VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

PŘEKLADAČ JAZYKA PROLOG PRO .NET

PROLOG COMPILER FOR .NET PLATFORM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. PETR HALJUK**
AUTHOR

VEDOUCÍ PRÁCE — **doc. Dr. Ing. DUŠAN KOLÁŘ**
SUPERVISOR

BRNO 2017

Zadání diplomové práce/20031/2016/xhalju00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2016/2017

Zadání diplomové práce

Řešitel: **Haljuk Petr, Bc.**
Obor: Informační systémy
Téma: **Překladač jazyka Prolog pro .NET**
Prolog Compiler for .NET Platform
Kategorie: Překladače

Pokyny:

1. Prostudujte překlad jazyka Prolog do WAM - Warren Abstract Machine. Seznamte se i s dalšími přístupy k analýze a vyhodnocení jazyka Prolog.
2. Navrhněte interpret pro WAM na platformě .NET, navrhněte překladač jazyka Prolog i provázání celého souboru s platformou .NET. Implementujte reprezentaci datových struktur a unifikaci a substituci u termů.
3. Po konzultaci s vedoucím implementujte překladač jazyka Prolog do WAM a interpret pro WAM jako knihovnu pro .NET s vhodným API.
4. Na vhodně zvolené sadě příkladů (konzultujte s vedoucím) otestujte vaši aplikaci, včetně testů zátěžových.
5. Zhodnoťte přínos vaší práce, diskutujte možná rozšíření.

Literatura:

- HASSAN AIT-KACI: Warren's Abstract Machine, A TUTORIAL RECONSTRUCTION, February 18, 1999, (REPRINTED FROM MIT PRESS VERSION)
- [online] SWIProlog, http://www.swi-prolog.org/, cit. 2016-09-26
- Dle doporučení vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- První dva body zadání.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Kolář Dušan, doc. Dr. Ing.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 24. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Náplní této diplomové práce je implementace interpretu logického programovacího jazyka Prolog. Práce shrnuje různé přístupy vyhodnocování programů v tomto jazyce, nejvíce prostoru je věnováno popisu Warrenova abstraktního stroje. V práci je navržen způsob začlenení Prologu do platformy Microsoft .NET a jeho propojení s objektovými jazyky na této platformě. Následně je navržen a implementován interpret a překladač vycházející z Warrenova abstraktního stroje včetně propojení s .NET.

Abstract

This Master's deals with the implementation of the interpreter of logic programming language "Prolog". It summarises the different approaches to evaluation of programs in this language with focus on description of The Warren Abstract Machine. A new way of integrating Prolog into The Microsoft.NET platform has been designed as well as its connection with object-oriented languages. Subsequently, an interpreter and a compiler based on The Warren Abstract Machine have been designed and implemented including the connection to The Microsoft.NET platform.

Klíčová slova

Prolog, Warren Abstract Machine, logické programování, platforma .NET

Keywords

Prolog, Warren Abstract Machine, logic programming, .NET platform

Citace

HALJUK, Petr. *Překladač jazyka Prolog pro .NET*. Brno, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. Dr. Ing. Dušan Kolář

Překladač jazyka Prolog pro .NET

Prohlášení

Prohlašuji, že jsem tento semestrální projekt vypracoval samostatně pod vedením pana doc. Dr. Ing. Dušana Koláře. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Petr Haljuk
23. května 2017

Poděkování

Chtěl bych poděkovat doc. Dr. Ing. Dušanu Kolářovi za odborné vedení, za pomoc a rady při zpracování této práce. Také bych rád poděkoval všem, kteří mě při psaní podporovali.

Obsah

Kapitola 1

Úvod

Existují různé přístupy k zápisu počítačových programů, souhrnně se nazývají paradigmata. V současné době mezi nejpopulárnější paradigmata patří objektově orientované a imperativní programování. V nich se algoritmus popisuje jako posloupnost kroků na různé úrovni abstrakce. Existují ale i jiné přístupy. Deklarativní jazyky nepopisují přesný výčet kroků, ale pouze cíl, kterého je potřeba dosáhnout. Typickými zástupci jsou funkcionální jazyky (např.: Haskell nebo F#, které výpočet chápou jako vyhodnocování matematických funkcí) a jazyky logické.

Logické jazyky jsou postavené na principech matematické logiky. Původně vznikly pro automatizaci matematických důkazů, ale jsou výpočetně úplné a lze je používat jako jazyky pro obecné použití. Jedním z nejznámějších logických jazyků je Prolog. Tento způsob programování umožňuje jednoduchý zápis některých typů programu jako například expertní systémy, úlohy pracující s prohledáváním stavového prostoru nebo tvorba syntaktických analyzátorů.

Platforma .NET je prostředí pro vývoj desktopových, mobilních a webových aplikací. Jedním z jejich základních rysů je, že umožňuje psát programy ve více jazycích a tyto programy bez větších problému kombinovat. Většina jazyků na této platformě je objektově orientovaná (lze ale nalézt i funkcionální jazyky). Nabídka logických jazyků na této platformě je velmi chudá. Většina pokusů o propojení s Prologem nebo přímo o implementace Prologu pro .NET zůstala nedokončená nebo rychle zastarala.

Cílem této práce je navrhnout a implementovat takový překladač jazyka Prolog, aby umožnil spolupráci s ostatními jazyky na platformě .NET. Tento přístup umožní napsat v Prologu pouze tu část, na kterou se hodí a usnadní práci. Ostatní části (například uživatelské rozhraní) pak lze vytvořit standardními prostředky.

Kapitola druhá stručně seznamuje se základními koncepty logického programování a syntaxí jazyka Prolog. V kapitole třetí jsou rozebrány přístupy k vyhodnocování těchto programů sekvenčním způsobem. V kapitole čtvrté je stručně popsána platforma .NET a jsou analyzovány předchozí pokusy o implementaci Prologu na této platformě. Kapitola pátá pak popisuje návrh nového interpretu. V šesté kapitole je popsána implementace nového překladače a interpretu, postavená na Warrenově abstraktním stroji. V závěrečné kapitole je potom vyhodnocena jeho funkčnost a výkon.

Kapitola 2

Logické programování

V této kapitole jsou popsány základy logického programování a jazyk Prolog na úrovni nutné k porozumění dalšímu návrhu překladače.

2.1 Základní koncepty

Logické programovací jazyky patří do kategorie jazyků deklarativních. Při programování se nepopisuje, jak výsledku dosáhnout, ale pouze, jak má vypadat výsledek. Program se skládá z množiny faktů a pravidel (souhrnně nazýváno databáze). Na program lze pokládat dotazy, na které se systém snaží odvodit odpověď právě podle faktů a pravidel. Výsledkem volání dotazu je to, zda je z databáze možno dokázat tvrzení v dotazu. Pokud ano, nazývá se takový dotaz úspěšným, v opačném případě se říká, že dotaz selhává. V případě úspěchu systém ještě vrátí uživateli ohodnocení proměnných, při kterém dotaz uspěl. Zajímavou vlastností logických programů je, že programy mohou být nedeterministické a program může vracet více řešení než jedno, jak je obvyklé např. u imperativních jazyků.

2.1.1 Hornovy klauzule

Hornovy klauzule jsou takové klauzule, které obsahují maximálně jeden pozitivní literál (jak je vidět v zápise disjunktivní formou v rovnici 2.3). Jsou podmnožinou klauzulí predikátové logiky. Jinými slovy: každou Hornovu klauzuli lze přepsat do predikátové logiky, ale ne každou klauzuli predikátové logiky lze zapsat jako Hornovu klauzuli. Skládá se z hlavičky a těla. Hlavička je predikát a tělo je seznam predikátů, které musí být splněny, aby byla splněna hlavička. Zapisuje se jako implikace následovně:

$$h \leftarrow p_1, p_2, ...p_n \tag{2.1}$$

Přepsání do predikátové logiky může vypadat následovně:

$$p_1 \wedge p_2 ... \wedge p_n \implies h \tag{2.2}$$

$$\neg p_1 \vee \neg p_2 ... \vee \neg p_n \vee h \tag{2.3}$$

kde h je hlavička a p jsou jednotlivé predikáty těla. V případě, že tělo neobsahuje žádný predikát, nazývá se taková klauzule jako fakt. Klauzule je základní jednotkou logického programu v jazyce Prolog. Více klauzulí se stejným jménem a aritou pak tvoří proceduru.

Predikát uspěje pokud ho lze unifikovat s nějakou hlavičkou klauzule v příslušné proceduře a pokud uspějí všechny predikáty v jeho těle. Rozsah platnosti proměnných je omezen maximálně na jednu klauzuli. [19]

2.1.2 SLD Rezoluce

Způsob, jakým je vyhodnocován konkrétní program, je závislý na pořadí jednotlivých Hornových klauzulí v programu a predikátů v jejich těle. Klauzule se procházejí od shora dolů a predikáty v těle klauzule zleva doprava. Tato vyhodnocovací strategie se nazývá SLD rezoluce (Selective Linear Definite clause resolution).

```
1  informatik(roman).
2  umi_programovat(roman).
3  informatik(X) ← programator(X).
4  programator(X) ← informatik(X), umi_programovat(X).
5  informatik(adam).
6  umi_programovat(adam).
```

Program 2.1: Ukázka kódu jednoduchého logického programu

Uvažujme kód v programu 2.1. Klauzule 1, 2, 5 a 6 jsou fakty, zbytek jsou pravidla. Proměnné jsou značeny velkými písmeny. Pokud bude položen dotaz zda je *roman* programátor, bude vyhodnocení podle SLD rezoluce probíhat následovně: 4, 1, 2. Tento způsob prohledávání se nazývá prohledávání do hloubky (Depth First Search). Systém tedy ověřuje klauzuli tak dlouho, dokud nějaký její predikát neselže. Stejně tak pro dotaz, zda je programátor *adam* lze najít pořadí tak, aby bylo vyhodnocení úspěšné. Ale v pořadí v jakém jsou klauzule uvedeny v programu 2.1 nelze SLD rezolucí dojít k výsledku a program bude nekonečně opakovat vyhodnocení klauzulí 4 a 3. Řešením by bylo přesunout klauzule 5 a 6 nad klauzuli 3. [18]

2.1.3 Unifikace

Unifikace je porovnávání vzorů (pattern matching) dvou libovolných termů. Výsledkem je buďto úspěch nebo neúspěch. Operaci unifikace lze definovat následovně: [9]

- Dvě konstanty se unifikují, pokud jsou si rovny
- Dvě struktury se unifikují, pokud jsou jejich funktory rovné (názvem a aritou) a všechny jejich členy se unifikují
- Dvě nenavázané proměnné se unifikují a jsou na sebe navázány
- Nenavázaná proměnná a konstanta nebo struktura se unifikují a je navázána na proměnnou

2.1.4 Zpětné navracení

Zpětné navracení (anglicky *backtracking*) je jeden ze základních konceptů jak vyhodnotit logický program. Na dotaz lze pohlížet jako na cíl, o kterém je potřeba rozhodnout, zda je splnitelný. Při vyhodnocování predikátu je jeho tělo postupně procházeno a jsou vyhodnocovány jednotlivé podcíle. V okamžiku, kdy některý podcíl selže (nelze ho pomocí databáze

dokázat), není výpočet ukončen, ale je znovu vyhodnocen předchozí podcíl a pokud znovu uspěje, pokračuje vyhodnocování od tohoto bodu. Díky tomuto mechanismu může systém projít postupně všechny nedeterministické možnosti (až na omezení SLD rezolucí popsané v kapitole 2.1.2). Díky zpětnému navracení může být vráceno více výsledků (více uspívajících ohodnocení proměnných). V okamžiku, kdy je vráceno nějaké uspívající ohodnocení proměnných a je požadováno další, je vyvoláno zpětné navracení a vyhodnocování pokračuje, jakoby předchozí pokus neuspěl. [19]

2.2 Jazyk Prolog

Vývoj logických jazyků začal v sedmdesátých letech dvacátého století, kdy byla uvedena první verze jazyka Prolog, dnes jednoho z nejznámějších a nejpoužívanějších jazyků v oblasti logického programování. Tento jazyk vychází z výše popsaných konceptů a doplňuje je o prvky nutné k praktickému programování, ačkoli nejsou vždy zcela postaveny na konceptech logiky. Následující text obsahuje popis základních vlastností jazyka Prolog. Podrobnější popis lze najít v citované literatuře. [18]

2.2.1 Syntaxe

Program v jazyce Prolog vypadá velmi podobně jako zápis Hornových klauzulí. Jen bylo nutné nahradit symbol šipky ($\leftarrow$) symboly z $ASCII$, konkrétně se zapisuje jako $:-$ a každá klauzule se ukončuje tečkou. Proměnné začínají velkým písmenem. Řetězec začínající malým písmenem nebo jakýkoli řetězec uzavřený v apostrofech je atom. Atom lze chápat jako pojmenovanou unikátní konstantu. Konstanty a atomy se nazývají jednoduchými termy. Struktury a seznamy pak složenými termy, přičemž souhrnně vše lze označit jako term. Dotaz je pak ve většině případů uvozován znaky $?-$.

2.2.2 Datové typy a struktury

Prolog neobsahuje mnoho datových typů ani datových struktur. Běžně se v jeho implementacích objevují celá čísla (která mají často neomezený rozsah), reálná čísla a řetězce. Z datových struktur Prolog podporuje seznamy a struktury. Obě jsou heterogenní, mohou tedy obsahovat různé datové typy.

Struktury se skládají z atomu, který představuje její název a seznamu termů, které obsahuje. Struktura představuje uspořádanou n-tici a zapisuje se takto: `atom(term, term, term)`. Je možné do sebe struktury libovolně zařazovat. Mimo jednoduchých termů lze v Prologu vše reprezentovat pomocí struktur (včetně seznamů a aritmetických výrazů, které jsou popsány dále). Například i klauzule je struktura s aritou 2 a názvem `:-` nebo tělo predikátu je struktura o n prvcích jejíž název je symbol čárka. Tento přístup umožňuje předávat klauzule a ostatní konstrukce jazyka jako termy.

Oproti strukturám nemají seznamy pevně danou délku. Seznam se skládá z hlavičky a zbytku seznamu. Nelze tedy přistupovat k termu na libovolné pozici, ale vždy je nutné ho procházet odpředu. Seznam se zapisuje do hranatých závorek a jednotlivé termy se oddělují čárkou. Prázdný seznam se pak zapisuje jako `[]`. Pro oddělení hlavičky od zbytku se používá znak `|` a hlavička může obsahovat libovolný počet termů. Zápis `[H|T]` tedy znamená, že `H` je první term v seznamu a `T` je seznam obsahující všechny termy seznamu mimo `H`. Jak již bylo zmíněno, seznam o n prvcích lze teoreticky chápat jako n zanořených struktur do sebe s aritou 2, kde je vždy na první pozici hodnota a na druhé struktura se zbytkem seznamu.

Poslední struktura pak obsahuje na druhé pozici symbol pro prázdný seznam (`[]`), který je chápán jako atom. Vzhledem k tomu, že by byl takový zápis nepřehledný, je pro seznamy zvolena výše uvedená speciální syntaxe.

Pro vyhodnocování aritmetických výrazů slouží operátor `is/2`[1]. Tento operátor vyhodnotí aritmetický výraz na pravé straně a následně ho unifikuje s termem na levé straně. Pokud se ve výrazu na levé i pravé straně vyskytuje stejná proměnná, operátor `is` vždy selže. Aritmetický výraz na pravé straně je reprezentován strukturou, jejíž název je symbol operace a seznam termů jsou operandy. Struktury pro aritmetické výrazy se nějčastěji zapisují infixovou notací. Program 2.2 demonstruje jednoduchou práci se seznamy a operátorem `is`. [18]

```
1     vratDruhy([F,S|X], S).
2
3     delkaSeznamu([], 0).
4     delkaSeznamu([H|T], S) :- delkaSeznamu(T,C), S is C + 1.
5
6     ?- vratDruhy([pes, kocka, kralik], Druhy).
7     Druhy = kocka
8
9     ?- vratDruhy([pes, kocka, kralik], pes).
10    False.
11
12    ?- delkaSeznamu([pes, kocka, kralik, liska], Delka).
13    Delka = 4
```

Program 2.2: Příklady jednoduché práce se seznamy v jazyce Prolog

2.2.3 Negace a operátor řezu

Operátor řezu (zapsaný znakem `!`) umožňuje omezit prohledávání všech řešení pomocí zpětného navracení. Vyhodnocování aktuálního těla predikátu se nikdy přes tento operátor nevrátí. Na operátor řezu se lze dívat také jako na predikát, který uspěje právě jednou. Použití operátoru řezu můžeme rozdělit na zelené a červené řezy. Liší se podle vlivu řezu na výsledek programu. Zelený řez pouze zmenší počet procházených možností, ale výsledky nijak neovlivní. Naopak červený řez mění naleznutá řešení.

Ačkoli predikát `not/1` svým názvem evokuje negaci, nejedná se o logickou negaci v pravém slova smyslu. Tento predikát neznamená, že dotaz zapsaný jako parametr neplatí. Znamená, že není dokazatelný podle faktů a pravidel v databázi programu. Jinými slovy `not(X)` uspěje, pokud `X` selže. Tento predikát lze implementovat pomocí operátoru řezu a vždy neuspívajícího predikátu `fail/0`. [18]

2.2.4 Operátor disjunkce

Tento operátor dlouhou dobu nebyl součástí Prologu. Lze ho totiž téměř nahradit rozepsáním klauzule s disjunkcí na více klauzulí bez disjunkce (jak je demonstrováno v programu 2.3). Zapisuje se znakem `;` mezi predikáty v těle klauzule a pomocí závorek ho lze libovolně zanořovat. Bez operátoru disjunkce platí, že rozsah platnosti názvů proměnných je

[1] Obvyklý zápis znamenající, že operátor `is` má aritu 2 (má dva parametry).

po celé klauzuli. Zavedením disjunkce v klauzuli vzniká více větví a pokud se proměnná poprvé vyskytne až v části klauzule s operátorem disjunkce, je rozsah platnosti jejího názvu omezen na tuto větev a její podvětve. V příkladu 2.4 je ukázána klauzule, jejíž proměnná `X` představuje v obou větvích jinou proměnnou (tedy, pokud je navázána v první větvi, nijak to neovlivní proměnnou `X` v druhé větvi).

```
1    a(X,Y,Z) :- b(X), (c(Y) ; d(Z)).
2    -------------
3    a(X,Y,Z) :- b(X), tmp(Y,Z)
4    tmp(Y,Z) :- c(Y).
5    tmp(Y,Z) :- d(Z).
```

Program 2.3: Příklad programu s operátorem disjunkce a ekvivalentního programu bez tohoto operátoru

```
1    a(Y) :- b(Y), (c(Y,X), e(X) ; d(Y,X), f(X)).
```

Program 2.4: Příklad klauzule s operátorem disjunkce

Způsob vynechání operátoru disjunkce uvedený v ukázce 2.3 nelze aplikovat na případ, kdy se v některé větvi vyskytuje operátor řezu. Taková situace je demonstrována na příkladu 2.5. [18]

```
1    a(1).
2    a(2).
3    b(3).
4    p(X,Y,Z) :- a(Z), (a(X), !, b(Y) ; b(X), a(Y)).
5
6    ?- p(X,Y,Z).
7    X = 1, Y = 3, Z = 1.
8    --------------------
9    a(1).
10   a(2).
11   b(3).
12   p(X,Y,Z) :- a(Z), tmp(X,Y).
13   tmp(X,Y) :- a(X), !, b(Y).
14   tmp(X,Y) :- b(X), a(Y).
15
16   ?- p(X,Y,Z).
17   X = 1, Y = 3, Z = 1 ;
18   X = 1, Y = 3, Z = 2.
```

Program 2.5: Příklad rozdílu klauzule s operátorem řezu

2.2.5 Změna programu za běhu

V Prologu lze, jako v málokterém jiném jazyce, měnit chování programu za jeho běhu. Slouží k tomu tzv. dynamické klauzule. V programu je nejdříve nutné takové klauzule

označit (např. pomocí `:- dynamic predicate/1.`). Pro přidání nové klauzule pak slouží predikát `assert/1`, pro odebrání `rectract/1` (odebere první klauzuli, se kterou se zadaná struktura unifikuje) a `rectractall/1` (odebere všechny klauzule, se kterými se zadaná hlavička unifikuje). Pro predikát `assert/1` ještě existují varianty `asserta/1` resp. `assertz/1`, které vynutí přidání klauzule na začátek resp. konec programu. [18]

```
1     :-dynamic test/1.
2
3     ?- test(42).
4     No.
5
6     ?- assert(test(X) :- number(X)).
7     Yes.
8
9     ?- test(42).
10    Yes.
```

Program 2.6: Demonstrace změny programu za běhu.

Pro určení, kdy je možné s dynamicky přidanou klauzulí pracovat, existují dva přístupy nazývané okamžitá a logická změna. Jak již název napovídá, přidaná (nebo odebraná) klauzule je přístupná (nebo nedostupná) pro vyhodnocování v okamžiku jejího přidání (nebo odebrání). Tento přístup ale již v současnosti nevyhovuje ISO standardu. V případě logické změny je při každé změně databáze vytvořena její nová generace. Na začátku vyhodnocování daného cíle je zaznamenáno aktuální číslo generace databáze a pro tento cíl jsou až do konce jeho vyhodnocení dostupné pouze klauzule, které existovaly v této generaci. [22]

Další dynamickou vlastností jazyka Prolog je možnost definovat operátorům (i většině vestavěných) za běhu prioritu, asociativitu a typ (prefix, infix, postfix). K tomu slouží direktiva `:- op(Priorita,Typ,Nazev).`, kde priorita je vyjádřena číslem větším než nula (vyšší znamená vyšší prioritu). Typ určuje asociativitu a druh (např.: `xfy` je infixový pravě asociativní operátor). Priorita 0 operátor odstraní. Operátor tedy v podstatě určuje, jak lze zapsat strukturu s daným názvem. Kompletní výčet typů a standardně definovaných operátorů lze najít v [18].

Kapitola 3

Přístupy k vyhodnocování programů

Protože se program v Prologu může za běhu měnit (jak bylo popsáno v kapitole 2.2.5), není ho možné překládat přímo do strojového kódu. Část musí být vždy vyhodnocována za běhu (dynamické predikáty, priorita operátorů atd.). Pokud se u Prologu mluví o překladu do strojového kódu, zpravidla se myslí pouze částečný překlad toho, co přeložit lze. Častým přístupem je také překlad do různých typů mezikódů, které jsou potom interpretovány pomocí tzv. virtuálních strojů. Princip jejich fungování ve více abstraktní rovině pak bývá často popsán formou abstraktního stroje, který umožní v popisu vynechat zbytečné implementační detaily. Jedním ze základních kritérií pro dělení Prologovských abstraktních strojů je způsob reprezentace datových struktur:

Sdílení struktur (Structure Sharing) využívá dva ukazatele. Jeden na kostru struktury a druhý na konkrétní navázání proměnných.

Kopírování struktur (Structure Copying) popisuje vždy konkrétní navázání proměnných a vytváří novou kopii celé struktury v okamžiku, kdy je navázána na proměnnou.

První implementace Prologu (např. DEC-10 Prolog) využívaly sdílení struktur. V současné době valná většina implementací využívá kopírování struktur, ačkoli praktická srovnání ukazují, že oběma způsoby lze dosáhnout srovnatelných výsledků. [11]

Následující text popisuje některé existující abstraktní stroje pro vykonávání Prologovských programů. Tento výčet není úplný. Z dalších architektur lze uvést například LAM[1], který využívá ještě jiný přístup než sdílení a kopírování struktur zvaný Program Sharing. Více informací lze nalézt v [11].

3.1 Warrenův abstraktní stoj (WAM)

Warren Abstract Machne vznikl v roce 1983, a stal se na dlouhou dobu velmi populární. Mnoho implementací a dalších abstraktních strojů z něho vychází i v současné době. Příkladem takové implementace je například GNUProlog[2] nebo komerční SICStus Prolog[3]. Tento stroj využívá kopírování struktur.

[1] Lakehead Abstract Machine
[2] http://www.gprolog.org
[3] http://sicstus.sics.se

WAM překládá klauzule do mezikódu, jehož vykonávání je podobné jako u imperativních jazyků. Hlavními stavebními bloky stroje jsou kódová oblast (kde je uložen přeložený program), adresovatelná paměť zvaná halda (anglicky *heap* nebo v některých zdrojích *main stack*), neomezený počet pracovních registrů a několik řídicích registrů obsahujících například adresu aktuálně vykonávané instrukce, adresu pro návrat po volání predikátu apod. Prvních n pracovních registrů se nazývá registry parametrů (anglicky *argument registers*), kde n je arita volaného podcíle. Tyto registry se nijak neliší od ostatních pracovních registrů, ale volaný podcíl předpokládá, že v těchto registrech nalezne parametry pro vyhodnocení. Pro přehlednost se registry použité v kontextu registru pro parametr označují jako `A` a obyčejné pracovní registry jako `X`. Dále abstrakní stroj obsahuje tři globální zásobníky: [4]

Zásobník prostředí (Environment Stack) slouží pro uložení řídicích registrů před voláním podcíle, aby řídící registry mohly být po návratu znovu obnoveny. Zároveň se zde ukládají tzv. permanentní proměnné, které jsou popsány později v této kapitole. Taktéž se sem ukládají adresy, kde se má pokračovat při selhání některého z podcílů.

Zásobník cesty (Trail Stack) obsahuje všechny proměnné, které byly do současné doby navázány. V případě zpětného navracení je určitý počet proměnných z vrcholu tohoto zásobníku odebrán a jsou nastaveny jako nenavázané.

Unifikační zásobnik (Unification Stack), někdy také nazýván PDL (Push-Down List), slouží pro odkládání termů během unifikace.

Překlad programu probíhá po jednotlivých klauzulích a dotaz je považován za speciální případ klauzule bez hlavičky. V příkladu 3.1 je uvedeno obecné schéma pro překlad klauzule. [9]

unifikace parametrů hlavičky p_0 s registry
vložení parametrů predikátu p_1 do registrů
`call` p_1
$\vdots$
vložení parametrů predikátu p_n do registrů
`call` p_n

Program 3.1: Schéma přeložené klauzule ve WAM

3.1.1 Instrukční sada

Jednotlivé instrukce, ze kterých se program pro WAM skládá, lze rozdělit do pěti kategorií, jejichž název je odvozen od prefixu, kterým název každé instrukce dané skupiny začíná. Kompletní seznam instrukcí včetně definic chování lze najít v [4].

GET instrukce se používají pro parametry v hlavičce klauzule. Zkontrolují, jestli se v příslušném registru parametrů vyskytuje očekávaný term.

UNIFY instrukce se taktéž vyskytují v hlavičce v případě unifikace parametrů složených termů.

PUT instrukce se používají při překladu těla, přesněji pro umístění parametrů volaného podcíle do příslušných registrů.

SET instrukce se používají pokud PUT instrukce vkládá do parametru složený term, parametry složeného termu poté nastaví tyto instrukce.

Řídicí instrukce zajišťují skok na správnou adresu při volání podcíle, návrat z volání, zajištění zpětného navracení, řez apod.

Aby bylo možné v hlavičce klauzule provést unifikaci i složených termů, pracují instrukce GET a UNIFY ve dvou režimech. Instrukce GET podle obsahu registru nastavují režim buď na čtení nebo zápis. Instrukce UNIFY se pak podle tohoto režimu řídí. Pokud je v hlavičce klauzule například struktura a v příslušném registru parametru je proměnná, přejde GET do režimu zápisu a do proměnné naváže reprezentaci struktury. Její prvky poté nastaví následující instrukce UNIFY. V opačném případě, tedy pokud je v registru parametru i v hlavičce na dané pozici struktura, nastaví GET instrukce režim čtení a pouze zkontroluje shodnost názvu a arity obou struktur. Následující instrukce UNIFY poté zkontrolují prvky složeného termu. Kód v ukázce 3.2 demonstruje unifikaci hlavičky za použití obou režimů. Každá instrukce může při vykonávání selhat (například není možné unifikovat dvě různé konstanty) a v tom případě je vyvoláno zpětné navracení.

Překlad dotazu je jednodušší. Cílem je pouze vytvořit v registrech reprezentaci parametrů a následně zavolat kód klauzule, kde se s těmito parametry provede unifikace. Jak je vidět na programu 3.3, nejdříve jsou v registrech vytvořeny zanořené struktury a ty jsou potom vkládány do méně zanořených struktur. [4]

```
1     get_variable X3, A1      % proměnná X (první výskyt)
2     get_structure a/2, A2    % nastaví mód na čtení
3     unify_variable X4        % zkopíruje první prvek struktury do X4
4     unify_constant b         % unifikace druhého prvku struktury
5     get_structure c/1, X4    % nastaví mód na zápis
6     unify_value X3           % proměnná X
7     proceed                  % návrat z volané klauzule
```

Program 3.2: Klauzule `p(X,a(c(X), b)).` přeložena do WAM

```
1     put_structure c/1, X3      % alokace struktury c
2     set_constant b
3     put_structure a/2, A1
4     set_value X3               % vložení struktury c do a
5     set_variable X4            % proměnná Y
6     put_variable X5, A2        % proměnná X
7     call p/2                   % skok na kód pro p/2
```

Program 3.3: Dotaz `p(a(c(b),Y), X).` přeložen do WAM

3.1.2 Klasifikace proměnných

Každá přeložená klauzule pracuje se stejnou sadou pracovních registrů. To znamená, že po vykonání instrukce `call` nelze předpokládat, že kód jiné klauzule nepřepsal obsah těchto registrů. Pro klauzule, které mají v těle víc než jeden predikát, tedy není možné proměnné ukládat pouze do pracovních registrů. Z tohoto důvodu WAM zavádí rozdělení proměnných

na dočasné a permanentní. Za permanentní je považována taková proměnná, která se vyskytuje ve více než jednom predikátu klauzule s tím, že hlavička se považuje za součást prvního predikátu těla. Sloučení hlavičky s prvním predikátem je zavedeno, protože mezi nimi není žádné volání a pracovní registry tedy nemohou být přepsány. Ostatní proměnné, které nejsou permanentní, jsou označovány jako dočasné. Například v klauzuli `p(X,Y) :- a(X,Z), c(Y,Z).` jsou permanentní proměnné Y a `Z`.

Proměnné, které musí přežít všechna volání v klauzuli se ukládají do záznamu na vrcholu zásobníku prostředí. Každá klauzule, která obsahuje více než jedno volání jiné procedury, začíná instrukcí `allocate`, jejímž parametrem je počet permanentních proměnných v klauzuli a končí instrukci `deallocate`. Právě tyto instrukce vytváří nebo odstraní záznam na zásobníku prostředí. Pro fakta a řetězová pravidla (pravidlo s jedním predikátem v těle) se tedy záznam na zásobníku prostředí vůbec nevytváří. Proměnné umístěné v registrech na zásobníku se značí `Y`. Pro úplnost je vhodné zmínit, že řetězová pravidla nemusí alokovat záznam na zásobníku kvůli optimalizaci posledního volání, která je podrobněji popsána v kapitole 3.4.1. Příklad 3.4 ukazuje překlad klauzule s permanentními proměnnými.

```
1     allocate 2              % alokovat prostor pro 2 perm. proměnné
2     get_variable X3, A1     % dočasná proměnná X
3     get_variable Y1, A2     % permanentní proměnná Y
4     put_value X3, A1
5     put_variable Y2, A2     % permanentní proměnná Z
6     call a/2
7     put_value Y1, A1        % Y ze zásobníku do registru
8     put_value Y2, A2        % Z ze zásobníku do registru
9     call c/2
10    deallocate
```

Program 3.4: Klauzule `p(X,Y) :- a(X,Z), c(Y,Z).` přeložena do WAM

Ve WAM se proměnné klasifikují ještě na bezpečné a nebezpečné (anglicky *safe* a *unsafe*). V případě implementace optimalizací pro správu paměti (zejména optimalizace posledního volání) se může stát, že permanentní proměnná vytvořená instrukcí `put_variable` bude navázána na proměnnou alokovanou později. Taková proměnná (např.: na novějším záznamu na zásobníku) může být smazána z paměti a nahrazena jinou a reference bude odkazovat na nerelevantní hodnotu. Řešení tohoto problému závisí na paměťovém modelu implementace. Možným řešením je povolit pouze odkazování od nověji alokovaných proměnných na starší a ne opačně. Detaily lze nalézt v [4].

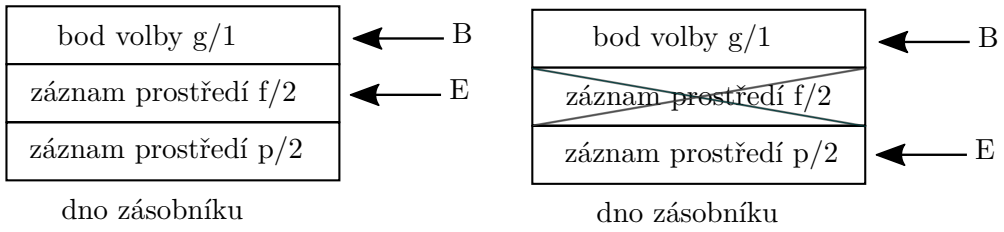
3.1.3 Zpětné navracení

Jak již bylo zmíněno, zpětné navracení je jedním ze základních kamenů logických programovacích jazyků a odlišuje je od ostatních. Je proto logické, že bude zásadně ovlivňovat návrh celého abstraktního stroje. V první řadě je nutné ukládat si informaci o tom, kde pokračovat v případě, že některý z podcílů selže. Tato informace se nazývá bod volby. Body volby jsou opět ukládány na zásobník prostředí. K tomuto zásobníku je nutné mít dva registry. První (označený `E`) ukazuje na nejvyšší záznam o prostředí (anglicky *Environment Frame*), který vytváří instrukce `allocate` a byl blíže popsán v předchozí kapitole. Druhý registr (označený `B`) pak ukazuje na nejvyšší bod volby zapsaný na stejném zásobníku. Důvod, proč jsou dva typy záznamu uloženy na stejném zásobníku je, že i když bude

dealokován nějaký záznam o prostředí (proběhne návrat z klauzule), může být smazaný záznam znovu potřeba po provedení zpětného navracení. Uvažujme program 3.5, obrázek 3.6 ilustruje stav zásobníku prostředí před návratem a po návratu z `f/2`. V tento moment je ohodnocení proměnných `X=a`, `Y=a`. Následující predikát `r/2` tedy selže a bude vyvoláno zpětné navracení k predikátu `g/1` v `f/2`. Vrací se tedy do míst, kde bude znovu potřebovat pracovat se záznamem o prostředí, který už byl dealokován instrukci `deallocate`. Smazat záznam o prostředí je tedy možné jen v případě, kdy nad ním v zásobníku není žádný bod volby. V jiném případě je pouze změněn ukazatel na vrchol zásobníku. Každý bod volby v sobě obsahuje obsah registrů parametrů a hodnotu registru `E` a obě hodnoty je tak možné v případě zpětného navracení obnovit.

```
1     e(a)
2     g(a).
3     g(b).
4     g(c).
5     r(a,c).
6     f(X,Y) :- e(X), g(Y).
7     p(X,Y) :- f(X,Y), r(X,Y).
8
9     ?- p(X,Y).
```

Program 3.5: Program pro demonstraci funkce zásobníku prostředí



Obrázek 3.6: Stav zásobníku prostředí ve WAM

Při vykonání každého zpětného navracení je nutné všechny proměnné, které byly od posledního bodu volby navázány, opět nastavit jako nenavázané. K tomu slouží zásobník cesty. Při každém navázání proměnné je reference na tuto proměnnou uložena na zásobník. Součástí bodu volby je pak informace o vrcholu zásobníku cesty v době vytvoření. Při vyvolání zpětného navracení jsou odebrány všechny prvky ze zásobníku až do tohoto bodu a proměnné nastaveny na nenavázané. Na zásobník cesty se zapisují pouze proměnné, které vznikly před vytvořením bodu volby. To lze rozlišit podle adresy na haldě, kde je proměnná alokována. Při každém vytváření bodu volby na zásobník je zaznamenána aktuální nejvyšší hodnota adresy na haldě. Pokud je proměnná na vyšší adrese, nemusí být ukládána na zásobník cesty, protože taková proměnná při zpětném navracení stejně zanikne (nepovede na ni žádná reference a může být na haldě přepsána).

Body volby jsou vytvářeny pomocí speciálních instrukcí, které obalují instrukce jednotlivých přeložených klauzulí. Instrukce `try_me_else` vytvoří nový bod volby na zásobníku a vloží do něj adresu klauzule, která se má vykonat v případě selhání. `retry_me_else` změní adresu další klauzule v již existujícím bodě volby a instrukce `trust_me` bod volby odstraní. [4] [20]

```
try_me_else <adresa další klauzule>
```
WAM kód první klauzule
```
retry_me_else <adresa další klauzule>
```
WAM kód druhé klauzule
```
retry_me_else <adresa další klauzule>
```
WAM kód třetí klauzule
⋮
```
trust_me
```
WAM kód poslední klauzule

Program 3.7: Schéma použití instrukcí pro zpětné navracení ve WAM

3.1.4 Operátor řezu

Operátor řezu musí odstranit ze zásobníku prostředí všechny body volby vzniklé od zavolání aktuální procedury. Instrukce `call` zapíše do speciálního registru adresu nejvrchnějšího bodu volby na zásobníku. Všechny novější záznamy jsou při volání řezu odstraněny. Problém nastává, pokud se řez vyskytuje v těle klauzule až za nějakým voláním predikátu. Další instrukce `call` totiž přepíše tento registr. Z tohoto důvodu rozděluje WAM dva případy řezu: mělký a hluboký. Mělký řez se vyskytuje v klauzuli před první instrukcí `call`, hluboký naopak kdekoli za ním. Registr potřebný pro řez je tedy v případě hlubokého řezu potřeba uložit do záznamu o prostředí. Vzhledem k tomu, že tento operátor se ve většině klauzulí nevyskytuje, není pro něj vyhrazeno speciální pole, ale ukládá se jako permanentní proměnná. Hned po unifikaci hlavičky je speciální instrukcí přepsán obsah registru na příslušné paměťové místo na zásobníku. [4]

3.1.5 Binary WAM

Jedná se o modifikací, někdy též nazývanou Binary Prolog. Modifikace spočívá v tom, že každá klauzule může mít maximálně jeden podcíl. Každý program lze převést do této formy tak, že se každé klauzuli přidá další parametr, který obsahuje následující podcíl. Díky tomu jsou po překladu do WAM všechny proměnné dočasné (nemohou se vyskytovat ve více než jednom podcíli) a není potřeba práce s zásobníkem prostředí. Díky tomu se sníží počet instrukcí, což vede k efektivnějším interpretům.

```
1    a(X) :- b(X), c(X).
2    b(atom).
3    c(atom).
4    ----------------------------------
5    a(X, Cont) :- b(X, c(X, Cont)).
6    b(atom, Cont) :- call(Cont).
7    c(atom, Cont) :- call(Cont).
```

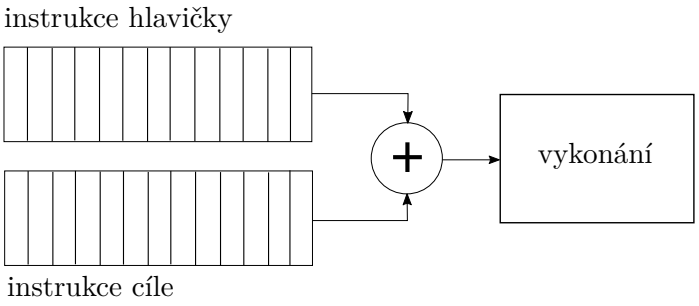
Program 3.8: Ukázka programu v Prologu a jeho alternativy v Binary Prologu

Vadou na kráse tohoto přístupu je, že všechny proměnné v následujících podcílech musí být uloženy na haldě a nelze použít některé běžné paměťové optimalizace. Efektivita implementace se tedy odvíjí od kvality správy paměti. [9]

3.2 Vienna Abstract Machine (VAM)

Jedná se alternativu k WAM vzniklou ve Vídni. Motivací k jejímu vzniku bylo odstranění úzkého místa v podobě předání parametrů do registrů a následné unifikaci. Zde jsou jednotlivé argumenty načítány postupně a hned unifikovány. VAM existuje ve dvou variantách VAM_{1P} a VAM_{2P}, jejichž rozdíly budou popsány později.

Reprezentace klauzulí v mezikódu VAM je velmi blízká tomu, jak je zapsána přímo v jazyce Prolog. Oproti WAM je instrukční sada menší a výsledný mezikód více připomíná vnitřní reprezentaci klauzule. Instrukční sada obsahuje tři skupiny instrukcí: unifikační (*unification instructions*), rezoluční (*resolution instructions*) a ukončovací (*termination instructions*). Unifikační instrukce slouží pro popis parametrů jak v hlavičce klauzule, tak i v těle (narozdíl od WAM, kde se používali dva rozdílné typy instrukcí). Rezoluční uvozují začátek podcíle (instrukce `goal`), případně konec faktu, řez nebo volání vestavěných predikátů. Ukončovací poté značí konec podcíle (instrukce `call` a `lastcall`, pro poslední podcíl). Specifický je způsob vyhodnocování programu nazývaný kombinování instrukcí. Při unifikaci je vždy načtena jedna instrukce z hlavičky a jedna z podcíle. Jejich operační kódy jsou zkombinovány (například sečtením operačních kódu) a kombinací vzniklá instrukce je poté vykonána. Například při unifikaci konstanty s prvním výskytem proměnné jsou načteny instrukce `const` a `fstvar`. Interpret následně vykoná kód přímo pro tuto kombinaci. [10] [9]



Obrázek 3.9: Princip vykonávání instrukcí ve VAM

3.2.1 VAM_{2P}

Tento typ VAM byl navržen pro implementaci v interpretech. Číslo uvedené u názvu znamená, že obsahuje dva instrukční ukazatele `goalptr` a `headptr`. První ukazuje na pozici v programu právě vyhodnocovaného cíle a druhý na hlavičku. Za běhu jsou obě instrukce načteny, zkombinovány a vykonány. Přeloženou klauzuli demonstruje program 3.10.

3.2.2 VAM_{1P}

Tento typ byl navržen pro přímý překlad do strojového kódu. Existuje zde pouze jeden ukazatel na pozici v programu. Instrukce jsou zkombinovány již v době překladu a pro každou možnou kombinaci unifikace je vygenerovaný speciální kód. Tento přístup umožňuje optimalizace jako eliminaci zbytečných instrukcí. [9]

```
1     fstvar 1     % proměnná X
2     list         % začátek seznamu
3     nxttmp 2     % dočasná proměnná H
4     fstvar 3     % proměnná Y
5     goal 3, a    % začátek podcíle
6     fstvar 4     % proměnná Z
7     nxtvar 1     % proměnná X
8     nxtvar 3     % proměnná Y
9     lastcall     % konec klauzule
```

Program 3.10: Přeložená klauzule `p(X,[H|Y]) :- a(Z,X,Y).` ve VAM

3.3 Abstraktní stroj ZIP

Hlavní myšlenka toho abstraktního stroje vychází z překladu programu v Prologu do instrukční sady, kterou lze jednoduše převádět zpět do formy Prologu. Datové struktury používané při vyhodnocování jsou podobné jako u předcházejících strojů. Lze zde najít část s programem, haldu, zásobník cesty a zásobník pro lokální proměnné. Instrukční sada obsahuje pouze 7 instrukcí, ačkoli při praktické implementaci může být z důvodu optimalizací počet instrukcí vyšší. Tento přístup využívá například SWI-Prolog [4].

Přeložená klauzule se skládá ze dvou částí. První je tabulka externích referencí (dále jako tabulka XR), která obsahuje konstanty, atomy, funktory a procedury, na které se potom odkazuje. Druhým je blok tzv. byte-kódů neboli instrukcí. Možné instrukce jsou následující:

enter odděluje hlavičku od těla klauzule

call volání prvku z XR tabulky

exit značí konec klauzule

const značí konstantu

var značí proměnnou

functor uvozuje složený typ, obsahuje referenci do XR tabulky

pop konec složeného termu

XR tabulka:

```
1: a
2: f/2
3: b
4: q/1
```

Blok byte-kodů:

```
var 1,
const 1,
functor 2, var 1, const 3, pop,
enter,
var 1, call 4,
exit
```

Program 3.11: Reprezentace klauzule `p(X,a,f(X,b)) :- q(X).` v ZIP

[4] http://www.swi-prolog.org

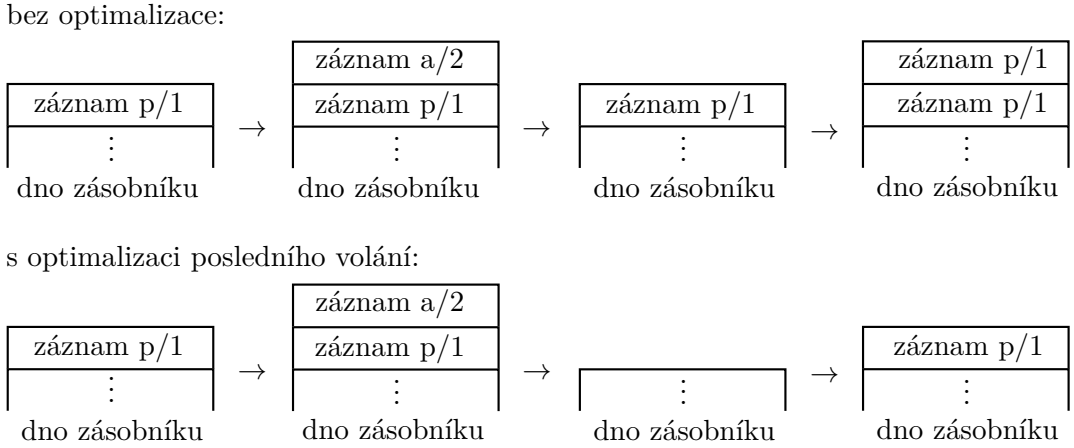
Za instrukcí v bloku byte-kódů následuje odkaz do XR tabulky (`const`, `call`, `functor`), pořadové číslo proměnné (`var`) nebo nic (`enter`, `pop`, `exit`). Na rozdíl od WAM, kde jsou parametry podcíle předávány pomocí registrů, v ZIP je k tomu využíván zásobník. Při vyhodnocení cíle jsou pak z takto přeložených klauzulí nalezeny takové, které odpovídají názvem a aritou, a ty jsou postupně unifikovány prováděním instrukcí dané klauzule s obsahem na zásobníku. [6]

3.4 Optimalizace kódu abstraktních strojů

Abstraktní stroje popsané v předchozích kapitolách jsou dostatečné pro vyhodnocení Prologovských programů. Často ale ne tak efektivně, jak by bylo možné. Z tohoto důvodu téměř každý návrh abstraktního stroje obsahuje různé optimalizace. V následující kapitole budou popsány ty nejběžnější z nich doplněné o příklady, jak jsou implementovány ve WAM, ačkoli tyto principy jsou používány i v ostatních strojích.

3.4.1 Optimalizace posledního volání

Tato optimalizace se vyskytuje nejen v logických jazycích, ale i u běžnějších (například imperativních) programovacích jazyků. Ačkoli v těchto jazycích jde spíše o vylepšení, v případě Prologu jde de facto o nutnost pro přijatelnou efektivitu programů. Tato optimalizace se uplatňuje u posledního predikátu v těle klauzule. V takovém případě je možné dealokovat kontext prováděné klauzule (např.: záznam o prostředí ve WAM) ještě před samotným voláním. To je možné, protože před voláním jsou parametry predikátu umístěny do registrů parametrů (nebo na zásobník) a uložení proměnné v kontextu již není potřebné. Tím se zajistí výrazně menší počet záznamů na zásobníku. Při volání je potřeba zajistit, aby návratová adresa po dokončení volané procedury vedla na předchozí proceduru. Ve WAM je pro volání použita speciální instrukce `execute`, která se chová velmi podobně jako `call`, ale nemění hodnotu registru s návratovou hodnotou.



Obrázek 3.12: Příklad zásobníku při vyhodnocování klauzule `p(X) :- a(X,Y), p(Y).`

Díky této optimalizaci je možné provádět v Prologu procházení seznamem iterativně i když je kód zapsán jako rekurzivní volání. Jedinou podmínkou je, že v programu musí být rekurzivní volání na posledním místě v těle klauzule (tzv.: tail recursion). Tato optimalizace je zřejmě efektivní pouze v případě, že se k žádnému predikátu v těle nelze vracet při

zpětném navracení. V opačném případě nelze kontext dealokovat, protože může být stále potřeba. Tento problém částečně řeší indexování klauzulí popsané v následující kapitole. [9] [4]

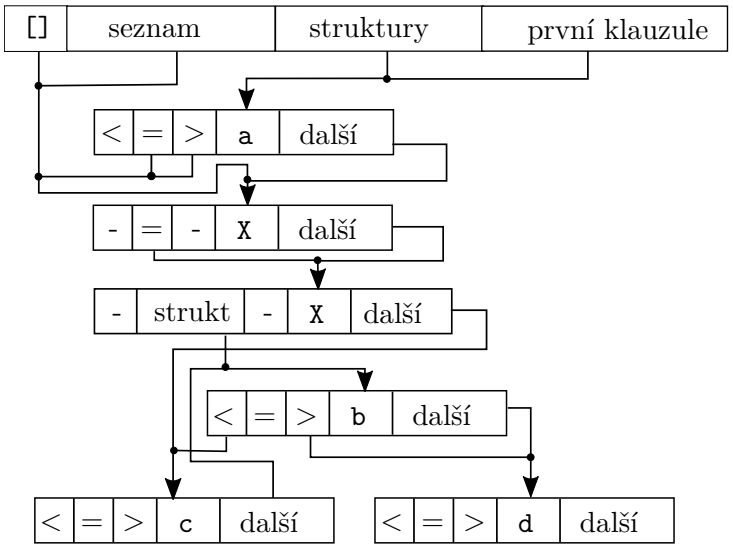
3.4.2 Indexování klauzulí

Při vykonávání procedury interpret zkouší postupně unifikovat obsah registrů parametrů s hlavičkami všech klauzulí v dané proceduře. V mnoha případech je ale již před začátkem unifikace zřejmé, že selže. Například nemá smysl zkoušet unifikovat predikát se strukturou s klauzulí, která má na stejné pozici v hlavičce seznam. Z tohoto důvodu je zavedeno indexování, které omezí počet klauzulí, se kterými se v proceduře pracuje. Jelikož indexování všech parametrů daného predikátu by bylo náročné, je často využit kompromis, kdy se indexuje pouze podle prvního parametru. Pokud je tedy vyhodnocován predikát, který má na místě prvního parametru konstantu, jsou vyhodnocovány pouze klauzule mající v hlavičce na prvním místě také konstantu. V případě, že je první parametr proměnná, musí se vyzkoušet všechny možností, neboť proměnná se unifikuje s čímkoli.

Samotná implementace indexování je poměrně rozmanitá. Častým přístupem jsou vyhledávací tabulky nebo vyhledávací stromy. Zajímavý je přístup použitý ve VAM. Pro indexování nejsou vytvořeny speciální instrukce, ale potřebné informace jsou uloženy přímo ve vnitřní reprezentaci procedury. Na obrázku 3.14 je znázorněna indexovací struktura pro klauzule z programu 3.13.

```
1    p(a, X) :- a(X,Y), b(X).
2    p(X, Y) :- c(Y), d(X).
3    p(X, a) :- f(X).
4    p(c, d).
5    p(b, X) :- f(X).
6    p(d, X) :- g(X).
```

Program 3.13: Klauzule pro indexovací strukturu na obrázku 3.14



Obrázek 3.14: Příklad indexovací struktury pro VAM

Jedná se o poměrně složitou strukturu, která v sobě zahrnuje binární strom a jednosměrně vázaný seznam. Z této struktury musí být možné zjistit, které klauzule je potřeba vyzkoušet unifikovat pro jakýkoli možný typ prvního parametru. V každé proceduře je uložen ukazatel na první klauzuli s prázdným seznamem, ukazatel na první klauzuli s neprázdným seznamem, na vyhledávací strom struktur a první klauzuli. Klauzule jsou pospojovány v pořadí, v jakém se vyskytovaly zapsané v programu. Tento vázaný seznam je použit v případě, že je prvním parametrem volaného predikátu proměnná a je nutné unifikovat ve všech klauzulích. Pro ostatní je vybrán ukazatel podle typu prvního parametru predikátu. Ukazatel `=` u klauzule značí, že se obě nachází ve stejném indexu. Tedy po unifikaci jedné klauzule bude jako následující indexována ta, na kterou ukazuje ukazatel `=`. Ukazatele `<` a `>` slouží pro vyhledávání v binárním stromu. Je podstatné si všimnout, že tato struktura obsahuje dva binární stromy (ačkoli první má pro stručnost příkladu pouze jeden prvek). V případě struktury jako prvního parametru je tedy nejdříve prohledán první strom (obsahující pouze `a`), potom proběhne unifikace s oběma klauzulemi s proměnnou `X` a poté je prohledán druhý strom. Pokud by byly nejdříve nalezeny všechny klauzule s odpovídající strukturou a až poté klauzule s proměnnou, byla by narušena SLD rezoluce. V případě, že se daný typ termu na první pozici v žádné klauzuli nevyskytuje, ukazatel vede na první klauzuli s proměnnou (v příkladu 3.14 nastala tato situace u seznamů). [9] [10]

V případě WAM se používá přístup nazývaný dvouúrovňové indexování. Tento název byl zvolen proto, že indexovaná procedura může vytvořit dva body volby na místo jednoho, který by byl vytvořen bez indexování. Na rozdíl od předchozího popsaného způsobu je zde index vytvořen pomocí instrukcí abstraktního stroje. Rozdělení typů podle kterých se klauzule dělí se také mírně liší. Typy se dělí na: konstanty (včetně prázdného seznamu a atomů), struktury, seznamy a proměnné. K vyhledání odpovídající klauzule je využita tabulka s rozptýlenými prvky. Jednotlivé klauzule jsou sloučeny do skupin (v pořadí v jakém se vyskytují v programu) tak, že každá skupina je nejdelší možná, aby neobsahovala klauzuli s proměnnou na prvním místě. Klauzule s proměnnou je vždy ve skupině sama. Index je poté tvořen pro tyto skupiny a každá skupina je dále brána jako jedna klauzule a obalena pomocí instrukcí pro zpětné navracení popsané v kapitole 3.1.3. Každou skupinu uvozuje instrukce `switch_on_term`, která rozhodne, podle typu prvního parametru, jaký kód se bude vykonávat. V případě struktur resp. konstant je použita druhá úroveň indexování pomocí instrukcí `switch_on_constant` resp. `switch_on_structure`, které podle názvu vyberou, kam se má v rámci kódu skočit. V případě, že pro jeden název existuje více klauzulí, je zavedena třetí úroveň indexování. K tomu slouží instrukce `try`, `retry` a `trust` s analogickým významem jako jejich obdoby z kapitoly 3.1.3. Mimo práce s body volby tyto instrukce ještě fungují jako volání. Například instrukce `try i` vytvoří bod volby s adresou návratu po selhání na následující instrukce a provede skok na adresu `i`. Překlad jedné skupiny klauzulí z procedury za pomoci indexování demonstruje okomentovaný příklad 3.15 [4] [9]

V některých implementacích Prologu se indexování nevyhodnocuje v době překladu, ale až za běhu. Tomuto přístupu se říka JIT[5] indexování klauzulí. Indexovací tabulka se buduje až po několika vykonání procedury a je na systému, aby zvolil nejvhodnější prvky, podle kterých indexovat. Díky tomu není nutné se omezovat pouze na první parametr predikátu a zachovat přijatelnou složitost samotného indexování. Tento přístup používá například SWI-Prolog. [22]

[5] Just In Time

```
1     p(a,X) :- ...    % konstanta
2     p(a,Y) :- ...    % konstanta
3     p(a,Z) :- ...    % konstanta
4     p(s(X)).         % struktura
5     p([]).           % konstanta
6     p(X).            % proměnná, ukončuje skupinu
7     ----------------------------------
8     % 1. úroveň indexování
9     % v případě proměnné: skok na 20
10    % v případě konstanty: skok na 14,
11    % v případě seznamu: selži,
12    % v případě struktry: skok na 19
13    switch_on_term 20, 14, fail, 19
14    switch_on_constant 2, {a: 16, []: 29}   % 2. úroveň indexování
15    % 3. úroveň indexování struktur
16    try 21
17    retry 23
18    trust 25
19    switch_on_structure 1, {s/1: 28}        % 2. úroveň indexování
20    try_me_else 22
21    % WAM kód pro p(a,X)
22    retry_me_else 24
23    % WAM kód pro p(a,Y)
24    retry_me_else 26
25    % WAM kód pro p(a,Z)
26    retry_me_else 28
27    % WAM kód pro p(s(X)).
28    retry_me_else 30
29    % WAM kód pro p([]).
30    trust_me
31    % WAM kód pro p(X).
```

Program 3.15: Příklad překladu jedné indexovací skupiny ve WAM

3.4.3 Ořezávání záznamu prostředí

Tato optimalizace je známá pod anglickým názvem *Environment Trimming*. Základní myšlenkou je, že permanentní proměnné nemusí být alokovány na záznamu prostředí po celou dobu vyhodnocování klauzule, ale jen do jejich posledního výskytu. Například ve WAM jsou permanentní proměnné alokovány na konci záznamu o prostředí. V případě, že už daná proměnná není potřeba, je velikost záznamu zmenšena a nově alokované záznamy mohou začínat na nižší adrese. Z tohoto důvodu je potřeba změnit pozice, na které jsou jednotlivé proměnné alokovány a to tak, že proměnná, která se jako první přestane vyskytovat, musí být alokována jako poslední. Například pro klauzuli `p(X,Y) :- a(X,Z), b(Z,Y,X), c(Z)` je pořadí následující: `Y`, `X`, `Z`.

Kapitola 4

Prolog a platforma .NET

V následující kapitole je stručně popsána platforma .NET. Dále jsou popsány již existující implementace Prologu pro tuto platformu. Na konci kapitoly je navrhnut způsob propojení Prologu s .NET a je vysvětleno, v čem se liší od již existujících.

4.1 Platforma .NET

Platformu .NET vyvíjí od roku 2000 firma Microsoft. Z pohledu programátora může být chápána jako běhové prostředí a rozsáhlá knihovna. Umožňuje vývoj desktopových, mobilních i webových aplikací primárně pro operační systém Windows, ale existují i implementace pro Unix a Mac OS. Programy jsou překládány do speciálního jazyka zvaného CIL a až při spuštění jsou převedeny na strojový kód cílové platformy (většinou je použit JIT[1] překlad). Díky tomu je pro .NET možné vyvíjet aplikace v mnoha jazycích jako například C#, VisualBasic.Net, IronPython[2] nebo F#. Díky tomu, že jsou všechny jazyky překládány do stejného mezikódu, je možné do jisté míry propojovat programy ve všech jazycích. Standard popisující, co každý jazyk v rámci .NET musí umožňovat, aby byla interoperabilita možná, se nazývá CTS (*Common Type System*).

Srdcem celé platformy je CLR (*Common Language Runtime*), který zajišťuje běh a překlad do kódu cílové platformy z jazyka CIL. Součástí CLR je také automatická správa paměti (*Garbage Collector*), správa vláken, zabezpečení kód atd. CLR není jedinou implementací běhového prostředí pro jazyky v CIL. Jedná se o implementaci distribuovanou s rámcem .NET na platfomě Windows. Specifikace popisující chování CLR se nazývá CLI (*Common Language Infrastructure*). Odtud také dostal jméno interpret popsaný v dalších částech této práce, CLIProlog. V novějších verzích přibyl ještě DLR (Dynamic Language Runtime), který je nadstavbou nad CLR a umožňuje implementaci dynamicky typovaných jazyků v .NET. Díky němu je v .NET přímá podpora pro pozdní vazbu ve všech hlavních jazycích. Využívá jej například implementace jazyka IronPython. [3] [17]

4.1.1 LINQ

Ve standardní knihovně obsahuje .NET mnoho zajímavých rozšíření. Jedním z nich je LINQ (*Language Integrated Query*). Jak již název napovídá, zjednodušuje a sjednocuje dotazování nad daty. Pracuje s jakýmkoli zdrojem implementující rozhraní `IEnumerable<T>`, které slouží pro procházení dat v objektu. LINQ je implementován pomocí rozšiřovacích metod

[1] Just In Time

[2] implementace Pythonu pro .NET - http://ironpython.net

(anglicky *Extension Methods*), které umožňují vytvořit statické metody, které lze volat nad objektem stejnou syntaxí, jako kdyby se jednalo přímo o metodu daného objektu. Druhou hojně používanou konstrukcí u LINQ jsou lambda funkce. Ty se používají především pro restrikci nebo projekci. Součástí LINQ jsou i agregační a další funkce. Díky konceptu rozšiřujících metod lze přidávat další operátory pouhým definováním nové rozšiřující metody. Více lze nalézt v citované literatuře.

Princip spočítá v tom, že pomocí metod technologie LINQ (často nazýváno také operátory LINQ) se nad kolekcí sestaví objekt implementující rozhraní `IQueryable<T>`, který se sestavuje postupným voláním operátorů nad kolekcí (viz příklad 4.1). V okamžiku, kdy uživatel chce přistupovat k datům, je na základě objektu `IQueryable` vytvořen enumerátor, který až v okamžiku volání prochází data a aplikuje na ně dané operace. Vytvoření enumerátoru proběhne buď explicitně voláním operátorů `ToList()`, `ToArray()` atd. nebo implicitně při procházení cyklem `foreach`.

```
1    IEnumerable<MyClass> collection = CreateSampleCollection();
2    IQueryable<MyClass> query = collection.Where(o => o.XYZ == 42)
3      .Skip(8).Take(7);
4    List<MyClass> result = query.ToList();
```

Program 4.1: Ukázka práce s technologií LINQ v jazyce C#

O tom, jak budou realizovány požadované operace rozhodují speciální objekty zvané poskytovatelé (anglicky *Providers*). LINQ díky tomu může pracovat s databází (poskytovatel operátory překládá do jazyka SQL) nebo jakýmkoli jiným datovým zdrojem. Mimo syntaxe zmíněné v příkladu 4.1, existuje v jazyce C# ještě speciální syntaxe připomínající jazyk SQL. V této práci je tato technologie zmíněna proto, že se od svého uvedení v .NET 3.5 stala defacto standardním způsobem zpracování dat v .NET a je vhodné navrhnout takové API pro propojení s Prologem, aby bylo možné výhodu technologie LINQ využít. [13]

4.2 Současné implementace

Pokusy o využití výhod Prologu na platformě .NET již proběhly, ale často mají jepičí život nebo nepřináší tak těsné propojení, které by umožnilo programovat na .NET v Prologu. Výsledkem je pouhý interpret Prologu napsaný v C# nebo jiném jazyce z .NET.

4.2.1 Překladač P#

Jedná se o implementaci z roku 2003, která se zdá být od roku 2004 neudržovaná. Je odvozena z projektu Prolog Café pro jazyk Java. Princip spočívá v tom, že Prologovský program je přeložen do WAM a poté do jazyka C# tak, že pro každý predikát je vytvořena skupina tříd se společným předkem pro pokrytí všech situací. Vygenerované třídy musí být před použitím přeloženy překladačem jazyka C# a načteny do programu jako knihovna pro .NET. Program lze přeložit i do spustitelného souboru, kde predikát, který se jako první volá, je předán pomocí parametru příkazové řádky. Pokud tento parametr není zadán, hledá se predikát `main`. Dynamické klauzule jsou implementovány pomocí hashovací tabulky. Kód v Prologu musí být dostupný při překladu aplikace buď již v přeložené formě nebo lze ve vývojovém prostředí (pokud to umožňuje) nastavit překlad Prologu do C# před samot-

ným překladem C# do CIL. Za běhu jde poté Prologovský program načíst pomocí reflexe. Zajímavou vlastností je podpora vláken a zámků.

Je možné volat z Prologu kód napsaný v některém z jazyků .NET pomocí vestavěných predikátů pro volání metod objektu a statických tříd, čtení datových členů, vytváření objektů a načítání .NET sestavení[3]. Pro lepší čitelnost je pro volání metod používán operátor `:/2` s typem `xfy`. Použití pak vypadá následovně: `Objekt:'Metoda'(parametr1, parametr2, NavratovaHodnota)`. Opačné volání je také podporováno, dotaz se sestaví z instancí tříd reprezentující jednotlivé termy. Volání pomocí Prologovského kódu ve formě řetězce není podporováno. [7]

4.2.2 Prolog.NET

Tato implementace překládá program na kolekci instrukcí WAM, kterou interpret následně vykonává. Program lze uchovávat pouze ve formě kódu v Prologu, výsledné WAM instrukce nelze uložit. Jedná se o jednodušší implementaci, které chybějí některé klíčové vlastnosti jako možnost změny programu za běhu (predikáty `assert/1` atd.). Též se zde neobvykle uvozují dotazy pomocí `:-` a nikoli `?-`. Z prostředí Prologu není možné pracovat s objekty nebo volat jejich metody. Naproti tomu je ale volání ze strany .NET a přebírání výsledků jednoduché. Dotazy se předávají interpretu formou řetězce. Není možné předat jiné konstanty, než ty, které umožňuje syntaxe jazyka Prolog. Výsledek se potom předává speciálním objektem, který obsahuje výsledné hodnoty proměnných pod indexy podle pořadí, jak se v dotazu vyskytly. [2]

4.2.3 C#Prolog

V době psaní tohoto textu se jedná o zřejmě jedinou dosud udržovanou implementaci. Nevychází z WAM, ale z architektury na podobném principu jako ZIP popsaný v 3.3. Podporuje téměř vše, co je v jiných implementacích Prologů běžné. Navíc obsahuje speciální vestavěné predikáty pro práci s databází, XML a JSON soubory, komplexní čísla a DCG[4]. Dotazy se interpretu předávají formou řetězců, není tedy možné předat nic víc než konstanty, které umožňuje zapsat syntaxe Prologu. Výsledky jsou poté vraceny formou enumerátoru[5], který vrací vždy jedno řešení. Objekt s výsledkem obsahuje informaci zda dotaz uspěl. V případě úspěchu také kolekci proměnných a jejich hodnot. Zajímavý je zvolený způsob propojení s platformou .NET. Tato implementace je šířena formou zdrojových kódu a umožňuje v jazyce C# definovat nové predikáty jejich doplněním na určité místo zdrojových souborech. Každý takový predikát má k dispozici seznam parametrů. Pokud uspěje, může provést unifikaci s proměnnými v parametrech, a vrací `true`. V případě, že predikát neuspěje, vrací `false`. Po zkompilování zdrojových kódů je tento predikát dostupný v interpretu Prologu. [1]

4.3 Návrh propojení .NET a Prologu

Aby nevznikl pouze interpret Prologu napsaný v prostředí .NET, ale interpret umožňující propojení obou prostředí a využívání jejich výhod, je nutné, aby byla zajištěna komunikace mezi objektově orientovaným prostředím a prostředím logických jazyků. Toto propojení by

[3] Balíček tříd pro platformu .NET

[4] Definite clause grammar

[5] obdoba iterátorů známých z C++ nebo Javy

mělo být realizováno pomocí standardních prostředků obou platforem a mělo by splňovat následující požadavky:

- Podpora objektů .NET jako typů Prologu
- Interakce a vytváření objektů .NET pomocí speciálních predikátů
- V prostředí .NET maximálně využívat výhod technologie LINQ
- Snadné volání Prologu z prostředí jazyků .NET včetně předávání reference na objekty
- Snadný převod kolekcí do Prologovských seznamů a zpět
- Podpora přístupu k prvkům Prologovskch struktur z .NET

4.3.1 Propojení z Prologu na .NET

Interpret by měl umožňovat pracovat s jakýmkoli datovým typem v .NET. Přímo jako konstanty lze zapsat desetinná čísla (datový typ `double`), celá čísla (podle velikosti buď typ `integer` nebo `BigInt`) a řetězce (typ `string`). Struktury a seznamy jsou definovány speciálními datovými typy (popsáno dále). Ostatní hodnoty jsou chápány jako reference na objekt v .NET. Při unifikace dvou referencí se využije metoda `Equals()`, kterou má každý objekt. Aby bylo možné pro reference použít predikáty `</2`, `>/2` atd., musí objekty implementovat rozhraní `IComparable`. Hodnoty typu `bool` reprezentují v Prologu atomy `cli_true` a `cli_false`. Konstantu `null` potom atom `cli_null`.

Pro vytváření nových instancí tříd, volání jejich metod nebo čtení dat budou sloužit speciální vestavěné predikáty. Návrh částečně vychází z [21] (liší se v sémantice predikátu `get`) a obsahuje predikáty:

- `cli_new('NazevTridy'(parametry...), Ref)` - Vytvoří novou instanci dané třídy. Pokud třída neexistuje nebo nemá konstruktor pro dané parametry, selže.
- `cli_send(Ref, 'NazevMetody'(parametry...))` - Volání metody daného objektu bez návratové hodnoty.
- `cli_send('NazevTridy', 'NazevMetody'(parametry...))` - Volání statické metody bez návratové hodnoty.
- `cli_send(Ref, 'NazevMetody'(parametry...), Output)` - Volání metody s návratovou hodnotou.
- `cli_send('NazevTridy', 'NazevMetody'(parametry...), Output)` - Volání statické metody s návratovou hodnotou.
- `cli_get(Ref, 'NazevDatovehoClenu', Output)` - Čtení obsahu datového členu objektu.
- `cli_set(Ref, 'NazevDatovehoClenu', NewValue)` - Zápis do datového členu objektu.
- `cli_instance_of(Ref, 'NazevDatovehoTypu')` - Obdoba operátoru `is` z jazyka C#. Ověří, zda je instance daného typu nebo jeho potomků.

- `cli_reference(Ref)` - Uspěje, pokud reference vede na objekt, který nereprezentuje přímo datový typ jazyka Prolog.

Predikát pro zrušení instance není potřeba, protože .NET má automatickou správu paměti. V případě objektů, které implementují rozhraní pro explicitní dealokaci zdrojů (objekty pracující se zdroji mimo .NET, například se soubory) lze dealokaci zajistit voláním metody `Dispose()` pomocí predikátu `cli_send/2`.

4.3.2 Propojení z .NET na Prolog

Všechny datové typy Prologu jsou dostupné i jako typy .NET. Číselné typy a řetězce jsou reprezentovány přímo typy .NET. Strukturu reprezentuje třída umožňující pod indexem přístup k jednotlivým prvkům a názvu struktury. Atom je pak speciální případ struktury s nula prvky. Seznam je reprezentován třídou implementující rozhraní `IEnumerable`, díky tomu na něj lze aplikovat všechny operátory technologie LINQ (například převod do kolekce, filtrování apod.). Pro převod jakékoli kolekce (včetně pole) na Prologovskou reprezentaci bude implementován speciální operátor LINQ `ToPrologList()`. Případně ho lze vytvořit přímo konstruktorem s výčtem jeho prvků.

K výsledkům lze přistupovat pomocí enumerátoru. Výpočet se vždy po nalezení řešení zastaví a předá výsledek. Pokud procházení pomocí enumerátoru vyžaduje další řešení, výpočet se znovu spustí. Pro zjištění, zda vůbec existuje nějaké řešení, lze použít LINQ operátor `Any()`, k získání pouze prvního výsledku operátor `First()` atd. Každé řešení je vráceno formou objektu, který obsahuje informaci o úspěšnosti řešení. Pokud je úspěšné, tak umožní podle názvu proměnné získat její hodnotu. Dotaz je možno zapsat buď řetězcem obsahující kód přímo v jazyce Prolog nebo vytvořením objektové reprezentace. Pro tvorbu složitějších dotazů jsou u tříd reprezentující Prologovský dotaz přetíženy operátory `&` a `|`. Příklad 4.2 demonstruje volání Prologovského kódu z jazyka C# pomocí dotazu zapsaného řetězcem. Je zde demonstrován způsob předávání referencí na objekt pomocí notace `@číslo_parametru`. Příklad 4.3 pak ukazuje tvorbu dotazu pomocí objektů. Podrobnější popis použitých tříd následuje v kapitole 5.

```
1     CliProlog program = CliProlog.FromFile("source.pl");
2     MyClass obj = new MyClass();
3     IEnumerable<PrologResult> results;
4     results = program.Query("p(atom, Y), q(@0, X)", obj);
5
6     foreach (PrologResult result in results)
7     {
8       Console.WriteLine("X: {0}, Y: {1}", result["X"], result["Y"]);
9     }
```

Program 4.2: Ukázka principu volání Prologu z jazyka C#

```
1     CliProlog program = CliProlog.FromFile("source.pl");
2     MyClass obj = new MyClass();
3     IEnumerable<PrologResult> results;
4
5     var X = Prolog.Variable("X");
6     var Y = Prolog.Variable("Y");
7     var p = Prolog.Predicate("p", Prolog.Atom("atom"), Y);
8     var q = Prolog.Predicate("q", obj, X);
9
10    results = program.Query(p & q);
11
12    foreach (PrologResult result in results)
13    {
14      Console.WriteLine("X: {0}, Y: {1}", result["X"], result["Y"]);
15    }
```

Program 4.3: Vytvoření dotazu v Prologu pomocí objektů

Kapitola 5

Návrh interpretu

Implementace překladače bude vycházet z Warrenova abstraktního stroje. A to z důvodu dostupného detailního popisu jeho funkčnosti včetně optimalizací a přijatelné složitosti implementace.

Jak je vidět, z již existujících implementací Prologu pro .NET popsaných v kapitole 4.2, existují v podstatě dva scénáře použití. Buď chce uživatel část aplikační logiky vytvořit v Prologu nebo plánuje použít Prolog jako skriptovací jazyk uvnitř aplikace tak, že kód (přímo nebo přes nějaký grafický editor) tvoří až uživatel aplikace za běhu. Aby bylo možné pokrýt oba tyto přístupy, je nutné rozdělit interpret na dvě části: překladač Prologu do objektové reprezentace a na samotný interpret, který převede objektovou reprezentaci do instrukcí WAM a umožní je vykonat.

V druhém případě, tedy že v Prologu je napsána část aplikační logiky, je do jisté míry inspirována implementací P# (viz. 4.2.1). Kód programu v Prologu je napsán ve speciálním souboru a před samotným překladem aplikace je tento soubor přeložen do jednoduchého kódu v jazyce C#, který třídou obaluje již předkompilovaný WAM kód. Při každém startu aplikace se tak ušetří překlad Prologu do WAM. Takový překladač bude implementován jako úloha pro aplikaci MSBuild[1], která umožní jak ruční překlad, tak integraci do vývojových prostředí jako Visual Studio. Tento přístup může v budoucnu umožnit využití ladicích nástrojů systému Visual Studia (zejména krokování kódu). Podobný přístup překladu používá také implementace jazyku pro tvorbu lexikálních a syntaktických analyzátorů ANTLR[2] pro .NET. Veřejná část API interpretu je detailně popsána na obrázku v příloze A. [16] [5] [14]

5.1 Architektura interpretu

Implementace bude, z důvodu větší přehlednosti, rozdělena do tří hlavních částí:

Model Prologu obsahuje třídy pro modelování Prologovského programu. Neobsahuje žádnou vyhodnocovací logiku.

Interpret WAM umožňuje vyhodnotit program zapsaný pomocí tříd z modelu.

CLIProlog spojuje obě předchozí části a přidává k ním analyzátor zdrojových kódů v jazyce Prolog, který zdrojové kódy překládá do objektové reprezentace. S tímto rozhraním přijde primárně do styku uživatel. Dále obsahuje integraci Prologu do MSBuild

[1] Překladač distribuovaný s rámcem .NET, umožňující rozšíření o další jazyky
[2] ANother Tool for Language Recognition - http://www.antlr.org

a další vývojové nástroje. Název CLIProlog byl zvolen podle označení *Common Language Infrastructure* používaný v kontextu spolupráce jazyků na platoformě .NET.

Při volání dotazu je nejprve dotaz přeložen do WAM kódu. Následně je vytvořena instance interpretu, která vychází z návrhového vzoru Iterator. V nekonečném cyklu jsou načítány instrukce z programu (popřípadě dotazu), které jsou předávány interpretu a ten je vykoná. Datové struktury WAM (jako registry nebo zásobníky) jsou součástí tohoto iterátoru. Teoreticky tedy lze nad jedním programem vykonávat více dotazů. Tato architektura otvírá možnost pro budoucí podporu více vláken v programu. Bylo by ale potřeba zajistit, aby metody pro práci s databází (vytváření nových generací) byly vláknově bezpečné. V implementaci popsané v této práci není více vláken uvažováno.

Vykonání instrukce končí jedním ze tří stavů: úspěch, úspěch při kterém má být výpočet přerušen (a předány výsledky) a neúspěch. Úspěch s přerušením výpočtu využívá pouze instrukce `query_end`, která se vyskytuje jako poslední v dotazu. Tato instrukce není přímo součástí WAM, ale byla přidána v této implementaci pro jednoduchou detekci konce dotazu. Po dosažení této instrukce je výpočet zastaven a po jeho opětovném zahájení se interpret chová jako kdyby instrukce `query_end` selhala. Na základě toho je vyvoláno zpětné navracení.

5.2 Překlad

Jak již bylo zmíněno, překlad je rozdělen na dvě části: překlad z Prologu do reprezentace pomocí objektů a následný překlad do instrukcí WAM. Důvodem rozdělení je, že v některých situacích je potřeba použít pouze jednu část překladu. Například při predikáte `read/1` se text převádí do objektové reprezentace, ale již se nepřekládá do instrukcí WAM. Naopak je tomu u predikátů `call/1` a `assert/1`, kdy je pouze potřeba přeložit objektovou reprezentaci do WAM, ale nepracuje se přímo se zdrojovým textem v jazyce Prolog. Především z časových důvodů nebude v interpretu implementována možnost definovat vlastní operátory. Díky tomu je analýza zdrojových textu výrazně jednodušší. V první fázi překladu bude využit pro lexikální a syntaktickou analýzu nástroj ANTLR4. Tento nástroj je generátor analyzátorů shora dolů typu *Adaptive LL(*)*, zkráceně *ALL(*)*. Jedná se o modifikaci analyzátorů *LL(k)*. Bližší informace lze nalézt v [15] a v [14]. Pro gramatiku definovanou v jazyce ANTLR je vygenerován kód v jazyce cílové platformy (v tomto případě C#, ale podporována je i Java a další), který vytvoří derivační strom. Spolu s ním jsou vygenerovány i podpůrné třídy podle návrhových vzorů Visitor a Observer. Pomocí těchto tříd lze procházením derivačního stromu sestavit abstraktní syntaktický strom. Ten bude realizován právě třídami z modelu. [14]

Překlad z objektové reprezentace do WAM zajišťuje přímo interpret. Tento přístup je nutný, protože překlad může probíhat i při vykonávání programu (například při predikátech `assert` apod.). Součástí modelu je i třída, umožňující procházení Prologovským programem reprezentovaným pomocí objektů. Tato třída je implementována podle návrhového vzoru Visitor. Díky potomkům této třídy je pak možné různými způsoby procházet program a generovat jednotlivé instrukce WAM. Výsledkem překladu je potom instance třídy `WamProgram` (nebo jejích potomků) obsahující již přeložené procedury složené z WAM instrukcí.

5.3 Reprezentace programu

Program se skládá z instrukcí. Každou instrukci reprezentuje struktura[3] `WamInstruction`. Typ instrukce je definován výčtovým typem. Struktura dále obsahuje dva členy typu `Int32`, o jejichž sémantice rozhoduje typ instrukce. Většinou se používají na index registru (ať už pracovní nebo registry na zásobníku) se kterými instrukce pracuje. Navíc ještě obsahuje jeden člen typu `object` sloužící pro referenci na libovolný objekt z .NET. Nejčastěji je využíván pro uložení konstant, jmen volaných predikátů nebo tabulek u indexování.

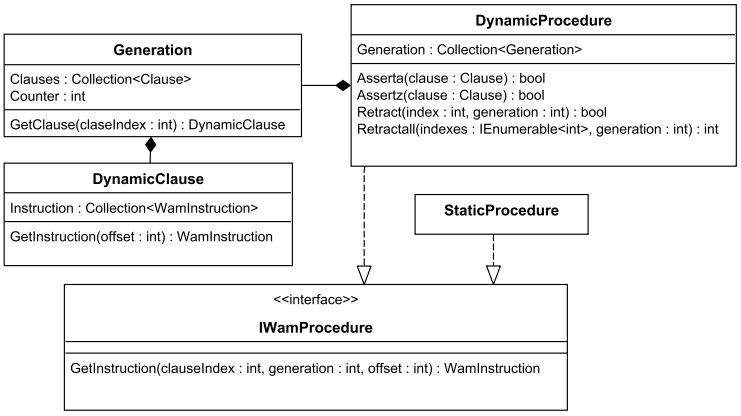
Reprezentace celého programu skládající se z instrukcí musí umožňovat jednoznačně adresovat každou instrukci. Zároveň je třeba vědět na jakých adresách začínají jednotlivé procedury (definovány názvem a aritou). Díky dynamickým klauzulím s logickou změnou (viz 2.2.5) je také nutné rozlišovat u jednotlivých klauzulí generaci databáze. Jelikož se vždy po dokončení procedury skáče na jiné místo v programu, nemusí být celý program za sebou v jedné datové struktuře. Lze ho rozdělit na jednotlivé procedury a klauzule. Ty poté obsahují instrukce WAM. Adresa instrukce je pak čtveřice: `(procedura, generace, klauzule, ofset)`. Ofsetem se myslí číslo instrukce v dané klauzuli dané procedury. Interpret pak musí udržovat informaci o aktuální generaci databáze. Aktuální generace, se kterou se pracuje při vyhodnocení klauzule, je pak uložena přímo v adrese. Každá procedura je reprezentována objektem a rozlišují se tři základní typy: statická procedura, dynamická procedura a vestavěná procedura.

Obrázek 5.1 znázorňuje objektový návrh reprezentace procedur. Každá procedura musí implementovat rozhraní `IWamProcedure`, které obsahuje metodu pro získání instrukce pro konkrétní klauzuli v dané generaci. Obsahuje také metody, které bude interpret volat před a po volání procedury. Interpret tedy nerozlišuje, s jakým typem procedury pracuje. Každý typ procedury volí jiný systém uložení instrukcí. Například u statických procedur není potřeba řešit generace nebo uvažovat změny za běhu v jejich obsahu a díky tomu lze zefektivnit přístup k instrukcím uloženým v proceduře.

U dynamických klauzulí je situace komplikovanější. Každá generace je reprezentována instancí třídy, která obsahuje jednotlivé klauzule. V rámci klauzule je uložen interpretovatelný WAM kód. Jak bylo uvedeno v kapitole 3.1.3, první instrukce dané klauzule se liší podle toho, zda je v rámci procedury první, poslední nebo někde mezi. Různá instrukce na začátku klauzule by zabránila opakovanému používání již přeložených dynamických klauzulí, neboť při odebrání první klauzule by se musela vytvořit kopie druhé klauzule v pořadí a změnit její první instrukci. Pouhá změna instrukce není možná, protože díky logické změně databáze (viz 2.2.5) mohou nějaký čas existovat obě generace současně. Řešením je vytvoření speciální instrukce `clause_start`, která podle pořadového čísla procedury v adrese zvolí správnou instrukci, kterou je v daném místě potřeba provést. Tím se výrazně zjednoduší tvorba generací databáze a zamezí se zbytečnému vytváření nových objektů reprezentující klauzuli.

Díky návrhu znázorněném v obrázku 5.1 lze vytvářet objekty reprezentující generaci jen pro metody, kterých se změna týká. Tento přístup je velmi výhodný, protože vestavěné predikáty pro dynamické klauzule pracují vždy jen s jednou procedurou. Na druhou stranu přináší komplikovanější vyhledání odpovídající verze procedury. Pokud se přímo požadována verze v proceduře nenachází, musí být vyhledána nejbližší nižší verze. Také je potřeba zajistit odstranění generací, se kterými už žádné vyhodnocování nepracuje. Každý objekt reprezentující generaci v rámci dynamické procedury obsahuje čítač, který ukazuje kolikrát je daná generace využívána při vyhodnocování. Každá procedura, která začne vyhodnocovat

[3] Hodnotový typ plaformy .NET



Obrázek 5.1: UML diagram vnitřní reprezentace procedur

klauzuli v dané generaci, tento čítač zvýší. Až vyhodnocování skončí, je tento čítač zase snížen. Ve vhodný okamžik (např.: konec volání dynamické procedury) může interpret určit, které generace jsou již zastaralé a nepoužívané a zrušit na ně referenci. Jejich paměť pak uvolní Garbage Collector. Taktéž pokud je vytvářena nová generace a starší verze není používána, může být rovnou odstraněna.

5.4 Reprezentace datových struktur

Protože jedním z požadavků na interpret je práce s objekty platformy .NET, je vhodné reprezentovat všechny datové struktury WAM pomocí objektů. Jako halda popsaná ve WAM pak funguje přímo paměť. Komplikací je, že WAM v několika případech předpokládá, že objekty jsou v paměti umístěny souvisle a adresy lze mezi sebou porovnávat (a zjistit, který zápis je v paměti novější). V případě určování nebezpečných proměnných (viz kapitole 3.1.2) to není problém, protože tento typ proměnných a instrukce s nimi pracující není potřeba vůbec uvažovat. Jak již bylo zmíněno, koncept nebezpečných proměnných řeší problém, že paměťová buňka bude odstraněna v době, kdy na ni povede reference. To z principu fungování automatické správy paměti v .NET není možné. V druhém případě je potřeba vědět, zda byla proměnná alokována před nebo po vzniku posledního bodu volby (kvůli rozhodnutí, zda má být přidána na zásobník cesty). Pro tento účel bude každá instance proměnné označena unikátním pořadovým číslem (přidělována na základě statické proměnné třídy).

Objekty reprezentující datové struktury WAM mohou být rozděleny na dva typy, interní pro interpret a veřejné, které se používají pro vstup a výstup (nacházejí se v části s modelem). Atomy jsou implementovány jako jednoduchá struktura obsahující řetězec reprezentující název. Zároveň má překrytou metodu `Equals`, která zajistí, že dvě instance se stejným názvem jsou ekvivalentní. Ostatní konstanty, jako čísla nebo řetězce jsou reprezentovány přímo svým typem z .NET. Práci s atomy a ostatními konstantami lze tedy z pohledu interpretu sloučit. V případě unifikace dvou konstant je výsledek unifikace závislý na výsledku metody `Equals`. Všechny typy jazyka Prolog navíc implementují rozhraní `ITerm`.

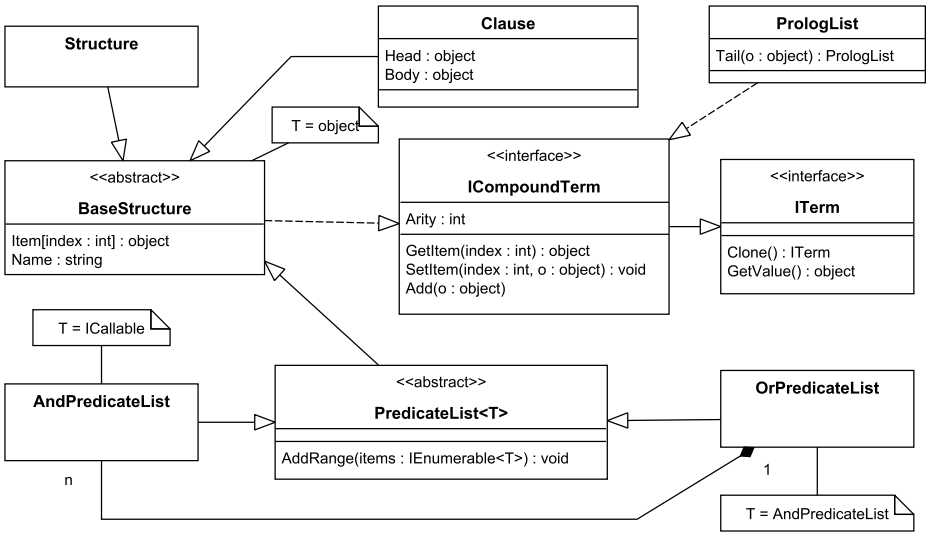
Speciální přístup vyžadují proměnné. Pro jejich reprezentaci se využívá více typů objektů. První se používá na vstupu (při definici programů nebo dotazů). Zde je proměnná podobně jako atom reprezentována strukturou, která obsahuje pouze její název. Díky tomu,

že je struktura hodnotový datový typ, není možné očekávat, že by po vykonání dotazu její objekt obsahoval řešení. Slouží pouze pro zapsání jména proměnné. Při vyhodnocování používá interpret vnitřně jinou její reprezentaci, která již umožňuje proměnné nastavit hodnotu. Jedná se o třídu (lze na ni tedy odkazovat referencí), která mimo názvu (který je volitelný) obsahuje ještě prvek typu `object`, představující hodnotu, která je na proměnnou navázaná. Nenavázaná proměnná má v tomto datovém členu hodnotu `null`. Unifikace je řešena obdobně jako ve WAM, tedy že pokud jsou unifikovány dvě nenavázané proměnné, je jedna nastavena jako hodnota druhé a před každou prací s danou proměnnou je nalezena poslední hodnota v tomto zřetězení. Poslední reprezentace proměnné se vyskytuje ve výsledku. Zde je potřeba postihnout situaci, kdy se dvě proměnné unifikují v jednu. Ve výsledcích jsou pak obě proměnné reprezentovány jedním objektem obsahujícím všechny jejich názvy a hodnotu. K nahrazení z vnitřní reprezentace na výstupní dojde po dokončení výpočtu. Díky tomu, že se nevrací přímo vnitřní reprezentace proměnné, je zaručeno, že mezi vrácením výsledku a vyvoláním zpětného navracení pro získání dalšího výsledku, nemůže uživatel změnit vnitřní stav interpretu a ovlivnit tak výpočet.

Prolog má dva typy složených termů: strukturu a seznam. Všechny ostatní termy jako klauzule, seznam predikátů v těle klauzule atd. jsou jen speciální případy struktury s konkrétním názvem. Tato struktura je pak často v programu zapisována v infixové variantě. Například klauzule `p(X) :- r(X), g(X).` je ekvivalentní s `:-(p(X), ','(r(X), g(X)))`. Z praktických důvodu je výhodné, aby například struktura s názvem `:-` byla v paměti reprezentována třídou speciálně pro klauzuli. Každý složený term implementuje rozhraní `ICompoundTerm`, které umožňuje přístup k jeho prvkům. V případě seznamu toto rozhraní reprezentuje složený term o dvou prvcích. První je hodnota a druhý zbytek seznamu. Tento přístup je výhodný, jak bude popsáno v kapitole o překladu do WAM. Všechny ostatní složené termy vycházející ze struktury jsou potomci abstraktní třídy `BaseStructure`. Obecnou strukturu pak reprezentuje třída `Structure`. Aby bylo zabráněno vytvoření instance obecné struktury, například s názvem `:-` místo speciální instance třídy pro klauzuli, jsou konstruktory jednotlivých tříd označeny jako `protected` a jejich instanci lze vytvořit pouze pomocí třídy implementující návrhový vzor Abstract Factory. Celou reprezentaci složených termů demonstruje UML diagram na obrázku 5.2.

Ne všechny termy se mohou vyskytovat na nejvyšší úrovni v rámci těla klauzule nebo v dotazu. Ty, které mohou, implementují rozhraní `ICallable` a patří mezi ně struktura, atom, seznam predikátů a proměnná. V případě proměnné ji v nejvyšší úrovni překladač překládá jako `call(X)`. Mezi některými termy jsou implementovány konverze, například atom lze implicitně přetypovat na strukturu s nula prvky nebo explicitně lze přetypovat strukturu na klauzuli (vznikne fakt). Jak již bylo naznačeno v předchozí kapitole, mezi objekty `BaseStructure` jsou přetíženy operátory `&` a `|`, které umožňují jednodušší a čitelnější tvorbu seznamu predikátů včetně zanořování pomocí závorek.

Aritmetické výrazy, používané zejména ve spojitosti s operátorem `is/2`, jsou běžné obecné struktury, jejichž název odpovídá symbolu aritmetické operace. Většina těchto struktur lze zapsat v infixovém zápisu. V případě, že by bylo požadováno, aby se výraz v infixové podobě i vypisoval (například v konzoli), je možné pro něj vytvořit vlastní třídu dědící od `Structure` a překrýt metodu `ToString()`. Následně stačí upravit abstraktní továrnu, aby pro dané názvy struktur vytvářela instance této nově definované třídy.



Obrázek 5.2: UML diagram objektového modelu složených termů

5.5 Předávání výsledků

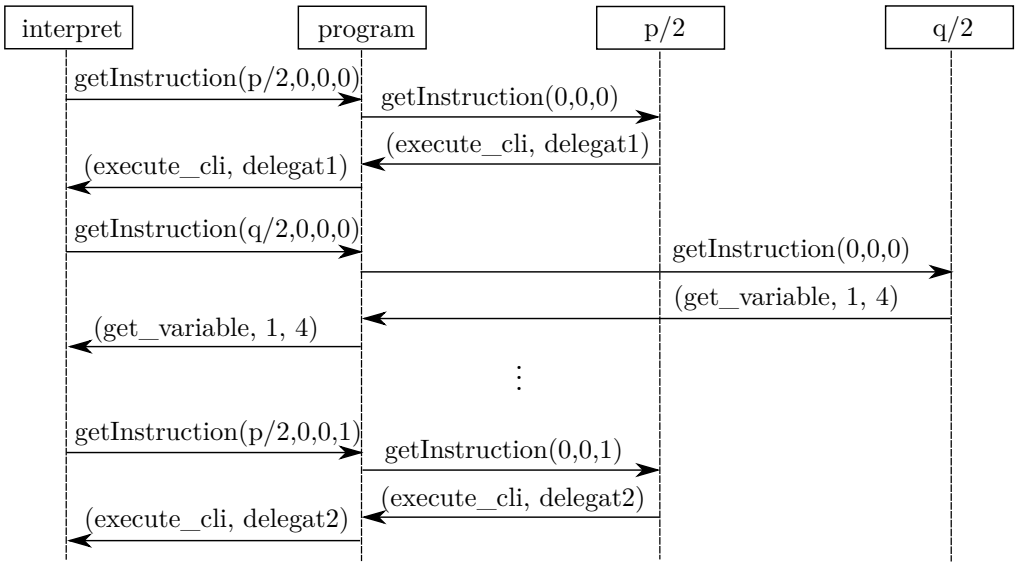
Každé volání dotazu z prostředí .NET vytváří novou instanci interpretu, která postupně čte instrukce a vykonává příslušný kód. Vždy, když narazí na instrukci, která přeruší výpočet, je vytvořen speciální objekt obsahující hodnoty jednotlivých proměnných v dotazu. Tohoto chování je dosaženo návrhovým vzorem Iterator, který je na platformě .NET implementovaný pomocí rozhraní `IEnumerable`. Toto rozhraní umožňuje získat objekt zvaný enumerátor sloužící k procházení dat. Tento objekt obsahuje metodu `MoveNext()` a datový člen `Current`. Při volání metody `MoveNext()` začne interpret vykonávat jednotlivé instrukce, dokud nenarazí na příkaz pro vrácení výsledků nebo nedojde k selhání, při kterém již neexistuje žádný bod volby. Pokud byl dotaz úspěšný, vrací hodnotu `true`, jinak `false`. Opětovné volání `MoveNext()` vyvolá zpětné navracení a výpočet pokračuje. Po úspěšném volání `MoveNext()` je vždy v datovém členu `Current` umístěn objekt implementující rozhraní `IPrologResult`, které obsahuje kolekci všech proměnných s hodnotami. K získání hodnoty konkrétní proměnné je využita syntaxe s hranatými závorkami, podobně jako u přístupu k prvkům pole. V tomto případě je hodnota indexována řetězcem.

Díky využití rozhraní `IEnumerable` z knihovny .NET lze použít na zpracování výsledků technologii LINQ nebo výsledky procházet cyklem `foreach`. V případě Prologu je možné v některých případech rozhodnout, jestli má smysl zkoušet další volání `MoveNext()`. Pokud na zásobníku prostředí není žádný bod volby, skončí toto volání vždy neúspěchem. To lze využít například při implementaci interaktivní konzole, kde v případě, že není možné spočítat žádný další výsledek, není zobrazena výzva uživateli, zda chce výpočet ukončit a nebo pokračovat. Tuto informaci lze získat přetypováním enumerátoru na nové rozhraní `IResultEnumerator`. To obsahuje metodu `CanHaveNextResult()`, která vrací `true`, pokud existuje nějaký bod volby. Přetypování je nutné pro zachování kompatibility s technologií LINQ, protože pracuje pouze přímo s rozhraním `IEnumerable`.

5.6 Vestavěné a vložené predikáty

Vestavěné predikáty jsou takové predikáty, které jsou přímo součástí interpretu. Některé z nich nejsou z výkonnostních důvodů implementovány přímo v Prologu (nebo ve WAM), ale přímo v jazyce, ve kterém je implementován interpret. Aby bylo možné zavolat interpretem i jiný kód než pouze vykonání WAM instrukce, byla do instrukční sady přidána instrukce `execute_cli`, která jako jediný parametr předává delegát na .NET metodu. Při jejím vykonání je zavolán tento delegát, který jako parametr dostane referenci na objekt interpretu. Aby bylo možné s interpretem pracovat, mají všechny jeho metody, implementující chování instrukcí, modifikátor přístupu `internal`[4].

Volání vestavěných predikátů není přímo překládáno jako instrukce `execute_cli`. Volají se standardně pomocí instrukcí `call` a `execute`. Instance třídy reprezentující vestavěný predikát je uložena, stejně jako ostatní predikáty, v tabulce procedur v programu. Po zavolání její metody `GetInstruction()` ale nevyhledává instrukci v již přeloženém WAM kódu, ale vrací instrukci `execute_cli` s delegátem na metodu, která má být zavolána. Toto řešení bylo zvoleno z důvodu, že implementace vestavěného predikátu se nemusí skládat pouze z jednoho volání metody. V některých případech je nutné předat řízení zpět interpretu uvnitř kódu vestavěného predikátu a po jeho vyhodnocení opět pokračovat další metodou. Tento případ nastává v situacích, kdy je volán další predikát nebo pokud predikát vytváří body volby. O tom, jaká instrukce bude interpretu vrácena, rozhoduje vestavěný predikát podle adresy (přesněji podle čísla klauzule a indexu instrukce). Obrázek 5.3 zobrazuje situaci, kdy vestavěný predikát `p/2` uprostřed svého kódu potřebuje zavolat `q/2`. Interpret nejdříve požaduje instrukci po `p/2`, který mu vrátí instrukci `execute_cli` s delegátem na konkrétní metodu, kterou interpret zavolá. V předané metodě je provedeno volání `q/2` (nastaví adresu další instrukce na `q/2` a adresu pro návrat na další instrukci v `p/2`). Interpret pak provede instrukce v kódu `q/2` a po návratu vyžaduje další instrukci po `p/2`. Podle adresy požadované instrukce `p/2` rozhodne, že se jedná o druhou část vykonávání a vrátí delegát na další metodu. Reálný příklad takového vestavěného predikátu je `not/1`.



Obrázek 5.3: Diagram volání při vyhodnocování vestavěného predikátů

[4] ve svém sestavení se chovají jako *public*, jinak jako *private*

Některé vestavěné predikáty nepoužívají zpětné navracení ani neprovádí volání jiných predikátů (např.: `==/2`, `var/1`). Lze je tedy volat vložením jejich instrukcí na místo volání a vynechat tak instrukce `call`, případně alokaci místa na zásobníku. Takový predikát se podle [8] nazývá vložený (anglicky *inline predicate*). V této implementaci je reprezentován implementací rozhraní `IBinaryInlinePredicate` resp. `IUnaryInlinePredicate` pro predikát s dvěma, resp. jedním parametrem. Zavolání vloženého predikátu zajistí skupina instrukcí `execute_inline_*`, kde je znak hvězdičky nahrazen kombinací jednoho až dvou písmen `X` a `Y`. Tyto znaky určují počet operandů a z jakých registru budou načítány. Díky tomu není nutné pro predikát připravovat parametry do předem určených registrů a výsledný WAM kód je kratší. Seznam vložených predikátů je uložen v tabulce přímo v interpretu a při jeho volání je nalezena odpovídající třída s implementací vloženého predikátu. Pomocí této třídy je predikát vykonán. Odpadá tak i použití instrukce `execute_cli`, která je používána u obecných vestavěných predikátů. Seznam těchto predikátů je neměnný a lze tedy při překladu rozhodnout, zda se jedná o vložený predikát a vygenerovat pro jeho volání speciální instrukci.

Kapitola 6

Implementace

Tato kapitola popisuje způsob implementace interpretu navrženého v předchozích kapitolách včetně implementačních detailů. Podrobně rozebírá obě fáze překladu, vyhodnocování instrukcí, způsob realizace propojení s .NET, integraci interpretu do nástroje MSBuild a integrovaného vývojového prostředí Microsoft Visual Studio.

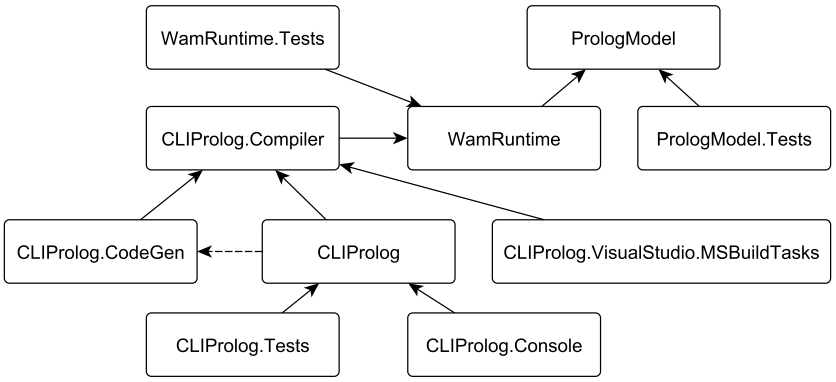
Dělení implementace do menších celků uvedené v kapitole 5 je v případě části CLIProlog nutné ještě více zjemnit. Je potřeba dosáhnout toho, že některé vestavěné predikáty lze implementovat přímo v jazyce interpretu. Pro překlad interpretu je tedy nutné již mít jeho překladačem přeložené predikáty do WAM. Z toho důvodu je implementace rozdělena do několika menších projektů[1], konkrétně na Compiler, který zajišťuje překlad ze zdrojových textů jazyka Prolog do objektové reprezentace. Dále CodeGen je nástroj pro příkazovou řádku, který generuje ze zadaného souboru s Prologovským zdrojovým kódem třídu jazyka C#, obsahující již přeložený WAM. Obdobně MSBuilTask, který je obdobou nástroje CodeGen pro systém MSBuild. Hlavním projektem, který všechny dříve zmíněné spojuje, je CLIProlog. Implementace ještě obsahuje jednoduchou konzolovou aplikaci pro interaktivní práci s Prologem. Vestavěné predikáty napsané v Prologu jsou součástí CLIProlog, je tedy nutné překládat nejdříve nástroj CodeGen, pomocí kterého se vzápětí přeloží vestavěné predikáty do formy knihovny a výsledná třída je použita v CLIProlog. Závislosti všech projektů zobrazuje obrázek 6.1, přerušovaná šipka znázorňuje, že mezi projekty neexistuje závislost na úrovni zdrojových kódu, ale projekt vyžaduje pro svůj překlad již přeložený jiný projekt.

Instrukční sada byla v této implementaci oproti WAM rozšířena. Hlavním důvodem bylo zvýšení efektivity programu a implementace operátoru disjunkce. Instrukce `put_value` a `get_variable` byly sloučeny do jedné pod názvem `move_register`, protože jejich sémantika je stejná a díky optimalizacím ztratilo smysl rozlišovat je podle toho, v které části programu se nachází. Instrukce, které mohou pracovat jak s permanentními tak dočasnými registry, byly rozděleny na dvě různé instrukce. Název instrukcí, které používají permanentní registry končí příponou `_y`. Ostatní přidané instrukce jsou popsány dále v této kapitole. Kompletní tabulku instrukcí lze nalézt v příloze B.

6.1 Překlad z Prologu do objektové reprezentace

Definice vlastních operátorů není součástí implementace popisované v této práci, nicméně je možné ji v budoucnu doplnit. Díky rozdělení implementace do více menších celků by

[1] základní dělící jednotka programů pro .NET, podobná například balíčkům v jazyce Java



Obrázek 6.1: Závislost jednotlivých projektů v rámci implementace

bylo nutné udělat změny pouze v překladu syntaktickém analyzátoru. Díky tomu je možné definovat prioritu operátorů přímo v gramatice Je rozlišováno celkem šest úrovní priorit. Jejich přehled zobrazuje tabulka 6.1.

priorita	**operátory**
1	`:-`
2	`;`
3	`,`
4	`is < = =.. =< == > >= \= \== @< @> @<= @>= =:= =\=`
5	`+ -`
6	`* /`

Tabulka 6.1: Tabulka priorit operátorů

Gramatika Prologu je bez definice uživatelských operátorů poměrně jednoduchá. Jediný problém způsobuje znak čárka, který má v programu dva významy. Slouží jako operátor v těle klauzule (a je to tedy název struktury) a zároveň je použit jako oddělovač prvků ve struktuře nebo seznamu. Je tedy potřeba navrhnout takovou gramatiku, která povolí jako prvek struktury nebo seznamu cokoli mimo operátoru čárka. Operátor čárka se může vyskytnout pouze v případě, že je uzavřen do závorek. Zároveň touto úpravou nesmí být porušena definice priority operátorů. V navržené gramatice se může v rámci jednoho pravidla vyskytovat levá rekurze, protože ANTLR od verze 4 umí tento problém odstranit sám (detaily lze nalézt v [14]). Alternativním řešením by bylo čárku rozlišovat až při tvorbě objektové reprezentace z derivačního stromu (který zajišťuje třída `TreeConverter`). Tento přístup nebyl zvolen, protože by komplikoval generování objektové reprezentace.

Speciální zacházení vyžaduje zpracování direktiv. Ty slouží pro definici dynamických predikátů a v případě této implementace i pro informace nutné k přístupu k Prologovskému programu z .NET. Konkrétněji jsou direktivy použity pro zápis názvu třídy, která bude v .NET reprezentovat daný program (toto samozřejmě neplatí při interpretování kódů načtených za běhu, v tom případě se žádná třída nevytváří). Hned, jak třída `TreeConverter` narazí na direktivu, vyvolá událost, kterou je možné zpracovat vně této třídy. V budoucnu bude tímto způsobem možné udělat například změny v precedenční tabulce atd.

6.2 Překlad objektové reprezentace do WAM

Druhá fáze překladu je také implementována na základě návrhového vzoru Visitor, který umožňuje procházení objektové reprezentace programu a zároveň lze v jeho potomcích měnit pořadí procházených uzlů. Celý překlad zastřešuje třída `WamGenerator`, která pro kolekci klauzulí vrací kolekci jejich překladů do WAM. Třída samotná překlad deleguje na další, specializovanější, a sama zajišťuje pouze zavedení optimalizací, především indexování (podrobněji popsáno dále v této kapitole). Z tohoto důvodu neplatí, že vstupní a výstupní kolekce mají stejný počet prvků.

K překladu jsou nutné dva průchody abstraktním syntaktickým stromem. První provede analýzu potřebnou pro druhý průchod, který již generuje konkrétní instrukce. Protože byly zavedeny optimalizace uvedené v [8], nedělí se klauzule na jednotlivé podcíle (jak je uvedeno ve [4]), ale části (anglicky *chunks*), které obsahují maximálně jeden nevložený predikát (nebo jsou ukončeny operátorem disjunkce). V případě prvního predikátu v těle se za součást jeho části považuje i hlavička. Tato změna pozmění i definici pro permanentní proměnnou. Za permanentní je v tomto případě považována taková proměnná, která se vyskytuje ve více než jedné části. Například klauzule `p(X) :- var(X), c(X,Y), d(Y).` obsahuje dvě části (pokud předpokládáme, že `var/1` je vložený predikát): `p(X) :- var(X), c(X,Y)` a `d(Y)`. Permanentní proměnnou je zde pouze `Y`.

Výsledkem analýzy klauzule je:

- počet výskytů proměnných
- výčet permanentních proměnných
- zda obsahuje klauzule hluboký řez (viz kapitola 3.1.4)
- pozice prvního a posledního výskytu proměnných
- na kterých pozicích v rámci parametrů predikátu se proměnná v dané části nachází (podrobněji popsán v kapitole 6.3)

Kvůli operátoru disjunkce je nutné rozdělovat klauzuli na tzv. větve. Uvažujme například klauzuli `p(X) :- d(X), (c(X,Y), e(Y) ; d(X,Z), e(Z)).`, která obsahuje dvě větve: `c(X,Y), e(Y)` a `d(X,Z), e(Z)`. Pro zobecnění lze i celou klauzuli považovat za větev a předtím zmíněné větve za podvětve. Ačkoli se v klauzuli vyskytují dvě permanentní proměnné (`Y` a `Z`), nacházejí se každá v jiné větvi a lze tedy znovu využít stejný registr na zásobníku. Na další klauzuli `p(X) :- a(X,Y), b(Y) ; c(X,Y).` bude demonstrován vliv větví na počet výskytů proměnných. Proměnné, které se v klauzuli vyskytují pouze jednou (anglicky označovány *void variables*), lze překládat úspornějším způsobem. Navíc je zvykem, že překladač na takovou proměnnou uživatele upozorní. V této klauzuli je taková proměnná `Y`. Je nutné si uvědomit, že ačkoli se v ní vyskytuje celkem třikrát, v každé větvi se jedná o jinou proměnnou. V druhé větvi se tedy vyskytuje jen jednou. V posledním příkladu bude demonstrováno, že v rámci jedné klauzule může být proměnná dočasná i permanentní. Pokud by byl operátor `is/2` implementován jako vložený predikát, mohla by klauzule vypadat takto: `p(X) :- c(X,Y), d(Y) ; Y is X + 1, e(X,Y)`. V první větvi je nutné `Y` alokovat na zásobník, v druhé mezi jejími výskyty není žádné volání a proto ji lze alokovat jen do dočasných registrů.

Při překladu musí překladač pro každou proměnnou vědět, jestli je v aktuální větvi permanentní, případně kolikrát se v dané klauzuli vyskytuje. Pro každou větev (včetně celé klauzule) je přiřazena struktura obsahující počet výskytů proměnných v rámci klauzule a počet výskytů v rámci jednotlivých částí klauzule. Struktura vždy obsahuje pouze proměnné nacházející se v dané větvi, nikoli v podvětvích. Podle prvního údaje lze rozhodnout, zda je nutné proměnnou zpracovávat (není *void variable*) a podle druhého, zda je permanentní. Ukázku výsledků výše popsané analýzy demonstruje tabulka 6.2. Z tabulky lze vyčíst, že `X` je permanentní proměnná, protože je vždy obsažena ve více než jedné části. Naopak proměnná `Y` je permanentní pouze ve větvi `c(X,Y), d(Y)`, kde se vyskytuje ve více než jedné části.

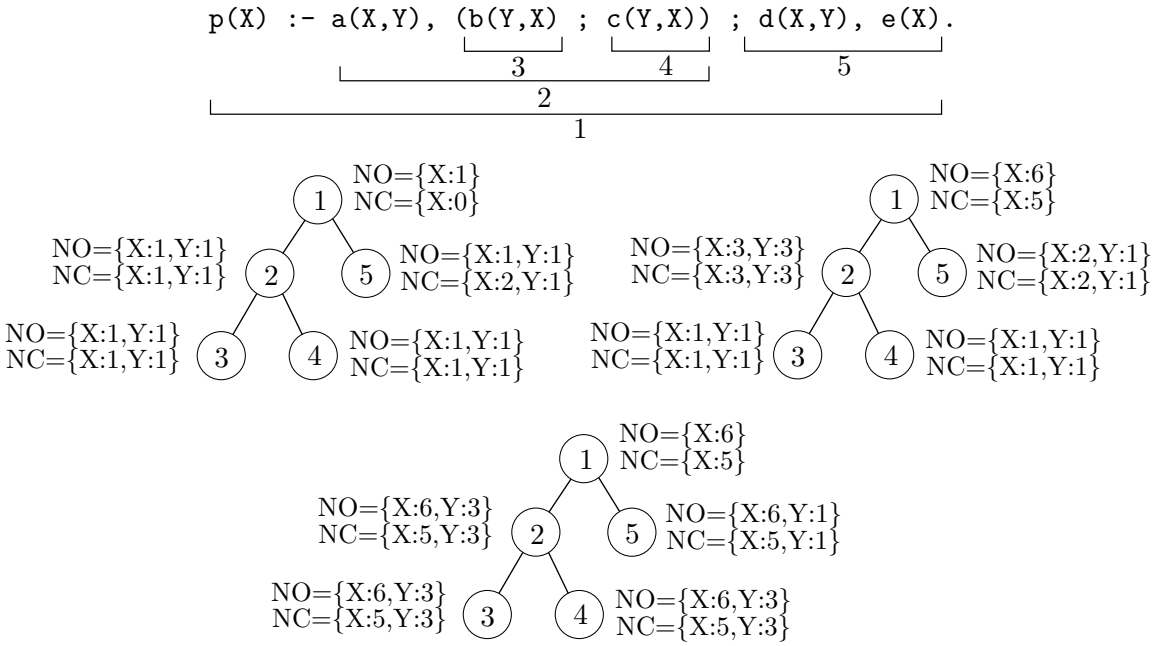
větev	**celkové výskyty `X`**	**celkové výskyty `Y`**	**výskyty v částech `X`**	**výskytu v částech `Y`**
celá klauzule	4	-	2	-
`c(X,Y), d(Y)`	4	2	2	2
`Y is X + 1, e(X,Y)`	4	2	2	1

Tabulka 6.2: Příklad analýzy klauzule `p(X) :- c(X,Y), d(Y) ; Y is X + 1, e(X,Y).`

V implementaci pro uložení informací o jednotlivých větvích je použita stromová struktura. Každý uzel má referenci na objekty reprezentující větve o úroveň nižší a na svou rodičovskou větev. Výsledné hodnoty jsou vypočítány tak, že se postupně prochází celá klauzule (průchodem typu pre-order) a vždy, když narazí na uzel reprezentující operátor disjunkce (třída `OrPredicateList`), je vytvořena nová instance třídy reprezentující informace o větvi. Tím, že je strom procházen rekurzivně, je vždy po dokončení procházení celé větve možné obnovit objekt reprezentující nadřazenou větev. Každý objekt obsahuje dvě tabulky, do kterých ukládá informaci o celkovém počtu výskytů proměnných v dané větvi (označena *NO*) a o počtu částí klauzule, ve kterých se proměnná vyskytuje (označena *NC*). Vždy, když je při procházení nalezena proměnná, zjistí se, zda je již v tabulce celkového počtu výskytů (*NO*). Pokud ano, zvýší se její hodnota v tabulce, pokud ne, je do tabulky vytvořen nový záznam s hodnotou 1. Složitější je situace u počítání počtu částí, kde se proměnná vyskytuje. Pro tento účel používá analyzátor speciální množinu, do které ukládá proměnné, které již nalezl. Vždy, když narazí na konec části klauzule, jsou všem prvkům obsaženým v této množině zvýšeny čítače v tabulce *NC* a množina vyprázdněna. Může se ale stát, že část klauzule zasahuje do více než jedné větve. Například: `p(X) :- r(X,Y) ; g(X,Y).` V tomto případě by se proměnná `X` nevyskytla v tabulce *NC* pro celou klauzuli, protože tato tabulka je naplněna až při dosažení konce části. Z tohoto důvodu, pokaždé když se při procházení narazí na proměnnou, která ještě není v tabulce *NC* pro aktuální větev, je do ní okamžitě přidána, ale s hodnotou 0. Tím je zaznamenáno, že se proměnná v dané větvi vyskytuje, ačkoli hodnota je chybná. Tento fakt ovšem nevadí, k nápravě dojde při druhé části algoritmu.

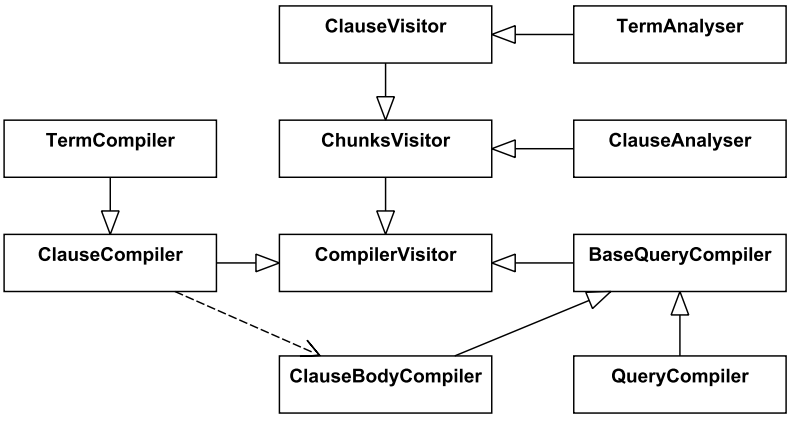
V druhé částí se zajistí, aby ve všech větvích byly o stejné proměnné stejné informace. Tedy každá podvětev ví to samé, co její rodič, ale může k tomu přidat ještě něco navíc. Nejprve jsou ve stromě odspodu sečteny všechny hodnoty v jednotlivých tabulkách. Tím se zajistí, že větev, ve které se proměnná poprvé vyskytla, obsahuje správné údaje. Nyní od vrchu každý uzel předá své hodnoty svým potomkům. Těmito hodnotami potomek nahradí své hodnoty (pokud ale měl nějakou informaci navíc, tak si ji ponechá). Postup výpočtu těchto hodnot demonstruje obrázek 6.2. Z tohoto stromu lze vyčíst, zda je v dané větvi proměnná permanentní (hodnota větší než jedna v *NC*), zda je nutné pro proměnnou alokovat

registr (hodnota větší než jedna v *NO*) a počet permanentních registrů, které je potřeba alokovat na zásobníku (maximum z počtu permanentních proměnných v jednotlivých větvích).



Obrázek 6.2: Příklad výpočtu výskytů proměnných

Aby bylo zabráněno opakování kódu při průchodu objektovou reprezentací klauzule, je implementace rozdělena do více tříd. Třída `ChunkVisitor` rozšiřuje děděním obecnější třídu `ClauseVisitor` o informaci, kterou část klauzule momentálně prochází. Tuto třídu dědí `CompilerVisitor`, který přidává funkce pro generování instrukcí. Tímto způsobem je implementována celá hierarchie tříd podle návrhového vzoru Visitor, která bez zbytečného opakování kódu pokrývá všechny potřebné způsoby průchodu klauzulí (viz kapitoly 6.2.1 a 6.2.2), které jsou při překladu potřeba. Obrázek 6.3 zobrazuje diagram všech těchto tříd.

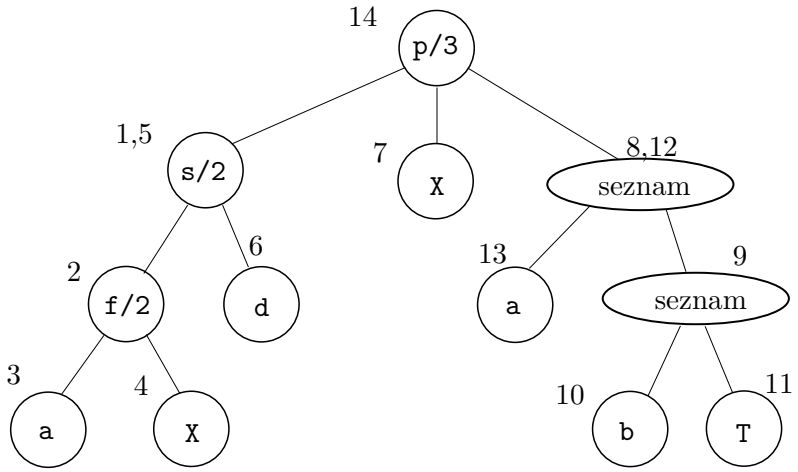


Obrázek 6.3: Diagram tříd pro překlad do WAM

6.2.1 Překlad dotazu

Úkolem dotazu je postupně připravit do registrů parametry pro volání jednotlivých podcílů. Pokud se v dotazu vyskytuje nějaká permanentní proměnná, je hned jako první vygenerována instrukce `allocate`. Přeložené dotazy nikdy nekončí instrukcí `deallocate`, ale jsou vždy zakončeny pomocí `query_end`. Tato instrukce, v případě že po dokončení není prázdný zásobník, provede jeho vyprázdnění (samozřejmě tak, aby nebylo smazáno cokoli, co by mohlo být potřeba při zpětném navracení).

Struktury jako parametry je nutné konstruovat odspodu. Tedy jako první je vytvořena nejvíce zanořená struktura. U složených termů jsou nejdříve zpracovány jeho prvky, které jsou taktéž složené termy a až poté termy jednoduché. Toto neplatí pro nejvyšší term reprezentující predikát. V tomto případě jsou všechny parametry překládány v pořadí, jak se v něm nacházejí. Jak je možné vidět na obrázku 6.4, uzly složených termů (mimo nejvyššího), které obsahují další složený term, jsou navštíveny dvakrát. Poprvé před zpracováním všech podřazených složených termů a podruhé před návštěvou jednoduchých termů. Při první návštěvě je dané struktuře přiřazen registr, při druhé je vygenerována instrukce pro vytvoření struktury do tohoto registru. Díky této strategii průchodu, jsou v době vytváření struktury instrukce pro vytvoření všech zanořených struktur již vygenerovány a samotná struktura je umístěna v tabulce symbolů. Lze tedy vygenerovat instrukci `set_value`, která vloží již vytvořenou strukturu do právě vytvářené struktury, jako kdyby to byla proměnná.



Obrázek 6.4: Pořadí průchodu uzly predikátu `p(s(f(a,X),d), X, [a,b|T])` při překladu

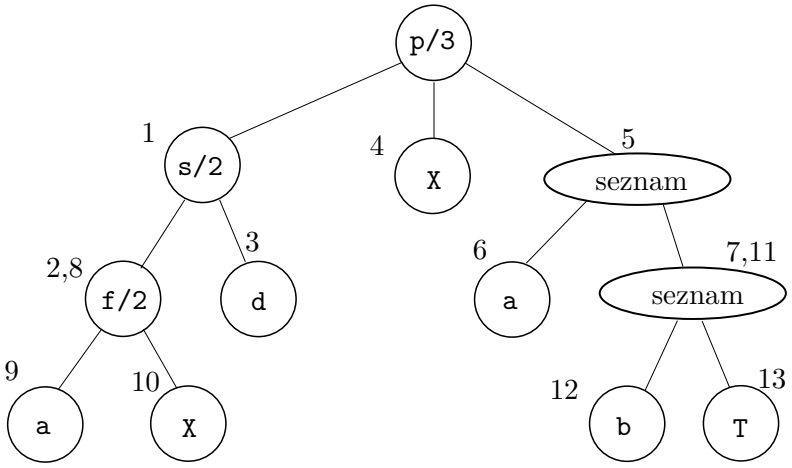
Byla zmíněna tabulka symbolů, která je detailně popsána v kapitole 6.3, stejně jako princip, podle kterého jsou alokovány registry. V tomto okamžiku stačí chápat tabulku symbolů jako vyhledávací tabulku, kde je pro jednotlivé proměnné, struktury, seznamy atd. přiřazen registr, ve kterém se zrovna nacházejí. Zásadní vliv na tuto tabulku má operátor disjunkce. Pokud je do tabulky přidán nový záznam v nějaké větvi, při dokončení překladu této větve musí být z tabulky zase odstraněn. Například v případě klauzule `p(X) :- a(X,Y), b(Y) ; c(X,Y), d(Y).` se v prvních predikátech obou větvích musí `Y` překládat jako kdyby byla viděna poprvé (tedy pomocí instrukcí `put_variable`). Pokud by nebyl záznam z tabulky odstraněn, překládal by se první predikát druhé větvě jako kdyby se již proměnná někde předtím vyskytla. Toto chování je zajištěno tak, že při každém vstupu do nějaké větve je vytvořena nová tabulka symbolů, do které jsou zkopírovány všechny zápisy z nadřazené

větve. V celé větvi se pak pracuje s novou tabulkou a po dokončení překladu větve je tato tabulka odstraněna a opět se pracuje s původní.

Pro dotazy byly do implementace WAM přidány nové instrukce rozšiřující `put_variable` a `set_variable`. Tedy instrukce, které vytvářejí objekt reprezentující proměnnou. Nové instrukce jsou pojmenovány `put_variable_r` a `set_variable_r`. Dělají to samé, co jejich předchůdci, ale navíc ještě proměnnou vloží do speciální kolekce proměnných, které mají být vráceny po úspěšném dokončení výpočtu. Jelikož princip překladu dotazu je používán i při překládání těla klauzule (kde se ale nevyskytují právě popsané instrukce), je implementace rozdělena do více tříd. Třída `BaseQueryCompiler` překládá obecně dotaz, ale bez instrukcí končící `_r`. Její potomek `QueryCompiler` potom překrytím metod zajistí používání `_r` instrukcí.

6.2.2 Překlad programu

Jak již bylo naznačeno v předchozí kapitole, tělo klauzule je překládáno stejně jako dotaz. Zbývá tedy popsat ještě způsob překladu hlavičky. V hlavičce je potřeba postupně unifikovat všechny parametry v registrech s parametry hlavičky, a to v takovém pořadí, aby v případě, že unifikace selže, bylo toto selhání zjištěno co nejdříve. Název predikátu v hlavičce není do kódu vůbec překládán. Předpokládá se, že výběr správného predikátu je zajištěn ještě před prováděním kódu. Jednotlivé parametry jsou postupně procházeny způsobem, že pokud je nalezen jednoduchý term, je okamžitě vygenerována instrukce pro unifikaci konstanty (`unify_constant`). V případě složeného termu je vygenerována instrukce, která tento term přesune do nějakého volného registru a instrukce pro unifikaci vygeneruje až po dokončení překladu všech termů v dané úrovně zanoření. Pořadí procházení jednotlivých termů demonstruje obrázek 6.5, u složených termů je opět uvedeno pořadí dvakrát, jednou pro přesun do registru a podruhé pro unifikaci.



Obrázek 6.5: Pořadí průchodu uzly hlavičky `p(s(f(a,X),d), X, [a,b|T])` při překladu

Výše popsaný průchod je implementován pomocí fronty. Na začátku jsou do fronty uloženy všechny parametry hlavičky. Obsah fronty je pak v cyklu odebírán, dokud není fronta prázdná. Každý odebraný prvek z fronty je předán k překladu. Pokud je na prvním místě jednoduchý term, je vygenerována instrukce pro unifikaci konstanty nebo proměnné (`unify_constant` a `unify_variable`). V případě složeného termu jsou zpracovány všechny

jeho prvky. Pokud je tento prvek taktéž složený term, je vygenerována instrukce pro unifikaci proměnné, složený term je přidán do tabulky symbolů a vložen na konec fronty.

Při překladu klauzule není k dispozici informace, kolik klauzulí bude obsahovat výsledná procedura. Proto každá přeložená klauzule začíná instrukcí `clause_start`. V případě statické procedury, je tato instrukce, po složení všech klauzulí do procedury, nahrazena konkrétní instrukcí. V případě pouze jedné klauzule je tato instrukce zcela vymazána. V případě instrukcí `try_me_else` a `retry_me_else` je v této implementaci drobná změna oproti WAM z [4]. Tyto instrukce mají jako parametr adresu, kam v programu skočit v případě, že vykonávaní aktuální klauzule selže. V této implementaci nemají tyto instrukce parametr, protože díky zvolenému formátu adresy lze určit adresu následující klauzule za běhu (inkrementací čísla klauzule a vynulování ofsetu v adrese). Z tohotu důvody byly instrukce přejmenovány na `Try_me` a `Retry_me`. Díky této úpravě lze v přeloženém kódu odebrat úvodní instrukcí `clause_start` bez nutnosti přepočítávat adresy.

Po přeložení hlavičky je tělo překládáno stejně jako dotaz. Překlad je realizován třídou `ClauseBodyCompiler`, která dědí od `BaseQueryCompiler` a zajišťuje, že třída převezme tabulku symbolů a další struktury od třídy `ClauseCompiler`. Na rozdíl od dotazu se v případě generování instrukce `allocate` musí vždy vygenerovat i `deallocate`. Také se zde uplatňuje optimalizace posledního volání. Překlad posledního podcíle těla probíhá tak, že jsou nejdřív standardním způsobem naplněny registry parametrů, poté je vygenerována instrukce `deallocate` a volání je místo `call` realizováno instrukcí `execute`.

6.2.3 Indexování klauzulí

Díky indexování (obecný princip popsán v kapitole 3.4.2) se klauzule slučují do větších celků. Tyto celky jsou následně předány interpretu a ten na ně pohlíží jako na jednu klauzuli v rámci procedury. Indexování zajišťuje třída `WamGenerator`, která postupně všechny klauzule přeloží pomocí třídy `ClauseCompiler`. Výsledky si ukládá do skupin podle názvu a arity klauzule. Vždy když narazí na klauzuli, která má na prvním místě jako parametr proměnnou, zpracuje celou skupinu a předá ji interpretu. Pokud je dokončen překlad všech klauzulí, jsou přeloženy všechny dosud nepřeložené skupiny.

Tabulka pro instrukci `switch_on_term` je implementována pomocí třídy `TermSwitcher`, která obsahuje čtyři celočíselné hodnoty představující velikost skoku v případě proměnné, konstanty, neprázdného seznamu a struktury. Její metoda `GetAddress(object term)` pro zadaný term, podle jeho typu, vrátí odpovídající hodnotu. Druhá úroveň indexování je realizována třídami `ConstantDictionary` a `StructureDictionary`. Jedná se o potomky generické kolekce `Dictionary` z .NET. Jako klíč je v případě struktur použit název struktury a její arita (reprezentována strukturou `ProcedureName`). Instance všech zmíněných tříd jsou předávány v rámci instrukce v jejím parametru typu `object`. Instrukce pro poslední úroveň indexování (`Try` apod.) fungují tak, jak byly popsány v kapitole 3.4.2, pouze s rozdílem, že adresa skoku není absolutní, ale relativní od adresy instrukce. Díky relativním skokům je možné i po překladu odebírat nebo přidávat instrukce na začátek dané klauzule bez nutnosti změny adres v instrukcích provádějící skok.

6.2.4 Seznamy

WAM podle [4] považuje symbol pro prázdný seznam za atom a pracuje s ním jako s konstantou. Aby byla zachována kompatibilita s .NET, musí být i prázdný seznam instancí třídy `PrologList`. Není zvykem, že by kolekce v .NET měly překrytou metodu `Equals()` tak, aby umožňovala přímé porovnání dvou různých instancí. Pokud bychom tedy použili

dva prázdné seznamy jako konstanty, unifikace by uspěla pouze v případě, že by se jednalo o jednu a tu samou instanci. V původním popisu WAM ([20]) jsou pro prázdné seznamy použity speciální instrukce `GetNil`, `PutNil`, `UnifyNil` a `SetNil`. Zavedením těchto instrukcí lze vzniklý problém vyřešit. Instrukce buď vytvoří do paměti novou instanci prázdného seznamu nebo zkontroluje, zda se na daném místě nachází seznam a je prázdný.

6.2.5 Operátor disjunkce

Pro implementaci operátoru disjunkce se nabízejí dvě možnosti. Buď rozložit jednu klauzuli do více klauzulí (podle transformace popsané v kapitole 2.2.4) a speciálně ošetřit chování operátoru řezu (například speciální instrukcí pro volání neovlivňující příslušný registr) nebo využít instrukcí jinak používaných pro zpětné navracení a celý kód ponechat v jedné klauzuli. Z důvodu, že druhá varianta nevytváří zbytečné zápisy na zásobníku a umožňuje jednodušší implementaci operátoru řezu, byla zvolena druhá varianta.

Ačkoli byly instrukce `try_me_else` a podobné v kapitole 6.2.3 přejmenovány a jejich sémantika lehce změněna, v případě operátoru disjunkce bude potřeba jejich původní podoba. Instrukční sada interpretu tedy obsahuje obě varianty. Instrukce opět pracují s relativní adresou. Dále je nutné zavést instrukci pro skok v rámci těla klauzule. Tato instrukce byla nazvána `Jump` a uplatní se v případě, kdy za disjunkcí následuje ještě další predikát. Například v klauzuli `p(X) :- (a(X) ; b(X)), c(X).` je nutné skočit po úspěšném provedení predikátu `a(X)` přímo na `c(X)`.

Operátor disjunkce může způsobit situaci, kdy není jednoznačné, zda se jedná o první nebo opakovány výskyt proměnné. Uvažujme dotaz `(a(X,Y) ; b(X)), f(Y).` Proměnná `Y` se v predikátu `f(Y)` vyskytuje v případě provedení první větve podruhé a v případě druhé větve je to její první výskyt. Nelze rozhodnout, zda použít instrukce `put_variable` nebo `move_register`. Obě instrukce v nějaké situaci způsobí nekorektní vyhodnocení programu. Při analýze klauzule, která probíhá před zahájením překladu, je pro každý složený term reprezentující disjunkci sestavena množina permanentních proměnných, které mají v alespoň jedné podvětvi svůj první výskyt. Pro proměnné, které jsou v této množině, ale v dané větvi nebyly vytvořeny, je vygenerována instrukce, která reprezentaci takové proměnné na zásobníku vytvoří.

Tento způsob implementace také mění optimalizaci posledního volání. Ta nebude uplatňována pouze na poslední predikát v těle, ale nově také na všechny poslední predikáty v každé větvi. Samozřejmě jen pokud je disjunkce na posledním místě v těle. Na všechny podtržené predikáty v následující klauzuli může být aplikována optimalizace posledního volání: `p(X) :- a(X,Y), b(Y) ; ((c(X,Z) ; f(X)), d(X,Z) ; e(X)).` Způsob překladu demonstruje program 6.6.

6.3 Alokace registrů

K dosažení co nejefektivnějších programů je potřeba přidělovat dočasné registry tak, aby se mezi nimi minimalizovaly přesuny a pokud už je takový přesun nutný, tak aby byl proveden co nejpozději. Podle [12] je nalezení nejlepšího přidělení registrů alespoň NP-úplný problém. Existují ale algoritmy, které vytvářejí dostatečně optimalizovanou práci s registry s nižší složitostí. V této implementaci byl použit algoritmus popsaný v [8]. Pro každou část klauzule jsou při analýze pro každou proměnnou sestaveny množiny USE, $NOUSE$ a $CONFLICT$. USE obsahuje pozice parametrů nevložených predikátů, kde se daná proměnná nachází. $NOUSE$ obsahuje pozice parametrů v posledním predikátu dané části, na kterých se vy-

```
1     allocate 2              % maximálně 2 perm. proměnné současně
2     get_variable_y 1, 1     % uloží X na zásobník do registru 1
3     try_me_else 6           % začátek první větve
4     put_variable_y 2, 2     % Y na zásobník a do registru parametru
5     call a/2                % volání a(X,Y)
6     put_value_y 1, 2        % Y ze zásobníku do registru parametru
7     deallocate              % poslední volání
8     execute b/1             % volání b(Y)
9     trust_me                % začátek druhé větvě
10    try_me_else 13          % začátek první podvětve druhé větvě
11    try_me_else 5           % začátek větve c(X,Z) ; f(X)
12    put_value_y 1, 1        % X ze zásobníku do registru parametru
13    put_variable_y 2, 2     % Z na zásobník a do registru parametru
14    call c/2                % volání c(X,Z)
15    jump 4                  % skok na konec větvě (řádek 19)
16    trust_me                % vstup do větvě f(X)
17    put_value_y 1, 1        % X ze zásobníku do registru parametru
18    call f/1                % volání f(X)
19    put_value_y 1, 1        % X ze zásobníku do registru parametru
20    put_value_y 2, 2        % Y ze zásobníku do registru parametru
21    deallocate              % poslední volání
22    execute d/2             % volání d(X,Z)
23    trust_me                % vstup do větvě e(X)
24    put_value_y 1, 1        % X ze zásobníku do registru parametru
25    deallocate              % poslední volání
26    execute e/1             % volání e(X)
```

Program 6.6: `p(X) :- a(X,Y), b(Y) ; ((c(X,Z) ; f(X)), d(X,Z) ; e(X)).` ve WAM

skytují jiné proměnné a které nejsou v USE. Platí, že $NOUSE \cap USE = \emptyset$. Pokud se daná proměnná nevyskytuje v posledním predikátu jako parametr (může být zanořená ve složeném termu), je množina $CONFLICT$ prázdná. V opačném případě obsahuje pozice všech parametrů, které nejsou daná proměnná. V případě, že se proměnná jako parametr v posledním predikátu nevyskytuje, je množina $CONFLICT$ prázdná. Uvažujme následující klauzuli: `p(X,c(Z),Y) :- var(Y), q(a(X), Y, Z).` Predikát `var/1` považujme za vložený a celá klauzule se tak skládá pouze z jedné části. Tabulka 6.3 zobrazuje výše zmíněné množiny pro tuto klauzuli.

proměnná	USE	$NOUSE$	$CONFLICT$
`X`	$\{1\}$	$\{2,3\}$	$\emptyset$
`Y`	$\{2,3\}$	$\emptyset$	$\{1,3\}$
`Z`	$\{3\}$	$\{1\}$	$\{1,2\}$

Tabulka 6.3: Příklad množin pro `p(X,c(Z),Y) :- var(Y), q(a(X), Y, Z).`

Jednotlivé parametry jsou ponechány v registrech, do kterých byly umístěny při volání. K přesunu proměnné mezi dočasnými registry dojde až v okamžiku, kdy nastane konflikt. Tedy v situaci, kdy je nutné použít registr, který je už obsazený. Jako nový registr je určen první volný, který se nenachází v $NOUSE \cup CONFLICT$. U vložených predikátů není potřeba připravovat parametry do registrů na konkrétní místo (čísla registrů, se kterými se pracuje, jsou předávány jako operandy instrukce) a tedy nezpůsobují konflikty. První

konflikt tak nastane v okamžiku, kdy je potřeba umístit do dočasného registru 1 strukturu `a/1`. Je tedy vygenerována instrukce `move_register`, která přesune proměnnou `X` z prvního registru do prvního volného, který není v $NOUSE(X) \cup CONFLICT(X)$. Dále je potřeba přesunout `Y` z třetího registru do druhého. V tomto případě nenastává konflikt, protože $2 \notin NOUSE(Y) \cup CONFLICT(Y)$. V případě proměnné `Z` sice konflikt nastává, ale vzhledem k tomu, že `Y` byla mezitím přesunuta, a nachází se jak v druhém tak třetím registru, není potřeba konflikt řešit. Obsah proměnné `Y` nebude jejím přepsáním ztracen.

Tento přístup ovlivňuje implementaci tabulky symbolů. Pro každý symbol (strukturu, atom, proměnnou atd.) se musí evidovat v jakých dočasných registrech se nachází (může jich být i více, pokud se proměnná nachází ve více parametrech hlavičky) a zda se proměnná nachází v permanentním registru a případně v jakém. U každého záznamu v tabulce je také uvedeno, do kdy je zápis platný. Například v klauzuli `p(X,Y) :- var(X), a(Y).` je `X` platné pouze do prvního parametru prvního predikátu těla. Poté lze registr obsahující tuto proměnnou přepsat bez toho, aby to byl konflikt. V případě, že se proměnná vyskytuje jako parametr v nějakém predikátu, je jeho platnost až po instrukcí `call`. Díky tomu, že jsou konflikty řešeny až v okamžiku kdy nastanou, lze v kódu hlavičky vynechat instrukce `get_variable`, které nepracují s permanentními registry. Program 6.7 demonstruje optimalizovaný a neoptimalizovaný WAM kód. [8] [12]

```
1     % neoptimalizovaný kód
2     get_variable 1, 4
3     get_variable 2, 5
4     get_value 3, 4
5     put_structure 6, +/2
6     set_constant 1
7     set_value 4
8     execute_inline_xx 5, 6, is
9     move_register 5, 1
10    move_register 4, 2
11    execute d/2
12    % optimalizovaný kód
13    get_value 1, 3
14    put_structure 4, +/2
15    set_constant 1
16    set_value 1
17    execute_inline_xx 2, 4, is
18    move_register 2, 1
19    move_register 3, 2
20    execute d/2
```

Program 6.7: Optimalizace kódu pro `p(X,Y,X) :- Y is 1+X, d(Y,X).`

6.4 Vyhodnocení WAM

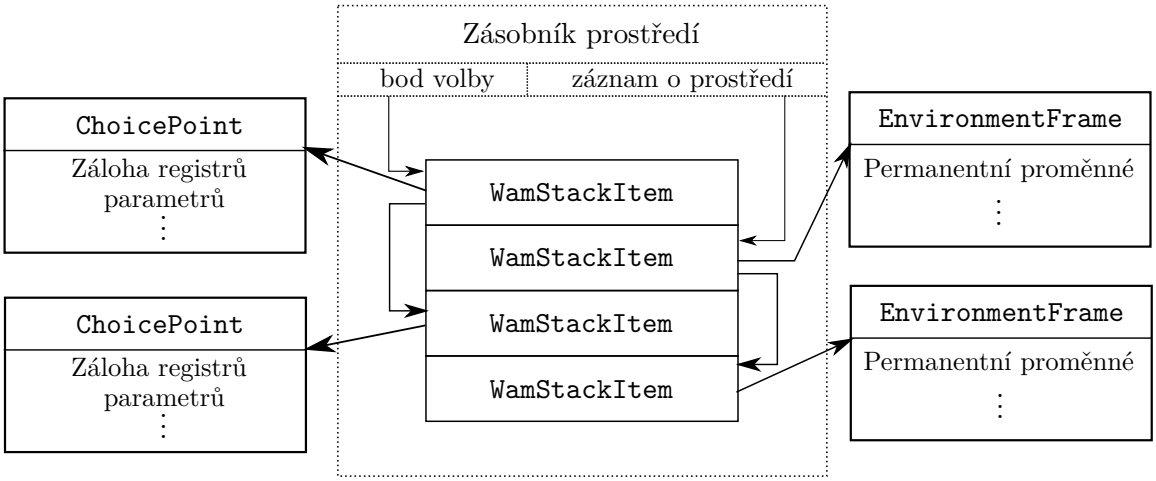
Instrukce v programu jsou určeny výčtovým typem. Po jejím načtení z adresy je pomocí konstrukce `switch` rozhodnuto, kterou metodu interpretu vykonat. Adresa je struktura skládající se ze čtyř celých čísel, kde první je index procedury v tabulce procedur, druhá je generace databáze, třetí index klauzule v rámci procedury a čtvrtá pozice instrukce v rámci klauzule. Načítání instrukcí je jedna z nejčastějších činností interpretu, proto je návrh pod-

řízen co nejrychlejšímu načtení instrukce. Tabulka procedur je rozdělena na kolekci objektů implementujícím `IWamProcedure` a vyhledávací tabulku přiřazující jednotlivým indexům jméno a aritu procedury. Díky tomu je potřeba vyhledávat proceduru v tabulce podle jména pouze při provádění instrukcí `call` a `execute`. Dále se pracuje pouze s nalezeným indexem, který je součástí adres instrukcí. Taktéž je optimalizován přístup k instrukcím ve statické proceduře. Není zde nutné řešit generace databáze a kód procedury se nemění. Všechny instrukce jsou proto uloženy v jedné kolekci. Pozice jednotlivých klauzulí v rámci této kolekce je uložena v dalším poli. Pro získání instrukce ze statické procedury stačí načíst podle indexu z tabulky procedur konkrétní proceduru, poté zjistit na které pozici v rámci procedury se nachází požadovaná klauzule (podle indexu klauzule v adrese) a k této pozici přičíst ofset z adresy. Na získaném indexu se pak nachází hledaná instrukce.

Dočasné registry jsou implementovány pomocí pole. V případě, že se při interpretaci program snaží přistoupit k registru na indexu mimo rozsah pole, je alokováno nové větší pole a obsah registrů překopírován. K tomuto zvětšování nedochází příliš často, protože u většiny klauzulí je využívaný počet pracovních registrů podobný. Jiná situace nastává u permanentních registrů alokovaných na zásobníku prostředí. Zde je v době překladu klauzule z objektové reprezentace do WAM známý počet permanentních proměnných, které je potřeba současně ukládat na zásobníku (ze stromu popsaném v 6.2). Tyto registry jsou taktéž implementovány pomocí pole, ale jeho velikost je po celou dobu vykonávání klauzule neměnná a je určena operandem instrukce `allocate`.

Pro realizaci zásobníku prostředí (popsaný v kapitole 3.1.3) není možné použít zásobník z knihovny .NET, neboť zásobník prostředí má nestandardní požadavky na práci s jeho prvky (vyžaduje přístup i na jiný prvek než vrchol). Zásobník prostředí je implementován kolekcí struktur `WamStackItem` a obsahuje indexy vrchního záznamu o prostředí a vrchního bodu volby. Každá struktura `WamStackItem` na zásobníku může obsahovat buď instanci třídy `EnvironmentFrame` (reprezentující záznam o prostředí) nebo `ChoicePoint` (repreznetující bod volby). Struktura `WamStackItem` také obsahuje index předchozího záznamu na zásobníku, ve kterém je stejný typ záznamu. V případě odebrání záznamu o prostředí ze zásobníku, je ukazatel na vrchní záznam nastaven na index předchozího záznamu o prostředí. Pokud je tento index vyšší než index vrchního bodu volby, je ze zásobníku odstraněna reference na objekt typu `EnvironmentFrame` (v opačném případě může být reference potřeba při zpětném navracení). Odstranění bodu volby funguje na podobném principu, pouze při odstranění vrchního bodu volby jsou odstraněny i reference na objekty `EnvironmentFrame` s vyšším indexem než je nový vrchol s bodem volby a aktuální index vrcholu záznamu o prostředí.

Vytváření a odstranění záznamů na zásobníku probíhá při vyhodnocování programu velmi často (u každé klauzule, která není fakt nebo řetězové pravidlo). Neustále vytváření a zahazování objektů přidává práci CLR, který musí zajistit jejich alokaci v paměti a potom je pomocí Garbage Collectoru zase odstranit. Z tohoto důvodu jsou instance tříd `EnvironmentFrame` a `ChoicPoint` recyklovány. O vytváření a rušení objektu se tak stará statická třída podle návrhového vzoru Pool. Tato třída obsahuje metody pro získání a vrácení požadovaného objektu. Počet uchovávaných vrácených objektů je omezen. V případě bodu volby je omezení 32 instancí, u záznamů prostředí 64. Tyto hodnoty dosahovaly nejlepších výsledků v testovaných programech). V případě, že je vrácen další objekt, ale paměť na vrácené objekty je již plná, je tento objekt zahozen. Obě třídy (`ChoicPoint` a `EnvironmentFrame`) obsahují pole, jehož požadovaná velikost může být v různých využitích objektu různá. Takové pole se používá pro uložení zálohy registrů parametrů nebo pro uchování permanentních proměnných (jak ukazuje obrázek 6.8). Při požadavku na nový ob-



Obrázek 6.8: Způsob implementace zásobníku prostředí

jekt tak Pool zkontroluje, jestli velikost pole vraceného objektu je dostatečná a pokud není, alokuje nové větší pole. Protože téměř každá klauzule vyžaduje velikost pole alespoň 5, je v případě požadavku na menší pole alokováno pole o velikosti 5, aby se předešlo nutnosti v budoucnu pole znovu alokovat.

6.4.1 Dynamické a meta predikáty

Způsob reprezentace dynamických struktur byl popsán v kapitole 5.3. V této podkapitole bude vysvětlen způsob implementace překladu klauzulí za běhu, způsob jakým jsou realizovány predikáty `retract/1` apod. Také bude popsáno, jak je realizováno odstraňování starých a nepotřebných verzí databáze.

Pro odebírání dynamických klauzulí z databáze je potřeba umět provést unifikaci buďto s celou klauzulí (v případě `retract/1`) nebo její hlavičkou (v případě `retractall/1`). Přeložený kód pro vykonání procedury se liší od kódu pro unifikaci struktury, který by v této situaci byl potřeba. Rekonstrukce klauzule z přeloženého kódu, zvláště po různých optimalizacích, je ve WAM netriviální úkol (na rozdíl od stroje ZIP). Z tohoto důvodu je k přeložené dynamické klauzuli uložen i kód pro unifikace klauzule jako struktury. Predikát `retract/1` je implementován tak, že podle názvu a arity struktury v parametru dohledá odpovídající dynamickou proceduru a následně interpretu předává instrukce pro unifikaci klauzule. Pokud žádná z instrukcí neselže, je pomocí instrukce `ExecuteCli` zavolán kód, který danou proceduru z databáze odstraní. V případě selhání je zkoušena unifikace s další klauzulí v proceduře. Pro informaci o tom, jakou klauzuli z procedury má zkoušet unifikovat, využívá číslo klauzule v adrese. Na podobném principu pracuje i `retractall/1`, pouze se unifikuje hlavička klauzule a v případě úspěšného odstranění je vždy vyvoláno zpětné navracení instrukcí `fail`.

Staré generace databáze, které již nemohou být volány, je potřeba odstranit. Každý objekt reprezentující generaci (třída `Generation` znázorněna na obrázku 5.1) obsahuje čítač, který ukazuje, kolik klauzulí s danou generací pracuje. Při volání procedury instrukcí `call` nebo `execute` je čítač volané generace zvýšen. Při ukončení vykonávání procedury instrukcemi `deallocate` nebo `proceed` je čítač snížen. Je nutné ošetřit, aby k odstranění generace nedošlo v okamžiku, kdy sice žádná klauzule z dané generace není vykonávána, ale stále může být vykonána po vyvolání zpětného navracení. Z tohoto důvodu se čítač snižuje

jen v případě, že návrat z procedury proběhl z poslední klauzule (což lze detekovat porovnáním čísla klauzule v adrese a počtu klauzulí v dané generaci). Pokud tento čítač dosáhne hodnoty 0, je z paměti odstraněn. Problém nastává v okamžiku, kdy je výpočet ukončen dříve, než vrátí všechna možná řešení. V tento okamžik se všechny čítače dynamických predikátů nestihly vynulovat, protože ještě existují další body volby. Enumerátor v .NET obsahuje metodu, která je volána při jeho odstranění z paměti (například po opuštění cyklu `foreach`). Při volání této metody je zkontrolováno, jestli je zásobník prostředí prázdný. Pokud ne, jsou postupně odmazány všechny záznamy, čímž se vynulují čítače u dynamických procedur a zastaralé generace jsou odstraněny.

Podobně jako dynamické predikáty musí i meta predikáty jako například `call/1` nebo `not/1` provádět za běhu překlad dotazu, který mají ve svých parametrech. V případě, že je takto volán pouze jeden predikát, není nutný žádný překlad. Pouze jsou parametry umístěny do příslušných registrů (ze struktury do registrů parametrů) a poté zavolána instrukce `execute`. V případě více predikátů je ale nutné nejprve provést překlad do instrukcí WAM. Například dotaz `call((a(X,Y),b(Y))), c(Y).` je interně přeložen jako: `tmp(X,Y), c(Y).` `tmp(X,Y) :- a(X,Y),b(Y).` Tento pomocný predikát je vytvořen jen dočasně. Pro jeho volání je definována speciální instrukce `call_inner`, která se chová obdobně jako `call`, ale jejím parametrem není název procedury, jejíž adresu si vyhledá v tabulce procedur, ale přímo adresa kam má v programu skočit. Tím je zajištěno volání bez nutnosti úpravy tabulky procedur.

6.5 Realizace propojení s platformou .NET

Propojení směrem z .NET do Prologu je poměrně jednoduché. Každý .NET objekt je brán jako konstanta. Interpret ale nabízí i další rozšiřování svého chování o další vestavěné predikáty případně o aritmetické funkce. Seznamy v Prologu a kolekce v .NET se liší a není možné zajistit jejich používání tak, jak je v Prologu obvyklé. Proto je v obou případech implementována možnost převodů mezi objekty implementující rozhraní `IEnumerable` a třídou `PrologList` a opačně. V případě .NET byl zvolen operátor technologie LINQ pojmenovaný `ToPrologList()`. Pro opačný směr převodu lze díky tomu, že `PrologList` implementuje rozhraní `IEnumarable`, použít standardní operátor `ToList()`. Běžný zápis v Prologu pro zbytek seznamu (například: `[X|T]`) lze u třídy `PrologList` simulovat pomocí metody `Tail()`, jejímž parametrem je term (například proměnná nebo seznam), který je vložen na konec seznamu. Pro přehlednější zápis programu ve formě objektové reprezentace je vytvořena statická třída `Prolog`, která obsahuje metody pro tvorbu různých Prologovských typů včetně struktur, pro které obsahuje abstraktní tovární metodu. Výše zmíněný zápis seznamu lze zapsat v jazyce C# takto:

```
Prolog.List(Prolog.Variable("X")).Tail(Prolog.Variable("T"));
```

Pro práci s objekty .NET v jazyce Prolog byly implementovány predikáty popsané v kapitole 4.3.1. Práce s objekty je realizována pomocí reflexe. Zde je potřeba zajistit, aby interpret měl přístup ke všem datovým typům vytvořeným nebo referencovaným v sestavení, kde je používán a ne jen k typům, které znal interpret v době překladu. Aplikace v .NET se skládají z více tzv. aplikačních domén. Do těchto domén jsou načítány sestavení (anglicky *assamblies*). Doména je taky nejmenší jednotkou, která může být za běhu z paměti odstraněna. Aplikace a interpret běží ve vlastních doménách a je třeba zajistit, aby interpret mohl používat skrz reflexi všechny typy aplikační domény. Platforma .NET umož-

ňuje z ostatních domén přístup do aplikační domény pomocí třídy `AppDomain`. Pokud tedy není typ nalezen přímo mezi základními typy z .NET, projde interpret všechny sestavení na aplikační doméně a postupně zkouší požadovaný typ dohledat. Ačkoli predikáty uvedené v 4.3.1 umožňují veškerou práci s objekty .NET, byly pro jednodušší programování a větší výkon (ne vždy je nutné použít reflexi) implementovány další predikáty. Jejich kompletní seznam lze nalézt v příloze B.

6.5.1 Rozšiřitelnost

Do interpretu lze přidávat nové predikáty implementované v jazyce C#, s nimiž lze dosáhnout podobné efektivity jako mají vestavěné predikáty. Do interpretu je lze vložit formou knihovny, což je .NETovské sestavení obsahující třídy označené atributem `PrologProcedure` (jako parametr přijímá název a aritu procedury) nebo atributem `PrologLibrary`. Druhé zmíněné musí dědit od třídy `WamLibrary`. Tato třída obsahuje enumerátor vracející jednotlivé procedury v knihovně. Samotná třída reprezenutjící predikát musí dědit od třídy `DeterministicUserPredicate` v případě predikátu, který nepoužívá zpětné navracení nebo v případě využití zpětného navracení od třídy `NonDeterministicUserPredicate`. Z této třídy nemá uživatel přímo přístup k vnitřním strukturám interpretu, ale jsou touto třídou poskytovány pomocí jejich metod. Tím se odstíní závislost implementace procedury na konkrétních implementačních detailech interpretu. Příklad použití těchto tříd lze nalézt v příloze D. Načtení knihovny do interpretu lze provést pomocí speciálních predikátů. `load_library_file/1`, který přijímá jako parametr cestu k souboru s knihovnou. V případě, že je knihovna již přímo součástí projektu pro .NET, lze použít vestavěný predikát `load_library/1`, který jako parametr přijímá název třídy s knihovnou (včetně jmenných prostorů). Oba predikáty lze v programu použít i jako direktivy (stejným způsobem jako `:- dynamic`). V tomto případě bude knihovna načtena hned při načítání programu.

V interpretu lze také ovlivnit vyhodnocování aritmetických výrazů. Implementace vychází z návrhového vzoru Bridge. Vyhodnocování provádí objekt implementující rozhraní `IExpressionEvaluator`, které obsahuje jedinou metodu `Evaluate()`, která jako parametr dostane strukturu, kterou je potřeba vyhodnotit. Instance používaná pro vyhodnocení je veřejně zpřístupněna pomocí statické vlastnosti (anglicky *property*) `Evaluator` třídy `CLIProlog`. Pokud chce uživatel pouze přidat nebo pozměnit chování některé operace, může podědit třídu `ExpressionEvaluator` a překrýt virtuální metodu `DoOperation`. Instanci nově vytvořené třídy pak stačí předat třídě `CLIProlog`.

6.5.2 Integrace do MSBuild a Visual Studia

Implementovaný překladač umožňuje uložit výsledný WAM kód jako třídu v jazyce C#. Výsledný kód buď může mít formu knihovny nebo programu (potomek třídy `WamProgram`). Druhá zmiňovaná možnost vytvoří takovou třídu, která je hned po přeložení připravena k použití s metodou `Query()`. Překlad je postaven na technologii zvané CodeDom[2], která umožňuje pomocí objektů sestavit strom programu a následně vygenerovat kód v jazyce C# (případně jiných jazycích .NET). Tato technologie původně vznikla pro vytváření zdrojových kódu pro grafická rozhraní generována grafickým editorem (například *WinForm*). Každá třída, kterou je možné exportovat jako zdrojový kód, implementuje rozhraní `ICliCompilable`, které poskytuje metody pro korektní vygenerování stromu pro daný objekt.

[2]https://msdn.microsoft.com/cs-cz/library/y2k85ax6.aspx

Vzniklý překladač do C# je integrován jako úloha do systému MSBuild. Tato úloha se zařazuje před samotný překlad programu, do dočasného adresáře vytvoří přeložený zdrojový soubor a přidá ho do souborů, které je nutné v dalším kroku přeložit ze C# do CIL. Aby nebylo nutné začleňovat úlohu do MSBuild ručně, byl vytvořen balíček systému NuGet[3] a podpůrný balík pro Visual Studio ve formátu VSIX[4]. Ten obsahuje šablony pro programy a knihovny v Prologu. V budoucnu může být tento balík rozšířen například o zvýraznění syntaxe apod.

[3] https://www.nuget.org
[4] https://msdn.microsoft.com/cs-cz/library/dd997148.aspx

Kapitola 7

Testování

Program byl testován na sadě přibližně 280 testů. Část z nich využívá jednotkové testování. Jednotkové testování bylo využito pro ověření korektních výsledků analýzy klauzulí, správného chování seznamů, skládání predikátů do klauzulí pomocí operátoru `&` a `|` nebo pro testování zásobníku prostředí. Další část testů ověřuje implementaci překladače z jazyka Prolog do objektové reprezentace. Překladači je vždy předán krátký program a je zkontrolován výstup, zda odpovídá očekávané reprezentaci. Největší část testů je věnována provádění krátkých programů a následně je zkontrolováno vrácené řešení. Část testů interpretu má jako vstup objektovou reprezentaci. Jedná se hlavně o testy jednodušších programů, které vznikly v době, kdy ještě nebyl dokončen syntaktický analyzátor. Pro vytvoření a spouštění testů byl použit Visual Studio Unit Testing Framework. Všechny testy úspěšně prošly.

7.1 Výkonnostní testy

Pro vyhodnocení výkonu implementovaného interpretu byla sestavena testovací sada tří programů. Prvním programem je řešení hlavolamu SUDOKU algoritmem bez pokročilejší inteligence. Tento algoritmus postupně generuje všechna řešení, kde se v řádku ani ve sloupci neopakují číslice. V dalším kroku pak ověří, zda se v 9 čtvercích o velikosti 3x3 číslice také neopakují. Jestliže se opakují, selže a je testována další kombinace. Tento algoritmus byl zvolen, protože pro vhodně vybrané zadání probíhá výpočet dostatečně dlouho, aby bylo možné zanedbat vliv dalších programů běžících na testovacím počítači. Zároveň při řešení SUDOKU tímto algoritmem dochází k velkému počtu alokací na zásobníku a za delší dobu běhu se tak lépe projeví efektivita těchto operací v interpretu.

Druhým testovacím programem je problém n dam. Tento problém spočívá v rozmístění n dam na šachovnici $n \times n$, tak aby se podle pravidel šachu neohrožovaly. Kód testu byl převzat z knihy [18]. Měřena je doba, za kterou dokáže interpret najít všechna možná řešení pro dané n. Testování probíhalo pro n v rozsahu 12 až 15. Pro $n = 12$ má tento problém celkem 14 200 řešení, pro $n = 15$ existuje 2 279 184 řešení. Třetím testem byl opět problém n dam, ale predikáty pro ověření, zda jedna dáma ohrožuje jinou, byly definovány jako dynamické. Tímto testem byl měřen rozdíl mezí přístupem ke statickým a dynamickým predikátům.

Každý test byl spuštěn celkem třináctkrát. Nejvyšší, nejnižší a první hodnota byla vyškrtnuta a zbytek hodnot zprůměrován. Testy běžely na čisté instalaci systému Windows s nainstalovaným rámcem .NET verze 4.5. Počítač nebyl připojen do sítě a mimo interpretu běžely pouze nezbytné systémové procesy. Mimo implementace popisované v této práci byl

dále testován SWI-Prolog (implementace, která není vytvořena v .NET), C#Prolog a P#. Prolog.NET nebyl do testu zahrnut, protože testovací programy by vyžadovaly nemalé úpravy, aby bylo možné je spustit. Poslední test nebyl spuštěn ani na interpretu P#, který nedokázal zpracovat dynamicky definovanou klauzuli obsahující tělo. Zdrojové kódy testu lze nalézt na přiloženém DVD.

Implementace Prologu	**Čas výpočtu**
SWI-Prolog	31 minut
C#Prolog	4 hodiny 2 minuty
P#	> 8 hodin
CLIProlog	**3 hodiny 53 minut**

Tabulka 7.1: Výsledky testování na hlavolamu SUDOKU

Implementace Prologu	**n=12**	**n=13**	**n=14**	**n=15**
SWI-Prolog	6 vteřin	26 vteřin	3 minuty 56 vteřin	26 minut 57 vteřin
C#Prolog	2 minuty 25 vteřin	13 minut 54 vteřin	1 hodina 19 minut	> 6 hodin
P#	46 minut 23 vteřin	4 hodiny 37 minut	> 6 hodin	-
CLIProlog	**1 minuta 14 vteřin**	**7 minut 28 vteřin**	**48 minut 8 vteřin**	**5 hodin 28 minut**

Tabulka 7.2: Výsledky testování na problému n dam

Implementace Prologu	**n=12**	**n=13**	**n=14**
SWI-Prolog	6 vteřin	33 vteřin	4 minuty 7 vteřin
C#Prolog	2 minuty 40 vteřin	16 minut 10 vteřin	1 hodina 41 minut
CLIProlog	**1 minuta 32 vteřin**	**9 minut 17 vteřin**	**1 hodina 3 minuty**

Tabulka 7.3: Výsledky testování na problému n dam s dynamickými predikáty

Implementovaný interpret je podle výkonnostních testů srovnatelný s ostatními implementacemi. V případě problému n dam byl výrazně rychlejší než ostatní implementace pro .NET. Z testu s dynamickými predikáty je patrné, že ostatní implementace pracují lépe s dynamickými predikáty. Zpomalení zhruba o 30% u CLIPrologu na testu s dynamickými predikáty je zřejmě způsobeno komplikovanou adresací instrukcí v dynamických procedurách. Všechny .NETovské implementace výrazně zaostávají za rychlostí SWI-Prologu. To je pravděpodobně způsobeno tím, že SWI Prolog nemá mezivrstvu v podobě CLR a při jeho vyhodnocování neprobíhá JIT překlad zdrojových kódu interpretu. Vliv může mít také rozdílná architektura interpretu, různé optimalizace (např. JIT indexování) a podobně. Výkon CLIPrologu jde velmi pravděpodobně dále zlepšovat úpravou návrhu a optimalizací, aby využil co nejvíce z CLR. Provedení takových úprav by ale bylo časově náročné. Dosáhnout výkonu SWI-Prologu na platformě .NET je pravděpodobně nemožné.

Kapitola 8

Závěr

V rámci práce byl vytvořen souhrn existujících způsobů, jak vyhodnocovat Prologovské programy. Detailně byl prostudován koncept Warrenova abstraktního stroje (WAM). Dále byla zmapována situace ohledně implementací Prologu pro platformu .NET. Na základě těchto zjištění byl sestaven návrh na nový překladač a interpret, který zajistí těsnější spojení s prostředím .NET. Dále byl navrhnut způsob implementace operátoru disjunkce a dynamických predikátů. Navržený interpret a překladač byl následně implementován. Systém byl úspěšně začleněn do platformy .NET a jejích obvyklých vývojových nástrojů.

8.1 Přínos práce

Práce shrnuje princip vyhodnocování jazyka Prolog a popisuje výsledný interpret, který umožňuje kombinování platformy .NET a jazyka Prolog výrazně intuitivnějším způsobem než všechny dosud vytvořené implementace. Díky tomu je možné kombinovat programy v .NET s Prologem v místech, kde může Prolog ušetřit práci jinak potřebnou při implementaci jinými jazyky. Také je možné začlenit Prolog do .NETovských aplikací jako skriptovací jazyk. Objektové rozhraní překladače umožňuje i vytváření vizuálních editorů, které budou generovat kód v Prologu. Výkonem je vytvořený interpret srovnatelný s nejrychlejšími interprety pro platformu .NET. V některých úlohách je dokonce mezi .NET implementacemi nejrychlejší.

8.2 Možná rozšíření

Projekt je možné dále rozvíjet. V oblasti překladače je ho možné doplnit o možnost definice vlastních operátorů. Z pohledu výkonu je možné dále optimalizovat překladač i interpret, aby překlad i samotné vyhodnocování trvalo kratší dobu nebo zavést indexování pro dynamické klauzule, což by vyžadovalo změnu jejich reprezentace. Případně je možné zavést JIT indexování podobně jako ho využívá SWI-Prolog. Pro omezení paměťových nároků je možné implementaci doplnit ořezáváním zásobníku prostředí (viz kapitola 3.4.3). Z pohledu vývojových nástrojů je možné rozšířit podporu pro Visual Studio například o ladicí nástroje umožňující program krokovat, případně analyzovat body volby, které program vytvořil. Ideálním řešením by bylo generování speciálních ladicích instrukcí mezi WAM kód, které by využívaly API pro ladění nástroje Visual Studio. Užitečným rozšířením by také bylo zvýrazňování syntaxe, párování závorek nebo podtrhávání chyb přímo v editoru zdrojového kódu. Opět se jako nejpraktičtější jeví implementace pomocí SDK pro Visual Studio. Sa-

motný Prolog by pak mohl být rozšířen o slovníky[1] podobně jako od verze 7 SWI-Prolog. Za rozšíření by stála také standardní knihovna, která je v této první verzi, ve srovnání s jinými distribucemi Prologu, poměrně chudá. Praktická by byla také možnost generování tříd skrz *MSBuild* i pro jiné jazyky než pouze C#, jako například Visual Basic.NET nebo IronPython. Současná implementace nedovoluje práci nad jedním programem více vláknům najednou. Rozšířením by tak mohla být i úprava jednotlivých metod interpretu tak, aby byly vláknově bezpečné. V tom případě by bylo vhodné, vytvořit nové predikáty pro zámky a další práci s vlákny.

Při práci s objekty z .NET nastává, podobně jako práce se vstupem nebo výstupem, problém se zpětným navracením. Jakmile nějaká metoda změní vnitřní stav objektu, je to operace nevratná a po vyvolání zpětného navracení tato změna přetrvá. Podobně jako u zmíněného vstupu a výstupu je takové řešení akceptovatelné, pokud s ním programátor dopředu počítá. Existují ale situace, kdy by se hodilo mít objekty, které se po zpětném navracení vrátí do stavu, ve kterém byly. Zajímavým rozšířením by tedy bylo umožnit objektům chovat se stejně jako typy jazyka Prolog. Aby bylo této vlastnosti dosaženo, bylo by nutné vytvářet na každém bodě volby, kam se může vrátit vykonávání programu po neuspění nějakého predikátu, kopii objektu pro obnovení původního stavu. Platforma .NET nemá jednotné rozhraní pro kopírovaní objektů a toto chování je navíc prostorově náročné a ne vždy nutné. Pokud by programátor vyžadoval od své třídy takové chování, musel by objekt implementovat rozhraní nazvané například `IBacktrackable`. Toto rozhraní by poskytovalo metodu pro vytvoření své vlastní kopie (je na programátorovi, jak hluboká kopie by kopie byla). Interpret by pak s takovým objektem zacházel tak, že se při každém bodě volby vytvoří kopii objektů implementující toto rozhraní. Implementace této funkce by měla poměrně velký negativní vliv na výkon interpretu, protože by bylo při vytváření každého bodu volby nutné kontrolovat, zda se někde v aktuálně používaných datech nevyskytuje objekt implementující toto rozhraní (například někde zanořený ve složeném termu). Aby se tomuto zpomalení předešlo, bylo by nutné návrh velmi dobře promyslet a upravit.

[1] http://www.swi-prolog.org/pldoc/man?section=dicts

Literatura

[1] C#Prolog – A Prolog interpreter written in managed C#. https://sourceforge.net/projects/cs-prolog/, navštíveno: 2017-01-05.

[2] Prolog.NET - CodePlex. https://prolog.codeplex.com, navštíveno: 2017-01-04.

[3] Albahari, J.; Albahari, B.: *C# 6.0 in a Nutshell: The Definitive Reference*. O'Reilly Media, 2015, ISBN 9781491927069.

[4] Aït-Kaci, H.: *Warren's Abstract Machine: A Tutorial Reconstruction (Logic Programming)*. The MIT Press, 1991, ISBN 0262510588.

[5] Bishop, J. M.: *C# : návrhové vzory*. Brno: Zoner Press, 2010, ISBN 978-80-7413-076-2.

[6] Bowen, D.; Byrd, L.; Clocksin, W.; aj.: *A Portable Prolog Compiler*. DAI research paper, Department of Artificial Intelligence, University of Edinburgh, 1983. URL https://www.researchgate.net/publication/273888197_A_portable_Prolog_compiler

[7] Cook, J.: P# Manual (version 1.1.3). 2003. URL http://www.dcs.ed.ac.uk/home/jjc/psharp/psharp-1.1.3/manual.pdf

[8] Debray, S. K.: *Register Allocation in Prolog Machine*. Symposium on Logic Programming, IEEE, 1986. URL https://pdfs.semanticscholar.org/be79/bf12014c53607e7933717b710ac8a7bd9261.pdf

[9] Krall, A.: Implementation Techniques for Prolog. In *Proceedings of the Tenth Logic Programming Workshop, WLP 94*, 1994. URL https://pdfs.semanticscholar.org/fdbf/aa46bf6ab2148595f638fe9afe97033583ee.pdf

[10] Krall, A.: The Vienna abstract machine. *The Journal of Logic Programming*, ročník 29, č. 1, 1996: s. 85 – 106, ISSN 0743-1066. URL http://www.sciencedirect.com/science/article/pii/S0743106696000647

[11] Li, X.: Structure Sharing and Structure Copying Revisited. In *Compulog Net Meeting on Parallelism and Implementation Technology*, 1996. URL http://www.cs.nmsu.edu/lldap/jicslp/xi.html

[12] Matyska, L.; Jergová, A.; Toman, D.: Register Allocation in WAM. In *Logic Programming, Proceedings of the Eigth International Conference, Paris, France, June 24-28, 1991*, 1991, s. 142–156.

[13] Paolo, P.; Marco, R.: *Microsoft LINQ Kompletní průvodce programátora*. Computer Press, 2009, ISBN 978-80-251-2735-3.

[14] Parr, T.: *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, druhé vydání, 2013, ISBN 1934356999, 9781934356999.

[15] Parr, T.; Harwell, S.; Fisher, K.: Adaptive LL(*) Parsing: The Power of Dynamic Analysis. *SIGPLAN Not.*, ročník 49, č. 10, Říjen 2014: s. 579–598, ISSN 0362-1340.
URL `http://doi.acm.org/10.1145/2714064.2660202`

[16] Pecinovský, R.: *Návrhové vzory: [33 vzorových postupů pro objektové programování]*. Computer Press, 2007, ISBN 978-80-251-1582-4.

[17] Robinson, S.: *C# : programujeme profesionálně*. Brno: Computer Press, 2003, ISBN 80-251-0085-5.

[18] Sterling, L.; Shapiro, E.: *The Art of Prolog, Second Edition: Advanced Programming Techniques (Logic Programming)*. The MIT Press, 1994, ISBN 0262193388.

[19] Tucker, A. B.; Noonan, R.: *Programming Languages: Principles and Paradigms*. McGraw-Hill Science/Engineering/Math, 2001, ISBN 0072381116.

[20] Warren, D. H. D.: An Abstract Prolog Instruction Set. Technická Zpráva 309, AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, Oct 1983.
URL `http://www.academia.edu/9970563/An_abstract_Prolog_instruction_set`

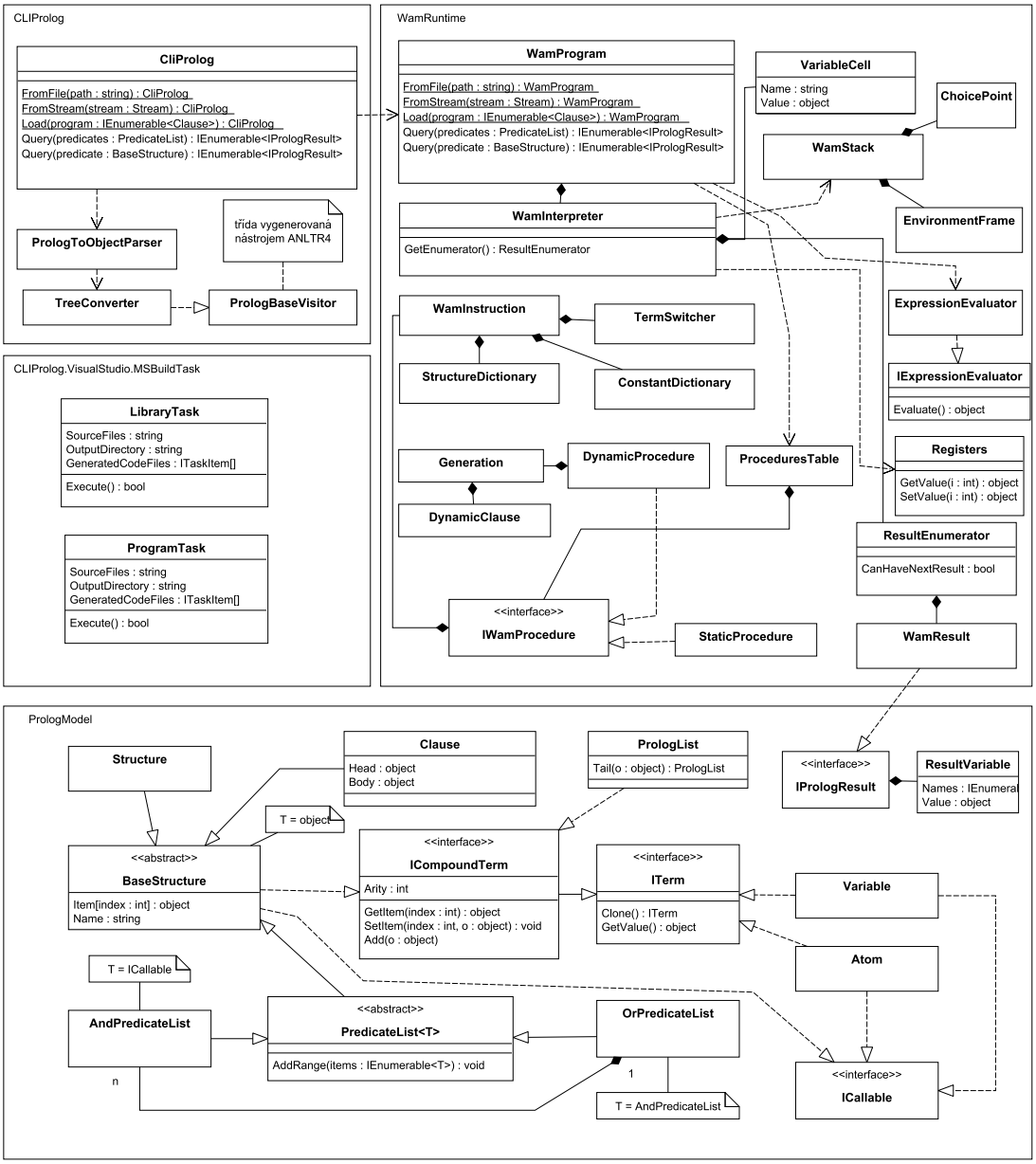
[21] Wielemaker, J.; Anjewierden, A.: An Architecture for Making Object-Oriented Systems Available from Prolog. In *WLPE*, 2002, s. 97–110.
URL `http://www.swi-prolog.org/download/publications/wlpe-02.pdf`

[22] Wielemaker, J.; De Koninck, L.; Fruehwirth, T.; aj.: *SWI Prolog Reference Manual 7.1*. BoD–Books on Demand, 2014.
URL `http://www.swi-prolog.org/download/devel/doc/SWI-Prolog-7.1.16.pdf`

Přílohy

Příloha A

UML diagram intepretu WAM



Příloha B

Přehled instrukcí interpretu

V následujících tabulkách je využito toto značení:

`X`$_i$ - index dočasného registru
`Y`$_i$ - index registru na zásobníku
CLI - libovolný CLI objekt reprezentující konstantu
[1] nevyskytuje se ve WAM popsaném v [4], ale vyskytuje se v [20]
[2] instrukce přidána v této práci

PUT instrukce

Název	**Popis**	**Parametry**		
		Int 1	**Int 2**	**Object**
`put_constant`	Vloží CLI konstantu do registru.	`X`$_i$	-	CLI
`put_list`	Vloží reprezentaci seznamu do registru.	`X`$_i$	-	-
`put_nil`	Vloží do registru prázdný seznam.	`X`$_i$	-	-
`put_structure`	Vloží reprezentaci struktury do registru.	`X`$_i$	arita	název
`put_value_y`	Přesune hodnota ze zásobníku do dočasného registru.	`Y`$_i$	`X`$_n$	-
`put_variable`	Vytvoří novou reprezentaci proměnné a umístí ji do registru.	`X`$_i$	-	-
`put_variable_r`[2]	Vytvoří novou reprezentaci proměnné a umístí ji do registru. Hodnota proměnné je po dokončení výpočtu vrácena uživateli.	`X`$_i$	-	název
`put_variable_y`	Vytvoří novou reprezentaci proměnné a umístí ji do registru na zásobníku.	`Y`$_i$	-	-
`put_variable_yr`[2]	Vytvoří novou reprezentaci proměnné a umístí ji do registru na zásobníku. Hodnota proměnné je po dokončení výpočtu vrácena uživateli.	`Y`$_i$	-	název

Řídicí instrukce

Název	Popis	Parametry		
		Int 1	Int 2	Object
`allocate`	Vytvoří záznam o prostředí na zásobníku s `n` permanentními proměnnými.	`n`	-	-
`call`	Provede volání podcíle.	arita	-	název
`call_inner`[2]	Volání podcíle v rámci procedury. Volá proceduru na `n`-tém pozici `k`-té klauzule.	`n`	arita	`k`
`clause_start`[2]	Začátek klauzule. Automaticky zvolí, zda vykonat `try_me`, `retry_me` nebo `trust_me`. Používá se u dynamických procedur.	-	-	-
`cut`	Hluboký řez. Parametr udává, v jakém registru je uložena úroveň řezu od instrukce `get_level`.	$\texttt{Y}_i$	-	-
`deallocate`	Dealokace záznamu o prostředí.	-	-	-
`execute`	Obdoba instrukce `call` využívána při volání posledního predikátu klauzule. Implementuje LCO (viz 3.4.1).	arita	-	název
`execute_cli`[2]	Provede delegát na .NET metodu.	-	-	delegát
`execute_x`[2]	Provede unární vložený predikát. Parametr udává, v jakém registru je operand.	$\texttt{X}_i$	-	název
`execute_xx`[2]	Provede binární vložený predikát. Parametry udávají, v jakých registrech jsou operandy.	$\texttt{X}_i$	$\texttt{X}_n$	název
`execute_xy`[2]	Provede binární vložený predikát. Parametry udávají, v jakých registrech jsou operandy.	$\texttt{X}_i$	$\texttt{Y}_n$	název
`execute_y`[2]	Provede unární vložený predikát. Parametr udává, v jakém registru je operand	$\texttt{Y}_i$	-	název
`execute_yx`[2]	Provede binární vložený predikát. Parametry udávají, v jakých registrech jsou operandy.	$\texttt{Y}_i$	$\texttt{X}_n$	název
`execute_yy`[2]	Provede binární vložený predikát. Parametry udávají, v jakých registrech jsou operandy.	$\texttt{Y}_i$	$\texttt{Y}_n$	název
`fail`[2]	Způsobí selhání. Do této instrukce se překládají predikáty `fail/0` a `false/0`.	-	-	-
`get_level`	Uloží do permanentního registru informaci o dosahu hluboké řezu.	$\texttt{Y}_i$	-	-
`move_register`[2]	Přesun mezi dočasnými registry.	$\texttt{X}_i$	$\texttt{X}_n$	-
`neck_cut`	Mělký řez.	-	-	-
`nop`[2]	Prázdná instrukce. Na tuto instrukci se překládá predikát `true/0`.	-	-	-

Název	Popis	Parametry		
		Int 1	Int 2	Object
`proceed`	Návrat z volání klauzule, na kterou nelze aplikovat LCO (např. fakt nebo klauzule končící vloženým predikátem).	-	-	-
`query_end`[2]	Ukončení dotazu. Zastaví výpočet a vrátí výsledky.	-	-	-
`retry`	Obnoví data z bodu volby, aktualizuje ho (návrat při selhání na následující instrukci) a provede skok o `n` instrukcí. Používáno při indexování klauzulí.	`n`	-	-
`retry_me`[2]	Obnoví data z bodu volby, aktualizuje ho (návrat při selhání na následující klauzuli) a pokračuje další instrukcí. Používáno na začátku klauzule.	-	-	-
`retry_me_else`	Obnoví data z bodu volby, aktualizuje ho (návrat při selhání na instrukci vzdálenou o `n` instrukcí) a pokračuje následující instrukcí. Používána pro operátor disjunkce.	`n`	-	-
`switch_on_constant`	Provede skok o `n` pozic podle hodnoty konstanty a její hodnoty v tabulce (instance třídy `ConstantDictionary`).Používáno při indexování.	-	-	tabulka
`switch_on_structure`	Provede skok o `n` pozic podle názvu a arity struktury a její hodnoty v tabulce (instance třídy `StructureDictionary`). používáno při indexování.	-	-	tabulka
`switch_on_term`	Provede skok o `n` pozic podle datového typu termu a jeho hodnoty v tabulce (instance třídy `TermSwitcher`). Používáno při indexování.	-	-	tabulka
`trust`	Odstraní bod volby na vrcholu zásobníku a provede skok o `n` pozic. Používáno při indexování.	`n`	-	-
`trust_me`	Odstraní bod volby na vrcholu zásobníku a pokračuje následující instrukcí.	-	-	-
`try`	Vytvoří bod volby (s návratem při selhání na následující instrukci) a provede skok o `n` instrukcí.	`n`	-	-

Název	Popis	Parametry		
		Int 1	Int 2	Object
`try_me`[2]	Vytvoří bod volby (s návratem při selhání na první instrukci další klauzule) a pokračuje následující instrukcí.	-	-	-
`try_me_else`	Vytvoří bod volby (s návratem při selhání na instrukci vzdálenou o `n` instrukcí) a pokračuje následující instrukcí. Používáno při operátoru disjunkce.	`n`	-	-

UNIFY instrukce

Název	Popis	Parametry		
		Int 1	Int 2	Object
`unify_constant`	V režimu čtení naváže na proměnnou ve složeném termu danou konstantu. Pokud je ve složeném termu konstanta a ne proměnná, porovná zda jsou konstanty shodné. V režimu zápisu funguje jako `set_constant`.	-	-	`CLI`
`unify_nil`[1]	V režimu čtení zkontroluje, jestli je ve složeném termu na aktuální pozici prázdný seznam. V režimu zápisu se chová jako `set_nil`.	-	-	-
`unify_value`	V režimu čtení unifikuje hodnoty v obou registrech. V režimu zápisu se chová jako `set_value`.	$\texttt{X}_i$	$\texttt{X}_n$	-
`unify_value_y`	V režimu čtení unifikuje hodnoty v obou registrech. V režimu zápisu se chová jako `set_value_y`.	$\texttt{Y}_i$	$\texttt{X}_n$	-
`unify_variable`	V režimu čtení načte další hodnotu z posledního čteného složeného termu do registru. V režimu zápisu se chová jako `set_variable`.	$\texttt{X}_i$	-	-
`unify_variable_y`	V režimu čtení načte další hodnotu z posledního čteného složeného termu do na zásobníku registru. V režimu zápisu se chová jako `set_variable_y`.	$\texttt{Y}_i$	-	-
`unify_void`	V režimu čtení přeskočí `n` prvků složeného termu. V režimu zápisu se chová jako `set_void`.	`n`	-	-

SET instrukce

Název	Popis	Parametry		
		Int 1	Int 2	Object
`set_constant`	Vloží do posledního vytvořeného složeného termu konstantu CLI.	`-`	`-`	`CLI`
`set_nil`[1]	Vloží do posledního vytvořeného složeného termu prázdný seznam.	`-`	`-`	`-`
`set_value`	Vloží do posledního vytvořeného složeného termu obsah registru.	$\texttt{X}_i$	`-`	`-`
`set_value_y`	Vloží do posledního vytvořeného složeného termu obsah registru.	$\texttt{Y}_i$	`-`	`-`
`set_variable`	Vytvoří novou reprezentaci proměnné, vloží ji to registru a do posledního vytvořeného složeného termu.	$\texttt{X}_i$	`-`	`-`
`set_variable_r`[2]	Vytvoří novou reprezentaci proměnné, vloží ji do registru a do posledního vytvořeného složeného termu. Hodnota proměnné je po dokončení výpočtu vrácena uživateli.	$\texttt{X}_i$	`-`	`-`
`set_variable_y`	Vytvoří novou reprezentaci proměnné, vloží ji do registr na zásobníku a do posledního vytvořeného složeného termu.	$\texttt{Y}_i$	`-`	`-`
`set_variable_yr`[2]	vytvoří novou reprezentaci proměnné, vloží ji do registr na zásobníku a do posledního vytvořeného složeného termu. Hodnota proměnné je po dokončení výpočtu vrácena uživateli.	$\texttt{Y}_i$	`-`	`-`
`set_void`	Vloží do posledního složeného termu `n` nových reprezentací proměnných.	`n`	`-`	`-`

GET instrukce

Název	Popis	Parametry		
		Int 1	Int 2	Object
`get_constant`	Pokud je v registru nenavázaná proměnná, naváže na ní konstantu CLI. Jinak porovná konstantu s obsahem registru, selže pokud se neshodují.	$\texttt{X}_i$	-	CLI
`get_list`	Pokud je v registru nenavázaná proměnná, naváže na ní novou reprezentaci seznamu. Pokud je v registru seznam, nastaví na něj ukazatel posledního složeného termu a nastaví mód na čtení. Jinak selže.	$\texttt{X}_i$	-	-
`get_nil`[2]	pokud je v registru nenavázaná proměnná, naváže na ní prázdný seznam, jinak zkontroluje, zda je v registru prázdný seznam, pokud není, selže	$\texttt{X}_i$	-	-
`get_structure`	Pokud je v registru nenavázaná proměnná, naváže na ní novou reprezentaci struktury. Pokud je v registru struktura, zkontroluje název a aritu, nastaví na ni ukazatel posledního složeného termu a nastaví mód na čtení. Jinak selže.	$\texttt{X}_i$	arita	název
`get_value`	Unifikuje obsah obou registrů.	$\texttt{X}_i$	$\texttt{X}_n$	-
`get_value_y`	Unifikuje obsah obou registrů.	$\texttt{Y}_i$	$\texttt{X}_n$	-
`get_variable_y`	Přesune obsah dočasného registru na zásobník.	$\texttt{Y}_i$	$\texttt{X}_n$	-

Příloha C

Přehled vestavěných predikátů

Vestavěné predikáty jsou inspirovány SWI-Prologem (malou částí). Navíc jsou přidány predikáty specifické pro platformu .NET. V tomto přehledu je využita následující notace:

`+` Parametr musí být instancionován.

`-` Výstupní parametr. Může obsahovat hodnotu, predikát pak uspěje pokud byla vrácena zadaná hodnota.

`--` Parametr musí být nenavázaná proměnná.

`@` Parametr nemusí být instanciván.

`?` Parametr může být vstupní i výstupní.

Aritmetické predikáty

-Číslo* is *++Výraz **vložený predikát**

Vyhodnotí aritmetický výraz *Výraz* a unifikuje ho s parametrem *Číslo*.
Dostupné aritmetické operace (lze rozšířit, viz kapitola 6.5.1):

`+` sčítání, je možné použít infixovou notaci

`-` odčítání, je možné použít infixovou notaci

`*` násobení, je možné použít infixovou notaci

`/` dělení, je možné použít infixovou notaci

`abs` absolutní hodnota

`floor` zaokrouhlení dolů

`ceil` zaokrouhlení nahoru

`round` matematické zaokrouhlení

`pi` konstanta π

+Výraz1 =:= +Výraz2 **vložený predikát**

Vyhodnotí oba výrazy a uspěje, pokud jsou jejich hodnoty shodné.

+Výraz1 =\= +Výraz2 **vložený predikát**

Vyhodnotí oba výrazy a uspěje, pokud se jejich hodnoty liší.

+Výraz1 > +Výraz2 **vložený predikát**

Vyhodnotí oba výrazy a uspěje, pokud je hodnota levého výrazu větší než hodnota pravého výrazu.

+Výraz1 >= +Výraz2 **vložený predikát**

Vyhodnotí oba výrazy a uspěje, pokud je hodnota levého výrazu větší nebo rovna hodnotě pravého výrazu.

+Výraz1 < +Výraz2 **vložený predikát**

Vyhodnotí oba výrazy a uspěje, pokud je hodnota levého výrazu menší než hodnota pravého výrazu.

+Výraz1 =< +Výraz2 **vložený predikát**

Vyhodnotí oba výrazy a uspěje, pokud je hodnota levého výrazu menší nebo rovna hodnotě pravého výrazu.

Predikáty pro práci s .NET

cli_new(*+Konstruktor*, *--Reference*)

Uspěje, pokud zadaný typ existuje a existuje zadaný konstruktor. Název třídy je nutné uvádět celý včetně jmenných prostorů. Parametrický konstruktor se pak zapisuje jako struktura.
Příklad: `cli_new('System.DateTime'(1992,4,20), Datum).`

cli_instance_of(*+Reference*, *+Typ*)

Uspěje, pokud je objekt *Reference* datového typu *Typ* nebo jeho potomků. Typ se zadává jako atom včetně jmenného prostoru.

cli_reference(*@Reference*) **vložený predikát**

Uspěje, pokud je objekt reference na typ z .NET a ne na typ Prologu.

cli_send(*+Reference*, *+Metoda*)

Zavolání metody objektu (nebo statické metody) bez návratové hodnoty. V případě statické metody je do *Reference* zadán atom s názvem třídy.
Příklad statického volání: `cli_send('MyNamespace.MyClass', 'MyMethod'(X,"abc")).`

cli_send(*+Reference*, *+Metoda*, *-Výstup*)

Zavolání metody objektu (nebo statické metody) s návratovou hodnotou. V případě statické metody je do *Reference* zadán atom s názvem třídy.

cli_get(*+Reference*, *+DatovýČlen*, *-Výstup*)

Čtení obsahu datového členu objektu nebo statické třídy.

cli_set(*+Reference*, *+DatovýČlen*, *+NováHodnota*)

Nastavení obsahu datového členu objektu nebo statické třídy

cli_throw(*+Výjimka*)

Vyhození .NET výjimky. *Výjimka* je reference na objekt dědíci od třídy `Exception`, kterou lze vytvořit pomocí `get_new/2`.

cli_to_list(*+Seznam*, *--Kolekce*)

Převede Prologovský seznam na kolekci `List`.

cli_to_prolog_list(*+Kolekce*, *-Seznam*)

Převede jakoukoli kolekci implementující rozhraní `IEnumerable` na Prologovský seznam.

cli_enumerable(*@Reference*) vložený predikát

Uspěje, pokud objekt na který vede *Reference* implementuje rozhraní `IEnumerable`.

cli_enumerate(*+Reference*, *-Hodnota*)

Vrací první prvek z objektu implementující rozhraní `IEnumerable`. Po každém zpětném navracení vrací vždy další prvek. Pokud již enumerátor nemůže najít další prvek, selže.

Predikáty pro hlášení chyb

uninstantiation_error

Vyvolá chybu (implementováno .NET výjimkou), která informuje o tom, že parametry predikátu nejsou instanciované.

domain_error(*+Zpráva*)

Vyvolá chybu (implementováno .NET výjimkou), která informuje o tom, že hodnota parametru není v požadovaném rozsahu hodnot. Parametr *Zpráva* je text informující o podrobnostech.

type_error(*+OčekávanýTyp*, *+ZadanýTyp*)

Vyvolá chybu (implementováno .NET výjimkou), která informuje o tom, že parametr je neočekávaného typu. Parametr *OčekávanýTyp* je atom a *ZadanýTyp* je objekt z parametru predikátu, který chybu způsobil.

Predikáty pro kontrolu typů

atomic(*@Term*) vložený predikát

Uspěje, pokud je term atom, číslo, řetězec nebo prázdný seznam.

atom(*@Term*) vložený predikát

Uspěje, pokud je term atom.

compound(*@Term*) vložený predikát

Uspěje, pokud je term složený (struktura nebo seznam).

integer(*@Term*) vložený predikát

Uspěje, pokud je term celé číslo.

float(*@Term*) vložený predikát

Uspěje, pokud je term číslo s plovoucí desetinnou částkou.

string(*@Term*) vložený predikát

Uspěje, pokud je term řetězec.

nonvar(*@Term*) vložený predikát

Uspěje, pokud term není nenavázaná proměnná.

var(*@Term*) vložený predikát

Uspěje, pokud je term nenavázaná proměnná.

Pomocné predikáty

findall(*+Šablona*, *+Cíl*, *-Výsledek)*

Vloží všechny výsledky podcíle *Cíl* do seznamu a unifikuje ho s parametrem *Výsledek*. Parametr *Šablona* ovlivňuje formát výsledku.
Příklad: `findall([X,Y], p(X,Y), List)` vrátí seznam seznamů o dvou položkách pro všechna řešení `p/2`.

cli_solutions_to_list(*+Šablona*, *+Cíl*, *-Výsledek)*

Obdobně jako `findall/3` vrací všechny výsledky, ale ve formě kolekce platformy .NET.

not(*+Cíl*)

Uspěje, pokud *Cíl* selže.

repeat

Vytvoří bod volby a při zpětném navracení vždy uspívá.

true **vložený predikát**

Uspívající predikát, který nevytváří bod volby.

fail **vložený predikát**

Vždy selže.

false **vložený predikát**

Obdoba `fail/0`.

! **vložený predikát**

Operátor řezu. Odstraní všechny body volby vzniklé od volání predikátu.

?Term =.. ?Seznam **vložený predikát**

Převede term na seznam nebo opačně. V případě převodu struktury na seznam, je první prvek seznamu název struktury a další jsou její prvky. Musí být instanciován alespoň jeden parametr, druhý se dopočítá.

@Term1 = @Term2 **vložený predikát**

Test na unifikovatelnost. Pokud uspěje, jsou termy unifikovány. Tedy pokud je jeden z nich nenavázaná proměnná, je tato proměnná navázána.

@Term1 \= @Term2 **vložený predikát**

Test na neunifikovatelnost. Uspěje pokud termy nelze unifikovat.

@Term1 == @Term2 **vložený predikát**

Uspěje, pokud jsou oba termy stejné. V případě .NET reference rozhoduje výsledek metody `Equals()`.

@Term1 \== @Term2 **vložený predikát**

Uspěje, pokud jsou oba termy různé. V případě .NET reference rozhoduje výsledek metody `Equals()`.

@Term1 @< @Term2 **vložený predikát**

Porovnání termů podle tzv. standardního pořadí termů. V případě různých termů je pořadí následující: proměnná < číslo < řetezec < reference na .NET objekt< atom < složený term. V případě, že jsou oba termy čísla jedno je desetinné, je druhé číslo převedeno taktéž na desetinné a porovnáno. Proměnné se porovnávají podle pořadí jejich vytvoření. Atomy a řetězce jsou porovnávány lexikograficky. Objekty .NET jsou porovnány podle výsledku metody `GetHashCode()` a struktury jsou porovnány nejprve podle arity, poté podle názvu a případně rekurzivně podle jejich prvků.

@Term1 @=< @Term2 **vložený predikát**

Stejné řazení jako u `@<`, ale uspěje i pokud jsou si termy rovné.

@Term1 @> @Term2 **vložený predikát**

Stejné řazení jako u `@<`, ale uspěje pokud je term na levé straně větší než term na pravé straně.

@Term1 @>= @Term2 **vložený predikát**

Stejné řazení jako u `@<`, ale uspěje pokud je term na levé straně větší nebo rovný než term na pravé straně.

Predikáty pro práci s řetězci

split_string(*?Řetězec*, *?Oddělovač*, *?Výsledek*)

Pokud jsou instanciovány parametry *Řetězec* a *Oddělovač* je s parametrem *Výsledek* unifikován seznam řetězců rozdělený podle zadaného oddělovače. V případě, že jsou instanciovány parametry *Oddělovač* a *Výsledek* , je s parametrem *Řetězec* unifikován řetězec vzniklý spojením výsledku oddělovači. Pokud jsou instanciovány parametry *Řetězec* a *Výsledek* , pokusí se určit oddělovač. Pokud takový oddělovač nelze najít, selže.

string_concat(*?Řetězec1*, *?Řetězec2*, *?Výsledek*)

Pokud jsou instanciovány parametry *Řetězec1* a *Řetězec2*, je do parametru *Výsledek* unifikována konkatenace obou řetězců. V ostatních případech, podle instanciovaných parametrů, zkusí určit prefix nebo postfix. Vždy musí být instanciovány alespoň dva parametry.

string_length(*+Řetězec*, *-Délka*) **vložený predikát**

Spočítá délku řetězce.

sub_string(*+Řetězec*, *?PoziceOd*, *?Délka*, *?Výsledek*)

V případě instanciace prvních tří parametrů unifikuje do parametru *Výsledek* podřetězec začínající na dané pozice o dané délce. V případě instanciace parametru *Výsledek*, zkusí nalézt *Výsledek* v řetězci *Řetězec* a s parametry *PoziceOd* a *Délka* unifikuje pozici, kde byl daný podřetězec nalezen. V tomto případě může mít predikát více řešení.

term_string(*?Term*, *?Řetězec*)

Převede term na řetězec nebo řetězec na term.

Predikáty pro práci se seznamy

member(*@Term*, *+Seznam*)

Uspěje, pokud seznam obsahuje zadaná term.

append(*+Seznam1*, *+Seznam2*, *-Výsledek*)

Spojí dva seznamy a jejich výsledek unifikuje s parametrem *Výsledek*.

reverse(*+Seznam*, *-Výsledek*)

Obrátí pořadí prvků v seznamu.

same_length(*+Seznam1*, *+Seznam2*)

Uspěje, pokud mají oba seznamy stejný počet prvků.

union(*+Seznam1*, *+Seznam2*, *-Výsledek*)

Provede sjednocení obou seznamů. Na rozdíl od `append/3` nepřidává prvky, které se v seznamu již nacházejí.

intersection(*+Seznam1*, *+Seznam2*, *-Výsledek*)

Určí průnik obsahů obou seznamů.

flatten(*+Seznam*, *-Výsledek*)

Odstraní ze seznamu zanořené seznamy a vloží všechny prvky do jednoho seznamu.

length(*+Seznam*, *-Délka*)

Spočítá délku seznamu.

delete(*+Seznam*, *+Term*, *-Výsledek*)

Odstraní všechny výskyty zadaného termu ze seznamu.

sort(*+Seznam*, *+Term*, *-SeřazenýSeznam*)

Seřadí zadaný seznam. Pro porovnání využívá predikáty `@<` apod.

Systémové predikáty

consult(*+Soubor*)

Načte do interpretu zadaný zdrojový soubor. Cesta k souboru je očekávána jako atom a bez přípony (očekává se přípona `.cpl`).

load_library_from_file(*+Soubor*)

Načte do interpretu knihovnu ve formě sestavení pro .NET. Cesta k souboru je očekávána jako atom a bez přípony (očekává se přípona `.dll`). Lze zapsat i jako direktivu.

load_library(*+Třída*)

Načte do interpretu knihovnu z aktuálního sestavení .NET. Název třídy s knihovnou je očekáván jako atom a musí být uváděn včetně jmenných prostorů. Lze zapsat i jako direktivu.

load_predicate(*+Třída*)

Zavede do interpretu predikát implementovaný třídou z .NET, tato třída se musí nacházet v aktuálním sestavení. Název třídy s predikátem je očekáván jako atom a musí být uváděn včetně jmenných prostorů. Lze zapsat i jako direktivu.

call(*+Cíl*)

Provede volání zadaného podcíle.

time(*+Cíl*)

Stejné jako `call/1`, ale na konci volání zobrazí informaci o době výpočtu.

Predikáty pro vstup a výstup

write(*@Term*) vložený predikát

Vypíše na standardní výstup zadaný term.

writeln(*@Term*) vložený predikát

Vypíše na standardní výstup zadaný term následovaný znakem nového řádku.

nl vložený predikát

Pošle na standardní výstup zalomení řádku.

read(*-Výstup*) vložený predikát

Načte term ze standardního vstupu a unifikuje ho s parametrem *Výstup*.

read_string(*-Výstup*) vložený predikát

Načte řetězec ze standardního vstupu a unifikuje ho s parametrem *Výstup*.

Predikáty pro změnu databáze

asserta(*+Klauzule*)

Přidá do databáze predikát obsažený v parametru *Klauzule* na začátek procedury.

assertz(*+Klauzule*)

Přidá do databáze predikát obsažený v parametru *Klauzule* na konec procedury.

assert(*+Klauzule*)

Stejné jako `assertz/1`.

retract(*+Klauzule*)

Odebere klauzuli z databáze.

retractall(*+Hlavička*)

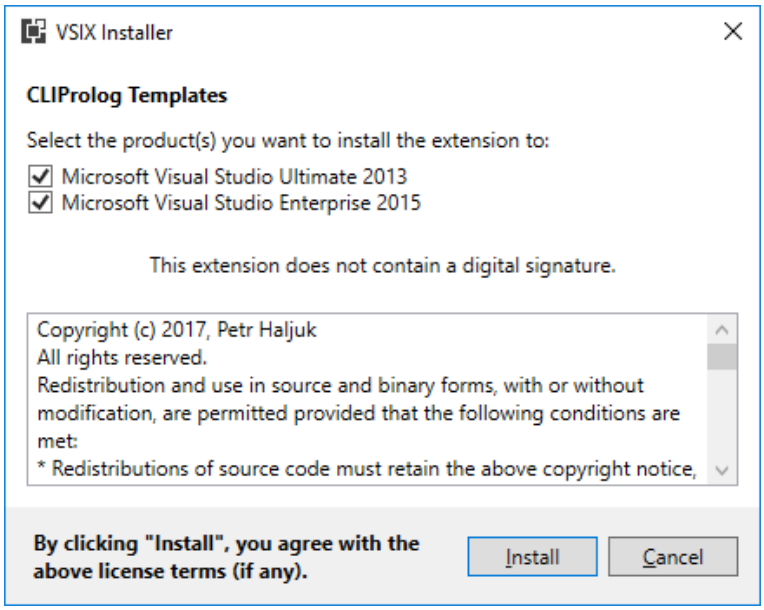
Odebere všechny klauzule, se kterými je unifikována zadaná hlavička.

Příloha D

Návod k použití

Instalace do Visual Studia

Pro instalaci šablon projektů a souborů pro Visual Studio stačí spustit instalační soubor `CLIPrologTemplates.vsix` nacházející se na přiloženém DVD v adresáři `Výsledky`. V zobrazeném okně vyberte, pro jakou verzi Visual Studia chcete balíček nainstalovat (v případě, že máte nainstalováno více verzí Visual Studia). Po instalaci tohoto balíčku se při přidání Prologovského souboru projektu automaticky přidají všechny potřebné závislosti a úlohy do MSBuild. Odinstalaci balíčku lze provést ve Visual Studiu v nabídce `Tools > Extensions and Updates`.

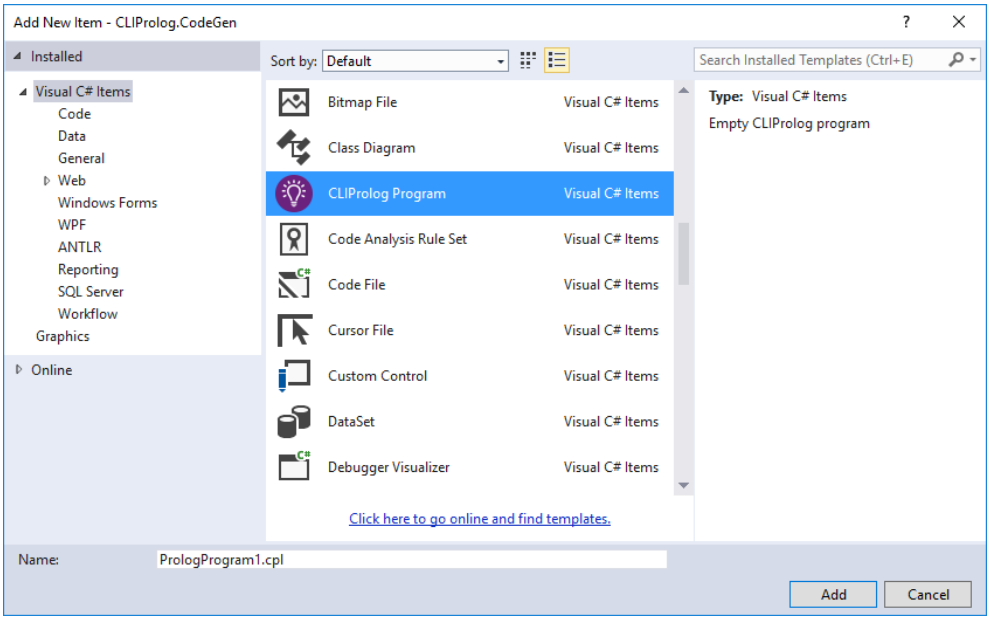


Obrázek D.1: Instalační okno VSIX balíčku

Přidání do existujícího projektu

Pokud máte nainstalované šablony z předchozí kapitoly, stačí ve Visual Studiu otevřít projekt, do kterého chcete CLIProlog přidat a v menu `Project` zvolit `Add New Item`. Následně

v sekci `Visual C# Items` najít šablonu `CLIProlog Program` a dvojklikem ho do projektu přidat. Po přeložení projektu je Prologovský program dostupný jako třída pod názvem uvedeným v direktivě `classname`.



Obrázek D.2: Dialogové okno se šablonami souborů

CLIProlog lze přidat do projektu i bez nainstalovaných VSIX šablon pomocí manuální instalace NuGet balíčku. Tento způsob je ale v porovnání s instalací balíčku VSIX pracnější a z tohoto důvodu není doporučován. NuGet balíček lze nalézt na DVD v souboru `CLIProlog.0.0.1.nupkg` v adresáři `Výsledky`. Pro jeho instalaci otevřete ve Visual Studiu okno `Package Manager Console` (v nabídce `View > Other Windows`). Do této konzole zadejte příkaz (název souboru s balíčkem se v cestě neuvádí):

```
Install-Package CLIProlog -Source <cesta adresáři s balíčkem>
```

V případě, že není nainstalován VSIX balíček, je ještě nutné ručně přidat do projektového souboru (s připonou `.cproj`) jednotlivé soubory, které má CLIProlog překládat. XML kód, který je nutné do projektového souboru přidat se nachází v ukázce D.3.

```
1  <ItemGroup>
2    <Cpl Include="PrologProgram1.cpl">
3      <Generator>MSBuild:Compile</Generator>
4    </Cpl>
5  </ItemGroup>
```

Program D.3: Kód pro přidání souboru pro překlad

Pokud nechcete překládat zvláštní soubor se zdrojovým kódem, ale chcete Prologovský program kompilovat do WAM za běhu, stačí pouze nainstalovat NuGet balíček bez úpravy projektových souborů. K získání instance Prologovského programu lze využít statické metody třídy `CliProlog`:

FromSource(řetězec) - překlad Prologovského programu předaný jako řetězec

FromObjects(IEnumrable<Clause>) - překlad Prologovského programu sestavený z objektové reprezentace

FromStream(StreamReader) - překlad Prologovského programu, který je čtený ze vstupního proudu

FromFile(string) - překlad Prologovského programu ze souboru, parametrem je cesta k souboru

Tyto metody vrací instaci objektu reprezentující program, nad kterým lze, pomocí jeho metody `Query()` volat dotazy.

Vytvoření nového projektu v CLIPrologu

Vytvoření nového projektu probíhá podobně jako přidání do existujícího. K dispozici je šablona pro konzolovou aplikaci, kterou lze nalézt v nabídce Visual Studia pro vytváření nového projektu. Aplikace hned po startu volá predikát `main/2`. Jeho prvním parametrem je seznam parametrů příkazové řádky, se kterým byl program spuštěn. Druhý parametr je nenavázaná proměnná. Očekává se, že uživatel na tuto proměnnou naváže návratový kód aplikace. V případě že kód není navázán, je použita hodnota 0.

Vytvoření knihovny

Knihovna pro CLIProlog je také druh projektu. Jeho šablonu lze nalézt na stejném místě, na kterém se nachází šablona pro konzolovou aplikaci. Tento projekt může obsahovat neomezeně Prologovských souborů (vycházejících ze šablony `CLI Prolog Library`). Může také obsahovat predikáty implementované v jazyce C#. Jak již bylo napsaná v kapitole 6.5.1, lze implementovat predikát jak s podporou zpětného navracení tak bez ní.

Jednodušším případem je predikát bez zpětného navracení. Pro jeho implementaci je nutné vytvořit třídu dědící od `DeterministicUserPredicate` a označenou atributem `PrologProcedure`. Kód predikátu je zapsán v překryté metodě `Execute()`, která vrací `bool` podle toho, zda predikát uspěl. Třída `DeterministicUserPredicate` poskytuje následující metody (s modifikátorem `protected`):

Unify(object, object) - unifikace dvou termů, vrací datový typ `bool`, který informuje, zda se unifikace zdařila

GetArgument(int - získání obsahu dočasných registrů (pokud je v registru navázaná proměnná, je vrácen term, který je na ni navázaný), na kterých jsou uloženy parametry volání predikátu (indexováno od 0)

SetArgument(int, object) - nastaví hodnotu dočasného registru

Jednoduchou implementaci predikátu bez zpětného navracení demonstruje program D.4. Obdobným způsobem jsou implementovány i predikáty s podporou zpětného navracení. Jejich předkem je třída `NonDeterministicUserPredicate` a obsahuje všechny metody jako `DeterministicUserPredicate`. Navíc přidává ještě dva datové členy (implementovány jako vlastnosti):

```
1     using System;
2     using WamRuntime.Extensions;
3
4     namespace MyNamespace
5     {
6       [PrologProcedure("test", 1)]
7       class Test : DeterministicUserPredicate
8       {
9         protected override bool Execute()
10        {
11          object par1 = this.GetArgument(0);
12
13          int year = DateTime.Now.Year;
14
15          return this.Unify(par1, year);
16        }
17      }
18    }
```

Program D.4: Kód predikátu v jazyce C#, který uspěje, pokud lze parametr unifikovat s aktuálním rokem

`IsFirstCall` - obsahuje hodnotu `true`, pokud se jedná o první volání v rámci procedury (nebylo provedeno zpětné navracení)

`Context` - objekt pro ukládání dat (systémem klíč-hodnota), které přetrvají do dalšího volání po zpětném navracení (tento objekt je uložen na zásobníku prostředí)

Ukázkovou implementaci predikátu se zpětným navracením demonstruje program D.5. Takto definované predikáty nemusí být pouze součástí knihovny. Lze je vytvořit i přímo v sestavení programu pro C# a do interpretu je načíst pomocí predikátu `load_predicate/1`. Podrobnější ukázky se nacházejí na přiloženém DVD.

Nástroj CodeGen

CodeGen je nástroj pro příkazovou řádku, který pro textový soubor s programem v jazyce Prolog vygeneruje třídu pro C#. Tuto třídu pak lze přidat do jakéhokoli projektu i v jiných vývojových prostředích než je Visual Studio. Samozřejmě je nutné k programu přiložit i knihovnu interpretu. Nástroj CodeGen se spouští s následujícími parametry příkazové řádky:

`-i` - vstupní soubor

`-o` - výstupní soubor, pokud není zadán, vypíše se program na standardí vstup

`-l` - pokud je zadán tento volitelný parametr, překládá se kód jako knihovna

```
1  using System;
2  using PrologModel.Exceptions;
3  using WamRuntime.Extensions;
4
5  namespace MyNameSpace
6  {
7    [PrologProcedure("example", 2)]
8    class Example : NonDeterministicUserPredicate
9    {
10     protected override bool Execute()
11     {
12       if (this.IsFirstCall)
13       {
14         object par1 = this.GetArgument(0);
15
16         if (!(par1 is int))
17         {
18           throw new NotSufficientlyInstantiatedException();
19         }
20
21         this.Context.SetData("number", par1);
22       }
23
24
25       object par2 = this.GetArgument(1);
26       int number = (int)this.Context.GetData("number");
27       number++;
28       this.Context.SetData("number", number);
29
30       return this.Unify(par2, number);
31     }
32   }
33 }
```

Program D.5: Kód predikátu v jazyce C#, který pro zadané číslo vrací vždy o jedno vyšší při každém zpětném navracení

Příloha E

Obsah DVD

Zde je popsán obsah jednotlivých adresářů na DVD.

Zpráva Technická zpráva ve formátu PDF a její zdrojové kódy v LaTeX

Výsledky Balíček pro Visual Studio, NuGet balíček, přeložená knihovna, nástroj CodeGen a interaktivní konzole

Příklady Příklady projektů pro Visual Studio a příklady pro spuštění v interaktivní konzoly

Testy Programy používané pro výkonnostní testy

Všechny výstupy vyžadují alespoň .NET verze 4.5. Ukázkové projekty a balíčky vyžadují Visual Studio 2013 nebo 2015 (na verzi 2017 nebylo testováno).