i instrukcja obsługi

Program działa w terminalu na linuksie (był testowany na Ubuntu w WSL). Do jego kompilacji zostały użyte: g++ 7.5.0, flex 2.6.4, GNU Bison 3.0.4 i GNU Make 4.1. WAM został napisany w C++ w standardzie C++14.

Sposoby użycia:

```
wam [<program>]
wam -c <input> [<output>]
wam -e <program> <query>
```

W pierwszym przypadku użycia `program` jest ścieżką do pliku z programem napisanym w języku Prolog. Program jest ładowany do pamięci, następnie WAM wyświtla "?-"i czeka na użytkownika, żeby wpisał zapytanie. Zapytanie musi być w jednej linii i kończyć się kropką. Po wykonaniu zapytania WAM resetuje swoją pamięć i oczekuje następnego zapytania od użytkownika i tak dalej w pętli aż użytkownik nie wyjdzię manualnie (ctrl+C).
W drugim przypadku użycia pobiera z pliku tektowego `input` program lub zapytanie i kompiluje go do listy instrukcji maszyny Warrena, które umieszcze w pliku tekstowym `output` jeśli został podany lub w `a.out` w przeciwnym wypadku. Kompilator rozróżnia program i zapytanie Prologa po tym, że zapytanie musi rozpoczynać się od "?-".
W trzecim przypadku użycia `program` i `query` są plikami tekstowymi zawierającymi instrukcje maszyny Warrena, takimi jakie można wygenerować używając opcji `-c`. WAM ładuje wykonuje te instrukcje pomijając kompilator.

WAM - Abstrakcyjna maszyna Warrena

WAM jest główną częścią programu i odpowida za przetwarzanie zapytań i programów dostarczanych jako lista instrukcji maszyny Warrena. Każde zapytanie składa się z termów następujących po sobie. Jeżeli maszynie uda się je wszystkie udowodnić, to zwraca wynik "prawda", w przeciwnym wypdaku zwraca "fałsz". Do kluczowych w implementacji operacji należą:

3.1 Unifikacja

Żeby udowodnić term z zapytania musi być on najpierw dopasowany do tzw. głowy predykatu. Polega to na sprawdzeniu, czy struktura oba termów jest taka sama i przypisaniu zmiennych tam gdzie to konieczne. W tym celu oba porównywane termy można przedstawić jako drzewa, gdzie dziećmi każdego wierzchołka są jego podtermy, a liśćmi są formuły atomowe. Wtedy oba drzewa się porównuje przechodząc po nich algorytmem przeszukania w szerz. Tam gdzie któryś z porównywanych wierzchołków jest nieprzypisaną zmienną, jest ona przypisywana do odpowiadającego termu z drugiego drzewa. Jeśli któreś porównywane wierzchołki różnią się to operacja $unify$ zwraca $fail$.

3.2 Stosowanie reguł

Term z zapytania może być udowodniony przez zunifikowanie go z termem faktu, ale może być też udowodniony jeśli da się go zunifikować z głową regułu, a następnie udowodnić warunki reguły. Termy warunków funkcjonują wtedy tak jak termy zapytania. Reguły mogą być wywoływane przez siebie na wzajem tak jak funkcje w innych językach. Ich implementacja wymaga wymiennych środowisk w których istnieją różne zmienne.

3.3 Nawroty

Jedną z ważniejszych cech Prologa jest jego niedeterminizm. Często jeden term można próbować udowodnić na wiele sposobów i maszyna musi być w stanie wykonać nawrót i spróbować odwołać się do innej klauzuli, jeśli obecna zwraca $fail$. Wymaga to tworzenia specjalnych punktów do których maszyna może wrócić, przywracając poprzedni stan swojej pamięci.

Struktura pamięci

W tym rozdziale opisane są wszystkie elementy pamięci maszyny. Zmienne reprezentujące elementy pamięci maszyny są zmiennymi globalnymi. Cała pamięć jest czyszczona (nie licząc części kodu) przed rozpoczęciem wykonywania każdego zapytania.

4.1 Komórka pamięci

Komórka pamięci to podstawowa składowa wielu elementów pamięci. Każda komórka ma jeden z dwóch typów. Komórka zmiennej zawiera tag, który może być REF lub STR, a także adres dowolnej innej komórki znajdującej się w pamięci. Komórka funktora zawiera reprezentację funktora f/n, gdzie f jest nazwą funktora, a n liczbą argumentów przez niego przyjmowanych. Domyślnie każda komórka jest komórką zmiennej o tagu REF i adresie pokazującym na samą siebie, czyli tzw. komórką nieprzypisaną. Nieprzypisana komórka odpowiada nieprzypisanej zmiennej.
W implementacji komórka jest reprezentowana przez własną klasę z polami `string tag` i `shared_ptr<Address> addr`.

4.2 `HEAP` - Sterta

Sterta to główny blok pamięci w którym przechowywane są komórki pamięci reprezentujące struktury i zmienne (w przeciwieństwie do rejestrów, w których wszystkie komórki są komórkami REF z adresami komórek ze sterty). Adres końca sterty jest pamiętany przez rejestr H. Rejestr H jest aktualizowany na końcu każdej instrukcji, która dodaje nowe komórki do sterty.
Sterta jest reprezentowana przez klasę `MemoryBloc`, która trzyma komórki w werktorze z przeładowanymi operatorami dostępu, które w przypadku próby dostępu do komórki o indeksie większym od długości wektora najpierw zapełnia brakujące pola nieprzypisanymi komórkami. Rejestr H jest reprezentowany przez liczbę naturalną równą długości sterty.

4.3 Rejestry tymczasowe

Rejestry zawierają komórki z adresami pokazującymi na stertę. Komórki w rejestrach służą jako wskaźniki do miejsc w stercie gdzie przechowywane są podtermy zawarte w obecnie ewaluowanym termie. Rejestry tymczasowe przestają być użyteczne po przejściu do następnego termu i są nadpisywane. Rejestry tymczasowe są przypisywane do podtermów przez kompilator.
Rejestry tymczasowe są reprezentowane przez klasę `MemoryBloc`, tak samo jak sterta.

4.4 Rejestry argumentów

Rejestry argumentów tak samo jak rejestry tymczasowe zawierają komórki będące wskaźnikami na stertę. Przechowują one adresy argumentów, czyli bezpośrednich podtermów ewaluowanego termu i służą do przekazywania argumentów przy wywołaniu instrukcji `call`. Są one ustawiane w trakcie ewaluowania termu z

zapytania i są one wczytywane do innych rejestrów w trakcie dopasowywania do głowy klauzuli.
Rejestry argumentów są reprezentowane przez klasę `MemoryBloc`.

4.5 Rejestry trwałe

Rejestry trwałe działają podobnie do rejestrów tymczasowych i rejestrów argumentów i służą do pamiętania zmiennych pojawiających się w wielu termach w zapytaniu lub regule. Rejestry trwałe są zapisywane w środowiskach i są wymieniane wraz ze zmianą aktywnego środowiska. Te rejestry są przydzielane do zmiennych przez kompilator.
Rejestry argumentów są reprezentowane przez klasę `MemoryBloc`.

4.6 `MODE` - Tryb maszyny

W momencie wykonywania zapytania maszyna w każdym momencie może być w jednym z dwóch trybów: *read* i *write*. Tryb jest ustawiany przez instrukcję `get_structure` i zmienia działanie instrukcji `unify_variable` i `unify_value`. Drugim rejestrem globalnym używanym tylko przez te instrukcje jest rejestr `S` przechowujący adres w stercie.
Zarówno `MODE` i `S` reprezentowane są przez liczby naturalne, przy czym *mode* może być w tylko dwóch stanach: 0 (*write*) i 1 (*read*).

4.7 `CODE` - Magazyn kodu

W magazynie kodu przechowywana jest załadowana lista instrukcji wygenerowana przez kompilator lub pobrana z pliku. Instrukcje programu i zapytanie ładowane są osobno przed rozpoczęciem wykonywania zapytania. Najpierw ładowane są klauzule wbudowane, które funkcjonują jak część programu, która jest niewidoczna w ładowanym pliku. Następnie ładuje program. Wszystkie etykiety są usuwane z kodu i pamiętane są osobno razem z numerami instrukcji następujących po nich. Na końcu ładowany jest kod zapytania i ustawiany jest rejestr `P` oznaczajacy miejsce w kodzie instrukcji obecnie wykonywanej i rejestr `CP` oznaczający miejsce w kodzie do którego maszyna ma wrócić po wywołaniu `proceed` lub `deallocate`. Rejestr `P` ustawiany jest na pierwszą instrukcję z zapytania, a `CP` na koniec kodu.
Magazyn kodu jest reprezentowany przez wektor wektorów stringów, gdzie każde pole zawiera osobno nazwę instrukcji i wszystkie argumenty jako stringi. Etykiety są pamiętane w `std::unordered_map` gdzie kluczem jest string będący nazwą etykiety, a warotścią jest liczba naturalna oznaczająca indeks instrukcji której odpowiada etykieta. Rejestry `P` i `CP` są reprezentowane przez liczby naturalne.

4.8 `TRAIL` - Ślad

Ślad służy do zapamiętywania kiedy nieprzypisane zmienne zostają przypisane tak, żeby w przypadku nawrotu dało się cofnąć odpowiednie przypisania. Przechowuje on adresy zmiennych w kolejności, w której zostają przypisane. Nowy element zostaje dodany na koniec śladu za każdym razem gdy zostaje wywołana operacja *bind*. W przypadku nawrotu i konieczności odłączania zmiennych rejestr `HB` pamięta wielkość sterty przed ostatnim punktem wyboru, żeby nie próbować odłączać zmiennych usuniętych przez nawrót.
Ślad jest reprezentowany przez wektor adresów przypisywanych komórek. Podczas nawrotu odpowiednia ilość elementów z końca śladu jest usuwana. Rejestr `HB` jest reprezentowany przez liczbę naturalną oznaczającą ilość komórek w stercie.

4.9 AND-STACK i OR-STACK - Stosy

AND-STACK jest stosem przechowującym środowiska. Środowisko przechowuje rejestry trwałe i punkt powrotu (CP) dla obecnie przetwarzanego zapytania lub reguły, a także indeks poprzedniego środowiska, które ponownie się uaktywni po dealokacji obecnego. Globalny rejestr E zawiera indeks obecnie aktywnego środowiska. OR-STACK jest stosem przechowującym punkty wyboru. Punkt wyboru zapamiętuje moment w którym maszyna dokonuje wyboru, w jaki sposób próbować udowodnić term z zapytania lub ciała reguły. W przypadku Niepowodzenia w ewaluacji maszyna wraca do ostatniego punktu wyboru i próbuje udowodnić term w inny sposób. Punkt wyboru przechowuje wartości wielu globalnych rejestrów w momenie jego utworzenia: E, CP, B, H. Oprócz tego punkt wyboru w momencie utworzenia zapamiętuje też długość śladu, rejestry argumentów (których ilość jest pamiętana w globalnym rejestrze num_of_args i numer instrukcji do której maszyna ma przejść w przypdaku konieczności powrotu do tego punktu. Każdy punkt wyboru chroni środowiska istniejące w momencie jego utworzenia przed usunięciem po dealokacji w razie konieczności ponownej aktywacji tego środowiska po nawrocie. Alternatywną metodą implementacji jest przechowywanie środowisk i punktów wyboru na jednym stosie [1]. Wtedy każdy punkt wyboru chroni przed usunięciem wszystkie środowiska pod nim na stosie i globalny rejestr B pamięta indeks na stosie aktywnego punktu wyboru. Ze względu na łatwość w implementacji zdecydowałem się na użycie dwóch stosów, a rejestr B służy do ochrony środowisk pamiętając aktywne środowisko w momencie utworzenia ostatniego punktu wyboru.
Oba stosy są reprezentowane przez wektory. Środowisko jest reprezentowane przez własną klasę z polami przechowującymi warotści rejestrów trwałych i rejestrów E i CP. W trakcie przełączania aktywnego środowiska wartości rejestrów trwałych są zapisywane w poprzednim środowisku, a na ich miejsce są wczytywane rejestry trwałe z nowego środowiska. Punkt wybory też jest reprezentowany przez własną klasę z polami E, CP, B, H, TR (długość śladu), BP (instrukcja do wykonania po nawrocie). Wszystkie te pola są liczbami naturalnymi.

Instrukcje Maszyny

Główna pętla progarmu polega na wywołaniu instrukcji `CODE[P]` i zwiększeniu `P` o 1 do momentu dojścia do końca kodu lub w trakcie wykonywania instrkcji zwrócenia $fail$. W pierwszym wypadku pojawia wykonywanie zapytania kończy się powodzeniem i użytkownik zostaje zapytany czy chce kontynuować. Jeśli wybierze "tak"to program wraca do wykonywania zapytania, tak jakby instrukcja zwróciła $fail$. Jeśli wykowana instrukcja zwróci $fail$, to jeśli jest aktywny punkt wyboru to wyknonywana jest operacja nawrotu i zapytanie wykonywane jest dalej. Jeżeli nie ma aktywnego punktu wyboru, to wykonywanie zapytania kończy się z wynikiem $fasz$.
Poniżej opisane są wszystkie obsługiwane instrukcje.

5.1 `put_structure` f/n,`X`i

Dodaje na koniec sterty komórkę STR i komórkę funktora f/n, której adres zawiera. Następnie kopiuje tą komórkę STR do rejestru o adresie `X`i.

5.2 `set_variable X`i

Dodaje na koniec sterty nieprzypisaną komórkę REF, a następnie umieszcza w rejestrze `X`i komórkę REF z jej adresem.

5.3 `set_value X`i

Dodaje na koniec sterty komórkę będącą kopią komórki z rejestru `X`i.

5.4 `get_structure` f/n,`X`i

Wywołuje $deref$(`X`i) i jeżeli otrzymany adres pokazuje na nieprzypisaną komórkę, to dodaje na koniec sterty komórkę STR i komórkę funktora f/n, której adres zawiera. Następnie wywołuje $bind$($deref$(`X`i),`H`) i ustawia tryb maszyny na $write$ i adres `S` na koniec sterty. W przeciwnym wypadku jeśli $deref$(`X`i) pokazuje na komórkę STR z adresem $addr$, to ustawia `S` na $addr + 1$ i tryb maszyny na $read$.

5.5 `unify_variable X`i

Jeśli maszyna jest w trybie $read$ to kopiuje komórkę z adresu `S` do rejestru `X`i.
Jeśli maszyna jest w trybie $write$ to dodaje na koniec stery nieprzypisaną komórkę i kopiuje ją do rejestru `X`i.

5.6 `unify_value` `X`i

Jeśli maszyna jest w trybie *read* to wywołuje $unify(\texttt{X}i, \texttt{S})$.
Jeśli maszyna jest w trybie *write* to kopiuje komórkę z rejestru `X`i na koniec sterty.

5.7 `put_variable` `X`i `A`j

Dodaje na koniec sterty nieprzypisaną komórkę, a następnie kopiuje ją do rejestrów `X`i i `A`j.

5.8 `put_value` `X`i `A`j

Kopiuje komórkę z rejestru `X`i do rejestru `A`j.

5.9 `get_variable` `X`i `A`j

Kopiuje komórkę z rejestru `A`j do rejestru `X`i.

5.10 `get_value` `X`i `A`j

Wywołuje $unify(\texttt{X}i, \texttt{A}j)$.

5.11 `call` *label*

label jest stringiem. Jeśli *label* jest postaci $*/n$, gdzie n jest liczbą naturalną, to ustawia `num_of_args`$= n$. Wyszukuje w kodzie wiersza p oznaczonego etykietą *label*. Jeśli takiej etykiety w kodzie nie ma to zwraca `fail`. W przeciwnym wypadku ustawia `CP`=`P` i `P`$= p - 1$.

5.12 `proceed`

Ustawia `P`=`CP`.

5.13 `allocate` N

Tworzy nowe środowisko *env* przechowujące obecne `E` i `CP` i wstawia je do stosu `and_stack` na indeksie $max(B,E) + 1$. Następnie przełącza się na środowisko *env*.
N jest argumentem oznaczającym ilość rejestrów argumentów, ale ponieważ rejestry są przechowywane w tablicy dynamicznej, w obecnej wersji ten argument nic nie robi.

5.14 `deallocate`

Ustawia `P`=`and_stack[E].CP`, a następnie przestawia aktywne środowisko na środowisko o indeksie przechowywanym przez obecnie aktywne środowisko.

5.15 `try_me_else` *label*

Tworzy nowy punkt wyboru gdzie `BP`= *label*. Następnie ustawia `B`=`E` i `HB`=`H`.

5.16 `retry_me_else` *label*

Przywraca stan maszyny do stanu bezpośrednio po utworzeniu ostatniego punktu wyboru. Także zamienia etykietę pamiętaną przez aktywny punkt wyboru na *label*. Nie zmienia rejestru `P`.

5.17 `trust_me`

Przywraca stan maszyny do stanu z przed utworzenia ostatniego punktu wyboru, usuwając go. Nie zmienia rejestru `P`.

5.18 Instrukcje specjalne

Niektóre instrukcje pojawiają się w kodzie generowanym przez samą maszynę przed załadowaniem programu, ale nie powinny być zawierane przez ładowany program.

5.18.1 `write`

Wypisuje na standardowe wyjście tektową reprezentacje termu na który pokazuje adres $deref(\texttt{A0})$. Jeśli otrzymany adres pokazuje na nieprzypisaną komórkę, wyświetlany jest jako $\_mn$, gdzie m oznacza block pamięci w której zanjduje się zmienna: H - sterta, X - rejestry tymczasowe, A - rejestry argumentów, Y - rejestry trwałe. n oznacza indeks w tym bloku pamięci. Jeżeli otrzymany adres pokazuje na komórkę STR, której adres pokazuje na komórkę funtora f/n, gdzie $n > 0$ to rekurencyjnie wypisywane są też podtermy.

Kompilator

Kompilator zawarty w programie służy do kompilowania programów i zapytań Prologa do instrukcji maszyny Warrena. Kompilator rozpoznaje czy na wejściu dostaje zapytanie czy program, po tym że zapytania zaczynają się od `?-`. Dla programu instrukcje generowane są dla każdej klauzuli osobno, a następnie łączone w ze sobą w całość. Nazwy struktur i zmiennych mogą zawierać wielkie i małe litery, cyfry i podkreślniki.
Kompilator akceptuje operator unifikacji, który nie należy do czystego Prologa: `X = Y` jest interpretowane jako term `=(X,Y)`.
Oprócz tego kompilator obsługuje listy. Pusta lista to `[]`, singleton `[X]` jest interpretowany jako `.(X, [])`. Dłuższe listy są konwertowane rekurencyjnie, np. `[X, Y, Z]` do `.(X, .(Y, .(Z, [])))`. Dopuszczalny jest też zapis `[X | Y]`, gdzie `Y` to ogon listy i jest konwertowany do `.(X, Y)`. Można też użyć zapis mieszany, np. `[X, Y | Z]`.

6.1 Gramatyka

Przez lekser dostarczane są 2 tokeny: `STRUCT` oznaczający nazwę struktury i `VAR` oznaczający nazwę zmiennej.
Schemat gramatyki:

```
program
| predicates
| ?- terms .
predicates
| predicates predicate
| predicate
predicate
| term :- terms .
| term .
terms
| terms , term
| term
term
| STRUCT ( terms )
| STRUCT
| VAR
| []
| [ terms ]
| [ terms | term ]
```

Przez lekser dostarczane są 2 tokeny: `STRUCT` oznaczający nazwę struktury i `VAR` oznaczający nazwę zmiennej.

6.2 Sposób alokacji pamięci

Alokowane są 3 typy rejestrów: tymczasowe (oznaczane przez `X`), argumentów (`A`) i trwałe (`Y`). Wszystkie typy rejestrów indeksowane są od 0.
Rejestry argumentów są przydzielane do bezpośrednich podtermów obecnie rozpatrywanego termu w kolejności ich występowania.
Rejestry tymczasowe są przydzielane do wszystkich podtermów w rozpatrywanym termie. Kolejność jest usutalana przeszukując nadrzędny term algorytmem BFS.
Rejestry trwałe są przydzielane wszystkim zmiennym, które w rozpatrywanym zapytaniu lub regule pojawiają się w wielu termach nadrzędnych. Przydzielane są w takiej kolejności, w jakiej pojawiają się ich drugie wystąpienia.

Porównanie z inną implementacją i analiza efektywności

W tym rozdziale wymienię kilka istniejących już implementacji Prologa (które zawierają WAM) i porównam szybkość działania jednej z nich z moją implementacją na wybranych testach.
3 z popularnych istniejących już implementacji Prologa to:
SWI-Prolog - dostępny od 1987 jest prawdopodobnie najpopularniejszą i najbogatszą w dodatkową funkcjonalność implementacją. Jest dostępny na Windowsa, Linuxa i Mac OS X.
GNU Prolog - dostępny od 1996 na Windowsa, Linuxa i Mac OS X.
YAP Prolog - dostępny od 1985 na Windowsa, Linuxa i Mac OS X. Jest to open-source'owa i działająca wyjątkowo szybko implementacja.
W tych implementacjach kompilator i WAM są połączone w jednej aplikacji. W moim rozwiązaniu są to 2 osobne aplikacje (co może ulegnąć zmianie).

Podsumowanie

TODO

Bibliografia

[1] H. Ait-Kaci. Warren's abstract machine. 1999.

Zawartość płyty CD

W tym rozdziale należy krótko omówić zawartość dołączonej płyty CD.