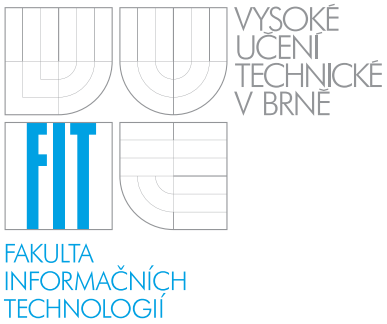


Funkcionální a logické programování
FPR
Studijní opora
Logické programování

Dušan Kolář
Ústav informačních systémů
Fakulta informačních technologií
VUT v Brně

Listopad '05 – Říjen '06
Verze 1.0

Tento učební text vznikl za podpory projektu „Zvýšení konkurenceschopnosti IT odborníků – absolventů pro Evropský trh práce“, reg.č. CZ.04.1.03/3.2.15.1/0003. Tento projekt je spolufinancován Evropským sociálním fondem a státním rozpočtem České republiky.

Abstrakt

Logické programovací jazyky vznikly jako reakce na potřebu automatizovat důkazy v predikátové logice. Během doby se však tyto jazyky staly, i když specifickým, jazykem, který měl daleko obecnější užití, než jen pro podporu důkazů. Kromě jistým způsobem známého jazyka Prolog se vyvinuly jazyky, které dále rozvíjely filozofii jazyka Prolog, ale přidávaly další vlastnosti (např. paralelní Prolog, systémy CLPR), nebo i jazyky, které přišly s propracovanějším mechanismem vyhodnocení (jazyk Gödel).

Tato publikace shrnuje základní principy a obraty z užití jazyka Prolog a jeho některých nástupců. Jejím cílem je poskytnout dostatečný podklad pro hrubou orientaci v dané kategorii a poskytnout dostatečný podklad pro zvládnutí základních programovacích technik v Prologu, kterou čitatel plně rozvine ve vlastní praktické činnosti.

Vřelé díky všem, kdo mě podporovali a povzbuzovali při práci na této publikaci.

Obsah

Notace a konvence použité v publikaci

Každá kapitola a celá tato publikace je uvozena informací o čase, který je potřebný k zvládnutí dané oblasti. Čas uvedený v takové informaci je založen na zkušenostech více odborníků z oblasti a uvažuje čas strávený k pochopení prezentovaného tématu. Tento čas nezahrnuje dobu nutnou pro opakované memorování paměťově náročných statí, neboť tato schopnost je u lidí silně individuální. Příklad takového časového údaje následuje.

Čas potřebný ke studiu: 2 hodiny 15 minut

Podobně jako dobu strávenou studiem můžeme na začátku každé kapitoly či celé publikace nalézt cíle, které si daná pasáž klade za cíl vysvětlit, kam by mělo studium směřovat a čeho by měl na konci studia dané pasáže studující dosáhnout, jak znalostně, tak dovednostně. Cíle budou v kapitole vypadat takto:

Cíle kapitoly

Cíle kapitoly budou poměrně krátké a stručné, v podstatě shrnující obsah kapitoly do několika málo vět, či odrážek.

Poslední, nicméně stejně důležitý, údaj, který najdeme na začátku kapitoly, je průvodce studiem. Jeho posláním je poskytnout jakýsi návod, jak postupovat při studiu dané kapitoly, jak pracovat s dalšími zdroji, v jakém sledu budou jednotlivé cíle kapitoly vysvětleny apod. Notace průvodce je taktéž standardní:

Průvodce studiem

Průvodce je často delší než cíle, je více návodný a jde jak do šířky, tak do hloubky, přitom ho nelze považovat za rozšíření cílů, či jakýsi abstrakt dané stati.

Za průvodcem bude vždy uveden obsah kapitoly.

Následující typy zvýrazněných informací se nacházejí uvnitř kapitol, či podkapitol a i když se zpravidla budou vyskytovat v každé kapitole, tak jejich výskyt a pořadí není nijak pevně definováno. Uvedení logické oblasti, kterou by bylo vhodné studovat naráz je označeno slovem „Výklad“ takto:

Výklad

Důležité nebo nové pojmy budou definovány a tyto definice budou číslovány. Důvodem je možnost odkazovat již jednou definované pojmy a tak významně zeštíhlet a zpřehlednit text v této publikaci. Příklad definice je uveden vzápětí:

Definice!

Definice 0.0.1 *Každá definice bude využívat* poznámku na okraji *k tomu, aby upozornila na svou existenci. Jinak je možné zvětšený okraj použít pro vpisování poznámek vlastních. První číslo v číselné identifikaci definice (či algoritmu, viz níže) je číslo kapitoly, kde se nacházela, druhé je číslo podkapitoly a třetí je pořadí samotné entity v rámci podkapitoly.*

Pokud se bude někde vyskytovat určitý postup, či konkrétní algoritmus, tak bude také označen, podobně jako definice. I číslování bude mít stejný charakter a logiku.

Algoritmus!

Algoritmus 0.0.1 *Pokud je čtenář zdatný v oblasti, kterou kapitola, či úsek výkladu prezentuje, potom je možné skočit na další oddíl stejné úrovně.*

Přeskoky v rámci jednoho oddílu však nedoporučujeme.

V průběhu výkladu se navíc budou vyskytovat tzv. řešené příklady. Jejich zadání bude jako jakékoliv jiné, ale kromě toho budou obsahovat i řešení s nástinem postupu, jak takové řešení je možné získat. V případě, že řešení by vyžadovalo neúměrnou část prostoru, bude vhodným způsobem zkráceno tak, aby podstata řešení zůstala zachována.

Řešený příklad

Zadání: Vyjmenujte typy rozlišovaných textů, které byly doposud v textu zmíněny.
Řešení: Doposud byly zmíněny tyto rozlišené texty:

- Čas potřebný ke studiu
- Cíle kapitoly
- Průvodce studiem
- Definice
- Algoritmus
- Právě zmiňovaný je potom Řešený příklad

Některé informace mohou být vypíchnuty, či doplněny takto bokem.

V závěru každého výkladové oddílu se potom bude možné setkávat s opětovným zvýrazněním důležitých pojmů které se v dané části vyskytly a případně s úlohou, která slouží pro samostatné prověření schopností a dovedností, které daná část vysvětlovala.

Pojmy k zapamatování

- Rozlišené texty
- Mezi rozlišené texty patří: čas potřebný ke studiu, cíle kapitoly, průvodce studiem, definice, algoritmus, řešený příklad.

Úlohy k procvičení:

Který typ rozlišeného textu se vyskytuje typicky v úvodu kapitoly. Který typ rozlišeného textu se vyskytuje v závěru výkladové části?

Na konci každé kapitoly potom bude určité shrnutí obsahu a krátké resumé.

Závěr

V této úvodní stati publikace byly uvedeny konvence pro zvýraznění rozlišených textů. Zvýraznění textů a pochopení vazeb a umístění zvyšuje rychlost a efektivnost orientace v textu.

Pokud úlohy určené k samostatnému řešení budou vyžadovat nějaký zvláštní postup, který nemusí být okamžitě zřejmý, což lze odhalit tím, že si řešení úlohy vyžaduje enormní množství času, tak je možné nahlédnout k nápovědě, která říká jak, případně kde nalézt podobné řešení, nebo další informace vedoucí k jeho řešení.

Klíč k řešení úloh

Rozlišený text se odlišuje od textu běžného změnou podbarvení, či ohraničením.

Možnosti dalšího studia, či možnosti jak dále rozvíjet danou tématiku jsou shrnuty v poslední nepovinné části kapitoly, která odkazuje, ať přesně, či obecně, na další možné zdroje zabývající se danou problematikou.

Další zdroje

Oblasti, které studují formát textu určeného pro distanční vzdělávání a samostudium, se pojí se samotným termínem distančního či kombinovaného studia (distant learning) či tzv. e-learningu.

Kapitola 1

Úvod

Čas potřebný ke studiu: 37 hodin 45 minut
Tento čas reprezentuje dobu pro studium celého modulu.

Údaj je pochopitelně silně individuální záležitostí a závisí na současných znalostech a schopnostech studujícího. Proto je vhodné jej brát jen orientačně a po nastudování prvních kapitol si provést vlastní revizi, neboť u každé kapitoly je individuálně uveden čas pro její nastudování.

Cíle modulu

Cílem modulu je na konkrétním příkladu logického programovacího jazyka prezentovat základní obraty, možnosti a charakter práce v takovémto typu jazyka. V úvodu potom je zmíněna i teoretická báze všech logických programovacího jazyků, predikátová logika. Význam modulu je v prezentaci deklarativního programovacího paradigmatu a jeho ovládnutí, neboť tak otevírá čitateli nové možnosti v přístupu k programovacím technikám a řešení některých problémů vůbec.

(Modul má do jisté míry charakter výuky programovacího jazyka.)

Po ukončení studia modulu:

- budete mít dostatečnou znalost jazyka Prolog pro další rozvoj práce v něm i pro tvorbu středně náročných aplikací;
- budete schopni jednoduché práce s paralelismem v jazyce Prolog;
- budete ovládat některé vestavěné a knihovní predikáty jazyka Prolog;
- dojde k revizi vašich znalostí predikátové logiky a jejich případnému doplnění pro potřeby znalosti vyhodnocení programů v logických programovacích jazycích.

Průvodce studiem

Modul začíná definicí formální báze logických programovacích jazyků, predikátové logiky. Následuje úvod do jazyka Prolog, který je reprezentantem deklarativního a logického programovacího jazyka. Je představen pojem unifikace. V dalších kapitolách je rozvíjena práce v jazyku Prolog na seznamech a jiných datových strukturách. V závěru modulu je prezentován koncept změny programu za běhu a možnosti využití paralelismu v některých systémech pro práci s jazykem Prolog.

Pro studium modulu je důležité mít dobré znalosti z oblasti výrokové a predikátové logiky, neboť tento modul, i když provádí jistou revizi těchto znalostí, tak není primárně určen pro jejich výuku a získání! Dále jsou třeba obecné znalosti a pojmy z algoritmizace, programovacích technik a technologií. Praktická zkušenost s logickými programovacími jazyky je výhodou. Praktické znalosti s imperativními programovacími jazyky mohou být výhodou, ale je však nutné počítat s tím, že postupy a koncepty užívané v imperativních jazycích se dají jen málokdy aplikovat do prostředí jazyků deklarativních.

Tento modul je jedním z modulů pro předměty funkcionální a logické programování. Modul je koncipován jako úvodní a proto je možné jej studovat zcela nezávisle, pouze s nezbytnými znalostmi vymezenými jinde v této kapitole. Dalšími moduly, které doplňují funkcionální a logické programování je modul pro funkcionální programování, který s tímto tvoří jeden celek.

Návaznost na předchozí znalosti

Pro studium tohoto modulu je nezbytné , aby studující měl znalosti z výrokové a predikátové logiky na dostatečné úrovni. Vhodná je i znalost programovacích technik a technologií a rozvinutá praktickou zkušenost s programováním na vysoké úrovni. Zkušenost s deklarativními programovacími jazyky není nutná, ale je výhodou. Stejně tak znalosti z oblasti zpracování a interpretace programovacích jazyků se mohou jevit jako výhoda.

1.1 Koncepce modulu

Následující kapitola rekapituluje formální logické báze pro logické programovací jazyky, výrokovou a zejména tedy predikátovou logiku.

Další kapitola je úvodem k jazyku Prolog. Prezentováno je samotné logické programování, základní typy a vlastnosti jazyka Prolog, demonstrovány vybrané vestavěné predikáty a zejména prezentován pojem unifikace, který v jazyku Prolog hraje klíčovou roli.

V následujících kapitolách jsou potom rozvinuty základní obraty užívané v jazyky Prolog, demonstrovány programovací praktiky,

zejména potom změna programové databáze/programu za běhu programu. V závěrečné kapitole je potom demonstrováno využití paralelismu v jazyku Prolog na dvou jeho implementacích.

Informace předkládané v kapitolách budou zejména náročné na pochopení celého konceptu. Nemá tedy smysl je memorovat (až na pár pojmů). Důležité je pochopit celou logiku věci. Bez důkladného pochopení nemá smysl dále rozvíjet práci v logických jazycích, neboť se v tomto případě jedná o absolutní základy.

1.2 Potřebné vybavení

Pro studium a úspěšné zvládnutí tohoto modulu není třeba žádné speciální vybavení. Naprosto dostačující je běžné PC s operačním systémem řady Windows či Linux. Na takovém PC je nutné mít nainstalován interpret či překladač jazyka Prolog (SWI Prolog, Ciao Prolog). Pro práci začátečníka je interpret jednodušší, proto doporučujeme nepoužívat kompilaci. Výhodou je přístup k Internetu, aby bylo možné znalosti průběžně aktualizovat a doplňovat z dalších zdrojů.

Další zdroje

Logické paradigma vyžaduje explicitní znalost predikátové logiky. Dále je dobré mít zvládnuté obecné techniky algoritmizace, algebry apod. Tento modul není úvodem do programování ani do predikátové logiky, ale úvodem do jistého druhu programování, proto není možné vysvětlovat základní obraty z programování ani predikátové logiky a je nutné je nastudovat jinde.
Řadu odkazů či přímo textů lze nalézt Internetu, či na stránkách doporučených interpretů jazyka Prolog. Z literatury tištěné lze doporučit:

- Nilsson, U., Maluszynski, J.: *Logic, Programming and Prolog (2ed)*, John Wiley & Sons Ltd., 1995.
- Bieliková, M., Návrat, P.: *Funkcionálne a logické programovanie*, Vydavateĺstvo STU, Vazovova 5, Bratislava, 2000.

V textu se také u vybraných klíčových termínů mohou objevit i odpovídající anglické termíny, které by měly umožnit rychlé vyhledání relevantních odkazů na Internetu, který je v tomto směru bohatou studnicí znalostí, jenž mohou vhodně doplnit a rozšířit studovanou problematiku.

Kapitola 2

Formální logika

Tato kapitola podává formální základy logických programovacích jazyků — výrokovou a predikátovou logiku. Od těchto formálních základů se odvíjí zpracování a funkčnost logických jazyků jako takových.

Čas potřebný ke studiu: 5 hodin 30 minut.

Cíle kapitoly

Cílem kapitoly je zvládnout orientaci v základních a důležitých pojmech z oblasti výrokové a predikátové logiky. Osvojit si terminologii, která je s těmito formalismy spojena. Zvládnout důležité vlastnosti uvedených formalismů a zejména důsledků, které z nich vyplývají pro zpracování logických jazyků.
Kapitola je míněna přehledově, aby připomněla terminologii z jiných, matematických, modulů. Ti, kteří terminologii znají, mohou přeskočit, ti, kteří vůbec netuší by měli sáhnout po odborně zaměřené publikaci.

Průvodce studiem

Před tím, než se budeme zabývat jednotlivými typy programovacích jazyků podrobně, tak je třeba si vybudovat jistou taxonomii a názvosloví, aby potom bylo jasné, jaký termín se v daných souvislostech jak chápe.
Ke studiu této kapitoly není třeba žádného speciálního vybavení. Jako u většiny ostatních se jedná o to si zapamatovat danou terminologii a důležité a podstatné věci v kapitole. Pro hlubší studium, nebo bližší objasnění termínů je vhodné využít informací na Internetu. Potom stačí počítač s prohlížečem WWW stránek a připojení k Internetu. Pro vyhledávání je vhodné využít anglickou terminologii, která je při zavádění termínů uváděna v závorkách.

Obsah

Výklad

2.1 Výroková logika

Neformální definice

Výroková logika je také často označována slovním spojením *propoziční logika*. Ve výrokové logice je každý fakt, jako například „Tonda jede autem“, reprezentován jednoduchým atomickým výrokem, který označíme například P. Z jednotlivých výroků je možné sestavit mnohem složitější výrazy — *věty*. Věty tvoříme ze základních výroků pomocí *logických spojek*: $\wedge$ (logické a), $\vee$ (logické nebo), $\neg$ (logická negace), $\rightarrow$ (implikace) a $\leftrightarrow$ (ekvivalence). Můžeme si vypomoci i závorkami.

základní terminologie

logické spojky

Máme-li tedy k původnímu výroku další, Q, „Tonda má auto“, tak můžeme mít například takovéto věty (fakty) vzniklé spojením svou předchozích:

- $P \vee Q$: „Tonda jede autem nebo Tonda má auto“
- $P \wedge Q$: „Tonda jede autem a Tonda má auto“
- $\neg P$: „Tonda nejede autem“
- $Q \rightarrow P$: „Jestliže má Tonda auto potom Tonda jede autem“

Obecně, pokud X a Y jsou věty ve výrokové logice, tak potom i $X \wedge Y$, $X \vee Y$, $\neg X$, $X \rightarrow Y$ a $X \leftrightarrow Y$ jsou platné věty výrokové logiky. Příkladem uplatnění tohoto pravidla v kombinaci se závorkami jsou tyto věty výrokové logiky:

tvoření vět

- $P \vee \neg Q$
- $P \wedge (P \rightarrow Q)$
- $(Q \vee \neg R) \rightarrow P$

Výrok může být v reálném světě buďto pravdivý, nebo nepravdivý. *Interpretace* výroku mu přiřazuje pravdivostní hodnotu pravda (truth, T) nebo nepravda (false, F). Pravdivost libovolné věty je potom možné zjistit na základě sestavení *pravdivostní tabulky*, která definuje pravdivostní hodnoty věty na základě pravdivostního ohodnocení svých součástí. Pravdivostní tabulka tak poskytuje jednoduchou *sémantiku* pro výrazy ve výrokové logice. Jednoduchost je dána tím, že hodnoty výroků a tím pádem i vět mohou být pouze pravda, či nepravda. Například:

pravdivost

A	B	A ∧ B
T	T	T
T	F	F
F	T	F
F	F	F

Pro odvození nových fakt se ve výrokové logice používají *odvozovací pravidla*. Sémantika logiky určuje, která odvozovací pravidla jsou obecně platná. Jedno velmi známé a často užívané pravidlo je *modus ponens*:

Definice!

Definice 2.1.1 *Modus ponens:*

$$\frac{A, A \rightarrow B}{B}$$

Pravidlo říká, že pokud je platné A a přitom z A nutně vyplývá B, potom také platí B. Je to nakonec možné ověřit užitím pravdivostní tabulky.

Formální definice

Koncept neformálně představený v předchozí stati nyní formalizujeme a doplníme několika tvrzeními (bez důkazů, ty je možné nalézt ve vhodné literatuře). Budeme se přitom snažit zejména o zavedení správné terminologie.

Definice!

Definice 2.1.2 Jazyk (syntaxe) výrokové logiky *je definován za pomoci abecedy a gramatiky nad touto abecedou.* Abeceda *obsahuje pouze:*

1. *výrokové symboly/proměnné (atomické formule) P, Q, R, ... (symboly výrokových proměnných)*
2. *logické spojky:* $\wedge$, $\vee$, $\neg$, $\rightarrow$, $\leftrightarrow$
3. *pomocné symboly (závorek):* (,), [,]

Gramatika *je definována takto:*

1. *výrokové symboly (P, Q, R, ...) jsou* správně utvořené formule *(s.u.f.), anglicky* well-formed formulas *(wffs)*
2. *jestliže A, B jsou s.u.f., pak* $(A \wedge B)$, $(A \vee B)$, $\neg A$, $(A \rightarrow B)$, $(A \leftrightarrow B)$ *jsou s.u.f.*

3. *každá formule vznikla konečným použitím pravidel (1) a (2) — nic jiného není s.u.f.*

Na základě definovaného jazyka výrokové logiky můžeme nyní definovat sémantiku:

Definice 2.1.3 Interpretace *(pravdivostní ohodnocení, valuace) je zobrazení ν, které každému výrokovému symbolu v s.u.f přiřazuje pravdivostní hodnotu T (0) nebo F (1).* **Definice!**

Definice 2.1.4 *Formule A nabývá pravdivostní hodnotu* 1, $\bar{\nu}(A) = 1$, *pokud:* **Definice!**

- $A = B$ *a zároveň:* $\nu(B) = 1$,
- $A = \neg B$ *a zároveň:* $\bar{\nu}(B) = 0$,
- $A = (B \wedge C)$ *a zároveň:* $\bar{\nu}(B) = \bar{\nu}(C) = 1$,
- $A = (B \vee C)$ *a zároveň:* $\bar{\nu}(B) = 1$ *nebo* $\bar{\nu}(C) = 1$,
- $A = (B \rightarrow C)$ *a zároveň:* $\bar{\nu}(B) = 0$ *nebo* $\bar{\nu}(C) = 1$,
- $A = (B \leftrightarrow C)$ *a zároveň:* $\bar{\nu}(B) = \bar{\nu}(C)$.

Pokud nabývá formule A pravdivostní hodnotu 1, tak je pravdivá. Definujme.

Definice 2.1.5 *Formule A je* pravdivá *při ohodnocení ν pokud* $\bar{\nu}(A) = 1$. *Jinak je formule A* nepravdivá. **Definice!**

V dalším definujeme velmi důležitý pojem — *model*. Tento pojem bude dále rozvinut v predikátové logice a je velmi důležitý pro podstatu fungování a práci vyhodnocovacích mechanismů logických programovacích jazyků.

Definice 2.1.6 *Je-li formule A pravdivá při ohodnocení ν, říkáme, že ν je* modelem *A a píšeme $\nu \models A$.* **Definice!**

V praxi se často ptáme, zda-li pro danou formuli existuje nějaká interpretace, která by byla modelem pro danou formuli. Pokud taková existuje, alespoň hypoteticky, mluvíme o *splnitelnosti*.

Definice!

Definice 2.1.7 *Formule A je splnitelná, pokud existuje alespoň jedna interpretace ν taková, že ν je modelem A, $\nu \models A$.*

Tento koncept můžeme dále zobecnit v následujících třech definicích.

Definice!

Definice 2.1.8 Tautologie *je formule (věta), A, která nabývá hodnoty pravda při každé interpretaci (tedy formule, která je splnitelná v každé interpretaci). Píšeme $\models A$.*

Definice!

Definice 2.1.9 *Říkáme, že množina formulí T je splnitelná, jestliže existuje pravdivostní ohodnocení ν takové, že každá formule $A \in T$ je pravdivá při ohodnocení ν. V takovém případě říkáme, že ν je modelem T.*

Definice!

Definice 2.1.10 *Říkáme, že formule A je (tautologickým)* důsledkem *množiny formulí T, jestliže každý model množiny T je také modelem A. Píšeme $T \models A$.*

Pro práci s formulemi lze užít řadu pravidel, která se mohou hodit při různých operacích (zmíněných dále). Dvě vybraná pravidla budeme prezentovat.

Teorém 2.1.1 *Tautologie zůstane tautologií, i když v ní nahradíme každý výskyt určité proměnné jednou a toutéž správně utvořenou formulí.*

Teorém 2.1.2 *Nechť A je s.u.f., která obsahuje alespoň na jednom místě podformuli B. Jestliže platí, že B je ekvivalentní s C, a jestliže formule A' vznikne nahrazením libovolného počtu výskytů formule B formulí C ve formuli A, pak A je ekvivalentní s A'.*

Podle pravidla substituce nahrazujeme proměnné, podle pravidla ekvivalentního nahrazení nahrazujeme formule. Při užití pravidla substituce dosazujeme za každý výskyt, při užití pravidla ekvivalentního nahrazení dosazujeme za libovolný počet výskytů.

Dosud prezentovaný počet logických spojek zdaleka není úplný, ale určitě ani minimální. Minimalitu počtu logických spojek ustanovuje následující *věta o reprezentaci*.

Teorém 2.1.3 *Každou pravdivostní n-ární funkci f lze reprezentovat formulí výrokové logiky, která obsahuje pouze spojky negace (¬) a konjunkce (∧), resp. disjunkce (∨): $A(p_1, p_2, \ldots, p_n)$.*

K vyjádření, jak naznačuje věta, stačí určitá *základní* množina logických spojek. O tom, že to je pravda, se lze lehce přesvědčit, pokud si vybereme negaci (¬) a konjunkci (∧):

- $(A \vee B)$ je zkratka za formuli $\neg(\neg A \wedge \neg B)$
- $(A \rightarrow B)$ je zkratka za formuli $\neg(A \wedge \neg B)$
- $(A \leftrightarrow B)$ je zkratka za formuli $\neg(\neg(A \wedge B) \wedge \neg(\neg A \wedge \neg B))$

Řešený příklad

Zadání: Ověřte, že náhrada pro ekvivalenci pomocí negace (¬) a konjunkce (∧) je správná. *Řešení:* Použijeme pravdivostní tabulky:

Tabulka 1: $A \leftrightarrow B$

A	**B**	**A ↔ B**
T	*T*	*T*
T	*F*	*F*
F	*T*	*F*
F	*F*	*T*

Tabulka 2: $\neg(A \wedge B)$

A	**B**	**A ∧ B**	**¬ (A ∧ B)**
T	*T*	*T*	*F*
T	*F*	*F*	*T*
F	*T*	*F*	*T*
F	*F*	*F*	*T*

Tabulka 3: $\neg(\neg A \wedge \neg B)$

A	**B**	**¬A**	**¬B**	**¬A ∧ ¬B**	**¬ (¬A ∧ ¬B)**
T	*T*	*F*	*F*	*F*	*T*
T	*F*	*F*	*T*	*F*	*T*
F	*T*	*T*	*F*	*F*	*T*
F	*F*	*T*	*T*	*T*	*F*

Z tabulek 2 a 3 vezmeme jako základ poslední sloupce a spojíme je konjunkcí, výsledek ukazuje následující tabulka 4.

Tabulka 4: $\neg(A \wedge B) \wedge \neg(\neg A \wedge \neg B)$

$\neg\ (\mathbf{A} \wedge \mathbf{B})$	$\neg\ (\neg\mathbf{A} \wedge \neg\mathbf{B})$	$\neg\ (\mathbf{A} \wedge \mathbf{B}) \wedge \neg\ (\neg\mathbf{A} \wedge \neg\mathbf{B})$
F	T	F
T	T	T
T	T	T
T	F	F

Negací výsledku tabulky 4 potom dostáváme hodnotu pro náhradu za původní výraz.

Tabulka 5: $\neg(\neg(A \wedge B) \wedge \neg(\neg A \wedge \neg B))$

$\neg\ (\mathbf{A} \wedge \mathbf{B}) \wedge \neg\ (\neg\mathbf{A} \wedge \neg\mathbf{B})$	$\neg(\neg\ (\mathbf{A} \wedge \mathbf{B}) \wedge \neg\ (\neg\mathbf{A} \wedge \neg\mathbf{B}))$
F	T
T	F
T	F
F	T

Na závěr výsledek původní, z tabulky 1, srovnáme s výsledkem novým, z tabulky 5. Vidíme, že se jedná o naprosto shodný výsledek, ostatně jak demonstruje tabulka 6 níže,

Tabulka 6: porovnání

$\neg(\neg\ (\mathbf{A} \wedge \mathbf{B}) \wedge \neg\ (\neg\mathbf{A} \wedge \neg\mathbf{B}))$	$\mathbf{A} \leftrightarrow \mathbf{B}$
T	T
F	F
F	F
T	T

Učiňme nyní dohodu, že výrazem konjunkce, resp. disjunkce, rozumíme formuli, která spojuje spojkou konjunkce, resp. disjunkce, n atomických formulí. Potom můžeme učinit následující sérii definicí, kterou podáme jako definic jednu.

Definice!

Definice 2.1.11 Literály *jsou atomické formule a negace atomických formulí.*

Formule A je normální disjunktivní formou *nad atomickými formulemi p_1, p_2, ..., p_n, je-li A tvaru disjunkce, jejíž každý člen je konjunkcí nějakých literálů z p_1, p_2, ..., p_n nebo literálem.*

Formule A je normální konjunktivní formou *nad atomickými formulemi p_1, p_2, ..., p_n, je-li A tvaru konjunkce, jejíž každý člen je disjunkcí nějakých literálů z p_1, p_2, ..., p_n nebo literálem.*

Elementární konjunkcí *nad p_1, p_2, ..., p_n je každá konjunkce literálů z těchto formulí, v níž se každý z těchto symbolů vyskytuje jako literál právě jednou.*

Elementární disjunkcí *nad* $p_1, p_2, \ldots, p_n$ *je každá disjunkce literálů z těchto formulí, v níž se každý z těchto symbolů vyskytuje jako literál právě jednou.*

Úplnou normální disjunktivní formou *(ÚDNF) je každá disjunkce různých elementárních konjunkcí nad* $p_1, p_2, \ldots, p_n$.

Úplnou normální konjunktivní formou *(ÚKNF) je každá konjunkce různých elementárních disjunkcí nad* $p_1, p_2, \ldots, p_n$.

Ke každé formuli A výrokové logiky, která není tautologií, nebo kontradikcí (opak tautologie), lze najít formuli B, která je ve tvaru ÚDNF (ÚKNF) a je ekvivalentní s A (ÚDNF nelze zkonstruovat pro kontradikce a ÚKNF tedy nelze zkonstruovat pro tautologie).

Formální systém výrokové logiky — axiomatizace

Formální systém je tvořen/obsahuje tyto prvky:

- Jazyk/redukce jazyka — z původní definice přebíráme vše, až na množinu logických spojek, ta se omezuje pouze na negaci ($\neg$) a implikaci ($\rightarrow$), což je také dostačující systém.
- Axiomy/schémata axiomů — jsou základní vybrané pravdivé věty (tautologie) daného sytému.
- Odvozovací pravidlo — modus ponens (viz definici 2.1.1)
- Důkazy
- Důkazy z předpokladů
- Věty o úplnosti
- Věta o kompaktnosti

V několika následujících definicích a větách (bez důkazů) si vysvětlíme a shrneme základní pojmy, které se dříve neobjevily, nebo jsou pro nás zvláště důležité dále.

Definice 2.1.12 Důkaz *je v daném systému konečná posloupnost správně utvořených formulí (kroků důkazu), z nichž každá je buď axiomem nebo byla odvozena aplikací některého pravidla odvození na některé předcházející správně utvořené formule.* **Definice!**

Definice!

Definice 2.1.13 *Určitá formule je* dokazatelná *v daném systému tehdy a jen tehdy, když v tomto systému existuje důkaz, jehož je tato formule členem.* ($\vdash$)

Od důkazu a dokazatelnosti je jen krok k tomu, co je to teorém.

Definice!

Definice 2.1.14 *Teorém je v systému S taková formule, která je v systému S dokazatelná.*

Definice důkazu říká, že vstupními body jsou axiomy a potom užitím odvozovacích pravidel z nich. Často však nejsme v situaci, kdy dokazujeme něco obecně platného. Spíše máme k dispozici nějakou hypotézu, nebo základní předpoklady, ze kterých se snažíme poté něco odvodit — dokázat.

Definice!

Definice 2.1.15 Důkaz z hypotéz*: Důkaz je v daném systému S konečná posloupnost správně utvořených formulí, přičemž formule A_i je v daném systému dokazatelná z množiny hypotéz H (tj. formulí, které obecně nejsou tautologiemi), jestliže pro každé i platí buď*

1. *A_i je jednou z hypotéz, nebo*
2. *A_i je axiómem, nebo*
3. *vzniklo jako závěr odvozovacího pravidla modus ponens, jehož předpoklady leží mezi $A_1, \ldots, A_{i-1}$.*

Řekneme, že formule A_i je dokazatelná z předpokladů H *($H \vdash A_i$).*

Teorém 2.1.4 Věta o dedukci (metateorém dedukce)*: Je-li věta B dokazatelná z vět A_1, A_2, …, A_n, pak věta $A_n \to B$ je dokazatelná z vět A_1, A_2, …, A_{n-1}, (tedy je-li A_1, A_2, …, A_{n-1}, $A_n \vdash B$, pak A_1, A_2, …, $A_{n-1} \vdash A_n \to B$).*

K pochopení dalšího je vhodné definovat i sémantiku některých pojmů, které se objevily výše. Následovat budou jen dvě.

- *Sémantický výklad pravidla modus ponens*: Jestliže formule A a formule $A \to B$ jsou tautologie, pak i B je tautologií. (Modus ponens je tedy pravidlo, které zachovává i pravdivost.)

- *Sémantická varianta věty o dedukci*: Formule/věta B vyplývá z $A_1, A_2, \ldots, A_{n-1}, A_n$ (tedy $A_1, A_2, \ldots, A_{n-1}, A_n \models B$), právě když $A_1, A_2, \ldots, A_{n-1} \models A_n \rightarrow B$.

V následujícím potom shrneme klíčové vlastnosti, které u logických (a nejen takových, obecně formálních) systémů zkoumáme. Tyto vlastnosti jsou: *rozhodnutelnost*, *bezespornost*=konzistence a *úplnost*.

Rozhodnutelnost

Definice 2.1.16 *Množina A je rozhodnutelná ve své nadmnožině B tehdy a jen tehdy, když existuje efektivní procedura (procedura o konečném počtu kroků) aplikovatelná na prvky množiny B taková, že umožní rozhodnout o každém prvku množiny B zda je, či není, prvkem množiny A.* **Definice!**

Teorém 2.1.5 *Abeceda musí být rozhodnutelná v množině všech možných symbolů.*

Množina správně utvořených formulí musí být rozhodnutelná v množině všech možných kombinací znaků abecedy.

Množina axiomů musí být rozhodnutelná v množině správně utvořených formulí.

Daný systém je rozhodnutelný tehdy a jen tehdy, když množina jeho teorémů je rozhodnutelná v množině všech správně utvořených formulí.

Zejména poslední z tvrzení shrnutých do jednoho bodu je velmi důležité a určuje řadu vlastností logických programovacích jazyků.

Bezespornost

Definice 2.1.17 *Daný systém je* syntakticky bezesporný*, platí-li, že v něm není dokazatelná věta $A \wedge \neg A$ (spor).* **Definice!**

Daný systém je sémanticky bezesporný *(korektní), platí-li pro každou větu A, že je-li tato věta dokazatelná, pak je taky logicky pravdivá (je tautologií). (Je-li $\vdash A$, pak $\models A$.)*

Podobně i bezespornost a následně úplnost významně určují chování evaluátorů logických jazyků, jak ukáže věta v závěru tohoto výkladového bloku, pro výrokovou logiku situace není až tak hrozivá. Jak uvidíme později, situace je mnohem vážnější pro logiku predikátovou.

Úplnost

Definice!

Definice 2.1.18 *Daný systém je* syntakticky úplný, *platí-li, že přidáme-li k množině axiomů správně utvořenou formuli (větu), která není teorémem, pak dostaneme sporný systém.*

Daný systém je sémanticky úplný, *platí-li pro každou větu A, že je-li tato věta tautologií, pak je v daném systému dokazatelná (je teorémem). (Je-li $\models A$, pak $\vdash A$.)*

Pozn.: Když je každý teorém tautologií, jde o sémantickou bezespornost (korektnost), když je každá tautologie teorémem, jde o sémantickou úplnost. Když je systém jak bezesporný, tak i úplný, tak množina teorémů je totožná s množinou tautologií.

Klíčový teorém pro výrokovou logiku:

Teorém 2.1.6 *Systém výrokové logiky je rozhodnutelný, bezesporný i úplný.*

Doplníme o teorém, který upravuje vztah bezespornosti a splnitelnosti.

Teorém 2.1.7 *Je-li T množina formulí výrokové logiky, potom množina T je bezesporná, právě když je splnitelná.*

Pojmy k zapamatování
Klíčových pojmů této kapitoly je celá řada. Snad nejdůležitější je spornost/bezespornost, rozhodnutelnost, úplnost a splnitelnost/nesplnitelnost.

Výklad

2.2 Predikátová logika

Neformální definice

Mezi hlavní nedostatky výrokové logiky patří to, že nedokáže vyjádřit obecná tvrzení typu „Tonda řídí všechna auta, která se mu líbí.“ Abychom vyjádřili, co vše má Tonda rád, co se mu líbí, bychom museli mít celou řadu pravidel. Proto nastupuje predikátová logika.

atomická věta

Věty v predikátovém kalkulu se staví z *atomických vět*. Atomická

věta je tvořena jménem predikátu následovaným všemi parametry tohoto predikátu. Parametrem přitom může být libovolný *term*. Mezi termy řadíme: term

- konstantní symboly — například „tonda“
- proměnné — například „X“ (notaci s velikostí počátečních písmen již přebíráme z Prologu)
- funkční výrazy — například „bratr(tonda)“ — výraz je tvořen funktorem následovaným potřebným počtem parametrů, což jsou zase libovolné termy

Mezi atomické věty tak lze zařadit i tyto příklady:

- přátelé(tonda, marcela)
- přátelé(otec(karel), matka(franta))
- umí_řídit(karel, X)

Věty se potom v predikátové logice utváří podobně jako v logice výrokové — atomické věty jsou spojovány do vyšších celků pomocí logických spojek. Mezi věty predikátové logiky tedy lze zařadit: věty

- přátelé(tonda, marcela) → má_rád(tonda, marcela)
- umí_řídit(karel, auto) ∨ umí_řídit(karel, motorka)
- (umí_řídit(karel, auto) ∨ umí_řídit(karel, motorka)) → (vlastní(karel, řidičák) ∧ koupí(karel, auto))

Oproti výrokové logice je však v predikátové logice možné pro tvorbu vět použít i *kvantifikátory*, pomocí kterých lze definovat, jak lze chápat kterou proměnnou uvnitř věty. Kvantifikátory jsou v predikátové logice dva *univerzální* (∀) a *existenční* (∃). Příkladem vět s kvantifikátory může být: kvantifikátory

- ∃X: pták(X) ∧ ¬létá(X)
- ∀X: (osoba(X) → ∃Y: má_rád(X, Y))

Věta v predikátové logice by měla mít všechny své proměnné kvantifikované. I když lze utvořit s.u.f. bez této vlastnosti — například ∀X: loves(X,Y) — tak se nejedná o větu. S.u.f., která má všechny své proměnné kvantifikované jsou také nazývány *uzavřené formule.* uzavřené formule

Sémantiku predikátové logiky, podobně jako u výrokové, definujeme v rovině pravdivostních hodnot vět. Pravdivostní hodnotu větu můžeme určit poměrně jednoduše, pokud známe pravdivostní hodnoty všech jejich základních komponent. Pravdivostní hodnoty a význam těchto základních komponent určuje *interpretace*, která tyto hodnoty interpretace

domény

určuje na základě jisté *domény*, kterou máme na mysli.

Ve výrokové logice byla interpretace jednoduchá — pouze přiřadila hodnoty pravda a nepravda atomickým výrokům/proměnným. V predikátové logice však máme navíc konstantní symboly, funkční výrazy, predikáty, kvantifikátory. Krom toho, predikátová logika pracuje s objekty z určité *domény*, která může být potenciálně nekonečná. Uvažme například predikát *je_otec*, který má dva parametry a doménu jen se třemi objekty *karel*, *pepa* a *mirek*. Interpretace musí potom definovat chování (pravdivostní hodnotu) predikátu pro všechny kombinace hodnot z dané domény, což je 6 možností a pravdivá je přitom kombinace *karel*, *pepa* v uvedeném pořadí. Podobně interpretace definuje chování pro ostatní prvky predikátové logiky. U kvantifikátorů je význam takový, že zápis $\forall X$*: je_otec(karel, X)* opravdu přiřazuje X všechny hodnoty, pro které je predikát *je_otec(karel, X)* pravdivý. Obdobně je definována sémantika existenčního kvantifikátoru, kde musí být splněna podmínka alespoň pro jeden objekt z domény.

Formální definice

Začneme definicí termů:

Definice!

Definice 2.2.1 Term*, v predikátové logice, je:*

- *každá proměnná;*
- *výraz* $f(t_1, \ldots, t_n)$*, je-li* f *n-ární funkční symbol a* $t_1, \ldots, t_n$ *jsou termy;*
- *každý výraz získaný konečným počtem aplikací předchozích dvou pravidel, nic jiného term není.*

Jelikož jsou termy definovány konečným počtem pravidel, říkáme, že to jsou konečná slova.

Termy jsou základní stavební kameny pro stavbu formulí. Formální definice následuje:

Definice!

Definice 2.2.2 *Formule, v predikátové logice, je:*

- *výraz* $p(t_1, \ldots, t_n)$*, je-li* p *n-ární predikátový symbol a* $t_1, \ldots, t_n$ *jsou termy, tento výraz je zván taktéž* atomická *formule;*
- *výraz:* $\neg A$, $(A \wedge B)$, $(A \vee B)$, $(A \rightarrow B)$, $(A \leftrightarrow B)$*, pokud* A *a* B *jsou formule;*
- *výraz:* $\forall x : A$, $\exists x : A$*, pokud* x *je proměnná a* A *je formule.*

Pro řadu operací s formulemi je nutné znát pojem volné a vázané proměnné. Formálně lze tyto pojmy definovat např. takto:

Definice 2.2.3 *Nechť t je term a A je formule, potom:* **Definice!**

- *podslovo s termu t, které je samo termem nazveme podtermem termu t;*
- *podslovo B formule A, které je samo formulí nazveme podformulí formule A;*
- *daný výskyt proměnné x ve formuli A je* vázaný*, je-li součástí nějaké podformule tvaru $\forall x : B$ nebo $\exists x : B$; není-li daný výskyt proměnné x vázaný, říkáme, že je* volný*;*
- *říkáme, že proměnná x je* volná *ve formuli A, má-li tam volný výskyt; proměnná x je* vázaná *v A, má-li tam vázaný výskyt;*
- *formule A je* otevřená*, pokud neobsahuje žádnou vázanou proměnnou; A je* uzavřená*, neobsahuje-li žádnou volnou proměnnou.*

Před tím, než se dostaneme k sémantice predikátové logiky, definujme formálně interpretaci jazyka, na jejímž základě budeme definovat interpretaci termů.

Definice 2.2.4 Interpretace jazyka *je definována množinovou (relační) strukturou $\mathcal{M}$ která ke každému symbolu jazyka a k množině proměnných přiřadí množinu individuí.* **Definice!**

Relační struktura $\mathcal{M}$ obsahuje:

- *neprázdnou množinu M pro individua;*
- *zobrazení $f_M : M^n \to M$ pro každý n-ární funkční symbol f;*
- *relaci $p_M \subseteq M^n$ pro každý n-ární predikát p.*

Interpretace termů je dána definicí:

Definice 2.2.5 *Mějme jazyk $\mathcal{L}$ a strukturu $\mathcal{M}$, která ho interpretuje. Chceme každému termu přiřadit jeho hodnotu v doméně M struktury $\mathcal{M}$.* **Definice!**

1. *Proměnné, v termech musíme ohodnotit nejdříve. Použijeme zobrazení e, které každé proměnné x přiřadí hodnotu $e(x)$ z domény M. Takovému zobrazení říkáme* ohodnocení proměnných.

2. Interpretaci termu t *při ohodnocení* e *označíme* $t[e]$. *Definujme:*

- $t[e] = e(x)$, *je-li* t *proměnná* x;
- $t[e] = f_M(t_1[e], \ldots, t_n[e])$, *je-li* t *tvaru* $f(t_1, \ldots, t_n)$.

Nyní nám už nic nebrání v definici pojmu pravda, použijeme *Tarského* definici pravdy.

Definice!

Definice 2.2.6 *Nechť* $\mathcal{L}$ *je jazyk,* $\mathcal{M}$ *jeho interpretace,* e *pravdivostní ohodnocení a* A *je formule jazyka* $\mathcal{L}$.

Je-li e *ohodnocení proměnných,* x *je proměnná a* m *je prvek z domény* M, *pozměněné ohodnocení* $e(x/m)$ *definujeme:*

$$e(x/m)y = \begin{cases} m & \text{je-li } y \equiv x \\ e(y) & \text{je-li } y \not\equiv x \end{cases}$$

splnitelná v $\mathcal{M}$

1. Říkáme, že A je splněna *v* $\mathcal{M}$ *při ohodnocení* e *a píšeme* $\mathcal{M} \models A[e]$ *jestliže (indukcí podle složitosti* A*):*

(a) A *je atomická,* $A \equiv p(t_1, \ldots, t_n)$, *kde* p *není rovnost, potom* $\mathcal{M} \models A[e]$, *jestliže* $(t_1[e], \ldots, t_n[e]) \in p_M$

(b) A *je atomická,* $A \equiv t_1 = t_2$ *a* $t_1[e] = t_2[e]$

(c) A *je tvaru* $\neg B$ *a* $\mathcal{M} \not\models B[e]$

(d) A *je tvaru* $B \to C$ *a* $\mathcal{M} \not\models B[e]$ *nebo* $\mathcal{M} \models C[e]$

(e) A *je tvaru* $\forall x : B$ *a* $\mathcal{M} \models B[e(x/m)]$ *pro každé* $m \in M$

(f) A *je tvaru* $\exists x : B$ *a* $\mathcal{M} \models B[e(x/m)]$ *pro nějaké* $m \in M$

pravdivá v $\mathcal{M}$

model

2. Říkáme, že formule A je pravdivá *v* $\mathcal{M}$ *a píšeme* $\mathcal{M} \models A$, *je-li* A *splněna v* $\mathcal{M}$ *při každém ohodnocení proměnných.* $\mathcal{M}$ *je modelem* A.

Z definice vyplývá, že je-li formule uzavřená, potom její splnění je pro všechna ohodnocení stejné. Stačí tedy ověřit zda je splněna, či nesplněna při jednom ohodnocení. Jinými slovy, je-li uzavřená formule splněna při alespoň jednom ohodnocení, je pravdivá.

Definice!

Definice 2.2.7 *Říkáme, že formule* A *je* validní (platná) *nebo* logicky pravdivá *a píšeme* $\models A$, *jestliže je pravdivá při každé interpretaci daného jazyka.*

Tedy $\models A$ *právě když* $\mathcal{M} \models A$ *pro každou interpretaci* $\mathcal{M}$.

Obdobně bychom pro formuli mohli definovat splnitelnost formule.

Definice 2.2.8 *Říkáme, že formule A je* splnitelná, *pokud existuje interpretace $\mathcal{M}$, ve které je pravdivá.* **Definice!**

Dále si zavedeme značení a pojem substituce, substituovat budeme termy za volné proměnné ve formuli.

Definice 2.2.9 *Jsou-li $x_1, \ldots, x_n$ různé proměnné formule A a $t_1, \ldots, t_n$ jsou termy, potom symbolicky* **Definice!**

$$A_{x_1,\ldots,x_n}[t_1, \ldots, t_n]$$

označíme výraz, který vznikne z A nahrazením každého volného výskytu proměnné x_i termem t_i, pro $1 \leq i \leq n$.

Substituce přitom musí být platná, tj. žádná volná proměnná v t_i, pro $1 \leq i \leq n$, se nesmí stát vázanou v $A_{x_1,\ldots,x_n}[t_1, \ldots, t_n]$.

Nyní se dostáváme ke klíčovým částem, kterými tato shrnující kapitola vyvrcholí, jen připomeňme, že pojem důkazu, důkazu z předpokladů a vět je v predikátové logice stejný jako ve výrokové logice.

Jen pro úplnost, definujme pojem uzávěru formule.

Definice 2.2.10 *Jsou-li $x_1, \ldots, x_n$ všechny proměnné s volným výskytem ve formuli A v nějakém pořadí, potom formuli* **Definice!**

$$\forall x_1 : \forall x_2 : \ldots \forall x_n : A$$

nazveme uzávěrem formule A.

I když, dle definice, může vzniknout mnoho uzávěrů, tak lze dokázat, že všechny uzávěry k dané formuli jsou ekvivalentní. Následuje věta o dedukci a konstantách, ze kterých vyvodíme zajímavé důsledky, na kterých je principiálně postaveno vyhodnocení Prologu (formální základ SLD rezoluce, není předmětem tohoto modulu).

Teorém 2.2.1 *Nechť T je množina formulí, A je uzavřená formule a B je libovolná formule. Potom $T \vdash A \to B$ právě když $T, A \vdash B$.*

Teorém 2.2.2 *Nechť T je množina formulí jazyka $\mathcal{L}$ a A je formule jazyka L. Nechť $x_1, \ldots, x_n$ jsou proměnné.*

Nechť jazyk $\mathcal{L}'$ vznikne rozšířením $\mathcal{L}$ o nové symboly $c_1, \ldots, c_n$ pro konstanty (nulární funktory). Potom platí

$$T \vdash_{\mathcal{L}'} A_{x_1,\ldots,x_n}[c_1, \ldots, c_n] \text{ právě když } T \vdash_{\mathcal{L}} A$$

Důsledkem je právě klíč k vyhodnocovacímu mechanismu v Prologu (všechny formule v Prologu jsou uzavřené — viz níže).

Důsledek 2.2.1 *Je-li A' uzávěr formule A a T je množina formulí, potom $T \vdash A$, právě když $T \cup \{\neg A'\}$ je sporná.*

Množinu T nazýváme teorií, či hypotézou. Formálněji:

Definice!

Definice 2.2.11 *Je-li $\mathcal{L}$ jazyk prvního řádu a T množina jeho formulí, říkáme, že T je* teorie prvního řádu *s jazykem $\mathcal{L}$.*
Formulím z množiny T říkáme speciální axiomy *teorie T.*

Predikátová logika je zvláštním případem teorie prvního řádu, která nemá žádné speciální axiomy.

Podobně jako pro formule, lze definovat pojem modelu i pro teorie.

Definice!

Definice 2.2.12 *Je-li T teorie s jazykem $\mathcal{L}$ a $\mathcal{M}$ je interpretace jazyka $\mathcal{L}$, říkáme, že $\mathcal{M}$ je* modelem teorie T *a píšeme $\mathcal{M} \models T$ jestliže každý speciální axiom teorie T, tedy každá formule z T je pravdivá v $\mathcal{M}$.*
Říkáme, že formule A je sémantickým důsledkem *teorie (množiny) T a píšeme $T \models A$, jestliže A je pravdivá v každém modelu teorie T.*

Oproti výrokové logice vidíme, že problém je v tom, že u predikátové logiky pracujeme s interpretacemi. Asi sami cítíme, že ne každá interpretace je modelem, což, bohužel přináší řadu problémů.

Zakončíme ekvivalentem věty 2.1.6 z části o výrokové logice, respektive částí této věty, nicméně nikoliv povzbudivým.

Churchova věta

Teorém 2.2.3 Churchova věta
Nechť $\mathcal{L}$ je jazyk prvního řádu bez rovnosti, který obsahuje alespoň dva binární predikáty, potom predikátová logika *s jazykem $\mathcal{L}$* je nerozhodnutelná.

Jelikož predikátová logika, tak jak jsme ji definovali a používáme ji, je bez rovnosti, tak toto platí i pro náš případ. Nicméně situace není zcela beznadějná (jinak by Prolog ani nepracoval), neboť:

Gödelova věta

Teorém 2.2.4 Gödelova věta o úplnosti
Každá logicky platná formule je dokazatelná (na úrovní predikátové logiky).

Pojmy k zapamatování
Klíčových pojmů této kapitoly je celá řada. Snad nejdůležitější je spornost/bezespornost, rozhodnutelnost, úplnost a splnitelnost/nesplnitelnost.

Závěr
Tato kapitola zrekapitulovala pojmy z výrokové a predikátové logiky, které významně ovlivňují chování a vlastnosti logických programovacích jazyků. Jejím cílem nebylo seznámit s těmito pojmy, ale spíše připomenout jejich existenci a obsah. Pokud jste tedy na konci této kapitoly pouze neoprášili své znalosti z formální logiky, tak je vhodné tuto mezeru dohnat studiem příslušné literatury, která je k tomu určena. V opačném případě tato kapitola svůj účel splnila a je možné začít s úvodem do Prologu.

Úlohy k procvičení:
Pro systém výrokové logiky ukažte, že spojky negace ($\neg$) a implikace ($\rightarrow$) pokrývají všechny logické spojky jazyka výrokové logiky.

Klíč k řešení úloh
Postupujte podle řešeného příkladu ve stati o výrokové logice.

Další zdroje
Zdroje k výrokové logice lze snadno najít na Internetu, tato stať například čerpala z:

- `http://www.phil.muni.cz/fil/logika/vl.php`
- `http://ktiml.mff.cuni.cz/downloads/` (VL0.pdf, PL0.pdf, ...)
- `http://en.wikipedia.org/wiki/Propositional_logic`
- `http://en.wikipedia.org/wiki/First_order_predicate_calculus`
- `http://www.macs.hw.ac.uk/~alison/ai3notes/section2_4_3.html`

Spoustu materiálů však lze zcela jistě nalézt i ve fakultních, univerzitních či vědeckých knihovnách.

Kapitola 3

Úvod do Prologu

Tato kapitola seznamuje s absolutními základy jazyka Prolog — jaké jsou základní datové typy, vybrané vestavěné predikáty, ukázky základních úloh řešitelných v Prologu. Je jakýmsi vstupem k Prologu.

Čas potřebný ke studiu: 8 hodin 45 minut.

Cíle kapitoly

Cílem kapitoly je naučit čtenáře vytvořit a spustit nejmenší programy v Prologu, práci s typickými představiteli prostředí pro vyhodnocení jazyka Prolog.
Kromě syntaxe jazyka, která je velmi jednoduchá, se jedná zejména o podání prvků způsobu zpracování programů a osvojení si základních zvyklostí a jisté minimální množiny vestavěných predikátů a datových typů, které jsou pro logické jazyky, zejména tedy pro Prolog, typické.

Průvodce studiem

Na odhalení a ovládnutí základních programovacích technik a praktik v jazyce Prolog je třeba se nejdříve seznámit se a osvojit si *úplné základy*, které hledají oporu v predikátové logice a jejich „školních“ úlohách, a v tom, *jakým způsobem Prolog pracuje*.
Ke studiu této kapitoly je naprosto nezbytné mít k dispozici počítač s instalovaným interpretem jazyka Prolog. Je možné mít i kompilátor, ale práce v něm, zejména pro začátečníky, je mnohem, ale mnohem náročnější. Systémů pro interpretaci programů v jazyce Prolog je na Internetu dostupných celá řada. V dalším však doporučujeme užití systému SWIProlog, který je dostupný jak pro operační systémy (OS) Windows, tak pro systémy odvozené od OS Unix. Další možností je potom třeba CiaoProlog, který však díky rozsáhlému systému knihoven již pro základní práci vyžaduje zvládnutí práce s moduly.

Obsah

Výklad

3.1 Úvod k logickému programování

Hlavní myšlenkou logického programování bylo a je využít počítač k vyvozování důsledků na základě deklarativního popisu, k automatizaci dokazovacího procesu. K tomu, aby se tato vize mohla uskutečnit je však třeba projít jistým transformačním procesem:

1. **reálný svět**, z něhož nás zajímá řešení jistého problému musíme omezit na
2. **zamýšlenou interpretaci**, kdy vybereme jen to, co je pro problém důležité a specifické, na základě čehož vytvoříme
3. **logický model**, který nese prvky hypotézy predikátové logiky a příslušné interpretace, u které hodláme ověřit, zda je modelem, a celé konečně transformujeme na
4. logický **program**, který již může být zpracován automatizovaně.

Programy v logických programovacích jazycích, logické programy, jsou jistou součástí predikátové logiky. Abychom získali výsledek, tak kromě programu potřebujeme znát/zadat ještě cíl/dotaz, který nás zajímá. Přesněji:

Definice 3.1.1 *Programy v logický jazycích sestávají ze dvou částí:* **Definice!**

- Vlastní program — *je tvořen* Hornovými klauzulemi*, které tvoří hypotetický základ pro důkaz.*
- Dotazy — *cíle, predikáty, které chceme za dané hypotézy ověřit, zda platí, či ne.*

Hornovy klauzule se mohou vyskytovat ve dvou tvarech, buďto se jedná o *fakta* nebo o plné klauzule s *hlavičkou* a *tělem*:

Definice 3.1.2 *Tvar* Hornových klauzulí *je z pohledu predikátové logiky tento:* **Definice!**

$$H \leftarrow B_1 \wedge \ldots \wedge B_n, n > 0$$

- H *je hlava, či hlavička*
- $B_1 \wedge \ldots \wedge B_n$ *je tělo tvořené* atomickými formulemi

$$H \leftarrow \blacksquare$$

- *H je faktem.*

Jelikož množina důsledků plynoucích z naší hypotézy je obecně nekonečná, tak si vybíráme pomocí dotazů — cílů.

Definice!

Definice 3.1.3 *Z hlediska predikátové logiky lze cíl zapsat takto:*

$$\square \leftarrow B_1 \wedge \ldots \wedge B_n, n > 0$$

- $B_1 \wedge \ldots \wedge B_n$ *je dotaz/cíl tvořený* atomickými formulemi

Jelikož v programovacích jazycích používáme pro zápis programů často znaky definované v ASCII, tak program i dotaz v prologu zapisujeme dle těchto syntaktických pravidel:

- klauzule: `h :- b1, b2, ... , bn.`
- fakta: `h.`
- dotazy, cíle jsou ve tvaru `b1, b2, ... , bn.`, kde levá implikace je doplněna samostatně interpretem Prologu, což reprezentuje výzva na příkazové řádce Prologu ve tvaru `?-`

Vidíme tak, že logická konjunkce je reprezentována čárkou ve větě a levá implikace dvojznakem minus a dvojtečka. Každá klauzule, fakt, či cíl jsou zakončeny tečkou.

Z formálního pohledu jsou taktéž všechny klauzule univerzálně kvantifikovány a proto je třeba všechny podklady, které máme z predikátové logiky na správný tvar převést. Na závěr si tedy vše demonstrujme na jednoduchém příkladu.

Řešený příklad

Zadání: V Prologu zapište, že Jan má jako děti Andulku a Tomáše, Tomáš má jako dítě Aničku a Jan je dítětem Marka.
Nejdříve v predikátové logice, poté v Prologu definujte vztah „je vnukem“.
Řešení: Pro vytvoření vztahů v rodině použijeme fakta, která je budou definovat:

```
ditetem(andulka, jan).
ditetem(tomas, jan).
ditetem(anicka, tomas).
ditetem(jan, marek).
```

Pro řešení druhého příkladu máme formalizovat známý fakt, že vnuk je dítě mých dětí:

$$\forall X \forall Y : \texttt{vnukem}(X, Y) \leftarrow (\exists Z : \texttt{ditetem}(X, Z) \wedge \texttt{ditetem}(Z, Y))$$

Tento zápis je sice pravdivý, ale v Prologu přímo nereprezentovatelný, proto jej upravíme, dle pravidel predikátové logiky, takto:

$$\forall X \forall Y \forall Z : \texttt{vnukem}(X, Y) \leftarrow \texttt{ditetem}(X, Z) \wedge \texttt{ditetem}(Z, Y)$$

Což v Prologu již podchytíme zcela jednoduše:

```
vnukem(X,Y) :- ditetem(X,Z), ditetem(Z,Y).
```

Když přihlédneme k našemu příkladu, tak by se mohlo zdát, alespoň vzdáleně, že máme popsány základní rodičovské vztahy a tedy je možné položit takovýto dotaz:

```
?- ditetem(marie, X).
```

který by měl odpovědět na otázku, kdo jsou rodiče Marie. Nicméně, odpovědět na tuto otázku nelze, neboť program neobsahuje data, ze kterých by to bylo možné odvodit. Takže odpověď bude anglické „no“ (ne) na znamení, že cíl nevyplývá z naší hypotézy. Prolog totiž je založen na *předpokladu uzavřeného světa*. Kromě toho v něm lze vyjádřit pouze pozitivní informace (nelze například vložit údaj, že Marie není Tomášovým dítětem).

uzavřený svět

pozitivní informace

3.2 Základní typy a vlastnosti

Historie

Počátkem 70. let 20. století se objevuje první implementace jazyka Prolog, napsaná v jazyce Fortran. Za tímto počinem stáli pánové G. Battani a H. Meloni z Marseille. I když to byl počin významný, tak výsledek nebyl oslnivý.

Až v druhé polovině 70. let 20. století se objevuje interpret spolu s kompilátorem pro DEC10, za který stál jako ústřední postava D.H.D. Warren. Tato implementace již dosahovala rychlosti srovnatelné s LISPovskými interprety/kompilátory.

Přehled typů

Jak napovídá první implementace, tak na základní atomické datové typy jazyk nebyl a není bohatý. Vždy můžeme spolehnout na celá čísla (klasický zápis v desítkové soustavě) a atomy (začínají malým písmenem, nebo jsou uzavřeny v apostrofech). Velmi často je možné pracovat se znaky, nebo znakovými řetězci, ale pravidlem to není a tak i manipulace s těmito typy je rozdílná, stejně tak, jako i notace

literálů (pokud máte Prolog, který tuto podporu má, tak nahlédněte do jeho manuálu). Podobně i desetinná čísla jsou spíše výjimkou.

SWIProlog všechny zmiňované základní typy podporuje.

seznamy

Typickým strukturovaným typem jazyka Prolog je seznam, který se velmi často zapisuje tak, že začátek a konec seznamu určují hranaté závorky a jednotlivé prvky jsou odděleny čárkou. Jelikož je Prolog

Prolog je netypovaný

jazyk netypovaný, tak je možné mít v Prologu seznamy jakožto heterogenní datovou strukturu.

Jinou strukturou jsou pojmenované n-tice — zápis shodný jako fakt, ale ne nutně se jedná o predikát, či funktor. Jelikož n-tice jsou z principu heterogenní, nejinak je tomu také v Prologu. Následující ukázka shrnuje všechny popsané typy v malé ukázce, pro zachování logiky příkladů však jde i do míst v Prologu, která jsou až na dalších stránkách.

```
hloubkaStromu(list,0).
hloubkaStromu(uzel(_,L,P),H) :-
  hloubkaStromu(L,LH), hloubkaStromu(P,PH),
  max(LH,PH,MAX),
  H is MAX+1.
delka([],0).
delka([_|T],D) :-
  delka(T,TD),
  D is TD+1.
```

Příklad užití by mohl být potom takovýto:

```
?- delka([1,2,3],X), delka([3,5,7),X).
?- delka([4,2],2]).
?- hloubkaStromu(
     uzel(2,uzel(1,list,list),
            uzel(4,uzel(3,list,list),list)),
     H).
```

Vyhodnocení

Ač je Prolog považován za jazyky deklarativní, což je správné, tak způsob, jakým určuje vyhodnocení je znám a má operační charak-

SLD rezoluce

ter. Vyhodnocení se totiž provádí na základě *SLD rezoluce* — podcíle jsou expandovány do hloubky. Z pohledu programátora to znamená, že je-li hledán nějaký podcíl cíle (nějaký atom), tak jsou hlavičky klauzulí/fakt vložených do programu prohledávány shora dolů, až je možné nějakou hlavičku s podcílem unifikovat (podobně probíhá prohledávání ve vestavěných predikátech). Jakmile proběhne unifikace (Robinsonův

unifikační algoritmus, bez kontroly výskytu), je tělo klauzule zpracováváno zleva doprava, s tím, že každý atom na pravé straně je opět vyhodnocován do hloubky. Pokud se jedná o fakt, nebo všechny atomy na pravé straně jsou postupně uspokojeny, tak *uspívá* i celý podcíl a následně celý cíl. Pokud nelze unifikovat při prohledávání hlaviček klauzulí, nebo není možné uspokojit všechny atomy (dílčí podcíle) v těle klauzule, tak dochází k *selhání*. To vyvolává proces navracení (anglicky backtracking), kdy se vyhodnocovací mechanismus snaží navrátit do místa prohledávání a znovu, pro právě selhávající (dílčí) podcíl, se snaží najít jinou hlavičku, kde by uspěl, pokud toto není možné, tak se pokouší stejným mechanismem dokud neuspěje, nebo do doby, než selže i hlavní cíl (výpis „no“ na příkazovém řádku).

uspění cíle

selhání

backtracking

Program 3.2.1 *Uvažme nyní tento program v Prologu:*

```
male(john).   male(paul).   male(bob).
male(kevin).
female(mary). female(jane). female(linda).
father(paul,bob). father(john,jane).
father(paul,kevin).
mother(mary,bob). mother(linda,jane).
mother(mary,kevin).
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
sibling(X,Y) :- parent(Z,X), parent(Z,Y), X\=Y.
```

a na příkazovém řádku tento dotaz:

```
?- sibling(bob,X).
```

Potom odpověď na tento dotaz bude probíhat dle postupu výše, což znamená toto:

1. Výběr aktuálního podcíle (dle pravidla zleva doprava — do hloubky), kterým je *sibling(bob,X)*
2. Vyhledání první hlavičky stejného jména (shora dolů) — tou je *sibling*(X_1, Y_2) (indexy jsou doplněny pro jednoznačnost — tzv. čerstvé proměnné)
3. Unifikace X_1 a *bob* dává jako výsledek substituci „dosaď boba za X_1“, což budeme značit jako *[bob/*X_1*]*, unifikace X a Y_1 dává substituci *[*X/Y_1*]*
4. Jelikož se nejedná o fakt, tak aplikujeme substituce do těla klauzule a postupujeme prohledávání do hloubky: *parent(*Z_1*,bob), parent(*Z_1*,X), bob\=X*

(a) Výběr aktuálního podcíle dle pravidel nám poskytuje nový aktuální podcíl *parent(Z_1,bob)*

(b) Vyhledání první hlavičky stejného jména (shora dolů) — tou je *parent(X_2, Y_2)*

(c) Unifikace, jejímž výsledkem je *[Z_1/X_2, bob/Y_2]* (zřetězený zápis), je úspěšná

(d) V těle klauzule dostáváme *father(Z_1, bob)* a prohledáváme do hloubky:

i. Aktuálním podcílem je *father(Z_1, bob)*

ii. Vyhledání první hlavičky stejného jména (shora dolů) — tou je *father(paul,bob)*

iii. Unifikací zjišťujeme, že bob a bob jsou identičtí a dále získáváme substituci *[paul/Z_1]*, kterou si budeme pamatovat

iv. Jelikož se jedná o fakt a unifikace proběhla úspěšně, tak tento dílčí podcíl je splněn, **uspívá**

(e) Uspěním předchozího podcíle, jelikož klauzule má na pravé straně jen jeden podcíl, **uspívá** i tento podcíl a aplikací substituce získáváme *parent(paul,bob)*; dílčí podcíl je taktéž splněn

5. Propagací nově získaných substitucí získáváme na této úrovni *parent(paul,bob), parent(paul,X), bob\=X*, přičemž první z dílčích podcílů byl již splněn.

(a) Výběr aktuálního podcíle dle pravidel nám poskytuje nový aktuální podcíl *parent(paul,X)*

(b) Vyhledání první hlavičky stejného jména (shora dolů) — tou je *parent(X_3, Y_3)*

(c) Unifikace, jejímž výsledkem je *[paul/X_3, X/Y_3]*, je úspěšná

(d) V těle klauzule dostáváme *father(paul, X)* a prohledáváme do hloubky:

i. Aktuálním podcílem je *father(paul, X)*

ii. Vyhledání první hlavičky stejného jména (shora dolů) — tou je *father(paul,bob)*

iii. Unifikací zjišťujeme, že paul a paul jsou identičtí a dále získáváme substituci *[bob/X]*, kterou si budeme pamatovat

iv. Jelikož se jedná o fakt a unifikace proběhla úspěšně, tak tento dílčí podcíl je splněn, **uspívá**

(e) Uspěním předchozího podcíle, jelikož klauzule má na pravé straně jen jeden podcíl, **uspívá** i tento podcíl a aplikací

substituce získáváme *parent(paul,bob)*; dílčí podcíl je taktéž splněn

6. Propagací nově získaných substitucí získáváme na této úrovni *parent(paul,bob), parent(paul,bob), bob\=bob*, přičemž i druhý z dílčích podcílů byl již splněn.

   (a) Výběr aktuálního podcíle dle pravidel nám poskytuje nový aktuální podcíl *bob\=bob*

   (b) Jedná se o vestavěný operátor/predikát pro test na nerovnost (neunifikovatelnost), který ovšem **selhává**, neboť bob je shodné s bob.

   (c) Je vyvoláno zpětné navracení (backtracking)

7. Návrat k předchozímu uspívajícímu podcíli nás dostává k predikátu *parent(paul,bob)*, skrze který budeme postupovat dále zpět

   (a) Predikát *parent(paul,bob)* uspěl na základě uspění všech svých podcílů (těla klauzule), proto zpětné navracení pokračuje dále, k predikátu father*(paul,bob)*

      i. Zde byla vytvořena substituce *[bob/X]* na základě unifikace s příslušným faktem, ta je zrušena a je obnoveno vyhledávání, kdy podcílem je znovu *father(paul, X)*.
      ii. Vyhledání další hlavičky stejného jména (shora dolů) — tou je *father(john,jane)*
      iii. Pokus o unifikaci **selhává** (nelze unifikovat paul a john), dochází k opětovnému vyvolání **navracení**
      iv. Vyhledání další hlavičky stejného jména (shora dolů) — tou je *father(paul,kevin)*
      v. Unifikací zjišťujeme, že paul a paul jsou identičtí a dále získáváme substituci *[kevin/X]*, kterou si budeme pamatovat
      vi. Jelikož se jedná o fakt a unifikace proběhla úspěšně, tak tento dílčí podcíl je splněn, **uspívá**

   (b) Uspěním předchozího podcíle, jelikož klauzule má na pravé straně jen jeden podcíl, znovu tedy **uspívá** i tento podcíl a aplikací substituce získáváme *parent(paul,kevin)*; dílčí podcíl je taktéž splněn (*znovusplnitelný predikát*) znovusplnitelnost

8. Propagací nově získaných substitucí získáváme na této úrovni *parent(paul,bob), parent(paul,kevin), bob\=kevin*, druhý z dílčích podcílů byl tak znovu splněn.

   (a) Výběr aktuálního podcíle dle pravidel nám poskytuje nový aktuální podcíl *bob\=kevin*

(b) Jedná se o vestavěný operátor/predikát pro test na nerovnost (neunifikovatelnost), který v tomto případě *uspívá*, neboť bob a kevin jsou rozdílní

9. Výpočet se zastavuje s následujícím výpisem (výpis platných substitucí na nejvyšší úrovni, kdy je očekávána vaše odpověď na výzvu — stisknutím klávesy Enter další výpočet zastavíte, pokud stisknete středník, tak se znovu vyvolá zpětné navracení):

```
X = kevin ?
```

Pojmy k zapamatování
Mezi klíčové pojmy této výkladové části patří zejména:

- Hornovy klauzule, fakt
- dotaz, cíl, podcíl
- SLD rezoluce — úspěch, selhání, neúspěch

Výklad

3.3 Vybrané vestavěné predikáty

Dále uvedeme některé vestavěné predikáty, se kterými je možné se setkat v Prologu. Jejich výčet se liší dle distribuce, ale přesto je značná část zachycena ve všech typech a některé z nich budou dále uvedeny. Pro konkrétní seznam predikátů té které distribuce Prologu je nutné si důkladně prostudovat manuál dodaný k ní. Zjistíte, že výčet je poměrně široký a v případě podpory modularizace prakticky nekonečný.

Mezi klíčové vlastnosti vestavěných predikátů patří:

- mimologický, operačně definovaný význam, případně zcela bez deklarativního významu — pouze vedlejší účinky (vstup/výstup, modifikace databáze, ...)
- obvykle nejsou znovusplnitelné
- nezbytné pro praktické programování

Aritmetické operace

Definice!

Definice 3.3.1 *Aritmetické výrazy se vyhodnocují pomocí vestavěného operátoru* **is**, *který se používá dle následujícího schématu:*

<proměnná> **is** <výraz>

Výraz je nejdříve vyhodnocen a poté unifikován s proměnnou vlevo od operátoru pro `X is X+1` vždy selže!

Testování typu dat

Definice 3.3.2 *Test na to, zda proměnná obsahuje/je navázána (celé) číslo:* **Definice!**

integer(<proměnná>)

Definice 3.3.3 *Test na to, zda proměnná obsahuje/je navázána na atom (základní, nestrukturovaná hodnota):* **Definice!**

atom(<proměnná>)

Definice 3.3.4 *Test na to, zda proměnná obsahuje/je navázána na atom, nebo číslo:* **Definice!**

atomic(<proměnná>)

Díky tomu, že Prolog je netypovaný jazyk, tak tyto predikáty dodávají možnost ověřit za běhu, co skutečně je obsaženo v dané proměnné. Dle distribuce Prologu je možné se setkat i s řadou dalších podobných predikátů.

Metalogické predikáty

Definice 3.3.5 *Test na to, zda proměnná je navázána na nějakou hodnotu, či ne. Uspěje, pokud zatím není vázaná:* **Definice!**

var(<proměnná>)

Definice 3.3.6 *Test na to, zda proměnná je navázána na nějakou hodnotu, či ne. Uspěje, pokud je vázaná:* **Definice!**

nonvar(<proměnná>)

Definice 3.3.7 *Test na to, zda proměnné obsahují totožné/identické objekty:* **Definice!**

<proměnná> == <proměnná>

Tyto predikáty dále rozšiřují dříve uvedenou sadu s tím, že posouvají úroveň testu na vyšší úroveň a můžeme ověřovat, zda proměnná vůbec nějakou hodnotu má, či zda jsou dva objekty identické.

Operace s klauzulemi

Prolog je praktiky jediný jazyk, který umožňuje změnu programu za jeho běhu. K tomu slouží sada predikátů, které umí odebírat, či přidávat klauzule/fakta z/do databáze. Tyto predikáty typicky neuspívají při navracení, ani se neinvertuje výsledek jejich operace.

Definice!

Definice 3.3.8 *Vložení klauzule, či faktu (přirozené, vynucené na začátek, vynucené na konec):*

assert(<klauzule>)
asserta(<klauzule>)
assertz(<klauzule>)

Definice!

Definice 3.3.9 *Výběr klauzule z databáze podle hlavičky:*

clause(<hlava>, <tělo>)

Definice!

Definice 3.3.10 *Odebrání klauzule z databáze:*

retract(<klauzule>)

schéma změny DB

Změna databáze může probíhat podle dvou schémat:

- Okamžitá změna (immediate update view) — je zastaralá a dle normy již nevyhovující
- Logická změna (logical update view) — je aktuální, dle normy, pracuje na základě tohoto principu
  - vytváří se generace databáze (s každou operací)
  - (pod)cíl uzavřený v intervalu vložení – zrušení příslušné klauzule s ní potom může pracovat

okamžitá změna

Příklad ukazuje, jak se chová systém při okamžité změně:

1. vložení faktu *a(1)* do databáze z příkazového řádku:

```
?- assert(a(1)).
yes
```

2. vyčtení obsahu faktu *a* z databáze a uložení modifikované hodnoty:

```
?- a(X), Y is X + 1, assert(a(Y)).
X = 1, Y = 2  ?  ;
X = 2, Y = 3  ?  ;
...
```

Vidíme, že při vyvolání navracení je ihned možné využít právě vloženou hodnotu faktu *a*.

Při užití systému s logickou změnou databáze by ve stejné situaci bylo chování takovéto (dle tohoto testu lze systémy jednoduše rozlišit): logická změna

1. vložení faktu *a(1)* do databáze z příkazového řádku:

   ```
   ?- assert(a(1)).
   yes
   ```

2. vyčtení obsahu faktu *a* z databáze a uložení modifikované hodnoty:

   ```
   ?- a(X), Y is X + 1, assert(a(Y)).
   X = 1, Y = 2  ?  ;
   no
   ```

3. vyčtení obsahu faktu *a* z databáze a uložení modifikované hodnoty:

   ```
   ?- a(X), Y is X + 1, assert(a(Y)).
   X = 1, Y = 2  ?  ;
   X = 2, Y = 3  ?  ;
   no
   ```

4. ...

Rozebírání struktury termů

Na to, abychom mohli s predikáty pracovat v době běhu programu zcela komfortně, tak slouží následující sada vestavěných predikátů. Jejich účelem je možnost skládat a vytvářet termy přímo za běhu aplikace, případně rozebírat strukturu existujících. Toto vše, ve spojení s předchozí sadou predikátů je jedinečný a velmi silný nástroj.

Definice 3.3.11 *Převodu termu na seznam a zpět — převod probíhá v tom směru, kde je volná proměnná, pokud jsou obě vázané, tak se „kontroluje“ shoda, jinak predikát selže:* **Definice!**

<proměnná/term> =.. <proměnná/seznam>

Definice!

Definice 3.3.12 *Zavolá predikát jako podcíl, jako by byl v daném místě zapsán, takto může být v proměnné:*

call(<proměnná/predikát>)

Tyto dva predikáty se často užívají spolu, například v následující ukázce kódu, kdy predikát *aplikuj* vyhodnotí predikát, jenž má význam funkce, vložený jako první parametr s tím, že mu předá dva argumenty, z nichž jeden je vlastní parametr a druhý reprezentuje výsledek:

```
aplikuj(Funkce,Parametr,Vysledek) :-
  Predikat =.. [Funkce,Parametr,Vysledek],
  call(Predikat).
inkrement(X,Y) :-
  Y is X+1.
main :-
  ...
  aplikuj(inkrement,3,V),
  ...
  /* užití V~*/
  ...
```

Vstup/výstup

Pro vstupní a výstupní operace slouží níže uvedené predikáty. Pro přesměrování vstupu a výstupy mezi soubory a konzolou, otevírání souborů, apod. slouží i další jmenované predikáty. I když mezi jednotlivými distribucemi Prologu panuje jistá shoda v jejich užití, tak na detailní práci konkrétní distribuce je třeba nahlédnout do příslušného manuálu.

Definice!

Definice 3.3.13 *Zápis na výstup:*

write(<prom./term>)
write(<prom./stream>, <prom./term>)

Definice!

Definice 3.3.14 *Čtení ze vstupu:*

read(<prom./term>)
read(<prom./stream>, <prom./term>)

Definice 3.3.15 *Zapsání konce řádku:* **Definice!**

nl
nl(<prom./stream>)

Definice 3.3.16 *Otevři soubor pro čtení:* **Definice!**

see(<prom./soubor>)

Definice 3.3.17 *Otevři soubor pro zápis (na konec):* **Definice!**

tell(<prom./soubor>)
append(<prom./soubor>)

Definice 3.3.18 *Nastav soubor na právě čtený:* **Definice!**

seeing(<prom./soubor>)

Definice 3.3.19 *Nastav soubor na právě zapisovaný:* **Definice!**

telling(<prom./soubor>)

Definice 3.3.20 *Zavři právě čtený vstup:* **Definice!**

seen

Definice 3.3.21 *Zavři právě zapisovaný výstup:* **Definice!**

told

Malý příklad užití ukazuje, jak se tyto predikáty typicky užívají. Predikát *consult* uspívá, pokud se mu podaří otevřít soubor předaný parametrem F a po načtení predikátů ze souboru, které jednak zobrazuje na standardní výstup a jednak ukládá do databáze. Po načtení, zobrazení a uložení prvního predikátu se testuje, zda to není predikát pro ukončení čtení ze souboru, pokud ne, dojde ke zpětnému navrácení až k predikátu (vestavěnému) *repeat*, který je vždy znovuuspívající. Teprve splnění ukončovací podmínky způsobí zavření vstupu pro čtení a splnění predikátu *consult*. Vykřičník v těle predikátu je operátor řezu (bude diskutován dále) a při navracení způsobí odříznutí navracení před místo jeho polohy, aby zamezil opakování díky predikátu *repeat*.

```
consult(F) :- see(F),
              repeat,
                read(X),
                nl, write(X),
                assert(X),
              X=end_of_file,   % Ukončovací podmínka
              !, seen.
```

Pomocné predikáty

Výčet predikátů, který následuje, doplňuje a uzavírá výčet nejdůležitějších vestavěných predikátů. Svým způsobem bychom asi podobné predikáty měli čekat, ale jejich explicitní výčet rozhodně není na škodu. V předchozích příkladech jsme se s některými setkali, i když v obměně.

Definice!

Definice 3.3.22 *Explicitní vynucení unifikace, test na unifikovatelnost (jinak probíhá automaticky při unifikace parametrů v době vyhledávání správného predikátu — viz program 3.2.1 s příkladem vyhodnocení v kapitole 3.2):*

<proměnná/term> = <proměnná/term>

Explicitní vynucení unifikace, test na neunifikovatelnost:

<proměnná/term> \= <proměnná/term>

Definice!

Definice 3.3.23 *Vždy uspívající predikát (neplatí pro navracení):*

true

Vždy selhávající predikát:

fail

Vždy uspívající a znovusplnitelný (i při navracení) predikát:

repeat

Definice!

Definice 3.3.24 *Test na neúspěch (nejedná se o negaci v logickém smyslu). Predikát uspívá, pokud podcíl v něm specifikovaný selhává:*

not(<proměnná/predikát>)

Nutnou a neodmyslitelnou součástí každého Prologu je vestavěný predikát pro natažení vašeho programu do interpretu. Výčtem uvedeme několik příkladů, které by bylo možné použít, vždy se však přesvědčte o chování své distribuce:

- **consult**(<atom_jména_souboru>)
- [<atom_jména_souboru>]
- **ensure_loaded**(<atom_jména_souboru>)

3.4 Unifikace v Prologu

Unifikace je v Prologu základní a jedinou operací, která řídí vyhodnocení a díky níž se dostáváme k výsledku. Probíhá při vyhodnocení SLD rezolucí, když zadáme cíl a potom u všech (dílčích) podcílů, nebo explicitně zadáním predikátu =, či jeho negované formy \=.

Unifikace probíhá podle Robinsonova unifikačního algoritmu (1965, nezávisle i Guard 1964), ale je možné ji provádět jakýmkoliv jiným stejným vlastností. Při unifikaci se však neprovádí *kontrola výskytu*. To znamená, že se neověřuje, že dva vstupující termy do unifikace neobsahují na různých úrovních stejnou proměnnou, což může vést na nekonečný výsledný term. Např.

kontrola výskytu

```
X = f(X)
```

je ukázkou takové unifikace, kdy na obou stranách je tatáž proměnná, ale na různých úrovních. Výsledkem by byl nekonečný term/predikát *f(f(f(f(. . . f(X) . . .))))*, což v praxi není možné, takže systém buďto skončí chybou okamžitě, nebo později, případně se dostane do nekonečné smyčky.

Unifikaci využíváme typicky v těchto případech:

- Přiřazení (navázání) hodnoty k nějaké proměnné:

  $$<hodnota> = <proměnná>$$

  Zápis lze uplatnit pochopitelně i v opačném pořadí.

- Test na rovnost (unifikovatelnost), i složitých struktur:

  $$<term_1> = <term_2>$$

- Selektor položek z datových struktur/seznamů:

$$<seznam> = [<hlavička\_seznamu> \mid <tělo\_seznamu>]$$
$$<term> = <jméno>(<prom._1>, \ldots, <prom._n>)$$

- Vytváření datových struktur/seznamů. Např:

$$<proměnná> = [1,2,3]$$
$$<proměnná> = f(1,a,X)$$

- Předávání parametrů hodnotou, kdy term je předán jako skutečný argument.
- Předávání parametrů odkazem, kdy je předána proměnná (volná, nenavázaná) jako skutečný argument.
- Sdílení proměnných, vytváření synonym. Např. současné uplatnění těchto bodů:
  - Volání, predikátu a, predikát jako podcíl v těle jiné klauzule: *a(P,Q)*
  - Hlavička klauzule, definice predikátu a: *a(X,X) :- ...*

Robinsonův unifikační algoritmus

nejobecnější unifikátor

Formální zápis Robinsonova algoritmu s kontrolou výskytu, která se však v Prologu nevyskytuje, následuje. Algoritmus hledá nejobecnější unifikátor (*mgu* — most general unifier), což je takový, že neexistuje obecnější. Unifikátor je přitom realizován substitucí.

Formálněji, jestliže je *mgu* nejobecnější unifikátor pro termy t_1 a t_2, potom jeho aplikací získáváme:

$$t_1 \xrightarrow{mgu} t \xleftarrow{mgu} t_2$$

a zároveň neexistuje jiný unifikátor *mgu'* a k němu neprázdný unifikátor *mgu"* nerealizující pouhé přejmenování takový, že by platilo:

$$t_1 \xrightarrow{mgu'} t' \xleftarrow{mgu'} t_2 \wedge t' \xrightarrow{mgu''} t$$

V Robinsonově algoritmu se také vyskytuje operace zřetězení substituce při vytváření *mgu*. Budeme jej značit $\circ$ a prakticky to představuje postupnou aplikaci jednotlivých unifikací tak, jak jdou v zřetězení za sebou.

podvýraz

Unifikační algoritmus pracuje pouze nad literály, ale lze ho rozšířit nad celé formule, nebo termy. *Podvýrazem* literálu φ budeme rozumět formule a termy, které se v rámci φ objevují jako jeho podvýrazy. Je-li

na i-té pozici literálu φ nějaký podvýraz, potom ho budeme značit $\varphi^{(i)}$, pokud na této pozici žádný podvýraz není, potom je $\varphi^{(i)}$ nedefinováno.

Nesouhlasným párem potom označujeme dvojici podvýrazů různých literálů takovou, že pokud φ a ψ jsou tyto literály a i je takové nejmenší i, že $\varphi^{(i)}$ a $\psi^{(i)}$ jsou definovány, tak platí, že $\varphi^{(i)} \neq \psi^{(i)}$. Například mějme: nesouhlasný pár

$$\varphi = P(g_1(c); f_1(a; g_1(x); g_2(a; g_1(b))))$$

$$\psi = P(g_1(c); f_1(a; g_1(x); g_2(f_2(x; y); z)))$$

potom nesouhlasný pár je $(a, f_2(x; y))$.

Algoritmus 3.4.1 *Robinsonův algoritmus* **Algoritmus!**

Vstup: Δ, *množina literálů*
Výstup: μ, mgu *nebo selhání/neúspěch*
$\mu = []$ (prázdná substituce)
dokud v $\mu(\Delta)$ *existuje nesouhlasný pár* {
  najdi první nesouhlasný pár p *v* $\mu(\Delta)$
  pokud v p *není žádná volná proměnná* {
    skončí selháním unifikace
  } *jinak* {
    nechť $p = (x, t)$, *kde* x *je proměnná*
    pokud se x *vyskytuje v* t { – kontrola výskytu!!
      skončí selháním unifikace
    } *jinak* {
      nastav $\mu = \mu \circ [t/x]$
    }
  }
}
vrať μ

Pojmy k zapamatování

V této výkladové části jsme se seznámili s vybranými vestavěnými predikáty, které patří mezi klíčové a proto je nutné si je zapamatovat a umět používat. Znát jejich vlastnosti: mimologický charakter, obvykle nejsou znovusplnitelné při navracení, atd.
Kromě toho je nutné si osvojit pojmy spojené s unifikací: substituce, *mgu*, Robinsonův unifikační algoritmus.

Závěr

Tato kapitola si kladla za cíl uvést do jazyka logického programování, respektive Prologu. Nyní byste měli vědět, co vás po spuštění Prologu čeká, jakým způsobem se tvoří program Prologu, jak se zadávají cíle, jak obdržíte výsledek a zejména to, jakým způsobem Prolog pracuje při vyhodnocení cíle. Měli byste si také poradit s velmi jednoduchými úlohami.

Úlohy k procvičení:

1. Pro řešený program 3.2.1 s příkladem vyhodnocení v kapitole 3.2 naznačte, co by se dělo, pokud by na výslednou výzvy někdo odpověděl zadáním středníku. Toto chování zdůvodněte a detailně popište.
2. Nalezněte *mgu* (existuje-li) pro tyto dvojice literálů, zapište i výsledek:
   - $$\varphi = P(g_1(c); f_1(a; g_1(x); g_2(a; g_1(b))))$$
     $$\psi = P(g_1(c); f_1(a; g_1(x); g_2(f_2(x; y); z))$$
   - $$\varphi = P(g_1(c); f_1(a; g_1(x); g_2(a; g_1(b))))$$
     $$\psi = P(g_1(c); f_1(a; g_1(x); g_2(f_2(x; y); a))$$
   - $$\varphi = P(g_1(c); b; g_2(a; g_1(b)))$$
     $$\psi = P(g_1(c); f_1(a; g_1(x); g_2(f_2(x; y); z))$$
3. Rozšiřte a upravte program 3.2.1 tak, aby nevykazoval chybné chování, které jste demonstrovali v první úloze, dále definujte predikát pro bratrance a sestřenici, rozšiřte zadání dat o lidech tak, abyste mohli tyto predikáty ověřit v praxi.

Klíč k řešení úloh

1. Uvědomte si, co zadání středníku vyvolá, zbytek je v podstatě v řešeném příkladě již uveden.
2. Nastudujte Robinsonův unifikační algoritmus. Na pomoc si přizvěte Prolog.

3. Změňte definici tak, aby u dětí stejnými rodiči nerozlišoval predikát parent vazbu přes matku i otce. Pro definici sestřenice/bratrance si definujte pojem strýce_či_tety s vlastnostmi, jako jsme právě nově definovali pro predikát parent.

Další zdroje

Řadu informací podaných v této kapitole najdete v literatuře citované níže, nebo přímo v nápovědě, či doprovodné dokumentaci, či manuálech, které jsou distribuovány spolu s nějakou distribucí Prologu (SWIProlog, CiaoProlog, ...).

- Robinson, J. A.: A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12, 23–41, 1965.
- Nilsson, U., Maluszynski, J.: *Logic, Programming and Prolog (2ed)*, John Wiley & Sons Ltd., 1995.
- Bieliková, M., Návrat, P.: *Funkcionálne a logické programovanie*, Vydavateĺstvo STU, Vazovova 5, Bratislava, 2000.

Kapitola 4

Základní obraty v Prologu

Tato kapitola podává jisté základy programování a programovacích technik v Prologu. Výklad je prováděn na několika vybraných příkladech, kdy je uveden příklad a krátký popis chování.

Čas potřebný ke studiu: 10 hodin 30 minut.

Cíle kapitoly

Cílem kapitoly je zvládnout základní práci se seznamy v Prologu a ukázka využití operátoru řezu.

Průvodce studiem

Na odhalení a ovládnutí typických programovacích technik a praktik v jazyce Prolog je třeba se nejdříve seznámit se a osvojit si základy, které hledají oporu v typických elementárních úlohách.
Kapitola je obsahově útlá, ale vyžaduje a předpokládá práci u počítače a samostatné procvičování i na dalších úlohách, než jen předkládaných.
Ke studiu této kapitoly je naprosto nezbytné mít k dispozici počítač s instalovaným interpretem jazyka Prolog. Je možné mít i kompilátor, ale práce v něm, zejména pro začátečníky, je mnohem, ale mnohem náročnější. Systémů pro interpretaci programů v jazyce Prolog je na Internetu dostupných celá řada. V dalším však doporučujeme užití systému SWIProlog, který je dostupný jak pro operační systémy (OS) Windows, tak pro systémy odvozené od OS Unix. Další možností je potom třeba CiaoProlog, který však díky rozsáhlému systému knihoven již pro základní práci vyžaduje zvládnutí práce s moduly.

Obsah

Výklad

4.1 Seznamy v Prologu

Zápis literálů a vzorů

Seznam v Prologu je základní datová struktura a základní notace využívá dva konstruktory: konstruktory seznamu

- prázdný seznam je konstruován jako: []
- pro neprázdný seznam, přesněji hlavičku seznamu a zbytek seznamu, se užívá tečka

Řešený příklad

Zadání: Zapište elementárními konstruktory seznam s prvky 1, 2 a 3, dále seznam začínající číslem 5 s tělem v proměnné X, konečně seznam s prvním prvkem v proměnné Y, dále s hodnotou 6 a s tělem v proměnné Z.
Řešení:

- `.(1, .(2, .(3, [])))`
- `.(5, X)`
- `.(Y, .(6, Z))`

Tento zápis se však často neuplatňuje a používá se zápis mnohem čitelnější. Dokonce některé distribuce Prologu tento zápis již vůbec nepodporují. Nově, častěji se seznam uzavírá do hranatých závorek vždy, elementy se oddělují čárkou a zbytek seznamu rourou.

Řešený příklad

Zadání: Zapište běžnými konstruktory seznam s prvky 1, 2 a 3, dále seznam začínající číslem 5 s tělem v proměnné X, konečně seznam s prvním prvkem v proměnné Y, dále s hodnotou 6 a s tělem v proměnné Z.
Řešení:

- `[1, 2, 3]`
- `[5 | X]`
- `[Y, 6 | Z]`

Pokud seznam zapíšeme nějakou povolenou formou do hlavičky klauzule, tak z něj vytvoříme vzor, který je také předmětem unifikace, v rámci celé své struktury. V hlavičkách klauzulí se tak kromě

proměnných a atomů mohou vyskytovat i struktury (termy) s různou úrovní pojmenování a konkretizace (prázdný seznam, neprázdný seznam, seznam délky 3, term se jménem f a třemi různými parametry — ne jejich místech budou proměnné, apod.). Postupně na tyto konstrukce narazíme dále v textu. Existuje také zástupná proměnná, která se uvádí na místo, kde bychom uvedli normálně proměnnou, pokud bychom dále s hodnotou, na kterou se naváže, chtěli pracovat, ale my zrovna teď o ni nestojíme — je to podtržítko _. V některých mutacích Prologu dokonce proměnnou, kterou neodkazujeme je vhodné za podtržítko zaměnit, aby nám interpret nehlásil varování.

Elementární operace nad seznamy

V této části uvedeme několik krátkých definic predikátů pro základní operace nad seznamy.

member

Program 4.1.1 *Predikát je splněn, pokud je prvek* X *obsažen v seznamu* L — member(X,L).

```
member(X, [X|_]).
member(X, [_|Tail]) :-
  member(X,Tail).
```

Funkce je taková, že pro prázdný seznam není definována žádná klauzule, protože nic nemůže být obsaženo v prázdném seznamu. První řádek reprezentuje fakt, že je-li daný prvek na prvním místě seznamu (vazba přes jméno, unifikací), je prvkem tohoto seznamu. Jinak dochází k unifikaci dle hlavičky na druhém řádku, kdy predikát uspívá, pokud se prvek X vyskytuje ve zbytku seznamu.

append

Program 4.1.2 *Predikát je splněn, pokud spojením seznamů* L1 *a* L2 *vzniká seznam* L12 — append(L1,L2,L12).

```
append([], L, L).
append([H|T1], L2, [H|T2]) :-
  append(T1,L2,T2).
```

První řádek definuje fakt, že je-li první seznam prázdný, tak výsledný a druhý se musejí shodovat. Druhý řádek říká, že že první prvek výsledného seznamu musí být prvním prvkem prvního seznamu a dále, na základě splnění podcíle, spojením zbytku prvního seznamu s druhým vzniká zbytek výsledného seznamu.

last

Program 4.1.3 *Predikát je splněn, pokud první parametr* X *je unifikovatelný s posledním prvkem seznamu* L — last(X,L).

```
last(X, [X]).
last(X, [_|Tail]) :-
  last(X, Tail).
```

První řádek vyjadřuje fakt, že prvek je posledním prvkem seznamu, je-li jednoprvkový. Pokud seznam není jednoprvkový, tak X je jeho posledním prvkem, pokud je posledním prvkem jeho zbytku. Je vidět, že prázdný seznam nemá poslední prvek a proto nemá žádný předpis/klauzuli v definici.

Predikáty vyššího řádu nad seznamy

V této části si ukážeme implementaci některých manipulátorů seznamů, které již nemají jako parametry pouze seznamy, či jiné hodnoty, ale mají jako parametry zcela nezbytně další predikáty.

Program 4.1.4 *Predikát je splněn, pokud první parametr* F *je dvouparametrický predikát, druhý parametr* L *je seznam hodnot, které jsou prvními parametry predikátu* F *a třetí parametr je seznam* LF*, který obsahuje hodnoty, které jsou druhými parametry predikátu* F*, pokud by byly po řadě aplikován vždy po dvojici na hodnoty obou seznamů* — map(F,L,LF). map

```
map(_, [], []).
map(F, [H|T], [NH|NT]) :-
  P =.. [F,H,NH], call(P),
  map(F, T, NT).
```

První řádek definuje fakt, že pro prázdné seznamy není co provádět. Pokud jsou seznamy neprázdné, tak aplikací predikátu na první prvky obou seznamů (řádek 3) musíme dostat uspívající podcíl a totéž musí platit pro zbytky obou seznamů. Příklad užití, uvažme tuto definici:

```
inc(X,Y) :- var(Y), Y is X+1.
inc(X,Y) :- nonvar(Y), Z~is Y-1, Z=X.
```

Potom může následovat tento scénář užití:

```
?- map(inc,[1,2,3],X).
X = [2,3,4] ?
?- map(inc,X,[2,3,4]).
X = [1,2,3] ?
```

Program 4.1.5 *Predikát je splněn, je-li seznam* L *prázdný, tak potom* B=R*, jinak, je-li seznam* L *složen z prvků* A1, ..., AN*, potom musí platit, že* F *je tříparametrický predikát a je realizována funkce (pro její zápis si predikát* F *představme jako funkci se dvěma parametry a třetí jako výsledek)* R = F(A1,F(A2, ...,F(AN,B) ...)) — foldr(F,B,L,R). foldr

```
foldr(_, B, [], B).
foldr(F, B, [H|T], BB) :-
  foldr(F, B, T, BT),
  P =.. [F,H,BT,BB], call(P).
```

foldl

Program 4.1.6 *Predikát je splněn, je-li seznam* L *prázdný, tak potom* B=R*, jinak, je-li seznam* L *složen z prvků* A1, ..., AN*, potom musí platit, že* F *je tříparametrický predikát a je realizována funkce (pro její zápis si predikát* F *představme jako funkci se dvěma parametry a třetí jako výsledek)* R = F(...(F(F(B,A1),A2), ...), AN) — foldl(F,B,L,R).

```
foldl(_, A, [], A).
foldl(F, A, [H|T], AA) :-
  P =.. [F,A,H,AT], call(P),
  foldl(F, AT, T, AA).
```

Užití těchto predikátu demonstruje následující fragment kódu, respektive v něm predikáty *sum* a *rev*:

```
add(X,Y,Z) :-
  ZZ is Y+X,
  ZZ=Z.
conS(T,H,[H|T]).

sum(L, S) :-
  foldr(add, 0, L, S).
rev(L, RL) :-
  foldl(conS, [], L, RL).
```

Predikát *sum* realizuje součet prvků seznamu, predikát *rev* realizuje vysoce sofistikovaným obrácení pořadí prvků seznamu L.

Další typy seznamů

Prolog zná a umožňuje, dle distribuce použít ještě několik typů seznamů, než ty, se kterými jsme doposud pracovali. Jejich detailní znalost není nezbytná, ale typově je vhodné se s nimi seznámit. Jsou totiž založeny na jiné metodě než přístup k prvnímu prvku a zbytku seznamu.

Diferenční seznamy Ideu diferenčních seznamů ukazují tyto dva příklady:

$$X - X \approx []$$

$$[1,2,3|X] - X \approx [1,2,3]$$

Operace s těmito seznamy jsou často destruktivní, což u klasických seznamů není pravda. Tím pádem často operace, které lze využít s různou konkretizací parametrů nefungují. Např. append:

```
dappend(A-ZA, ZA-ZB, A-ZB)
```

Funkcionální seznamy Ideu funkcionálních seznamů ukazují tyto dva příklady:

$$z \setminus z \approx []$$
$$z \setminus [1,2,3|z] \approx [1,2,3]$$

Operace s těmito seznamy nejsou destruktivní a lze je proto používat tak, jako seznamy klasické, s různou konkretizací parametrů. Např. append:

```
fappend(L, R, z\(L (R z)))
```

Pojmy k zapamatování
Mezi klíčové pojmy této výkladové části především náleží:

- diferenční seznamy
- funkcionální seznamy

Důležité jsou však získané dovednosti, mezi které by mělo patřit zvládnutí práce se seznamy.

Výklad

4.2 Operátor řezu

Úvodní definice

Operátor řezu (vykřičník, !) je právě jednou uspívající operátor. Pokud dojde k zpětnému navracení v programu, tak dostane-li se navracení až k operátoru řezu, tak selže celý podcíl a všechny bodu znovuuspění, které se mezi operátorem řezu a hlavičkou klauzule vyskytly, jsou zapomenuty. Názorně:

$$foo \quad :- \quad \underbrace{a,\ b,\ c}_{navracení},\ !,\ \underbrace{d,\ e,\ f}_{navracení}$$

Dokud není splněno c, tak dochází k navracení až po a (které musí uspívat, jinak selže celé foo). Jakmile je ale splněno c, může navracení probíhat jen mezi d, e a f. Pokud však d selže, selže celý podcíl foo a body návratu až po foo včetně, jsou zapomenuty.

Použití I

Chceme Prologu sdělit, že již byla nalezena správná varianta podcíle a všechny ostatní již nejsou relevantní. Např.:

```
sum_to(1,1) :- !.
sum_to(N,Res) :-
  N1 is N-1,
  sum_to(N1,Res1),
  Res is Res1+N.
```

Pokud bychom na řádku 1 neměli operátor řezu, při navracení by byla využita další definice predikátu *sum_to*, na řádku 2, čímž by se výpočet dostal na nesprávnou větev (podtečení).

Použití II

Chceme, aby cíl ihned selhal. Bylo dosaženo stavu, kdy další prohledávání nevede k řešení. Potom operátor řezu řetězíme s predikátem *fail*. Ukážeme si na definici operátoru *not*, který je sice vestavěn, ale v nejhorším je možné si pomoci právě touto definicí:

```
not(P) :- call(P), !, fail.
not(P).
```

Použití III

Chceme ukončit generování alternativních řešení. Existují např. permutace existujícího řešení, které by sice byly objeveny, ale pro naši úlohu nemají význam, nebo jiná řešení nejsou správná. Např.:

```
is_integer(0).
is_integer(X) :-
  is_integer(Y),
  X is Y+1.
divide(N1, N2, Result) :-
  is_integer(Result),
  Product1 is Result*N2,
  Product2 is (Result+1)*N2,
  Product1 =< N1, Product2 > N1, !.
```

V tomto případě (celočíselné dělení *N1 / N2 = Result*), kdybychom nepoužili operátor řezu, tak při navracení se dostane program do nekonečné smyčky, která skončí až chybou v aritmetice, nedostatkem paměti, či podobně.

Nesprávné užití

Operátor řezu však může zamezit nalezení všech správných řešení i tam, kde to vyžadujeme. Dokonce je možné obdržet i řešení nesprávná. Kromě toho, s operátorem řezu mají predikáty silně vymezený charakter vzhledem k tomu, u kterých parametrů je možné mít volnou proměnnou a kde je nutné mít již konkretizovanou hodnotu. Uvažme tento program:

```
num_of_parents(adam, 0) :- !.
num_of_parents(eve, 0) :- !.
num_of_parents(X, 2).
```

Na první pohled je, dle jisté hypotézy, vše v pořádku, neboť podle ní kromě Adama a Evy mají všichni dva rodiče. Co ale tento cíl?

```
?- num_of_parents(eve, 2).
```

Odpověď bude *yes*. Je to ale to, co bychom chtěli, či očekávali? Zcela určitě ne!

Seznamy a operátor řezu

V následujícím oddílu ukážeme několik predikátů, které pracují se seznamy a bez operátoru řezu by nedávali korektní výsledek.

Program 4.2.1 *Predikát je splněn, pokud po odstranění parametru* X *ze seznamu* L *ve všech místech výskytu a při zachování pořadí ostatních prvků seznamu* L *získáme seznam* LD — delete(X,L,LD). delete

```
delete(_, [], []).
delete(X, [X|L], M) :-
  !,
  delete(X, L, M).
delete(X, [Y|L1], [Y|L2]) :-
  delete(X, L1, L2).
```

Pokud bychom operátor řezu nepoužili, docházelo by k tomu, že při navracení bychom postupně získávali seznamy, kde nejsou všechny výskyty X zcela odstraněny, což je jistě nesprávné chování.

Program 4.2.2 *Predikát je splněn, pokud uvážíme, že parametry* L1 *a* L2 *splňují vlastnosti množiny a seznam* LI *by vznikl průnikem takových množin, nicméně se zachováním pořadí shodných prvků dle seznamu* L1 — intersection(L1,L2,LI). intersection

```
intersection([], X, []).
intersection([X|R], Y, [X|Z]) :-
  member(X,Y),
  !,
  intersection(R,Y, Z).
intersection([X|R], Y, Z) :-
  intersection(R, Y, Z).
```

Při absenci operátoru řezu bychom v prvním případě získali sice správné řešení, ale při navracení by výsledná množina postupně ztrácela své členy, až by byla úplně prázdná, což opět není správné řešení.

range

Program 4.2.3 *Predikát je splněn, pokud jsou první dva parametry identické a třetí je jednoprvkový seznam právě s takovou hodnotou, nebo jsou první dva parametry celočíselné, první je menší jak druhý a poslední je seznam čísel (celých) od* F *do* T — range(F,T,L).

```
range(S, S, [S]) :-
  !.
range(S, E, [S|T]) :-
  S~< E, SS is S+1,
  range(SS, E, T),
  !.
range(_, _, []).
```

Predikát slouží jako generátor posloupností, např.:

```
?- range(1, 10, L).
L = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] ?
```

Pokud by chyběly operátory řezu, tak je predikát zcela nefunkční — chyběla by podmínka zastavení celého rekurzivního výpočtu, případně by k zastavení došlo na nesprávném místě (poslední řádek). Celý predikát by tak pozbyl smysl.

filter

Program 4.2.4 *Predikát je splněn, pokud první parametr,* P*, predikát s jediným parametrem, druhý parametr je seznam a třetí je seznam, který obsahuje pouze ty prvky seznamu* L*, které splňují predikát* P*, při zachování pořadí* — filter(P,L,LF).

```
filter(_, [], []).
filter(P, [H|T], [H|TT]) :-
  PP =.. [P,H], call(PP),
  !,
  filter(P, T, TT).
filter(P, [_|T], TT) :-
  filter(P, T, TT).
```

Uvažme třeba ještě tento predikát, který je splněn, pokud má jako argument liché číslo:

```
odd(X) :-
  Y is X // 2,
  YY is Y * 2,
  YY \= X.
```

Potom může využít predikát *filter* např. takto:

```
?- filter(odd, [1, 2, 3, 4, 5, 6], L).
L = [1, 3, 5] ?
```

Kdybychom však operátor řezu nepoužili, tak při navracení bychom postupně v dalších variantách dostávali seznam s menším a menším počtem správných prvků, což není správné chování.

Program 4.2.5 *Predikát je splněn, pokud parametry* L *a* LN *jsou prázdné seznamy, nebo je první z nich číslo,* N*, druhý seznam a třetí je seznam, který ze seznamu* L *obsahuje prvních* N *členů (pokud jich tolik nemá, tak méně)* — take(N,L,LN). take

```
take(_, [], []) :-
  !.
take(N, [H|T], [H|TT]) :-
  N > 0,
  !,
  NN is N-1,
  take(NN, T, TT).
take(N, [_|_], []) :-
  N =< 0.
```

Užití je potom například takovéto:

```
?- take(5, [1, 2, 3, 4, 5, 6, 7, 8], X).
X = [1, 2, 3, 4, 5] ?
```

Odstranění operátoru řezu by zcela změnilo význam predikátu — měli bychom vždy dále menší počet prvků v různých kombinacích.

Pojmy k zapamatování
V této výkladové části se vyskytuje jediný pojem — operátor řezu. Mnohem důležitější je ale zvládnutí práce s ním tak, aby byly využity jeho výhody a nedocházelo k jeho špatnému nasazení.

Závěr

V této kapitole jsme se soustředili na práci se seznamy v predikátech v různých drobných příkladech. Kromě toho jsme definovali vlastnosti operátoru řezu a demonstrovali jeho užití jak v typických situacích, tak ve spojení s predikáty na manipulaci seznamů.
Na konci této kapitoly byste tedy měli mít již značné dovednosti v programování v jazyce Prolog, zejména co se týká zpracování a práce se seznamy a dále byste měli umět využívat vlastností operátoru řezu. Pokud máte stále pochybnosti, že zvládáte práci se seznamy, nebo s operátorem řezu, projděte si znovu příklady v této kapitole, nebo se podívejte po internetu, či do příkladů dodaných k vaší distribuci Prologu.

Úlohy k procvičení:

1. Predikáty definované v kapitole prověřte i z hlediska funkce při vložení různě konkretizovaných argumentů. Např. *append([1,2],[3,4],X)*, *append([1,2],X,[1,2,3,4])*, *append(X,[3,4],[1,2,3,4])*, *append(X,Y,[1,2,3,4])*, atd.

2. Implementujte tyto predikáty:

   - permutation(L1, L2) — L2 je permutací L1
   - reverse(L1, L2) — L2 je L1 v opačném pořadí (pro definici užijte pouze predikát append a elementární konstrukce jazyka)
   - sublists(XS, XSS) — XSS je seznam všech podseznamů XS

3. Implementujte tyto predikáty s využitím operátoru řezu:

   - remove(X,L,LR) — odstraní z L první výskyt X, výsledek je shodný s LR
   - union(L1,L2,LU) — seznamy L1 a L2 reprezentují množinu, LU je sjednocení obou množin
   - dropWhile(P,L,LP) — P je predikát s jediným argumentem, L je seznam a LP je seznam, který obsahuje první prvky seznamu L, pro které uspívá predikát P
   - sort(L,LS) — L je seznam čísel a LS je vzestupně seřazený seznam L

Klíč k řešení úloh

Pro řešení úloh neexistuje žádné přímé doporučení. Pokud vám jednotlivé úkoly dělají stále potíže, zkuste implementaci v programovacím jazyku, který je vám blízký. Potom zkuste takovou implementaci převést na implementaci v Prologu (zvláště vhodný je jiný druh deklarativního programovacího jazyka).

Další zdroje

Řadu informací k tématu podaného v této kapitole najdete v literatuře citované níže. Jelikož se však tento modul zabývá především programováním, tak nejlepších výsledků dosáhnete tím, že si sednete k počítači a budete zkoušet a programovat a zkoušet a programovat a tak dále a tak dále.

- Nilsson, U., Maluszynski, J.: *Logic, Programming and Prolog (2ed)*, John Wiley & Sons Ltd., 1995.
- Bieliková, M., Návrat, P.: *Funkcionálne a logické programovanie*, VydavateĺStvo STU, Vazovova 5, Bratislava, 2000.

Kapitola 5

Praktiky v Prologu

Tato kapitola prezentuje další vybrané techniky programování v jazyku Prolog — práce s vlastními datovými strukturami, definice vlastních operátorů, využití změny databáze a dynamických klauzulí. Výklad je vždy prováděn na několika vybraných příkladech, kdy je uveden příklad a krátký popis chování.

Čas potřebný ke studiu: 8 hodin 30 minut.

Cíle kapitoly

Cílem kapitoly je kompletní zvládnutí práce s daty v Prologu, neboť práce se seznamy již byla zvládnuta, tak zbývá doplnit ostatní možnosti tvorby a manipulace datových struktur. Důležitým prvkem výkladu je také dynamická tvorba dat za běhu programu.

Průvodce studiem

Rozšíření znalostí a dovedností práce v Prologu znamená, kromě jiného, plně zvládnout práci s datovými strukturami a zejména jejich obměnu za běhu programu. Svým obsahem navazuje na kapitolu předchozí.

Kapitola není obsahově rozsáhlá, ale právě proto vyžaduje a předpokládá práci u počítače a samostatné procvičování i na dalších úlohách, než jen předkládaných.

Ke studiu této kapitoly je naprosto nezbytné mít k dispozici počítač s instalovaným interpretem jazyka Prolog. Je možné mít i kompilátor, ale práce v něm, zejména pro začátečníky, je mnohem, ale mnohem náročnější. Systémů pro interpretaci programů v jazyce Prolog je na Internetu dostupných celá řada. V dalším však doporučujeme užití systému SWIProlog, který je dostupný jak pro operační systémy (OS) Windows, tak pro systémy odvozené od OS Unix. Další možností je potom třeba CiaoProlog, který však díky rozsáhlému systému knihoven již pro základní práci vyžaduje zvládnutí práce s moduly.

Obsah

Výklad

5.1 Datové struktury

Pojmenované n-tice

Z hlediska datových struktur se na dále nestrukturovaný term můžeme dívat jako na pojmenované n-tice. Velmi často s nimi pracujeme jako s fakty (viz např. program 3.2.1 v kapitole 3) a opravdu je dále nestrukturujeme. Jak se ukáže dále, tak jejich použití je daleko širší.

Můžeme je totiž manipulovat přímo a předávat do predikátů/klauzulí jako parametry, kde jsou v době unifikace rozpoznány a správně unifikovány (viz. algoritmus 3.4.1, někdy také hovoříme o rozpoznávání vzorů — pattern matching). Uvažme tuto jednoduchou definici:

```
isRGB(color(X)) :-
  member(X,[red,green,blue]).
```

Tu lze potom použít např. tímto způsobem:

```
?- isRGB(color(brown)).
no
?- isRGB(color(red)).
yes
```

Počet členů n-tice přitom není nijak zvláště omezen a i hloubka struktury není omezena jinak než praktickými možnostmi počítače.

Díky tomu je možné takto budovat i rekurzivní datové struktury, jako např. stromy, či další grafové struktury. Díky tomu, že vliv typů v Prologu je malý, tak lze struktury libovolně kombinovat i vnořovat a volně do nich začlenit i Prologu vlastní seznamy.

Vyhledávací stromy

Stromové datové struktury v různých obměnách se vyskytují prakticky ve všech odvětvích informatiky. V tomto oddílu si ukážeme, jak lze např. vyhledávací stromy definovat v Prologu a jak je možné s nimi manipulovat. Klíčem v tomto stromu bude celé číslo, jako data je možné použít prakticky cokoliv. Strom bude binární a data budou pouze ve vnitřních uzlech stromu, listy zůstanou prázdné. Následující program ukazuje predikáty pro definici/unifikaci s prázdným stromem a dále predikát, co přidává do stromu další uzel s tím, že pokud je hodnota s daným klíčem již zastoupena, zůstává strom nezměněn.

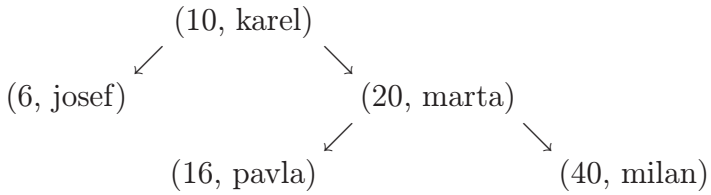
empty_tree, add2tree

Program 5.1.1 *Predikát* empty_tree *je splněn tehdy, pokud má jako parametr prázdný strom, neboli list, nebo je s ním unifikovatelný. Predikát* add2tree *je splněn tehdy, pokud by* Data *s daným* Klíčem *po vložení do* Původního *stromu vytvořila strom* Nový*, pokud již data takového klíče jsou zastoupena,* Nový *strom zůstává nezměněn —* empty_tree(T), add2tree(Klíč,Data,Původní,Nový)*.*

```
empty_tree(leaf).

add2tree(K, V, leaf, node(K,V,leaf,leaf)).
add2tree(K, _, node(K,V,X,Y), node(K,V,X,Y)) :-
  !.
add2tree(Kn, Vn, node(K,V,L,R), node(K,V,LL,R)) :-
  Kn < K, !,
  add2tree(Kn, Vn, L, LL).
add2tree(Kn, Vn, node(K,V,L,R), node(K,V,L,RR)) :-
  add2tree(Kn, Vn, R, RR).
```

Vidíme, že je opět využit operátor řezu a vazby mezi parametry pomocí unifikace. Takto budovaný strom je typický strom, kde od daného uzlu vlevo jsou data s menším klíčem a od daného uzlu vpravo data s větším klíčem, než je v daném uzlu. Je možné vytvořit i literál pro nějaký, zřejmě malý, strom. Uvažme tento strom:



Zápis stejného stromu dle naší definice jediným literálem by byl tento:

```
node(10,karel,
  node( 6,josef,leaf,leaf),
  node(20,marta,
    node(16,pavla,leaf,leaf),
    node(40,milan,leaf,leaf)
  )
)
```

Mezi typické operace se stromy patří vyhledání dat podle klíče a převod stromu na seznam (linearizace) dle požadovaného algoritmu. Následující dva programy ukázky takových algoritmů demonstrují.

in_order

Program 5.1.2 *Predikát je splněn tehdy, pokud se první parametr* Strom *unifikuje se stromem dle předchozí definice a druhý,* Seznam *se*

seznamem, který je tvořen in-order průchodem stromem s uloženými datovými položkami. — in_order(Strom,Seznam).

```
in_order(leaf, []).
in_order(node(_,Value,L,R), List) :-
  in_order(L, LL),
  in_order(R, RL),
  append(LL, [Value|RL], List).
```

I když je algoritmus zapsán v programu poměrně elegantně a krátce, bez použití operátoru řezu, tak má jistou nevýhodu, tou je časová složitost. Věděli byste, jak změnit program tak, aby se časová složitost významně snížila, jak?

Program 5.1.3 *Predikát je splněn tehdy, pokud se první parametr* search Strom *unifikuje se stromem dle předchozí definice a druhý,* Seznam *se seznamem, který je tvořen in-order průchodem stromem s uloženými datovými položkami.* — search(Klíč,Strom,Data).

```
search(_, leaf, _) :-
  !, fail.
search(Key, node(Key,Value,_,_), Value) :-
  !.
search(Key, node(KeyT,_,L,_), Value) :-
  Key < KeyT, !,
  search(Key, L, Value).
search(Key, node(_,_,_,R), Value) :-
  search(Key, R, Value).
```

První užití operátoru řezu v tomto programu/predikátu je jedno z typických užití — chceme vyvolat okamžité selhání. Ano, je to ten případ, kdy strom hledaná data neobsahuje a proto musí hledání jako takové selhat. Ostatní užití operátoru řezu není nutné, ale může urychlit program, který by tento predikát použil, neboť zamezuje prohledávání těch pasáží stromu, kde se data stejně vyskytovat nemohou, neboť daný klíč se vyskytuje ve stromě jen jedenkrát.

Výrazy ve stromu

Stromové struktury se hodí i pro reprezentaci výrazů a jejich manipulaci za běhu — vyhodnocení, transformace, atd. Uvažme jednoduchý strom, který může nést operace pro sčítání, odčítání, násobení, celočíselné dělení a číselné hodnoty. Literály pro různé typy výrazů následují.

- 3

```
val(3).
```

- $3 + 8$

```
op(plus, val(3), val(8)).
```

- $3 * 5 - 2$

```
op(sub,
  op(mul, val(3), val(5)),
  val(2)
).
```

Řešený příklad

Zadání: Vytvořte predikát, který vyhodnotí výrazový strom odpovídající výše uvedenému schématu.
Řešení: Při řešení je třeba si uvědomit několik základních momentů:

- je třeba vzít do úvahy každý operátor — typ uzlu ve stromu
- je nutné vzít do úvahy typy listů — jen číselné konstanty
- vyhodnocení probíhá post-order průchodem stromem

Potom je již řešení jednoduché:

```
eval(val(X),X).
eval(op(plus,L,R,),X) :-
  eval(L,LX), eval(R,RX),
  X is LX+RX.
eval(op(minus,L,R),X) :-
  eval(L,LX), eval(R,RX),
  X is LX-RX.
eval(op(mul,L,R),X) :-
  eval(L,LX), eval(R,RX),
  X is LX*RX.
eval(op(div,L,R),X) :-
  eval(L,LX), eval(R,RX),
  X is LX//RX.
```

Z řešení je možné vyčíst, že pokud správně volíme (a je to přitom umožněno) skladbu parametrů, tak operátor řezu není třeba použít, neboť při navracení neexistuje jiná varianta, kde by se mohl výpočet zachytit a způsobit tak nalezení chybného řešení. Pokud bychom však operátor řezu použili, tak můžeme u některých distribucí Prologu zrychlit výpočet, pokud by predikát *eval* byl vyžíván často v části

programu, kde probíhá navracení — zamezilo by se neúspěšnému prohledávání a testování selhávajících variant.

Pojmy k zapamatování
Mezi klíčové pojmy této výkladové části především náleží:

- datové struktury, z nichž jsme odvodili
- n-tice
- a stromy, jako reprezentanty rekurzivní datové struktury

Důležité jsou však získané dovednosti, mezi které by mělo patřit zvládnutí práce s datovými strukturami různých typů, jejich tvorba, zpracování a manipulace za běhu programu.

Výklad

5.2 Operátory definované uživatelem

Prolog umožňuje, jako řada deklarativních jazyků, definici vlastních operátorů, tj. predikátů, které nejsou nutně tvořeny identifikátorem a v závorkách uzavřenými parametry, ale jsou tvořeny znaky netextovými, jako např. znaménko plus „+“, je větší než „>“, apod. Kromě takovýchto netextových operátorů umožňuje i definici operátorů složených z textových znaků, které je možné (dle distribuce)

Definice vlastního operátoru

Definice vlastního operátoru. Je možná pomocí speciálního vestavěného predikátu.

Definice 5.2.1 *Nový operátor se definuje pomocí tohoto vestavěného predikátu:* **Definice!**

op(+Precedence, +Type, :Name)

kde Precedence *značí prioritu operátoru v rozsahu 0–1200, přičemž číslo 0 deklaraci operátoru ruší a 1200 je nejnižší priorita;* Type *určuje typ operátoru, je to jedna z těchto možností (f je poloha operátoru): xf, yf, xfx, xfy, yfx, yfy, fy, fx; a* Name *určuje pojmenování nového operátoru, může to být i seznam operátorů se stejnými vlastnostmi.*

Na to, abychom mohli takový nově definovaný operátor použít v programu, musíme vestavěný predikát *op* umět správně použít, např. takto (operátor + pro sčítání, který je zleva asociativní, definován tak, jak je to v SWI Prologu):

```
:-op(500,yfx,+).
```

Tímto totiž přinutíme Prolog při zpracování našeho programu zavést okamžitě definici daného operátoru a dále v textu umožnit jeho použití.

ukázka užití

V další ukázkové situaci budeme opět konstruovat strom jako speciální případ grafu, ale vytvoříme si k tomu operátor.

```
:- op(500, xfx, ':->').

root(a).

a :-> b.  a :-> c.  a :-> d.
b :-> e.  b :-> f.
c :-> g.  c :-> h.  c :-> i.
d :-> j.
e :-> k.
f :-> l.  f :-> m.
h :-> n.
i :-> o.  i :-> p.
j :-> q.  j :-> r.  j :-> s.
m :-> t.
```

Operátor umožnil zapsat vždy vztah dvou uzlů, kdy uzel vlevo je rodičovským a uzel vpravo je uzlem synovským. Na to, abychom ověřili, zda uzel je vnitřní uzel stromu, nebo je listem je možné vytvořit jednoduché predikáty, využijí nově definovaný operátor:

```
is_leaf(X) :- _ :-> X, not(X :-> _).
is_inner(X) :- X :-> _.
```

Pro list musí platit, že do něj vede nějaká hrana z rodičovského uzlu a zároveň z něj nevede žádná hrana dál. Vnitřní uzel je sice doplňkem k listovému, ale výpočetně bychom situaci hodně komplikovali, takže mnohem jednodušší je využít toho, že vnitřní uzel je takový, který má alespoň jeden synovský.

To, že operátory mohou být i textové si ukážeme na ukázce operátoru, který bude ověřovat, zda dva uzly, zadané jako parametr, mají stejný rodičovský uzel:

```
:- op(500, xfx, 'is_sibling_of').

X is_sibling_of Y :-
  Z~:-> X,
  Z~:-> Y,
  X \== Y.
```

Vidíme, že operátor je možné využít jak jako konstruktor dat/fakt, tak pro definici predikátu. Užití je potom obdobné jako u jiných predikátů:

```
?- h is_sibling_of i.
Yes
?- b is_sibling_of X.
X = c ;
X = d ;
No
```

Na závěr tohoto oddílu si ještě ukážeme jeden řešený příklad.

Řešený příklad

Zadání: Vytvořte predikát *locate*, který vypíše cestu od kořene k zadanému uzlu.
Řešení: Důležité je si uvědomit, že cesta má být od kořene, k zadanému uzlu. Tzn., že musíme hledat rodičovské uzly postupně až ke kořeni a až jsou nalezeny, tak využít rozvinutí rekurzivního prohledávání a výpis dílčího úseku cesty nechat až na dobu po nalezení a tisku té předchozí.

```
locate(Node) :-
  path(Node),
  write(Node),
  nl.
path(Node) :-
  root(Node).
path(Node) :-
  Mother :-> Node,
  path(Mother),
  write(Mother),
  write(' :-> ').
```

Výsledný program je opět poměrně jednoduchý a průzračný. Jeho použití a výsledek při zadání konkrétního uzlu je očekávaný, pokud však zadáme volnou proměnnou, tak dostaneme takovouto odezvu (zkráceno):

```
?- locate(X).
a

X = a ;
a :-> b

X = b ;
a :-> c

X = c ;
a :-> d

X = d ;
a :-> b :-> e

X = e ;
a :-> b :-> f

X = f
```

Vidíme tedy, že použití operátoru může vést k výpisu celého grafu-stromu.

Pojmy k zapamatování
V této výkladové části se vyskytuje jediný pojem — operátor. Mnohem důležitější je ale zvládnutí tvorby a užití vlastních, uživatelem definovaných operátorů. A to jak v roli tvůrců dat, tak v roli predikátů.

Výklad

5.3 Změna databáze, dynamické klauzule

Poslední, ale nikoliv nejméně důležitou, výkladovou částí této kapitoly je stať, která se hlouběji věnuje využití změny databáze za běhu a s tím úzce souvisejícím pojmem — dynamická klauzule.

Změnu databáze můžeme chtít provádět ze dvou důvodů:

1. chceme měnit za běhu program, algoritmus, dle kterého se provádí nějaký náš výpočet — používá se pro náročné výpočty jako simulace, verifikace apod.
2. chceme si za běhu ukládat stav výpočtu — ukládáme nalezené znalosti, naši pozici, při průchodu stavovým prostorem apod.

Standardní operace

Mezi standardní operace pro manipulaci databáze patří tyto (již byly uvedeny v kapitole 3 v definicích 3.3.8 a 3.3.10 na straně 40):

- assert(+Term)
- asserta(+Term)
- assertz(+Term)
- retract(+Term)
- retractall(+Head) — odstraní všechny klauzule s danou hlavičkou

Navíc, pokud je nějaká klauzule měněna za běhu programu, tak se nazývá *dynamická klauzule*. To, že se jedná o dynamickou klauzuli musíme sdělit i Prologu, neboť tím eliminujeme spoustu chyb hlášených Prologem za běhu programu. Chceme-li např. definovat, že klauzule se jmény *tmp* a *pos* jsou dynamické a první má dva parametry a druhá jeden, tak to učiníme takto:

dynamická klauzule

```
:- dynamic
  pos/1,
  tmp/2.
```

Pokud bychom toto neučinili, tak při pokusu o vyhledání podcíle se jménem této klauzule, bychom obdrželi chybové hlášení, že taková neexistuje a program by byl zastaven. Pokud ji však definujeme jako dynamickou, tak se to projeví prostým neúspěchem, jako selhávající podcíl.

Prohledávání stavového prostoru

V našem výkladu se soustředíme na využití dynamických klauzulí pouze při prohledávání stavového prostoru. Při prohledávání stavového prostoru musíme obecně mít k dispozici:

- definici prostoru, kterým procházíme
- výchozí bod v prostoru
- operaci pro výběru jednoho kroku ve stavovém prostoru z pozice stávající
- hodnotící funkci, podle které poznáme, že jsme nalezli cíl

Kromě toho by náš algoritmus měl zaručovat, že nezačneme chodit v kruhu.

Vezměme v úvahu modelový případ, kdy náš prostor je tvořen maticí bodů 9x9, výchozí a koncový bod je zadán parametrem, jeden krok nemůžeme učinit pouze vodorovně, či pouze svisle, ale úhlopříčně, vždy na nejbližšího souseda a konec poznáme tak, že dorazíme do cílového bodu. Potom program pro učinění jednoho kroku může vypadat takto:

nextStep

Program 5.3.1 *Predikát* nextStep *je splněn tehdy, pokud z pozice* X, Y *se můžeme v jednom kroku pohnout na pozici* NX, NY*; jako pomocný využívá predikát* test*, který ověřuje, že nová pozice není mimo stavový prostor* — nextStep(X,Y,NX,NY)

```
nextStep(X, Y, XX, YY) :-
  XX is X+1, YY is Y+1, test(XX, YY).
nextStep(X, Y, XX, YY) :-
  XX is X+1, YY is Y-1, test(XX, YY).
nextStep(X, Y, XX, YY) :-
  XX is X-1, YY is Y+1, test(XX, YY).
nextStep(X, Y, XX, YY) :-
  XX is X-1, YY is Y-1, test(XX, YY).

test(X, Y) :- X>0, Y>0, X<10, Y<10.
```

Činnost predikátů je průzračná a nebudeme se jí zabývat do hloubky.

Mnohem důležitější je samostatný algoritmus pro řízení průchodu stavovým prostorem. Uvedeme si variantu, která cestu, kterou se k cíli dostala, průběžně ukládá do seznamu.

searchPathL

Program 5.3.2 *Predikát uspívá tehdy, pokud ze startovní pozice* S *existuje cesta uložené v seznamu* L *do cílové pozice* E — searchPathL(S,E,L)

```
:- dynamic
  pos/2.

searchPathL(start(X,Y), end(X,Y), [p(X,Y)]) :-
  !.
searchPathL(start(X,Y), E, [p(X,Y)|T]) :-
  assert(pos(X,Y)),
  nextStep(X, Y, XX, YY),
  not( pos(XX,YY) ),
  searchPathL(start(XX,YY), E, T).
searchPathL(start(X,Y), _, _) :-
  pos(X, Y),
```

```
  retract(pos(X,Y)),
  fail.
```

Na začátku programu si všimněme, že predikát definuje dynamickou klauzuli *pos*, kterou program využívá pro ukládání těch míst, která na dané cestě již navštívil, aby zamezil chození v kruhu. První definice klauzule *searchPathL* potom řeší situaci, kdy je startovní a koncová pozice shodná. Druhá potom realizuje jeden krok v prostoru:

1. uloží do databáze aktuální pozici
2. realizuje jeden krok
3. ověří, že nejde o krok do místa, kde již cesta vede (případně přes zpětné navracení nalezne jiný krok)
4. odstartuje prohledávání z nového místa

Pokud nelze učinit krok, nebo prohledávání danou cestou selže, tak se dostáváme k poslední klauzuli *searchPathL*, která ověří, že pozice aktuální pozice je uložena v databází, poté ji z databáze odstraní a způsobí selhání, aby bylo vyvoláno navracení pro hledání další, jiné cesty.

Pokud nás výsledek uspokojí a dále již budeme vyhledávat v nové situaci, tak před dalším vyvoláním predikátu pro hledání musíme odstranit všechny klauzule *pos* z databáze. Buďto vestavěným predikáte *retractall* nebo si můžeme pomoci takovýmto prográmkem, který funguje jen s predikátem *retract*.

Program 5.3.3 *Predikát uspívá tehdy, pokud z databáze odstraní* clearPos *všechny klauzule/fakta* pos *se dvěma parametry* — clearPos

```
clearPos :-
  pos(X, Y),
  retract( pos(X,Y) ),
  !, clearPos.
clearPos.
```

Vidíme, že násobnému uspění predikátu je zabráněno užitím operátoru řezu. Predikát uspívá i tehdy pokud v databázi není obsažen žádný predikát *pos*.

Jinou možností, jak provádět procházení stavovým prostorem, je nebudovat průběžně cestu v seznamu, ale využít toho, že si cestu poznamenáváme do klauzule *pos*. Stačí zaručit, aby klauzule *pos* byla v databázi uložena ve správném pořadí. Řešení ukazuje následující program.

searchPath

Program 5.3.4 *Predikát uspívá tehdy, pokud existuje cesta z pozice* S *do pozice* L*; předpokladem je, že klauzule* pos/2 *není využívána a v databázi není obsažena, neboť výsledek je uložen v posloupnosti právě těchto fakt.* — searchPath(S,E)

```
:- dynamic
  pos/2.

searchPath(start(X,Y), end(X,Y)) :-
  assert( pos(X,Y) ).
searchPath(start(X,Y), end(X,Y)) :-
  pos(X, Y),
  retract( pos(X,Y) ),
  !, fail.
searchPath(start(X,Y), E) :-
  assert( pos(X,Y) ),
  nextStep(X, Y, XX, YY),
  not( pos(XX,YY) ),
  searchPath(start(XX,YY), E).
searchPath(start(X,Y), _) :-
  pos(X, Y),
  retract( pos(X,Y) ),
  fail.
```

Vidíme, že program si zachovává charakter předchozího prohledávacího programu. Druhá definice klauzule však přidává kód, který se využívá při zpětném navracení — odstraní aktuální cílovou pozici a vynutí selhání (dvojice operátor řezu a fail). Pokud je cesta nalezena, tak výpočet končí, tak jak v předchozím případě, u první klauzule predikátu.

Pojmy k zapamatování

V této výkladové části se vyskytuje jediný pojem — dynamická klauzule. Mnohem důležitější je ale dovednost využívat takové klauzule spolu s operacemi změny databáze a zvládnutí problematiky prohledávání stavového prostoru.

Závěr

V této kapitole jsme se soustředili na změnu databáze, prohledávání stavového prostoru a dynamické klauzule.
Na konci této kapitoly byste tedy měli mít již značné dovednosti v programování v jazyce Prolog, obohacené zejména o změnu databáze za běhu a dynamické klauzule. Měli byste být také schopni řešit úlohy spojené s prohledáváním stavového prostoru. Pokud máte stále pochybnosti, že něco není v pořádku, projděte si znovu příklady v této kapitole, nebo se podívejte po internetu, či do příkladů dodaných k vaší distribuci Prologu.

Úlohy k procvičení:

1. K definici stromu z programu 5.1.1 a následujících stránek přidejte tyto predikáty:
   - pre_order(T,L) — T je strom a L je seznam z klíčů stromu vytvořený pre-order průchodem stromem
   - post_order(T,L) — T je strom a L je seznam dvojic klíč a data vytvořený post-order průchodem stromem
   - modifikujte predikát add2tree tak, aby při vkládání dat s klíčem, který je již ve stromu obsažen, provedl změnu dat na nová a zbytek zachoval stejný
   - list2tree(L,T) — L je seznam s páry klíč a data, T je strom vytvořený na základě těchto dat (využijte již existující predikáty)
2. K úloze o výrazech ze strany 69 přidejte tyto modifikace
   - využijte vyhledávacího stromu, který rozšiřte o vlastní predikát pro porovnávání tak, abyste mohli výrazový strom obohatit o proměnné a operátor přiřazení; strom využijte jako tabulku identifikátorů
   - změňte strom tak, aby nenesl celočíselné výrazy, ale boolovské výrazy, upravte i jeho evaluátor
   - zaveďte do stromu proměnné, udělejte predikát replace(T,V,E,N) — T je strom s výrazem, který je modifikován tak, že každý výskyt proměnné V je nahrazen výrazem E a výsledek je unifikován se stromem v parametru N

3. Využijte predikát z definice 5.2.1 k definici predikátů pro rozšíření manipulace s grafem stromu, který byl zaveden ve stejném oddílu jako zmíněná definice.
   - is_same_level_as(N1,N2) — operátor v roli predikátu uspívá, pokud jsou dva uzly na stejné úrovni, stejně vzdáleny od kořene
   - has_depth(N,D) — uspívá, pokud uzel N má ve stromu hloubku D (vzdálenost od kořene)
4. Program 5.3.4 doplňte o další predikáty
   - writePath — vypíše nalezenou cestu; ověřte činnost ve spojitosti se zpětným navracením(!)
   - listPost — uloží nalezenou cestu ve správném pořadí do seznamu

Klíč k řešení úloh

Pro řešení úloh, pokud vám dělají problémy, využijte informaci v této kapitole. Důkladně ji znovu prostudujte, případně hledejte v knihách obecně o algoritmech v grafech, či prohledávání stavového prostoru. Nicméně tato kapitola by měla stačit!

Další zdroje

Řadu informací k tématu podaného v této kapitole najdete v literatuře citované níže. Případně v manuálech a tutoriálech k distribuci vašeho Prologu. Jelikož se však tento modul zabývá především programováním, tak nejlepších výsledků dosáhnete tím, že si sednete k počítači a budete zkoušet a programovat a zkoušet a programovat a tak dále a tak dále.

- Nilsson, U., Maluszynski, J.: *Logic, Programming and Prolog (2ed)*, John Wiley & Sons Ltd., 1995.
- Bieliková, M., Návrat, P.: *Funkcionálne a logické programovanie*, Vydavateľstvo STU, Vazovova 5, Bratislava, 2000.

Kapitola 6

Paralelismus v Prologu

Tato kapitola podává jisté základy tvorby paralelně zpracovaných podcílů ve dvou distribucích Prologu — SWI Prologu a Ciao Prologu. Po obecném představení vestavěných predikátů je využití demonstrováno na algoritmu Merge-Sort, pro obě distribuce.

Čas potřebný ke studiu: 4 hodiny 30 minut.

Cíle kapitoly

Cílem kapitoly je zvládnout základní práci s vlákny a paralelním vyhodnocení v Prologu. Ukázat specifika související s různými distribucemi Prologu a naznačit typickou problematiku paralelismu.

Průvodce studiem

Před vstupem do této kapitoly je naprosto nutné, abyste zvládli všechny předchozí kapitoly, neboť v této kapitole se jednotlivé prvky již bez vysvětlení použijí a uplatní při demonstraci možností paralelního vyhodnocení v Prologu.
Cílem je demonstrace mechanismů a předvedení na jedné aplikaci — algoritmu Merge-Sort (jehož znalost se také předpokládá). Pro zvládnutí se dále předpokládá základní znalost s paralelního zpracování procesů a vláken obecně! Kapitola je obsahově velmi útlá, ale vyžaduje a předpokládá práci u počítače a samostatné procvičení i na dalších úlohách.
Ke studiu této kapitoly je naprosto nezbytné mít k dispozici počítač s instalovaným interpretem jazyka Prolog — vhodné je mít jednu z prezentovaných distribucí. Je možné mít i kompilátor, ale práce v něm, zejména pro začátečníky, je mnohem, ale mnohem náročnější. Ale jak bylo řečeno, v dalším však doporučujeme užití systému SWI-Prolog. Další možností je potom zmiňovaný CiaoProlog, který však díky rozsáhlému systému knihoven již pro základní práci vyžaduje zvládnutí práce s moduly, což je i případ paralelismu.

Obsah

Výklad

Klíčovým problémem skutečného paralelismu v deklarativních programovacích jazycích, přesněji v jazycích, kde je využíván garbage collector jakožto správce paměti, je paralelní běh správy paměti a vyhodnocovacího stroje. Proto, pokud chceme uplatnit a ověřovat skutečný paralelismus, musíme vždy dát pozor, jestli naše distribuce skutečný paralelismus podporuje, anebo je vhodná pouze pro virtuální paralelismus — jediný procesor.

6.1 Paralelismus v Ciao Prologu

Parametry a výsledky mezi vlákny se předávají pomocí databáze, na to slouží dynamické klauzule, které musí být ale v Ciao Prologu deklarovány jakožto paralelní, neboť potom se na ně vztahuje speciální režim, který lze využít pro spouštění vláken a jejich synchronizaci. Pro naše potřeby budeme definovat dvě takové klauzule, podobně jako klauzule dynamické ale jiným vestavěným predikátem:

```
:- concurrent
  res/1,
  run/1.
```

Zvláštní vlákno výpočtu se vytváří příslušným predikátem z příslušné knihovny. Predikát je definován takto:

Definice 6.1.1 *Vytvoření vlákna v Ciao Prologu je umožněno predikátem:* **Definice!**

eng_call(+Goal, +EngineCreation, +ThreadCreation, -GoalId)

První parametr je (pod)cíl, který má být paralelně vyhodnocen. Parametr EngineCreation *definuje způsob vytvoření vyhodnocovacího stroje (wait, či create). Parametr* ThreadCreation *definuje způsob vytvoření vlákna (self, wait, či create). Poslední parametr vrací identifikaci vlákna/cíle. Pokud jde o vlastní vlákno (self), predikát uspívá, pokud uspěje i cíl* Goal.

Vzhledem k tomu, že uplatnění skutečné paralelismu určitě nebude možné u vytváření vlastních vláken, vlákna budou muset být oddělená, tak musíme nějakým způsobem detekovat, že došlo k ukončení výpočtu v jiném vláknu, případně na něj počkat. Pro tento případ je Ciao Prolog vybaven následujícím knihovním predikátem.

Definice!

Definice 6.1.2 *Pro čekání na dokončení výpočtu paralelního vlákna je možné využít predikát:*

eng_wait(+GoalId)

Jediným parametrem je identifikátor vlákna/cíle, který je paralelně vypočítáván. Pokud by výpočet nebyl ještě dokončen, volající čeká na dokončení výpočtu cíle specifikovaného v GoalId.

Když jsme se ujistilo, že paralelní vlákno doběhlo, tak je možné uvolnit ho z paměti a tak uvolnit všechny prostředky, které potřebovalo pro svou činnost. K tomuto uvolnění (po ověření, že vlákno již doběhlo), slouží následující predikát.

Definice!

Definice 6.1.3 *Pro uvolnění vlákna, které je ve stavu* idle*:*

eng_release(+GoalId)

Na test stavu idle *se dá použít predikát* eng_wait*. Pokud se pokusíte vlákno uvolnit aniž by dosáhlo stavu* idle*, dojde k pádu systému.*

Jak jsme již řekli, pro komunikaci mezi vlákny se používá databáze a operace na její obměnu. Ciao Prolog pro případ práce s fakty (klauzule bez těla) definoval sadu predikátů, které mají stejnou funkci jak obdobně pojmenované „klasické“ verze predikátů pro změnu databáze. Jsou však určeny výhradně pro fakty a doporučeny pro použití v paralelním zpracování, neboť v sobě skrývají spouštěcí a synchronizační mechanismy. Jedná se o:

- asserta_fact — odpovídá asserta
- assertz_fact — odpovídá assertz
- retract_fact — odpovídá retract

Jako blokující (operace retract „čeká“ na operaci assert) se chovají nad těmi fakty, co byla označena jako paralelní. Jejich užití ale může být zrádné.

Případová studie — Merge-Sort

V tomto oddílu si postupně ukážeme obsah souboru, který definuje paralelní Merge-Sort. V úvodu je nutné importovat potřebné moduly a deklarovat paralelní fakty:

```
:- use_module(library(concurrency)).
:- use_module(library(lists)).
:- use_module(library(system)).

:- concurrent res/1, run/1.
```

Jako další v pořadí definujme predikát, který spojí dva vzestupně seřazené seznamy do jednoho vzestupně seřazeného seznamu (vzhledem k vlastnostem Prologu musejí být číselné, ale lehce by šlo upravit, na základě vlastního predikátu pro porovnání, na jakákoliv jiná data):

```
mergeL([],L,L) :- !.
mergeL([H|T],[],[H|T]) :- !.
mergeL([H1|T1],[H2|T2],[H1|T]) :-
  H1 =< H2,
  mergeL(T1,[H2|T2],T),
  !.
mergeL([H1|T1],[H2|T2],[H2|T]) :-
  H1 > H2,
  mergeL([H1|T1],T2,T).
```

Další v pořadí je pomocný predikát, který rozdělí seznam na dva v místě, které mu označíme. Vstupní seznam je první parametr, místo dělení druhý a další dva parametry jsou výsledné seznamy:

```
separate(L,0,[],L) :- !.
separate([H|T], N, [H|L] ,R) :-
  N>0,
  NN is N - 1,
  separate(T,NN,L,R).
```

Následuje poslední predikát, který pracuje pouze sekvenčně. Je to sekvenční verze algoritmu Merge-Sort. Celá idea paralelního zpracování (ve dvou vláknech) je totiž založen na tom, že se vstupní seznam rozdělí na dva, ty se seřadí paralelně a potom se sloučí do jednoho. Proto potřebujeme plně funkční paralelní verzi:

```
msrts([],[]) :- !.
msrts([X],[X]) :- !.
msrts(L,SL) :-
```

```
  length(L,X), M is X // 2,
  separate(L,M,LL,RL),
  msrts(LL,SLL),
  msrts(RL,SRL),
  mergeL(SLL,SRL,SL).
```

V závěru prezentujeme hlavní tělo, které rozdělí vstupní seznam na dva a spustí paralelní provádění. Ke své činnosti potřebuje ještě jeden malý predikát, který řídí výpočet v druhém vláknu (převzetí seznamu pro seřazení a předání seřazeného seznamu):

```
msrtp([],[]) :- !.
msrtp([X],[X]) :- !.
msrtp(L,SL) :-
  length(L,X), M is X // 2,
  separate(L,M,LL,RL),
  eng_call(msrtphx,create,create,GID1),
  assertz_fact(run(LL)),
  msrts(RL,SRL),
  retract_fact(res(SLL)),
  eng_wait(GID1),
  eng_release(GID1),
  mergeL(SLL,SRL,SL),
  !.

msrtphx :-
  retract_fact(run(L)),
  msrts(L,R),
  assertz_fact(res(R)).
```

Vidíme, že začátek obou verzí algoritmu Merge-Sort jsou shodné (prvních 5 řádků). Potom paralelní verze vytvoří druhé výpočetní vlákno, dále vloží seznam pro zpracování tímto druhým vláknem do databáze, sám seřadí druhou část seznamu, po té čeká na vložení výsledků od druhého vlákna, jakmile je převezme, tak se nasynchronizuje na ukončení tohoto vlákna, uvolní ho z paměti, spojí dvě poloviny seznamů a ukončí výpočet. Pomocný predikát *msrtphx* čeká na vložení seznamu pro seřazení, jakmile je vložen, seřadí ho, potom uloží výsledek a tím ukončí svou činnost, přejde do stavu *idle*.

Celý trik tedy není nijak složitý na pochopení. U skutečně paralelního výpočtu (více skutečných procesorů, nebo vícejádrový jeden) však buďte obezřetní a pečlivě ověřte, že vaše distribuce Ciao Prologu skutečný paralelismus podporuje a zvládá.

6.2 Paralelismus v SWI Prologu

Možnosti SWI Prologu ve srovnání s Ciao Prologem jsou prakticky stejné, akorát SWI Prolog nemá vše uloženo ve speciálních knihovnách a pro řadu operací nepoužívá specializované predikáty.

SWI Prolog nemusí definovat žádné dynamické klauzule používané v paralelním zpracování pro synchronizaci vláken.

Pro vytvoření vlákna používá zvláštní predikát.

Definice 6.2.1 *Vytvoření vlákna v SWI Prologu je umožněno predikátem:* **Definice!**

thread_create(:Goal, -GoalId, +Options)

První parametr je (pod)cíl, který má být paralelně vyhodnocen. Jestliže je vlákno úspěšně vytvořeno, tak je identifikátor vlákna navázán na parametr GoalId. *Poslední parametr,* Options*, je seznam nastavení pro oddělované vlákno (velikost pamětí různého typu, které se pojí s výpočtem; alias; oddělení vlákna). Tento seznam je ale možné nechat prázdný, neboť implicitní přednastavení jsou pro běžnou činnost dostačující.*

Pro spojení s jiným vláknem (které ukončilo svou činnost) slouží následující predikát. Pokud výpočet druhého vlákna ještě běží, tak predikát čeká, až svou činnost dokončí, třeba i výjimkou.

Definice 6.2.2 *Vytvoření vlákna v SWI Prologu je umožněno predikátem:* **Definice!**

thread_join(+GoalId, -Status)

První parametr je vlákno, na jehož dokončení se má čekat. Druhý parametr je výstupní a říká, v jakém stavu byl výpočet dokončen: true — uspívající, false — selhání, exception(Term) — ukončeno výjimkou, exited(Term) — násilně ukončen. Zejména první dva výsledky jsou pro běžnou práci dostačující. Prostředky vlákna daného GoalId *jsou po spojení uvolněny.*

Žádné další zvláštní predikáty nejsou pro naši případovou studii třeba. Vše již známe.

Případová studie — Merge-Sort

Dále uvedeme jen to, co je rozdílné oproti implementaci v Ciao Prologu. Předně se jedná o import modulů a definici dynamických fakt. V SWI Prologu plně dostačuje tento kód:

```
:- dynamic
  run/1,
  res/1.
```

Celá sekvenční výpočtová část je shodná. Formální rozdíl je až v paralelní části:

```
msrtp([],[]) :- !.
msrtp([X],[X]) :- !.
msrtp(L,SL) :-
  length(L,X), M is X // 2,
  separate(L,M,LL,RL),
  assert(run(LL)),
  thread_create(msrtphx,GID1,[]),
  msrts(RL,SRL),
  thread_join(GID1,_),
  retract(res(SLL)),
  mergeL(SLL,SRL,SL),
  !.

msrtphx :-
  retract(run(L)),
  msrts(L,R),
  assert(res(R)).
```

Jelikož rozdíly jsou opravdu jen formální, tak se nebudeme objasněním činnosti dále podrobně zabývat. Jen dodáme, že tak jako u Ciao Prologu je dobré si ověřit chování vaší verze distribuce SWI Prologu. I v sekvenční verzi si totiž můžete ověřit, jaký nejdelší seznam se vám podaří seřadit. Pokud budete chtít zpracovat delší, budete si muset „pohrát“ s nastavením pamětí, aby vám nepřetékal globální zásobník, nebo vymyslet jiný způsob zpracování, či řazení, který by pracoval *in situ*.

Pojmy k zapamatování

V této kapitole se nevyskytuje žádný pojem, který by byl typický pro Prolog a s ním spojený. Pracovali jsme však s pojmem známým z jiného odvětví informatiky, s pojmem *vlákno*. Tvorbu a synchronizaci vláken v Prologu v základní podobě byste potom měli zvládnout dovednostně.

Závěr

V této kapitole jsme se soustředili na jediné — vícevláknové zpracování úloh v Prologu, respektive dvou vybraných distribucí. Na případové studii jsme demonstrovali způsob práce s více vlákny, jejich vytváření, předávání dat a synchronizaci.
Na konci této kapitoly byste tedy měli mít v podstatě osvojeny všechny klíčové vlastnosti pro programování v Prologu, které byste měli být schopni využít i při tvorbě paralelních algoritmů, které, jak se ukazuje, jsou pro budoucnost ve výpočetní technice klíčové. Pečlivě zrevidujte své schopnosti a případné nedostatky z některých oblastí programování v Prologu zacelte řešením dalších úloh, které si buďto sami najdete ve své praxi, nebo v rámci cvičení na Internetu. Také tato publikace vám bude v řadě věcí jistě užitečná.

Úlohy k procvičení:

1. Najděte jiný řadicí algoritmus pro paralelizaci a paralelizujte jej. Porovnejte časovou složitost před a po paralelizaci (ideálně na stroji s jediným procesorem i na stroji s více procesory).
2. Paralelizujte výpočet čísla z Fibonacciho posloupnosti, srovnejte časovou složitost vašich řešení (sekvenčního i paralelního). Vyberte si jak časově náročnou, tak optimalizovanou verzi.
3. Pokuste se paralelizovat úlohu prohledávání stavového prostoru. V čem tkví problém paralelizace této úlohy?

   Pokud tento úkol nezvládnete, nic se neděje, je to opravdu těžký úkol.

Klíč k řešení úloh

Pro řešení paralelizace úloh nejprve vyřešte úlohy sekvenčně. To by neměl být žádný problém, jedná se o základní školní úlohy algoritmizace (až na ten stavový prostor). Poté nalezněte místo, kde se dva podobné výpočty (charakterem i složitostí) nacházejí blízko sebe a jeden z nich oddělte do jiného vlákna.
Pro prohledávání stavového prostoru je především nutné najít bod rozdělení, který není dopředu znám. Typicky to může být takové místo ve stavovém prostoru, ze kterého vede nejvíce cest, nebo alespoň dvě, které „by mohly být“ stejně dlouhé. Poté rozdělíme výpočet — spustíme dvě nezávislá vlákna (při dělení na 2) a hlavní zastavíme.

Oddělené části musejí pracovat se společnou částí dat, která se po dobu výpočtu ve vláknech nemění, a navíc se svými daty, která musejí být odlišná od dat druhého vlákna (např. dynamické klauzule se musejí jmenovat jinak). I když je to náročná úloha, výsledek stojí za to.

Další zdroje

Řadu informací k tématu podaného v této kapitole najdete na níže zmiňovaných stránkách distribucí Prologu, které poskytují plnou nápovědu k podpoře paralelního zpracování a jdou až za rámec této kapitoly. Nicméně i tato kapitola se zabývá především programováním a tak nejlepších výsledků dosáhnete tím, že si sednete k počítači a budete zkoušet a programovat a zkoušet a programovat a tak dále a tak dále.

- SWI Prolog: http://www.swi-prolog.org
- Ciao Prolog: http://www.ciaohome.org

Kapitola 7

Závěr

Publikace představuje v několika kapitolách typické rysy několika základních a vybraných pokročilých obratů a technik v Prologu. Přitom vysvětluje i samotné principy jazyka jak formální, tak operační.

Po absolvování by student měl být schopen tvořit jednoduché a středně obtížné aplikace v jazyku Prolog s tím, že tvorba náročných aplikací by, po jisté době vlastních praktických zkušeností, neměla zdaleka být problematická.

Modul je základním modulem pro logické programovací jazyky předmětu Funkcionální a logické programování. Doplňuje modul Funkcionální programování, se kterým tak tvoří celek vhodný pro úvodní studium problematiky funkcionálního a logického programování. Obě tato paradigmata jsou přitom uvedena jak teoreticky, tak zejména na praktických ukázkách a výuce práce v jednotlivých čelních představitelích programovacích jazyků obou směrů.

Modul obsahuje pouze prvotní a pokročilé informace a nástiny problematiky. Pro úplné zvládnutí problematiky je vhodné rozšířit si vědomosti nějakou vhodnou doporučenou literaturou jak z oblasti funkcionálního i logického programování, tak i z oblasti teorie kolem zmíněných paradigmat.

Literatura

[1] Abadi, M., Cardelli, L.: *A Theory of Objects*, Springer, New York, 1996, ISBN 0-387-94775-2.

[2] Aho, A.V., Hopcroft, J.E., Ullman, J.D.: *The Design and Analysis of Computer Algorithms*, Addison Wesley, 1974.

[3] Aho, A.V., Sethi, R., Ullman, J.D.: *Compilers: Principles, Techniques, and Tools*, Addison Wesley, Reading MA, 1986.

[4] Aho, A.V., Ullman, J.D.: *The Theory of Parsing, Translation, and Compiling, Volume I: Parsing*, Prentice-Hall, Inc., 1972, ISBN 0-13-914556-7.

[5] Aho, A.V., Ullman, J.D.: *The Theory of Parsing, Translation, and Compiling, Volume II: Compiling*, Prentice-Hall, Inc., 1972, ISBN 0-13-914564-8.

[6] Appel, A.W.: Garbage Collection Can Be Faster Than Stack Allocation, *Information Processing Letters 25 (1987)*, North Holland, pages 275–279.

[7] Appel, A.W.: *Compiling with Continuations*, Cambridge University Press, 1992.

[8] Augustsson, L., Johnsson, T.: *Parallel Graph Reduction with the* $< \nu, G >$*-Machine*, Functional Programming Languages and Computer Architecture 1989, pages 202–213.

[9] Barendregt, H.P., Kennaway, R., Klop, J.W., Sleep, M.R.: *Needed Reduction and Spine Strategies for the Lambda Calculus*, Information and Computation 75(3): 191-231 (1987).

[10] Beneš, M.: *Object-Oriented Model of a Programming Language*, Proceedings of MOSIS'96 Conference, 1996, Krnov, Czech Republic, pp. 33–38, MARQ Ostrava, VSB - TU Ostrava.

[11] Beneš, M.: *Type Systems in Object-Oriented Model*, Proceedings of MOSIS'97 Conference Volume 1, April 28–30, 1997, Hradec nad Moravicí, Czech Republic, pp. 104–109, ISBN 80-85988-16-X.

[12] Beneš, M., Češka, M., Hruška, T.: *Překladače*, Technical University of Brno, 1992.

[13] Beneš, M., Hruška, T.: *Modelling Objects with Changing Roles*, Proceedings of 23rd Conference of ASU, 1997, Stara Lesna, High Tatras, Slovakia, pp. 188–195, MARQ Ostrava, VSZ Informatika s r.o., Kosice.

[14] Beneš, M., Hruška, T.: *Layout of Object in Object-Oriented Database System*, Proceedings of 17th Conference DATASEM 97, 1997, Brno, Czech Republic, pp. 89–96, CS-COMPEX, a.s., ISBN 80-238-1176-2.

[15] Bieliková, M., Návrat, P.: *Funkcionálne a logické programovanie*, Vydavatelstvo STU, Vazovova 5, Bratislava, 2000.

[16] Brodský, J., Staudek, J., Pokorný, J.: *Operační a databázové systémy*, Technical University of Brno, 1992.

[17] Bruce, K.B.: A Paradigmatic Object-Oriented Programming Language: Design, Static Typing and Semantics, *J. of Functional Programming*, January 1993, Cambridge University Press.

[18] Cattell, G.G.: *The Object Database Standard ODMG-93*, Release 1.1, Morgan Kaufmann Publishers, 1994.

[19] Češka, M., Hruška, T., Motyčková, L.: *Vyčíslitelnost a složitost*, Technical University of Brno, 1992.

[20] Češka, M., Rábová, Z.: *Gramatiky a jazyky*, Technical University of Brno, 1988.

[21] Damas, L., Milner, R.: *Principal Type Schemes for Functional Programs*, Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 1982, pages 207–212.

[22] Dassow, J., Paun, G.: *Regulated Rewriting in Formal Language Theory*, Springer, New York, 1989.

[23] Douence, R., Fradet, P.: *A taxonomy of functional language implementations. Part II: Call-by-Name, Call-by-Need and Graph Reduction*, INRIA, technical report No 3050, Nov. 1996.

[24] Ellis, M.A., Stroustrup, B.: *The Annotated C++ Reference Manual*, AT&T Bell Laboratories, 1990, ISBN 0-201-51459-1.

[25] Finne, S., Burn, G.: *Assessing the Evaluation Transformer Model of Reduction on the Spineless G-Machine*, Functional Programming Languages and Computer Architecture 1993, pages 331-339.

[26] Fradet, P.: *Compilation of Head and Strong Reduction*, In Porc. of the 5th European Symposium on Programming, LNCS, vol. 788, pp. 211-224. Springer-Verlag, Edinburg, UK, April 1994.

[27] Georgeff M.: *Transformations and Reduction Strategies for Typed Lambda Expressions*, ACM Transactions on Programming Languages and Systems, Vol. 6, No. 4, October 1984, pages 603–631.

[28] Gordon, M.J.C.: *Programming Language Theory and its Implementation*, Prentice Hall, 1988, ISBN 0-13-730417-X, ISBN 0-13-730409-9 Pbk.

[29] Gray, P.M.D., Kulkarni, K.G., Paton, N.W.: *Object-Oriented Databases*, Prentice Hall, 1992.

[30] Greibach, S., Hopcroft, J.: Scattered Context Grammars, *Journal of Computer and System Sciences*, Vol: 3, pp. 233–247, Academia Press, Inc., 1969.

[31] Harrison, M.: *Introduction to Formal Language Theory*, Addison Wesley, Reading, 1978.

[32] Hruška, T., Beneš, M.: *Jazyk pro popis údajů objektově orientovaného databázového modelu*, In: Sborník konference Některé nové přístupy při tvorbě informačních systémů, ÚIVT FEI VUT Brno 1995, pp. 28–32.

[33] Hruška, T., Beneš, M., Máčel, M.: *Database System G2*, In: Proceeding of COFAX Conference of Database Systems, House of Technology Bratislava 1995, pp. 13–19.

[34] Issarny, V.: *Configuration-Based Programming Systems*, In: Proc. of SOFSEM'97: Theory and Practise of Informatics, Milovy, Czech Republic, Novemeber 22-29, 1997, ISBN 0302-9743, pp. 183-200.

[35] Jensen, K.: *Coloured Petri Nets*, Springer-Verlag Berlin Heidelberg, 1992.

[36] Jeuring, J., Meijer, E.: *Advanced Functional Programming*, Springer-Verlag, 1995.

[37] Jones, M.P.: *A system of constructor classes: overloading and implicit higher-order polymorphism*, In FPCA '93: Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark, June 1993.

[38] Jones, M.P.: *Dictionary-free Overloading by Partial Evaluation*, ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, Orlando, Florida, June 1994.

[39] Jones, M.P.: *Functional Programming with Overloading and Higher-Order Polymorphism*, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, Springer-Verlag Lecture Notes in Computer Science 925, May 1995.

[40] Jones, M.P.: *GOFER, Functional programming environment, Version 2.20*, mpj@prg.ox.ac.uk, 1991.

[41] Jones, M.P.: *ML typing, explicit polymorphism and qualified types*, In TACS '94: Conference on theoretical aspects of computer software, Sendai, Japan, Springer-Verlag Lecture Notes in Computer Science, 789, April, 1994.

[42] Jones, M.P.: *A theory of qualified types*, In proc. of ESOP'92, 4th European Symposium on Programming, Rennes, France, February 1992, Springer-Verlag, pp. 287-306.

[43] Jones, S.L.P.: *The Implementation of Functional Programming Languages*, Prentice-Hall, 1987.

[44] Jones, S.L.P., Lester, D.: *Implementing Functional Languages.*, Prentice-Hall, 1992.

[45] Kleijn, H.C.M., Rozenberg, G.: On the Generative Power of Regular Pattern Grammars, *Acta Informatica*, Vol. 20, pp. 391–411, 1983.

[46] Khoshafian, S., Abnous, R.: *Object Orientation. Concepts, Languages, Databases, User Interfaces*, John Wiley & Sons, 1990, ISBN 0-471-51802-6.

[47] Kolář, D.: *Compilation of Functional Languages To Efficient Sequential Code*, Diploma Thesis, TU Brno, 1994.

[48] Kolář, D.: *Overloading in Object-Oriented Data Models*, Proceedings of MOSIS'97 Conference Volume 1, April 28–30, 1997, Hradec nad Moravicí, Czech Republic, pp. 86–91, ISBN 80-85988-16-X.

[49] Kolář, D.: *Functional Technology for Object-Oriented Modeling and Databases*, PhD Thesis, TU Brno, 1998.

[50] Kolář, D.: *Simulation of LL_k Parsers with Wide Context by Automaton with One-Symbol Reading Head*, Proceedings of 38th International Conference MOSIS '04—Modelling and Simulation of Systems, April 19–21, 2004, Rožnov pod Radhoštěm, Czech Republic, pp. 347–354, ISBN 80-85988-98-4.

[51] Latteux, M., Leguy, B., Ratoandromanana, B.: The family of one-counter languages is closed under quotient, *Acta Informatica*, 22 (1985), 579–588.

[52] Leroy, X.: *The Objective Caml system, documentation and user's guide*, 1997, Institut National de Recherche en Informatique et Automatique, Francie, Release 1.05, http://pauillac.inria.fr/ocaml/htmlman/.

[53] Martin, J.C.: *Introduction To Languages and The Theory of Computation*, McGraw-Hill, Inc., USA, 1991, ISBN 0-07-040659-6.

[54] Meduna, A.: *Automata and Languages: Theory and Applications*. Springer, London, 2000.

[55] Meduna, A., Kolář, D.: Regulated Pushdown Automata, *Acta Cybernetica*, Vol. 14, pp. 653–664, 2000.

[56] Meduna, A., Kolář, D.: One-Turn Regulated Pushdown Automata and Their Reduction, In: *Fundamenta Informaticae*, 2002, Vol. 16, Amsterdam, NL, pp. 399–405, ISSN 0169-2968.

[57] Mens, T., Mens, K., Steyaert, P.: *OPUS: a Formal Approach to Object-Orientation*, Published in FME '94 Proceedingss, LNCS 873, Springer-Verlag, 1994, pp. 326-345.

[58] Mens, T., Mens, K., Steyaert, P.: *OPUS: a Calculus for Modelling Object-Oriented Concepts*, Published in OOIS '94 Proceedings, Springer-Verlag, 1994, pp. 152-165.

[59] Milner, R.: A Theory of Type Polymorphism In Programming, *Journal of Computer and System Sciences*, 17, 3, 1978.

[60] Mycroft, A.: *Abstract Interpretation and Optimising Transformations for Applicative Programs*, PhD Thesis, Department of computer Science, University of Edinburgh, Scotland, 1981. 180 pages. Also report CST-15-81.

[61] Nilsson, U., Maluszynski, J.: *Logic, Programming and Prolog (2ed)*, John Wiley & Sons Ltd., 1995.

[62] Okawa, S., Hirose, S.: Homomorphic characterizations of recursively enumerable languages with very small language classes, *Theor. Computer Sci.*, 250, 1 (2001), 55–69.

[63] Păun, Gh., Rozenberg, G., Salomaa, A.: *DNA Computing*, Springer-Verlag, Berlin, 1998.

[64] Robinson, J. A.: A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12, 23–41, 1965.

[65] Rozenberg, G., Salomaa, A. — eds.: *Handbook of Formal Languages; Volumes 1 through 3*, Springer, Berlin/Heidelberg, 1997.

[66] Salomaa, A.: *Formal Languages*, Academic Press, New York, 1973.

[67] Schmidt, D.A.: *The Structure of Typed Programming Languages*, MIT Press, 1994.

[68] Odersky, M., Wadler, P.: *Pizza into Java: Translating theory into practice*, Proc. 24th ACM Symposium on Principles of Programming Languages, January 1997.

[69] Odersky, M., Wadler, P., Wehr, M.: *A Second Look at Overloading*, Proc. of FPCA'95 Conf. on Functional Programming Languages and Computer Architecture, 1995.

[70] Reisig, W.: *A Primer in Petri Net Design*, Springer-Verlag Berlin Heidelberg, 1992.

[71] Tofte, M., Talpin, J.-P.: *Implementation of the Typed Call-by-Value λ-calculus using a Stack of Regions*, POPL '94: 21st ACM Symposium on Principles of Programming Languages, January 17–21, 1994, Portland, OR USA, pages 188–201.

[72] Traub, K.R.: *Implementation of Non-Strict Functional Programming Languages*, Pitman, 1991.

[73] Volpano, D.M., Smith, G.S.: *On the Complexity of ML Typability with Overloading*, Proc. of FPCA'91 Conf. on Functional Programming Languages and Computer Architecture, 1991.

[74] Wikström, Å.: *Functional Programming Using Standard ML*, Prentice Hall, 1987.

[75] Williams, M.H., Paton, N.W.: *From OO Through Deduction to Active Databases - ROCK, ROLL & RAP*, In: Proc. of SOFSEM'97: Theory and Practise of Informatics, Milovy, Czech Republic, Novemeber 22-29, 1997, ISBN 0302-9743, pp. 313-330.

[76] Lindholm, T., Yellin, F.: *The Java Virtual Machine Specification*, Addison-Wesley, 1996, ISBN 0-201-63452-X.