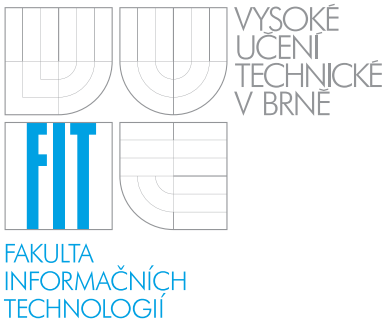

Principy programovacích jazyků a objektově orientovaného programování

Studijní opora

IPP – III

Dušan Kolář
Ústav informačních systémů
Fakulta informačních technologií
VUT v Brně

Únor '06 – 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

Programovací jazyky vnímáme často jako text, který slouží k popisu algoritmu, který se má vykonat s tím, že přesně specifikujeme co a jak se má provádět, abychom dosáhli kýženého výsledku. Existují však programovací jazyky, kde není nutné specifikovat, jak se má docílit výsledku, ale jenom co se má provádět. Dále existují i jazyky, které nemají textovou podobu, případně, pokud ji mají, tak není určena pro programátory — program se zapisuje pomocí grafických primitiv a jejich vzájemným provázáním. Takovým jazykům říkáme deklarativní. Řada z nich si vydobyla poměrně stabilní místo na programátorském nebi, jsou však i takové, bez kterých už si ani jisté činnosti neumíme představit (např. SQL).

Tato publikace dává nahlédnout na vybrané kategorie deklarativních jazyků, seznamuje s principy, na kterých jsou založeny, provádí srovnání s jazyky imperativními. Jelikož se často můžeme setkat s formální bází takových jazyků, tak výklad těchto formalismů omezíme na absolutní minimum a budeme odkazovat do literatury s tím, že předpokládáme, že si čtenář potřebné formalismy dostuduje z doporučených pramenů sám.

V publikaci se tak setkáme s jazyky funkcionálními, logickými, databázovými a jazyky s grafickým rozhraním. Nejvíce se budeme věnovat prvním dvěma.

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 jakýsi 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. K jejich zadání bude jako jakékoliv jiné, ale krom 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

Koncepce textu

Čas potřebný ke studiu: 16 hodin 10 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 seznámit studujícího s oblastí deklarativních programovacích jazyků. Modul se zaměřuje na jazyky funkcionální a logické. Ty studuje z hlediska možností, typických vlastností, možnosti užití apod., přičemž dává nahlédnout na klíčové prvky, které se pojí s implementací překladačů či interpretů takových jazyků. Dále modul zmiňuje jazyky pro databázové operace a jazyky s grafickým programátorským rozhraním.
(Z tohoto pohledu je možné modul považovat za do jisté míry encyklopedický.)
Po ukončení studia modulu:

- budete schopni klasifikovat jisté třídy deklarativních programovacích jazyků;
- budete schopni zvážit jejich užití pro daný problém;
- bude mít znalosti o formálních aparátech spojených s programovacími jazyky, které tak můžete samostatně dále rozvíjet;
- budete schopni (v případě dalších znalostí z jiných oborů) rozvíjet potřebné teoretické znalosti, které stojí v základu těchto jazyků tak, abyste mohli případně implementovat překladače těchto jazyků, nebo je intenzivně používali pro svoji práci.

Průvodce studiem

Modul začíná obecným úvodem do deklarativních programovacích jazyků, výčtem sledovaných tříd. Po té se zaměřuje na jazyky funkcionální (prezentuje formální bázi jazyka v nejmenší možné formě) a logické. Snaží se vypíchnout jejich charakteristické vlastnosti a zprostředkovat jejich výjimečné vlastnosti, které jsou u jiných jazyků ojedinělé, nebo zcela chybějí.
Následovat bude zmínka o jazycích jiných tříd a vše bude uzavřeno jakýmsi shrnutím a porovnáním s imperativním paradigmatem.
Pro studium modulu je důležité být aktivním programátorem a, ideálně, nově nabyté znalosti prakticky verifikovat. Aktivní užití některého z typických představitelů dané kategorie se tak stává jasnou výhodou. Nestačí pochopit jen to, jak jazyk vypadá zvenčí, ale i co se skrývá za tím, že se chová tak, jak se chová.

Tento modul úzce souvisí s moduly, které předkládají problematiku teorie formálních jazyků a automatů a dále problematiku zpracování formálních jazyků (překladačů/interpretů). Dále navazuje na předchozí moduly, které hovoří o jazycích imperativních a objektově orientovaných.

Návaznost na předchozí znalosti

Pro studium tohoto modulu je tedy nezbytné , aby studující měl základní znalosti ze zpracování formálních jazyků, architektury počítačů, assembleru (jazyk symbolických instrukcí) a, jak již bylo zmíněno, byl aktivním programátorem alespoň v jednom vyšším programovacím jazyce. Krom toho by měl absolvovat, nebo měl odpovídající znalosti z předcházejících modulů na téma programovacích jazyků (jazyk imperativní a jazyky objektově orientované).

1.1 Koncepce modulu

Následující kapitola provádí základní rozdělení a definici deklarativních programovacích jazyků.

Další kapitoly se potom postupně zabývají jazyky funkcionálními, jazyky logickými a následují ostatní typy deklarativních jazyků a shrnutí tématu. Vždy se text snaží o jakousi paralelu s imperativním programováním, aby do sebe vše zapadlo a bylo možné odlišit, v čem se tato dvě paradigmata zásadně liší.

Ve dvou kapitolách narazíme na teoretické báze pro dané třídy deklarativních programovacích jazyků, přitom jedné se budeme věnovat u druhé budeme předpokládat hluboké studium v rámci jiných, matematických, modulů. U probíraného teoretického konceptu se však dostaneme pouze k prvotní definici, takže pro celkové pochopení a nastudování doporučujeme čtenáři další literaturu k dostudování.

1.2 Potřebné vybavení

Pro studium a úspěšné zvládnutí tohoto modulu není třeba žádné speciální vybavení. Je však *vhodné* doplnit text osobní zkušeností s nějakým reprezentantem dané skupiny jazyků. Potom je ovšem nutné mít přístup k odpovídající výpočetní technice a k zdrojům nabízející překladače, či interprety (které doporučujeme) daných jazyků, což je v současnosti typicky Internet.

Další zdroje

Programovací jazyky jsou jednotlivě představovány v řadě publikací. Podobně i formalismy a otázky spojené s během programu, či jeho analýzou můžeme najít v řadě publikací. Pro zástupce hlouběji probíraných jazyků to jsou např. Nilsson, U., Maluszynski, J.: Logic, Programming and Prolog (2ed), nebo Thompson, S.: Haskell, The Craft of Functional Programming. Pro teorii kolem překladačů či programovacích jazyků potom je to např. Aho, Sethi, Ullman: Compilers—Principles, Techniques, and Tools, nebo Sebesta: Concepts of Programming Languages, či Reynolds: Theories of Programming Languages. Pro teorii zmíněnou v kapitole funkcionálních jazyků např. skripta Češka, M., Motyčková, L., Hruška, T.: Vyčíslitelnost a složitost.
V textu jsou také u klíčových termínu uváděny 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

Úvod

Tato kapitola prezentuje pojem deklarativních jazyků, specifikuje některé jejich třídy, které jsou dále rozpracovány v navazujících kapitolách.

Čas potřebný ke studiu: 2 hodiny 20 minut.

Cíle kapitoly

Cílem kapitoly je naučit čtenáře definovat a rozpoznat deklarativní programovací jazyky a odlišit je od jazyků imperativních.
Budou prezentovány čtyři kategorie jazyků, které mají jiný přístup k deklarativnímu pojetí programování. Tím dojde k vymezení pojmu deklarativních programovacích jazyků jako takových. Jedná se pouze o úvodní náhled, který bude dále rozvíjen, proto se jedná o důležitou kapitolu, kterou je nutné velmi dobře zvládnout.

Průvodce studiem

Tato kapitola předpokládá dobrou orientaci v základních pojmech z klasifikace a popisu imperativních programovacích jazyků a terminologií s tím spjatou. Na tyto pojmy navazuje ve srovnání s paradigmatem deklarativním. Neznalost pojmů z imperativního programování bude působit těžkosti v pochopení paradigmatu deklarativního zejména v pasážích, kde dochází ke srovnání.
Ke studiu této kapitoly není třeba žádného speciálního vybavení. Jako u většiny ostatních se jedná o to si zapamatovat danou terminologii a pochopit klíčové principy předkládaných konceptů. Pro hlubší studium, nebo bližší objasnění prezentovaných skutečností je vhodné využít informací na Internetu a vlastní praktické ověření vlastností na konkrétním jazyce.

Obsah

Výklad

2.1 Úvod k deklarativním jazykům

imperativní paradigma

Imperativní programovací jazyky vyžadují, aby programátor specifikoval

- co se má zpracovat,
- jak se to má zpracovat.

Zpracování se potom odehrává jako postupná modifikace vnitřního stavu programu, krok za krokem. U některých jazyků je standardem dokonce dáno, kdy je dosaženo jistého sekvenčního bodu výpočtu a kdy je známo, co která proměnná obsahuje za hodnotu (např. pro jazyk C je středník to, co definuje kroky výpočtu, mezi nimi je možné provádět prakticky libovolné optimalizace).

deklarativní paradigma

Deklarativní paradigma přitom vyžaduje, aby programátor specifikoval

- co se má zpracovat.

Interní stav výpočtu a pořadí vyhodnocení jednotlivých částí programu je tak pro nás „neznámé“. Slovo neznámé je v uvozovkách proto, že základní vyhodnocovací strategie je pochopitelně programátorovi známa, jinak by nebyl schopen program sestavit, ale nelze hovořit o modifikaci stavu tím jak je program vykonáván. Postup vyhodnocení je tedy určen strategií a detailněji potom překladačem, či aktuálním stavem výpočtu (jak uvidíme dále). Jelikož tyto jazyky nemají sekvenci kroků, tím méně potom cyklus. Opakování nějakého výpočtu je proto definováno výhradně jako rekurze.

význam rekurze

Mezi deklarativní jazyky řadíme zejména tyto typy programovacích jazyků:

- (čistě) funkcionální jazyky
- logické jazyky (i když formálně sem spadají jen některé)
- jazyky pro definici a manipulaci s daty (DDL a DML — Data Definition Language, Data Manipulation Language)
- některé jazyky s grafickým uživatelským rozhraním
- některé jazyky, či jejich části, pro popis hardware
- a jistě i další

2.2 Typy deklarativních jazyků

Čistě funkcionální jazyky

Mezi jazyky funkcionální, či přesněji a abychom jasně vyjádřili deklarativitu takového jazyka, čistě funkcionální, řadíme např. tyto jazyky:

- Haskell,
- Miranda,
- Gofer,
- FP,
- Hope,
- Orwell.

Jejich formální bází je λ-kalkul, na který lze každý funkcionální jazyk přeložit. Jeho výrazové schopnosti jsou na stejné úrovni jako např. Turingova stroje či rekurzivně spočetných jazyků (jazyky typu O Chomského hierarchie).

Základní stavební jednotkou jak pro řízení toku vyhodnocení, tak pro *data* je zde funkce, v co nejpřísnějším matematickém pojetí z hlediska vlastností — tj. pro danou kombinaci parametrů vrací vždy stejnou výslednou hodnotu.

Řešený příklad

Zadání: Srovnejte implementaci faktoriálu v jazyku C a jazyku Haskell.
Řešení: Implementace faktoriálu v jazyku C:

```
int ftrl(int n) {
  int result = 1;
  if (n<0) {
    error("Negative input");
    exit(1);
  }
  while (n>1) {
    result *= n;
    n--;
  }
  return result;
}
```

A pro srovnání implementace tatáž funkce v jazyku Haskell:

```
ftrl n | n < 0 = error "Negative input"
       | n < 2 = 1
       | otherwise = n * ftrl (n-1)
```

Jak vidíme, tak ač je jazyk Haskell pro řadu z nás neznámý, tak zápis výpočtu je čitelný i tak. Oproti jazyku C působí zcela určitě učesaněji, elegantněji a zejména je kratší. Pro ty, co by chtěli namítat, že používá rekurzi, lze dodat, že řada překladačů funkcionálních jazyků je optimalizujících a krom toho úpravou na dopředně rekurzivní funkci by to převedl na cyklus i neoptimalizující, nebo méně optimalizující překladač.

Podobně, jak v řešeném příkladu, bychom mohli začít srovnávat další. Určitě budeme srovnávat nesrovnatelné, ale zcela jistě bychom viděli velkou vyjadřovací schopnost takových jazyků. Proto nikoliv pro srovnání demonstrace dvou dalších funkcí v jazyku Haskell. Nejdříve funkce realizující algoritmus quick-sort v seznamu:

```
qsort [] = []
qsort (x:xs) = qsort [y | y<-xs, y<x]
               ++ [x] ++
               qsort [y | y<-xs, y>=x]
```

Prázdný seznam netřeba řadit, zbytek rozdělíme na dvě části podle mediánu (bereme první prvek) a rekurzivně seřadíme.

Jako další, jistou „demonstraci síly“, prezentujme konstantu obsahující celou Fibonacciho posloupnost. Určitě jen teoreticky, ale díky strategii vyhodnocení jazyka Haskell se vypočte jen tolik členů, kolik je třeba, takže k přeplnění paměti nedojde:

```
allfibs = 0:1:
          [allfibs!!n + allfibs!!(n+1) | n <- [0..]]
```

Algoritmus napevno definuje první dva prvky. Další již vypočítává z nich, přesně dle definice. Velice úsporně zapsáno a přitom je algoritmus i efektivní, neboť má zajímavou časovou složitost pro určení n-tého členu posloupnosti.

Logické jazyky

Typickými představiteli logických programovacích jazyků jsou tyto:

- Prolog

- Parlog
- Gödel
- CLP($\mathcal{R}$)
- KL1 (KLIC)

I když bychom našli i další jazyky, zejména odvozené od Prologu, ale poskytující řadu vylepšení.

Formální bází těchto jazyků je predikátová logika, přesněji její podmnožina, protože ne všechny konstrukce z predikátové logiky jsou v těchto jazycích zachyceny. I přesto je jejich výpočetní síla stejná, jako u jakýchkoliv jiných jazyků jako C, Pascal, či třeba Haskell.

Základními stavebními bloky jsou entity jako klauzule, predikáty, termy apod. Z nich se sestavuje celý program a na základě vyhodnocovací strategie potom dochází k vyhodnocení — logickému odvození, kdy musejí být splněna stejná pravidla, jako by se odvození dělo „ručně“ na základě pravidel predikátové logiky.

I když logické programovací jazyky nejsou určeny pro početní operace, tak budeme demonstrovat v Prologu, jak by se dala realizovat funkce pro výpočet faktoriálu:

```
ftrl(N,_) :- N<0, !, fail.
ftrl(N,R) :- N<2, R=1, !.
ftrl(N,R) :- NN is N-1, ftrl(NN, RR), R is RR*N.
```

Opět vidíme jasné využití rekurze, ale na druhou stranu, pro neznalého „pozorovatele“, je patrné, že v jazyku jsou jisté prvky, které nejsou hned na první pohled čitelné. To je, v tomto případě, jistá daň za to, že Prolog byl navržen počátkem 70. let dvacátého století. Nicméně i tak je zápis poměrně stručný.

Co se týká dalších ukázek, podobně jako u jazyka Haskell, tak zajímavý je kód pro řadicí algoritmus quick-sort. Kompletní Fibonacciho posloupnost by sice šlo asi vytvořit, ale stejně, jako v jazyku C, by to bylo poměrně náročné. Algoritmus pro quick-sort je tento:

```
quick([],[]).
quick([H|T], S) :- split(T, H, A, B),
                   quick(A, A1), quick(B, B1),
                   append(A1,[H|B1],S).
split([], _, [], []).
split([X|X1], Y, [X|Z1], Z2) :-
                             X < Y, split(X1, Y, Z1, Z2).
split([X|X1], Y, Z1, [X|Z2]) :-
                            X >= Y, split(X1, Y, Z1, Z2).
```

Vidíme, že oproti jazyku Haskell přibyl predikát `split`, který ověřuje rozděleni seznamu řazeného dle mediánu (opět první prvek), na dva. Ty jsou potom seřazeny a následně spojeny do jednoho. Zajímavý je právě řádek, kde dochází k řazení dvou částí rozděleného seznamu. Ty totiž mohou probíhat zcela nezávisle a tudíž paralelně. Ve verzi Prologu, která se jmenuje Parlog, je paralelismus na prvním místě a právě v něm by se tato konstrukce s úspěchem uplatnila.

Jazyky pro definici a manipulaci dat

U jazyků pro definici a manipulaci dat sice existuje jeden hodně známý zástupce, přesto si uvedeme, alespoň pro relační databáze i další možnosti:

- SQL (a jeho varianty a z něj odvozené jazyky)
- QUEL (Query Language)
- QBE (Query By Example)

Některé z jazyků v této kategorii jsou jak jazyky pro manipulaci, tak i definici dat. Někdy mají však jen jednu z těchto funkcí.

Nějaký formální základ může u těchto jazyků chybět, ale stejně tak to může být relační algebra či kalkul, nebo i jiný formální prvek, který tyto nějak doplňuje, či rozšiřuje.

Mezi typické vlastnosti patří, že tyto jazyky mají vazbu na přirozený jazyk, často tedy angličtinu. Příkaz, nebo častěji dotaz zapsaný v tomto jazyku potom často pasivně dekódujeme poměrně dobře. Aktivní použití je však často horší.

Další poměrně charakteristickou vlastností je, že tyto jazyky většinou *nejsou výpočetně úplné.* Takže v nich nelze popsat libovolný algoritmus. Jak je ale vidět z praxe, tak to ničemu nebrání v tom, aby byly široce nasazovány a využívány.

výpočetní úplnost

Jako ukázku kódu zvolíme dotaz v jazyku SQL v mutaci pro server Oracle s využitím jeho objektově-relační nástavby pro podporu práce s prostorovými daty:

```
SELECT
  A.GID,
  SDO_GEOM.SDO_AREA(A.geometry, MD.DIMINFO)
FROM
  STUFF A, STUFF C,
  USER_SDO_GEOM_METADATA MD
WHERE
  C.GID='Table' AND MD.table_name='STUFF' AND
  A.GID<>C.GID AND
```

```
sdo_relate(A.geometry, C.geometry,
        'mask=ANYINTERACT querytype=WINDOW') = 'TRUE';
```

Volně „přeloženo“ požadujeme vypsat všechny věci a jejich plochu, které leží, nebo se dotýkají stolu (Table). Tento příklad byl zvolen záměrně, aby bylo vidět, jak různá rozšíření mění původní koncept, takže se stává kód i pro znalce SQL prakticky nečitelný.

Jazyky s grafickým rozhraním

U jazyků s grafickým rozhraním máme na paměti především ty jazyky, kdy program, respektive jeho zdrojový tvar je v grafické formě a textová reprezentace se buď nepoužívá, nebo vůbec neexistuje. Typicky sem patří jazyky pro popis toku dat či signálů (data flow, signal flow). Takové jazyky ani nemusejí mít jméno, nebo přejímají jméno podle vývojového prostředí, ve kterém se používají. Podobné rozhraní najdeme například v nástrojích *SCADE Suite*, či *Simulink* pro *Matlab*.

Formálně jsou tyto systémy založeny na reaktivních systémech, komunikujících sekvenčních procesech či jiných matematicko-analytických formalismech. Velmi často, zvláště u systémů, které naplňují některé standardy (např. *SCADE*), jsou tyto formalismy velmi precizně vymezeny a zejména vazba programu na ně.

Mezi typické vlastnosti patří, jak lze očekávat, manipulace s grafickými entitami, jejich kombinace a propojování liniovými entitami podobně, jako ve vektorovém grafickém editoru. Typickou ukázku lze nalézt na domovských stránkách výše zmíněných produktů. Pro první představu lze za podobný typ jazyka považovat i deterministický konečný automat v grafické notaci a že tomu tak opravdu je, je možné se přesvědčit u řady produktů.

Pojmy k zapamatování

V této kapitole patří mezi pojmy k zapamatování především imperativní a deklarativní paradigma, jaký je mezi nimi rozdíl, jak se liší model vyhodnocení. No a potom jsou to jednotlivé typy deklarativních jazyků s důrazem na blíže popsané: funkcionální jazyky, logické jazyky, jazyky pro definici a manipulaci s daty, jazyky s grafickým uživatelským rozhraním a jazyky, či jejich části, pro popis hardware. Znalost rozčlenění a typických představitelů je vstupní bránou pro další porozumění textu a zejména orientaci v praxi.

Závěr

Cílem kapitoly bylo definovat deklarativní paradigma a jeho srovnání s paradigmatem imperativním. Nyní byste měli být schopni programovací jazyk jasně vsadit do jednoho z těchto paradigmat. Dalším cílem potom bylo definovat nejmarkantnější typy deklarativních jazyků a u vybraných z nich provést detailnější obeznámení s jejich charakterem. Zejména tyto typy byste měli být také schopní přímo detekovat.

Úlohy k procvičení:

1. Které algoritmy (i/zejména jednoduché) nelze realizovat v jazyku SQL?
2. Součástí jazyka UML jsou i stavové diagramy. Tak, jak jsou navrženy v UML (2.0 a výše), díky vazbám na okolí, jsou výpočetně úplné? Jak byste realizovali algoritmus pro quick-sort?
3. Zvolte nějaký imperativní jazyk a nějaký deklarativní jazyk (funkcionální, či logický) a zkuste v obou implementovat nějaký jednoduchý, ale pro vás neznámý algoritmus, nějakou hříčku. Nedbejte na větší časovou náročnost v deklarativním jazyku, to je samozřejmé. Srovnejte však užité techniky v programování!

Klíč k řešení úloh

1. Jaký typ definice SQL chybí, který je velmi důležitý pro vyjádření jisté typické programové konstrukce?
2. Zvažte vazbu i na teoretické okolí, třeba v rámci nějaké implementace. Algoritmus nejprve zakreslete pomocí vývojového diagramu.
3. Volte zejména algoritmy na výpočet nějakých hodnot, či nalezení nějakých hodnot.

Další zdroje

Další informace k této kapitole najdete na internetu, nebo přímo u jednotlivých představitelů různých typů deklarativních jazyků — na WWW stránkách, v manuálech, tutoriálech, či v diskusních fórech.

Kapitola 3

λ-kalkul

Lehký a stručný úvod do λ-kalkulu, který je formální bází funkcionálních programovacích jazyků, je jediným obsahem této kapitoly.

Čas potřebný ke studiu: 4 hodiny 40 minut.

Cíle kapitoly

Po zvládnutí této kapitoly byste měli být schopni rozpoznat výrazy zapsané v λ-kalkulu, měli byste znát základy tvorby syntakticky správných výrazů v něm a pasivně, na jednoduchých příkladech, provádět redukce výrazů v λ-kalkulu.
Cílem je, abyste si vytvořili dostatečný základ pro případné další rozšíření znalostí tohoto formalismu, jenž je využíván i v jiných odvětvích informatiky (např. denotační sémantika).

Průvodce studiem

Před tím, než se budeme zabývat funkcionálními jazyky, seznámíme se s teoretickým základem těchto jazyků, abychom vytvořili odrazový můstek pro další studium v dané kategorii. Krom toho objevíte nový způsob chápání pojmu funkce v programech, k čemuž je λ-kalkul ideální.
Ke studiu této kapitoly bude stačit papír a tužka, případně pomocná literatura ať klasická, či na Internetu (potom tedy potřebujete počítač s prohlížečem WWW stránek a připojení k Internetu). Jako u většiny ostatních se jedná o to si zapamatovat danou terminologii, zvládnout prezentované mechanismy a techniky a zapamatovat si důležité a podstatné věci v kapitole. Pro hlubší studium, nebo bližší objasnění termínů, je však vhodné využít informací na Internetu, či v další literatuře.

Obsah

Výklad

3.1 Syntaxe λ-kalkulu

Formálně lze syntaxi λ-kalkulu pomocí BNF definovat takto:

Definice 3.1.1 **Definice!**

```
<λ-výraz>  ::=  proměnná
             |  (<λ-výraz> <λ-výraz>)
             |  (λ<proměnná> . <λ-výraz>)
```

Přitom proměnné, pokud se nejedná o nějaké konkrétní, budeme označovat V, λ-výrazy označíme jako E. Konkrétní proměnné potom malými písmeny z konce abecedy, např. x, y, z.

Jak vidíme, tak se jedná o bezkontextovou gramatiku. Stejně tak je patrné, že celý kalkul má pouze tři konstrukty. Jako ukázku jednoduchých výrazů můžeme např, uvést:

- Výraz reprezentující identitu:

$$(\lambda x.x)$$

- Výraz provádějící prohození pořadí aplikace svých argumentů:

$$(\lambda x.(\lambda y.(y\ x)))$$

- Výraz pro opakované aplikování funkce:

$$(\lambda f.(\lambda x.((f\ x)\ x))$$

Proměnné

Proměnné v λ-kalkulu jsou jako každé jiné, definují vazbu s okolím a reprezentují pojmenované entity uvnitř výrazu. Na základě pravidel uvedených níže je možné ji v rámci λ-výrazů zaměňovat za jiné proměnné, či za jiné výrazy.

Aplikace

λ-výraz tvaru `(<λ-výraz> <λ-výraz>)` nazýváme λ-aplikace, či aplikace, pokud nemůže dojít k záměně. Máme-li v λ-kalkulu dva výrazy, E_1 a E_2, potom, je-li to jinak možné, tak je možné jeden aplikovat na druhý. Pokud zvolíme aplikaci v tomto pořadí (E_1 E_2), tak výraz (nebo úplně λ-výraz) E_1 je operátorem (rator) a E_2 je operandem (rand) v dané aplikaci.

Abstrakce

Výraz tvaru (λ<proměnná> . <λ-výraz>) nazýváme λ-abstrakce, či abstrakce. Tyto výrazy reprezentují funkce s jednou vázanou proměnnou (hlavičkou abstrakce) a tělem, které je opět tvořeno λ-výrazem; pokud nějaká operace vyžaduje více parametrů, tak bezprostředním vnořením λ-abstrakcí dosáhneme výsledku. Toto je jediný λ-výraz, který je možné aplikovat (účelně).

Volné a vázané proměnné

Proměnné především zajišťují vazbu mezi výrazem a okolím. Vazbu proměnné definuje hlavička abstrakce, přičemž proměnná je vázána nejbližší příslušnou hlavičkou nalevo od svého výskytu.

Definice!

Definice 3.1.2 *Volné a vázané proměnné si definujme na následujících příkladech:*

$$(\lambda x.y_{\text{volná}}x_{\text{vázaná}})(\lambda y.x_{\text{volná}}y_{\text{vázaná}})$$

$$\lambda \underbrace{x\,.\,x}_{\text{vazba}}\; y\,(\lambda \underbrace{x\,.\,x}_{\text{vazba}})\,y$$

Přitom vazba se provádí vždy nejtěsnější lambdou vlevo od výskytu proměnné.

Řešený příklad

Zadání: U zadaného výrazu vyznačte která proměnná je vázána kterou hlavičkou λ-abstrakce.

$$(\lambda f.((x\ (\lambda f.(\lambda x.((f\ x)\ y)(f\ x))))\ f)\ w)$$

Řešení:

$$(\lambda\ \text{f}.\ ((\mathbf{x}\ (\lambda\ \mathbf{f}.\ (\lambda\ \mathbf{x}.\ ((\mathbf{f}\ \mathbf{x})\ \mathbf{y})\ (\mathbf{f}\ \mathbf{x}))))\ \text{f})\ \mathbf{w})$$

- Proměnné **x**, **y**, **w** jsou v daném výrazu zcela volné.
- Hlavička λ **x** váže všechny výskyty **x**.
- Hlavička λ **f** váže všechny výskyty **f**.
- Hlavička λ f váže všechny výskyty f.

3.2 Konvence v λ-kalkulu

Jak jste si mohli všimnout už třeba v posledním řešeném příkladu, tak počet závorek už u poměrně jednoduchých výrazů je poměrně vysoký. Proto byly zavedeny konvence, které umožňují počet závorek rozumně redukovat.

Definice 3.2.1 *Aplikace je vždy zleva asociativní, takže zápis* $((\ldots(E_1\ E_2)\ldots)E_n)$ *je možné zkrátit na* $E_1\ E_2 \ldots E_n$. **Definice!**

Takže například platí:

- $E_1\ E_2$ lze použít namísto $(E_1\ E_2)$
- $E_1\ E_2\ E_3$ lze použít namísto $((E_1\ E_2)\ E_3)$
- $E_1\ E_2\ E_3\ E_4$ lze použít namísto $(((E_1\ E_2)\ E_3)\ E_4)$
- atd.

Definice 3.2.2 *Dosah hlavičky abstrakce „*λV*“ sahá tak daleko doprava, jak to je jen možné, takž zápis* $(\lambda V.(E_1\ E_2 \ldots E_n))$ *je možné zkrátit na* $\lambda V.E_1\ E_2 \ldots E_n$. **Definice!**

Jak vidíme, tak v definici 3.2.2 se již uplatnila zjednodušení definovaná v 3.2.1. V poslední definici umožňující zjednodušení vnořeným λ-abstrakcí.

Definice 3.2.3 *Bezprostředně vnořené* λ*-abstrakce je možné zřetězit, takže zápis* $(\lambda V_1.(\ldots(\lambda V_n.E)\ldots))$ *je možné zkrátit na* $\lambda V_1 \ldots V_n.E$. **Definice!**

Tak například platí:

- $\lambda\ x\ y.E$ lze použít namísto $(\lambda\ x.(\lambda\ y.E))$
- $\lambda\ x\ y\ z.E$ lze použít namísto $(\lambda\ x.(\lambda\ y.(\lambda\ z.E)))$
- $\lambda\ x\ y\ z\ w.E$ lze použít namísto $(\lambda\ x.(\lambda\ y.(\lambda\ z.(\lambda\ w.E))))$
- atd.

Vezmeme-li v úvahu všechny konvence, tak výraz:

$$(\lambda f.((x\ (\lambda f.(\lambda x.((f\ x)\ y)(f\ x))))\ f)\ w)$$

lze zjednodušit na

$$\lambda f.x\ (\lambda\ f\ x.(f\ x\ y)(f\ x))\ f\ w$$

Aby bylo možné v λ-kalkulu s výrazy lépe pracovat, tak byla zavedena notace, která umožňuje pojmenovat nějaký výraz a toto nové jméno potom užívat namísto takového výrazu. Píše se to takto:

$$\texttt{LET}\ \sim = \lambda\text{-výraz}$$

kde $\sim$ reprezentuje nové označení a často se pro něj používá tučné, či podtržené písmo.

Příkladem takové definice může být trojice kombinátorů (λ výrazů, na které lze převést jakýkoliv jiný λ-výraz):

- $\texttt{LET}\ \mathbf{S} = \lambda\ f\ g\ x.(f\ x)(g\ x)$
- $\texttt{LET}\ \mathbf{K} = \lambda\ x\ y.x$
- $\texttt{LET}\ \mathbf{I} = \mathbf{S}\ \mathbf{K}\ \mathbf{K}$

3.3 Redukce/konverze

Aby bylo možné nějak měnit λ-výrazy, byla stanovena pravidla, která určují, jaké změny jsou povoleny a kdy je možné je provést. Celkem jsou povoleny 3 typy těchto transformací, kterým se v λ-kalkulu říká redukce, či konverze.

substituce
platnost

- α-konverze: libovolná abstrakce tvaru $\lambda V.E$ může být redukována na abstrakci $\lambda V'.E[V'/V]$. Zápis $E[V'/V]$ označuje substituci proměnné V' za volné výskyty proměnné V ve výrazu E, přičemž substituce musí být *platná*. Pojmem platná rozumíme, že při substituci $E[E'/V]$ se žádná volná proměnná ve výrazu E' nestane vázanou.
- β-konverze: libovolná aplikace tvaru $(\lambda V.E_1)E_2$ může být redukována na $E_1[E_2/V]$, pokud je substituce platná.
- η-konverze: libovolná abstrakce tvaru $\lambda V.(EV)$, kde V není volné v E, může být redukována na E.

Abychom označili, že nějaká redukce se provedla podle určité konverze, tak píšeme:

- $E_1 \xrightarrow[\alpha]{} E_2$ pokud se redukce provedla pomocí α-konverze
- $E_1 \xrightarrow[\beta]{} E_2$ pokud se redukce provedla pomocí β-konverze
- $E_1 \xrightarrow[\eta]{} E_2$ pokud se redukce provedla pomocí η-konverze

Řešený příklad

Zadání: Určete platnost uvedených redukcí.

1. $\lambda\, x.x \xrightarrow[\alpha]{} \lambda\, y.y$
2. $\lambda\, x.f\ x \xrightarrow[\alpha]{} \lambda\, y.f\ y$
3. $\lambda\, x.\lambda\, y.F\ x\ y \xrightarrow[\alpha]{} \lambda\, y.\lambda\, y.F\ y\ y$
4. $(\lambda\, x.f\ x)\ E \xrightarrow[\beta]{} f\ E$
5. $(\lambda\, x\ y.F\ x\ y)\ E \xrightarrow[\beta]{} (\lambda\, y.F\ E\ y)$
6. $(\lambda\, x\ y.F\ x\ y)\ (E\ y) \xrightarrow[\beta]{} (\lambda\, y.F\ (E\ y)\ y)$

Řešení: Konverze pod čísly 1, 2, 4 a 5 jsou v pořádku, zatímco konverze 3 a 6 jsou chybné, neboť substituce u nich provedené nejsou platné.

Výraz, který je možné podle nějaké redukce změnit budeme nazývat *redex* podle zkratky z anglických slov „reducible expression“. Jedná-li se o redex podle příslušné konverze, tak se nazývá α-, β-, či η-redexem. redex

Výklad

3.4 Relace v λ-kalkulu

Nad jednotlivými výrazy lze definovat relace, které takové dva výrazy uvádějí do jistého vztahu.

Existence konverzí tak umožňuje definovat rovnost a identitu dvou λ-výrazů. Neformálně, dva výrazy jsou identické, pokud je jejich textový zápis shodný, zatímco jsou si rovny, pokud je možné posloupností konverzí převést tyto na takové tvary, které jsou identické.

Definice 3.4.1 *Dva λ-výrazy, E_1 a E_2 jsou identické, pokud jsou zapsány toutéž posloupností znaků, píšeme: $E_1 \equiv E_2$* **Definice!**

Definice 3.4.2 *Jsou-li E a E' dva λ-výrazy, potom jsou si rovny, píšeme $E = E'$, pokud buďto $E \equiv E'$, nebo existují takové výrazy $E_1, \ldots, E_n$, že:* **Definice!**

- $E \equiv E_1$
- $E' \equiv E_n$

- $\forall i \in \{1, \ldots, n\} : E_i \xrightarrow[\alpha]{} E_{i+1} \vee E_i \xrightarrow[\beta]{} E_{i+1} \vee E_i \xrightarrow[\eta]{} E_{i+1} \vee E_{i+1} \xrightarrow[\alpha]{} E_i \vee E_{i+1} \xrightarrow[\beta]{} E_i \vee E_{i+1} \xrightarrow[\eta]{} E_i$

Relace $\rightarrow$ je potom, neformálně, rovnost omezená na jednosměrnou konverzi (všechny konverze se např. provádějí pouze z E_i na E_{i+1}). Formálně potom budeme definovat takto.

Definice!

Definice 3.4.3 *Jsou-li E a E' dva λ-výrazy, potom $E \rightarrow E'$, pokud buďto $E \equiv E'$, nebo existují takové výrazy $E_1, \ldots, E_n$, že:*

- $E \equiv E_1$
- $E' \equiv E_n$
- $\forall i \in \{1, \ldots, n\} : E_i \xrightarrow[\alpha]{} E_{i+1} \vee E_i \xrightarrow[\beta]{} E_{i+1} \vee E_i \xrightarrow[\eta]{} E_{i+1}$

---

zobecněná substituce

Aby bylo možné všechny konverze provádět bez obav z neplatné substituce, definuje se *zobecněná substituce*.¨ Ta, neformálně, říká, že kdykoliv by se během substituce mohl objevit konflikt, který by zneplatnil substituci, tak se provede ještě před tím α-konverze, která konfliktní proměnné přejmenuje na nekonfliktní.

Řešený příklad

Zadání: Pro definice:

- LET **True** $= \lambda\ x\ y.x$
- LET **False** $= \lambda\ x\ y.y$
- LET **Not** $= \lambda\ t.t$ **False True**

Ukažte redukci výrazu **Not True**.
Řešení:

$$\begin{aligned} \textbf{Not True} &= (\lambda\ p.p\ \textbf{False True})\ \textbf{True} \\ &= \textbf{True False True} \\ &= (\lambda\ x\ y.x)\ \textbf{False True} \\ &= (\lambda\ y.\textbf{False})\ \textbf{True} \\ &= \textbf{False} \end{aligned}$$

3.5 Rekurze v λ-kalkulu

Zápis LET **F** $= \ldots$ **F** $\ldots$ nelze v λ-kalkulu ze zřejmých důvodů použít (nelze se odkázat na něco, co není definováno — např. makra

v jazyku C). Pro řešení situace se využívá operátor pevného, např. $\mathbf{Y}$. Ten je definován tak, že platí $\mathbf{Y}\ E\ =\ E\ (\mathbf{Y}\ E)$.

Máme-li potom pracovní definici nějakého výrazu ve tvaru:

$$\mathbf{f} a_1 \ldots a_n\ =\ \ldots\ \mathbf{f}\ \ldots$$

tak korektní λ výraz a definici utvoříme takto:

$$\texttt{LET}\ \mathbf{f}\ =\ \mathbf{Y}\,(\lambda\ a_1 \ldots a_n . \ldots .\ \mathbf{f}\ \ldots)$$

Užití a další vlastnosti a rozšíření λ-kalkulu již sahají za rámec této publikace.

Pojmy k zapamatování

Pojmů v této kapitole je poměrně dosti, všechny se pojí s tím základním — λ-kalkul. Kromě jejich znalosti, znalosti jejich definic je naprosto nutné umět i *pasivně* tento kalkul používat, tj. umět ho číst a u předložených jednoduchých konstrukcí umět α- a β-redukovat. Z pojmů, které je tedy nutné umět připomeňme alespoň následující: abstrakce, aplikace, konverze, redukce, redex, substituce, zobecněná substituce, volná/vázaná proměnná, identita, rovnost, relace $\rightarrow$, rekurze.

Závěr

Cílem kapitoly bylo seznámit čtenáře s teoretickou bází funkcionálních programovacích jazyků. Jednak proto, že jsou to představitelé deklarativního programovaní, jednak proto, že tento formalismus je využíván i v dalších odvětvích informatiky. Po nastudování této kapitoly by čtenář měl být schopen pasivního užití λ-kalkulu a měl by být schopen rychleji přejít k dostudování celého formalismu na libovolné vyšší úrovni.

Úlohy k procvičení:

1. Určete volné a vázané proměnné ve výrazu $\lambda\ x\ y.f\ (x\ (\lambda\ f\ y.f\ y\ y)\ w)\ y$?
2. Na co se redukuje výraz v λ-kalkulu $(\lambda\ x\ y.y)\ u\ v$?
3. Jak se řeší v λ-kalkulu opakování a jak se řeší prakticky?

Klíč k řešení úloh

1. Postupujte dle definice. Ukažte, čím je která proměnná vázána.
2. Uvažte dvojnásobnou β-redukci, postupujte dle definice.
3. Viz příslušnou pasáž v textu.

Další zdroje

Zdroje k λ-kalkulu jsou poměrně snadno nalezitelné jak v papírové, tak elektronické formě, tato stať například čerpala z:

- Češka, M., Motyčková, L., Hruška, T.: *Vyčíslitelnost a složitost*, s. 217, červen 1992, Vysoké učení technické v Brně.
- `http://en.wikipedia.org/wiki/Lambda_calculus`

Spoustu dalších materiálů však lze zcela jistě nalézt i ve fakultních, universitních či vědeckých knihovnách. Počítejte potom s prodloužením doby studia, pochopitelně.

Kapitola 4

Funkcionální programovací jazyky

V této kapitole se seznámíme s charakterem a vlastnostmi funkcionálních programovacích jazyků, jakožto reprezentanta jazyků deklarativních.

Čas potřebný ke studiu: 3 hodiny 40 minut.

Cíle kapitoly

Cílem kapitoly je seznámit čtenáře se základními a charakteristickými vlastnostmi funkcionálních jazyků. Jak vypadají data, jaké jsou typické řídicí struktury, jak jsou takové jazyky zpracovávány, jaké jsou jejich speciality, apod.
Na konci kapitoly by mělo být zřejmé, co lze od takových jazyků očekávat a zejména to, že tyto jazyky nejsou nikterak exotické a dají se použít i pro běžné programování. To že jsou výhodné pro specializované programy, to je asi samozřejmé.

Průvodce studiem

Tato kapitola předpokládá dobrou orientaci v základních pojmech z klasifikace a popisu imperativních programovacích jazyků a terminologií s tím spjatou. Dále předpokládá znalost deklarativního paradigmatu a nezbytně nutné minimum znalosti λ-kalkul.
Ke studiu této kapitoly není třeba žádného speciálního vybavení. Jako u většiny ostatních se jedná o to si zapamatovat danou terminologii a pochopit klíčové principy předkládaných konceptů. Pro hlubší studium, nebo bližší objasnění prezentovaných skutečností je vhodné využít informací na Internetu a vlastní praktické ověření vlastností na konkrétním jazyce.

Obsah

Výklad

4.1 Základní popis

Funkcionální paradigma je poměrně jednoduché:

Definice 4.1.1 *Paradigma funkcionálních programovacích jazyků: Funkce je základním a jediným výrazovým prostředkem pro popis a definici algoritmů i dat.*

Definice!

Mezi hlavními reprezentanty, ostatně zmíněny již v kapitole 2, vyzdvihněme např. jazyk Haskell (do jisté míry následník a pokračovatel jazyka Gofer), který nabízí zajímavou strategii vyhodnocení, tzv. *lazy evaluation* (bez překladu), které umožňuje definici formálně nekonečných datových struktur. Dalším zajímavým jazykem je jazyk Miranda, která má stejnou strategii vyhodnocení, nicméně jeho největší zvláštností je, že se jedná o komerční produkt. Deklarativní je i jistá podmnožina jazyku ML (a jeho derivátů), který nabízí striktní strategii vyhodnocení, která je bližší imperativnímu pojetí vyhodnocení jako např. u jazyků C, či Pascal. Tento jazyk je navíc zajímavý tím, že byl jako první formálně definován.

Formální bází je λ-kalkul, který je v základním pojetí beztypový. Nicméně i k němu je možné zavést typovou teorii a to jak v jednoduchém pojetí, tak zavést typy druhého řádu a dokonce typy vyšších řádů. Podobné modely se uplatňují v práci s typy ve funkcionálních jazycích a je naprosto běžné, že tyto jazyky jsou silně typované, ale přitom typy se neuvádějí explicitně (jazyk C, Java, apod,), ale typy jsou odvozovány (type inference)! (Systém automatického odvození dle typového systému pánů Hindleye a Millnera.) U funkcionálních jazyků se pro tyto účely používá jazyk *core-ML*, což není nic jiného než λ-kalkul v jiném hávu, který obsahuje více textové konstrukce místo různým symbolů, jako např. λ, které nejsou na klávesnici.

automatické typové odvození

Pokud se s těmito jazyky setkáme, tak málokdy narazíme na kompletní popis syntaxe v nějaké více či méně formální podobě. Velmi často je to řešeno na dílčích úrovních, na konkrétních příkladech u vysvětlování konkrétních vlastností. Syntaxe je často velmi jednoduchá, zejména ve srovnání s jazyky imperativními. Někdy však může trpět přemírou oddělovačů. Pokud je syntaxe někde uceleně uvedena, tak je to spíše něco navíc, co je určeno pro experty, kteří se navíc zajímají o překlad a zpracování takových jazyků.

popis syntaxe

U sémantiky je často popis neformální, případně s odkazem na formální bází jazyka, která dodává potřebný dovysvětlující rámec. Pokud

se však podíváme do literatury více odborné, či specifické pro daný, jazyk, nebo jeho překladač, či interpret, tak je možné, že se setkáme s formalismy, které definují sémantiku, nebo ji aspoň vysvětlují. Mezi takové formalismy a často vysvětlované mechanismy patří:

- formální systém automatické typové inference,
- formální důkaz toho, že celý jazyk a jeho zpracování a vyhodnocení je vnitřně bezesporné a správné,
- denotační sémantika (formalizace sémantiky vyhodnocení),

4.2 Data a jejich zpracování

Datové typy ve funkcionálních jazycích najdeme jak velmi jednoduché, skalární, tak složité, složené, variantní i rekurzivní. Předdefinované typy, se kterými se setkáváme přitom nejsou shodné s těmi, co najdeme běžně u imperativních jazyků. Jazyky jak funkcionální, tak deklarativní potom to, co chybí, není přímo v jazyku, doplňují na úrovni knihoven, či abstraktních datových typů.

Mezi bázové typy patří často typy jinde označované jako skalární, nebo z nich přímo odvozené (na implementační úrovni). Jedná se o celá čísla, čísla s plovoucí desetinnou čárkou, celá čísla s proměnlivou velikostí, znaky, apod. U všech těchto typů je typická vazba na hardware a také to, že tyto typy nelze na úrovni jazyka dekódovat (např. implementačními triky z jazyka C, či Pascal).

Kromě základních typů jsou zastoupeny i typy strukturované, což jsou zejména:

- seznamy,
- n-tice.

Seznamy lze vybudovat nad libovolným typem, jak vestavěným, tak uživatelským, což pochopitelně platí i pro funkce, či částečně aplikované funkce (funkce nedostává tolik parametrů, aby byl spuštěn výpočet, který definuje). Velice často, díky silné typovosti, jsou seznamy typem homogenním. Syntaxe je udržována na velmi jednoduché úrovni, často touto formou:

částečná aplikace

```
[1,2,3,4,5]
```

U n-tic se situace liší v tom, že jsou to typicky heterogenní datové struktury, které často nemají explicitní pojmenování, stejně jako seznamy. U beztypových jazyků, nebo jazyků s heterogenními seznamy se nemusejí vyskytovat, neboť jsou nahrazeny právě seznamy. Zápis je různý, ale nikoliv netypická forma je tato:

```
('a', 1, 45.6)
```

Vedle vestavěných mohou existovat i typy uživatelské. Lze nadefinovat prakticky libovolné uživatelské typy, typickou to jsou: uživatelské typy

- výčtové type;
- záznamy, i pojmenované;
- variantní záznamy (jméno u každé varianty);
- rekurzivní datové typy (mohou kombinovat předchozí vlastnosti).

Za všechny si uveďme příklad z jazyku Haskell:

```
data Osoba = Osoba String String Int
data Barva = RGB Int Int Int
           | CMYK Int Int Int Int
           | Grayscale Int
data Strom = List
           | Uzel Int Data Strom Strom
```

Typ `Osoba` je klasickým záznamem s tím, že si musíme pamatovat, co znamenají jednotlivé složky, v našem případě to je jméno, příjmení a věk. (Některé jazyky mají prostředky, jak ošetřit i toto, např. přes typová synonyma.) Typ `Barva` je potom ukázkou variantního záznamu, jinak se stejnými vlastnostmi jako typ `Osoba`. Poslední ukázkou je potom rekurzivní datový typ pro binární strom. Jak vidíme, tak typ `Strom` nese zřejmě celočíselný klíč, data a dva podstromy na vnitřních uzle, zatímco listové uzly jsou prázdné.

Kromě takovýchto typů jsou ve funkcionálních jazycích přítomny i poměrně speciální typy, zastoupené v knihovnách, ale často neimplementovatelné v jazyku samotném. Především se jedná o typy pro realizaci vstupně-výstupních operací. Obtíž je v tom, že bezestavový vyhodnocovací model deklarativního jazyka váží na stavové okolí. V současnosti jsou známé dva hlavní principy, jak se tyto operace realizují: vstupy a výstupy

- kontinuace (continuations) — často se pojí s proudovými (stream) vstupy a výstupy, základní idea je taková, že funkci pro vstup a výstup předám parametry pro realizaci vlastní operace a dále dvě funkce (kontiunuace), kde jedna představuje zbytek programu, co se bude vykonávat, pokud operace sama skončí úspěchem, a druhá reprezentuje zbytek programu, který se bude vykonávat, pokud operace skončí neúspěchem;

- monády (monads) — výstupem funkcí v případě V/V operací implementovaných tímto způsobem jsou akce, aby bylo zachováno základní paradigma funkce, kdy její výsledek je dán kombinací parametrů, nikoliv historií, akce je potom něco, co nám to zaručí, tedy formálně;

pole

Co s týká polí, tak se přímo v jazyce, díky své imperativní povaze, typicky nevyskytují. U jazyků, které nejsou čitě funkcionální (např. ML) se s nimi setkáme jako s tzv. mutovatelnými položkami (je možné měnit část jejich podstruktury, aniž by zbytek byl ovlivněn a je tedy možné z více míst sdílet plný přístup k položce). U čistě funkcionálních, tedy deklarativních jazyků implementace polí často zcela chybí, nebo se k nim přistupuje jako k monádům, případně je nějaká knihovní implementace, která však disponuje, přes řadu příjemných funkcí, i řadou omezení.

typové proměnné

Velmi dobrou vlastností funkcionálních jazyků je možnost implementace generických algoritmů díky polymorfismu na úrovni funkcí. To je umožněno zejména možností mít *typové proměnné* místo konkrétních typů. Konkrétní typ se doplní za proměnnou až podle parametrů v místě aplikace. Nejlepším příkladem je délka seznamu, Nechť `Int` označuje typ celých čísel a `Float` typ pro čísla v plovoucí desetinné čárce. Dále nechť `[Int]` a `[Float]` je po řadě typ pro seznam celých a desetinných čísel. Potom pro zjištění délky seznamu bychom měli mít např. takovéto funkce:

```
lengthInt :: [Int] -> Int
lengthFloat :: [Float] -> Int
```

Pozn.: syntaxe zápisu typu je převzata z jazyka Haskell, kdy zdvojená dvojtečka odděluje jméno funkce od typové signatury*; jednotlivé typy parametrů a výsledku jsou odděleny šipkami a tak jak jdou typy zleva doprava, tak jsou zpracovávány i parametry a poslední údaj je o výsledku.*

To by však bylo velice nešikovné, mít pro každý typ extra funkci pro zjišťování délky nad ním. Proto je v praxi jediná funkce, která má tento typ:

```
length :: [a] -> Int
```

kde `a` je právě typová proměnná, které se přiřadí typ v době aplikace funkce.

Typové proměnné lze využít i v uživatelských typech, například již zmíněný rekurzivní typ:

```
data Strom = List
           | Uzel Int Data Strom Strom
```

je nešikovný v tom, že typy klíče `Int` a dat `Data` jsou dány dopředu, což nemusí být výhodné. Místo toho je možné specifikovat typ takto:

```
data Strom key data = List
                    | Uzel key data Strom Strom
```

Potom `key` a `data` jsou typové proměnné, které mi umožňují volbu klíče a dat pro konkrétní instanci stromu. V rámci jednoho stromu, jedné datové struktury, musejí být po všech uzlech klíče stejného typu a data stejného typu. Nicméně v jiném stromě už to může být jiné.

práce s daty

Práce s daty je velice příjemná a jednoduchá. Závorky, které se často uplatňují při práci s daty, zejména při uplatnění rozpoznávání vzorů (pattern matching), mají typicky význam oddělovačů, aby díky prioritnímu zpracování bylo k sobě přiřazeno, co má být logicky u sebe. Nejsou tedy, jak se někdo může domnívat, součástí datových konstruktorů. Díky různým implementačním trikům lze v manipulaci s daty použít řadu technik vedoucích k zefektivnění jejich manipulace, což vede k úspoře paměti i času v době vyhodnocení.

4.3 Řídicí struktury

deklarace/definice

Základními strukturami jsou funkce, jejichž umístění ve zdrojovém textu je buďto zcela libovolné, neboť rekurze je u takových jazyků všudypřítomná, nebo se drží obvyklého schématu, kdy se mohu odvolávat jen na to, co již bylo definováno a pro vzájemně rekurzivní části používám prvky jazyka, které takové konstrukce dovolují. Funkce lze často staticky vnořovat, takže vlastně vytvářet si lokální funkce, které pomáhají v práci globálně dosažitelným funkcím. Deklarace jsou ve funkcionálních jazycích velmi zřídkavé, ne-li zcela neexistující.

moduly

Jako všechny moderní programovací jazyky jsou i funkcionální jazyky často modulární. Rozhraní modulu je přitom velmi často odvozeno z modulu a není nutné vytvářet takové rozhraní explicitně. To, jestli se rozhraní odvodí ze zdrojového textu, nebo z binární formy modulu je už jazykově závislé.

větvení výpočtu

Pro řízení toku vlastního výpočtu se využívá úplné větvení jak na úrovni těla funkcí, tak na úrovni aplikace funkce, kdy se využívá rozpoznávání vzorů a unifikace. Větvení musí být úplné (např. if-then příkaz musí mít vždy else část) proto, aby pro všechny kombinace vstupů bylo možno dospět k výsledku. U rozpoznávání vzorů, na úrovni aplikace funkcí, se u řady jazyků dovoluje programátorovi, aby nemusel vyčerpat všechny kombinace vzorů, které mohou do funkce vstupovat. V takovém případě však v době překladu dojde k dogenerování těch kombinací, které chybějí, spolu s kódem, který ohlásí chybu a

zastaví výpočet. Tak se daří zabránit neřízenému pádu vyhodnocení v případě, že se objeví neočekávaná kombinace vstupních parametrů.

Návrh programu

Základním charakterem je, že zpracování dat je navrženo a implementováno zcela odděleně od implementace vstupně-výstupních operací a k vzájemnému propojení dojde zcela na závěr celého vývoje. Oproti imperativním jazykům neleží tyto operace „vespod“ aplikace, ale spíše „nahoře“ nad částí pro zpracování dat.

Data jsou zpracovávána postupnou manipulací a změnou dat sekvencí funkcí, které jsou postupně aplikovány na zpracovávaná data — dochází k vysokému vyžití funkcí vyššího řádu a mapovacích funkcí, kdy jsou určité operace mapovány na všechny členy nějaké rekurzivní datové struktury. Výsledný program je tak vlastně jediný výraz, který je třeba vyhodnotit, abychom dostali výsledek.

Při návrhu programu se techniky známé z imperativního programování prakticky nedají uplatnit. Například je třeba se rozloučit s tím, že

- proměnná je pojmenovaná adresa v paměti,
- operace jsou prováděny v pořadí uvedeném ve zdrojovém textu,
- je možné využít ukazatelů,
- známe stav výpočtu a ten manipulujeme.

Naopak by měl návrh programu sledovat strategii vyhodnocení (viz oddíl této kapitoly 4.4), protože tak lze vytěžit z jazyka všechny jeho výhody, což se projeví na efektivitě zpracování.

Shrnutí vlastností

Funkcionální jazyky umožňují rychlý vývoj programů, které jsou ve srovnání s programy imperativními často „menší“. Díky existenci formální báze je možné provádět formální důkazy vlastností algoritmů, které by v jiných jazycích byly obtížné. Strategie vyhodnocení (viz 4.4) nabízejí vlastnosti, které by se jinak dosahovaly jen stěží.

Na druhou stranu popularita těchto jazyků, přes řadu výhod a kladů, není veliká. A to i přesto, že již vznikla řada „velkých“ a náročných aplikací, které se za náročné považují i na úrovni jazyků imperativních. Jedním z důvodů může být zpracování vstupně-výstupních operací, nebo nutnost některé operace, pro které nejsou knihovny, nebo jsou příliš systémové, implementovat např. v jazyku C a potom přes rozhraní pro import modulů z jiných jazyků na ně přistupovat.

Asi nejvážnějším důvodem je však jistý odpor programátorů, kteří mají imperativní paradigma již tak vžité, že přechod na čistě deklarativní paradigma je pro ně již nemožný.

4.4 Překlad, interpretace

Překlad funkcionálních jazyků je velmi náročný. Zejména přední část překladače je velmi zesílena, syntaktická a sémantická analýza v podstatě probíhá několikrát, na různých úrovních jazyka. Nejsložitější úlohou překladače totiž je převést deklarativní popis na serializovaný výpočet realizovaný na von Neumannovské architektuře dnešních počítačů.

Abychom objasnili, proč může syntaktická analýza proběhnout několikrát, naznačíme postup zpracování funkcionálního jazyka:

1. expanse syntaktických zkratek
2. redukce výrazových schopností jazyka (výrazově bohaté konstrukce jsou postupně překládány na jednodušší konstrukce, až zbývá malá množina základních), probíhá v několika krocích
3. automatické odvození typů, typová kontrola
4. serializace operací (závislá na strategii vyhodnocení)
5. kompletace větvení na úrovni rozpoznávání vzorů
6. převod rozpoznávání vzorů na testování značek a kaskádu příkazů if-then-else
7. závěrečné fáze, příprava pro optimalizace a generování kódu

Generování kódu je úzce spjato s vyhodnocovacími strategiemi, které vlastně určují klíčové vlastnosti jazyka. Kromě ovlivnění toku řízení a předávání toku řízení mezi funkcemi ovlivňují i tvorbu programu Všechny tyto strategie jsou založeny na známých konceptech, jsou však dovedeny do značné dokonalosti.

vyhodnocovací strategie

Volání hodnotou

Volání hodnotou (call-by-value) je strategie známá z imperativních jazyků. Parametry jsou vyčísleny před voláním funkce. Počet vyhodnocení nějakého výrazu je tak minimálně jeden, horní omezení však není.

U funkcionálních jazyků se tato strategie vyvinula do *striktního* vyhodnocení (strict evaluation). Optimalizace je v tom, že se sdílí hodnoty výrazů, takže k vyhodnocení dochází právě jednou. Navíc nedochází k vyhodnocení pod λ-abstrakcemi.

striktní vyhodnocení

Volání v případě potřeby

Volání v případě potřeby (call-by-need) je strategie, kdy dojde k zavolání (aplikaci) funkce bez toho, aby se vyhodnotily parametry — pro ně jsou vytvořeny kontextové obálky (context closures). Parametr je vyhodnocen až tehdy, pokud tělo funkce potřebuje jeho hodnotu k dalšímu postupu výpočtu. Počet vyhodnocení nějakého výrazu je tak minimálně žádný, horní omezení však není.

lazy evaluation

U funkcionálních jazyků se tato strategie vyvinula do strategie *lazy evaluation* (nepřekládáme). Díky vylepšení jako u striktní strategie jsou potom výrazy vyhodnocovány nejvíce jedenkrát.

Další strategie

Další strategie, se kterými se můžeme setkat, jsou někde mezi uvedenými. Často využívají analýzy striktnosti parametrů k tomu, aby dopředu předepsaly strategii vyhodnocení pro jednotlivé případy volání funkcí. V některých případech je možné využít informací od programátora, jak postupovat v kterém případě. Systémů je celá řada, proto další možné strategie odložíme na samostudium.

vlastní generování

To, že bychom při generování kódu ušetřili práci, pokud použijeme striktní vyhodnocení, je iluzorní. Problémy z ostatních strategií jsou plně suplovány konstrukty jako částečná aplikace funkce, funkce je výsledkem operace, apod. Pro řešení takovýchto problematických míst se proto užívají kontextové obálky, někdy také kontextové zámky (hodnoty, výrazy a funkce, které jsou v aktuálním čase volání potřeba jsou uloženy a zakonzervovány pro pozdější užití).

Při generování kódu přicházejí v úvahu dvě možnosti:

- přímá vazba na cílovou platformu — velmi obtížné, nutné další snižování síly jazyka, úzce spjato s knihovnami pro běh programu (runtime libraries), některé vlastnosti jazyka mohou tak vymizet;
- speciální cílový kód určený pro funkcionální jazyky — překlad je jednoduchý, interpretace takového kódu, případně jeho semi-interpretace.

interprety

Interprety funkcionálních jazyků v čisté podobě neexistují, protože je to prakticky nemožné. V praxi se jedná o překlad do specializovaného virtuálního kódu určeného pro funkcionální jazyky.

Takový virtuální kód může být třeba graf, reprezentace funkcionálních programů takovouto formou je přirozená a jednoduchá. Zpracování/vyhodnocení programu je potom redukce takového grafu. Pro

rychlost zpracování je to však nevýhodné. Proto je virtuální kód lineárním zápisem takového grafu. Uveďme dva případy takových kódů (bez překladu):

- three instruction code,
- spineless tag-less G-machine.

Vyhodnocení funkcionálních programů je prakticky vždy spojeno s automatickým managementem paměti, který se nazývá garbage collector (sběrač smetí, ale nepřekládá se). Jeho studium je za rámcem této publikace, ale existuje řada materiálů, kde je možné tuto problematiku nastudovat.

4.5 Na závěr

Zpracování typů

Zpracování typů je ve funkcionálních jazycích zcela speciální kategorie. Pokud jsou to jazyky netypované, tak se vše odehrává podobně, jako u jazyků imperativních, což představuje kontrolu na poslední chvíli.

Velmi často jsou to však jazyky typované. Potom se tedy využívá automatického typového odvození a velmi striktní typové kontroly. V praxi to může vést k velkému počtu operátorů, protože musíme mít pro sčítání různých typů (celá čísla, desetinná, racionální, komplexní) vždy specializovaný operátor, aby kontrola správně proběhla.

Jako jisté řešení z této svízelné situace (např. jazyky třídy ML) se objevují *typové třídy*, které se objevily v jazyku Gofer a nyní Haskell. Tyto třídy nemají nic společného s objekty a objektovým programováním. Nicméně sdružují funkce a operátory do skupin, kde jsou svázány jistým společným jmenovatelem. Potom je možné do takové třídy přidat typ tehdy, pokud pro každý operátor či funkci dodáme algoritmus, který se má použít pro vyhodnocení. typové třídy

Výsledkem je to, že např. pro sčítání máme pro všechny užité typy jeden operátor. Nevýhodou může být to, že schéma funkcí a operátorů v třídách je vlastně dopředu dáno (ze základních knihoven) a tak není možné použít operátory, či pojmenování funkcí i pro algoritmy/typy, které do systému typových tříd nezapadají.

Formální verifikace

Možnost formálně verifikovat programy je u funkcionálních jazyků výrazně vyšší, než jinde. Je to dáno jasným vztahem k λ-kalkulu a tím, že většina těchto jazyků má svou vlastní formální specifikaci.

Existují však i funkcionální jazyky, které se dokonce používají pouze pro formální specifikaci a verifikace (jazyk Z). U jazyků užitých pro programování je situace jiná. Díky typovému odvození a silné typové kontrole je řada chyb odhalena již v době překladu. Pro formální důkaz, pokud je třeba, je potom možné využít strukturální indukce. Pokud není možné využít tu, tak potom nezbývá než kontrolovanou část programu z něj vyjmout, dle formální specifikace překladu snížit úroveň jazyka na potřebnou úroveň (core-ML, λ-kalkul) a tam provést formální rozvahu.

Možnosti a vhodnosti využití

Funkcionální programovací jazyky nabízejí řadu standardních vlastností, které jsou vhodné pro týmovou spolupráci a tvorbu rozsáhlých programů, jako jsou modularita, strukturované datové typy, apod. Kromě toho, v případě automatické typové inference, nabízejí vlastnosti vhodné pro detekci chyb v raném stádiu. Čistě funkcionální jazyky však svou popularitou zaostávají za jazyky imperativními i když s funkcionálními prvky.

Největší překážkou je asi v učení se porozumění deklarativismu a funkcionálnímu paradigmatu, zejména po dlouhém užívání imperativních jazyků. Pokud je však tento moment překonán, tak se otevírají cesty k jazykům, které nabízejí efektivnější práci, podobně jako u imperativních jazyků řadu knihoven, apod.

Jazyky jsou vhodné zejména pro složitější díla tvořená jedním programátorem, nicméně je stejně tak možné je využít pro týmovou práci. Další nespornou oblastí užití jsou simulace a řešení jiných extrémních problémů, např. z oblasti umělé inteligence. Výhodně se dá použít pro prototypování algoritmů a ověřování jejich vlastností. V neposlední řadě mohou výborně sloužit jako jazyky obecného využití/nasazení.

Pojmy k zapamatování

Pojmů k zapamatování je v této kapitole celá řada. Uveďme ty nejdůležitější z nich: funkcionální paradigma, automatické typové odvození, seznamy, n-tice, částečná aplikace, výčtové typy, záznamy, variantní záznamy, rekurzivní datové typy, řešení vstupu a výstupu, typová proměnná, typové třídy, strategie vyhodnocení (typy strategií).

Závěr

Cílem kapitoly bylo bližší seznámení s funkcionálními programovacími jazyky. Ukázat jejich klíčové vlastnosti, způsob zpracování, pokusit se srovnat některé jejich vlastnosti s jazyky imperativními. Měli byste nyní znát tyto věci a být schopni správně takový jazyk rozpoznat, určit jeho vlastnosti a případnou vhodnost, či nevhodnost pro řešení nějaké úlohy. Klíčovou je zejména znalost vlastností, které nejsou v imperativních jazycích běžné, protože je možné je, jistým způsobem, zúročit i při programování právě v imperativních jazycích.

Úlohy k procvičení:

1. Zvolte si nějaký imperativní jazyk a v něm implementujte nekonečný seznam prvočísel. Pro přístup k němu používejte pouze dvě funkce/procedury — pro nastavení přístupu k prvnímu prvočíslu, pro poskytnutí následujícího prvočísla.
2. Vyhledejte co nejvíce funkcionálních jazyků, případně i ty, které nejsou čistě deklarativní. Který z nich je nejstarší? Který z nich je nejmladší?

Klíč k řešení úloh

1. První číslo musíte explicitně definovat, ostatní již z definice určíte. Co je již jednou vypočítané, to máte schované no a počítáte jen tolik, pokud se ptá uživatel na další hodnoty.

Další zdroje

Řadu informací podaných v této kapitole najdete v literatuře citované níže, nebo přímo v nápovědě, či doprovodné dokumentaci, či manuálech, které jsou k dispozici k funkcionálním programovacím jazykům na WWW stránkách.

- Thompson, S.: *Haskell, The Craft of Functional Programming*, ADDISON-WESLEY, 1999, ISBN 0-201-34275-8
- Jones, S.P.: *Haskell 98 Language and Libraries*, Cambridge University Press, 2003, p. 272, ISBN 0521826144.
- Bieliková, M., Návrat, P.: *Funkcionálne a logické programovanie*, Vydavateĺstvo STU, Vazovova 5, Bratislava, 2000.
- www.haskell.org

Kapitola 5

Logické programovací jazyky

V této kapitole se seznámíme s charakterem a vlastnostmi logických programovacích jazyků, jakožto reprezentanta jazyků deklarativních.

Čas potřebný ke studiu: 3 hodiny 45 minut.

Cíle kapitoly

Cílem kapitoly je seznámit čtenáře se základními a charakteristickými vlastnostmi logických jazyků. Jak vypadají data, jaké jsou typické řídicí struktury, jak jsou programy v takových jazycích vyhodnocovány, jaká je jejich formální báze, apod.
Na konci kapitoly by mělo být zřejmé, co lze od takových jazyků očekávat. Zejména by mělo vyplynout na povrch, že tyto jazyky jsou určeny zejména pro úlohy z umělé inteligence, simulace, formální verifikace apod. Jsou použitelné i jinak, ale na prohledávání stavového prostoru jsou ideální.

Průvodce studiem

Tato kapitola předpokládá dobrou orientaci v základních pojmech z klasifikace a popisu imperativních programovacích jazyků a terminologií s tím spjatou. Dále předpokládá znalost deklarativního paradigmatu a, i když je v kapitole elementárně zmíněna, tak hlubší znalost predikátové logiky.
Ke studiu této kapitoly není třeba žádného speciálního vybavení. Jako u většiny ostatních se jedná o to si zapamatovat danou terminologii a pochopit klíčové principy předkládaných konceptů. Pro hlubší studium, nebo bližší objasnění prezentovaných skutečností je vhodné využít informací na Internetu a vlastní praktické ověření vlastností na konkrétním jazyce.

Obsah

Výklad

5.1 Základní popis

Definovat paradigma pro logické jazyky, díky jejich různorodosti, není jednoduché, ale aspoň si tento pojem můžeme vymezit:

Definice 5.1.1 *Paradigma logických jazyků: Program v logických programovacích jazycích sestává z predikátů případně kombinovaných s omezeními, které pracují nad atomy, termy, či predikáty.* **Definice!**

Mezi nejznámější a nejrozšířenější reprezentanty patří jazyky Prolog, kolem kterého budeme soustřeďovat i výklad. Dnes již poměrně zapomenutá verze Prologu zvaná Parlog kombinovala Prolog s využitím paralelismu na všech úrovních. Tyto dva jazyky pracují s predikáty, které mají specifický tvar zvaný Hornovy klauzule. Dalším jazykem, který je nutné zmínit, je jazyk Gödel, který používal progresivní systém na vyhodnocení programu. Ač v řadě směrů jazyk Prolog překonával, tak nedošlo k takovému rozšíření, aby se jeho vývoj udržel až do dnešních dní. Zejména výrazová síla (mnohem bohatší, než Hornovy klauzule), práce s typy a plná úroveň deklarativity tento jazyk předurčovaly pro další rozvoj. Dalším významným členem rodiny logických jazyků jsou jazyky CLP($\mathcal{R}$) a příbuzné. Predikáty umožňují doplnit o různá omezení, takže výsledkem programu nemusí být hodnota, ale omezení, které hodnotu nějakým způsobem vymezuje.

Predikátová logika

Formální bází logických programovacích jazyků je predikátová logika. Nikoliv však v plné síle, ale nějakým způsobem omezená, přitom jakou podmnožinu predikátové logiky daný systém využívá záleží na přístupu a požadovaných vlastnostech.

automatizace důkazů

Musíme mít vždy na paměti, že přítomnost formální báze je u logických jazyků naprostou nezbytností, neboť původní určení těchto jazyků bylo automatizovat formální důkaz v predikátové logice. Proto je často i síla jazyků nižší, než u predikátové logiky samotné.

jazyk predikátové logiky

Připomeňme si jazyk predikátové logiky, syntakticky se skládá z termů, predikátů a formulí. Termy jsou konstanty (nulární funkční symboly), proměnné, funkční symboly. Formálně jsou termy definovány takto:

Definice 5.1.2 Term, *v predikátové logice, je:* **Definice!**

- *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.

Základními predikáty jsou potom pravda (T, true) a nepravda (F, false), dále je predikát syntakticky shodný s funkčním symbolem. Formule potom definujeme z predikátů pomocí logických spojek a kvantifikátorů. Formálně takto:

Definice!

Definice 5.1.3 *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.*

---

interpretace

Sémantika predikátové logiky není dána hned, ale nejdříve je třeba definovat *interpretaci*. Interpretace přiřazuje význam konstantám, funkcím a predikátům. Také je nutné přiřadit hodnoty volným proměnným. O tom, jestli je formule splnitelná, či platná, však nejde vždy zcela rozhodnout a proto mluvíme u predikátové logiky o *částečné rozhodnutelnosti*. Říkáme, že interpretace *je modelem* nějaké formule, pokud ta je pravdivá pro každé ohodnocení volných proměnných. Formule je *splnitelná*, pokud existuje taková interpretace, která je modelem. Formule je *platná* (validní), jestliže je pravdivá při každé interpretaci.

částečně rozhodnutelná

splnitelnost

platnost

Důkazy se v predikátové logice tvoří za pomocí správně definovaných formulí (well formed formulas) a odvozovacích, dedukčních, pravidel. Pravidla jsou:

- modus ponens: if $P \wedge (P \rightarrow Q)$ then Q
- zobecnění: z P odvoď $\forall x : P$

Je-li. potom k správně definovaná formule a W množina takových formulí tak píšeme

$$W \vdash k$$

pokud k je platná ve všech modelech W. Říkáme, že k je dokazatelná z W.

Syntaxe a sémantika

Kromě neformálních popisů, či popisů na příkladech (podobně jako u funkcionálních jazyků), se můžeme setkat i s formální definicí syntaxe pomocí vhodné formy. Pokud je to tak, tak důvodem je zejména snaha prezentovat jednoduchost syntaxe jazyka. Formálně se jedná o bezkontextové gramatiky — jedná se o zápis termů a predikátů (klauzulí) a zejména o notaci pro logické operátory a spojky.

Sémantika však formálně není prezentována prakticky vůbec. Můžeme se však setkat s odkazem na literaturu, kde je možné se tento formalizmus dozvědět. Velmi často je jazyk vysvětlován na příkladech, které jsou buďto v textu, nebo přímo součástí instalace, aby si je mohl uživatel okamžitě vyzkoušet. Jelikož je obecný koncept jazyka a způsob vyhodnocení poměrně jednoduchý, soustřeďuje se popis často na objasnění vestavěných, nebo v knihovnách ukrytých, predikátů. Ty totiž mají často mimo-logický význam a proto je třeba detailně objasnit jejich funkci.

5.2 Data a jejich zpracování

Datové typy ve funkcionálních jazycích najdeme jak velmi jednoduché, skalární, tak složité, složené, Jednoduchých, *atomických*, datových typů není v logických programovacích jazycích mnoho. V zásadě se jedná pouze o celá čísla, případně znaky. Jako rozšíření se objevují čísla s plovoucí desetinnou čárkou, Strukturu těchto typů nelze výrazovými prostředky jazyka nijak rozebrat (výjimku může tvořit přítomnost knihovní funkce).

termy jako data

Prvotní datovou abstrakcí jsou *termy*. Díky implementačním trikům je možné je konvertovat na predikáty a nechat okamžitě vyhodnotit. Logické jazyky tak mají poměrně ojedinělou vlastnost, že vlastně část programu mohou sestavit až v době běhu programu samotného. Pokud se na termy podíváme z jiného úhlu, tak to jsou pojmenované n-tice, jak je známe z funkcionálních jazyků:

```
osoba('Jan','Kadlec',1969).
adresa('Brno','Horova',22,'616 00').
data(podnik,typ(maly),[praha, brno]).
```

Termy lze vnořovat, uzavírají se nad atomy. Atomy hrají velikou roli, neboť mohou nést prakticky jakoukoliv hodnotu.
Pozn.: V syntaxi jazyku Prolog začínají atomy, či jména termů malým písmenem, nejedná-li se o čísla, velkým písmenem začínají proměnné. Chceme-li tedy mít hodnotu atomickou s velkým písmenem na

začátku, musíme ji uzavřít do apostrofů, stejně tak, pokud obsahuje mezery uvnitř.

seznamy

V příkladu jsme již viděli zápis seznamů známý z jazyka Haskell. Zápis literálů i práce se seznamy je v logických jazycích buďto naprosto shodná, nebo s drobnými odlišnostmi v syntaxi. Seznamy v logických jazycích jsou však často heterogenní datovou strukturou, jelikož jsou to jazyky netypované, kdy ke kontrole typu dochází až při konkrétní operaci. Proto se někdy se seznamy v jazyku Prolog zachází podobně jako s n-ticemi v jazycích funkcionálních.

Pro práci s daty, ať jednoduchými, tak termy, či seznamy, lze použít několik technik, jak data vytvořit, nebo přistoupit k jejich složkám, jsou-li strukturované:

- rozpoznávání a unifikace vzorů (pattern matching) v hlavičce klauzule — práce s parametry predikátu;
- převodem na jiný typ — např. převod mezi termem a seznamem u jazyka Prolog;
- přímý přístup k termům.

Termy lze potom vytvořit modifikací existujících, nebo operací pro převod na seznam v opačném směru.

5.3 Řídicí struktury

deklarace/definice

Základními strukturami jsou predikáty/klauzule, jejichž umístění ve zdrojovém textu je typicky zcela libovolné. Deklarace v takovýchto jazycích nejsou pro obecné užití povoleny — není potřeba. Jistý zjednodušený typ deklarace potom můžeme najít pro deklaraci dynamických klauzulí, pro deklaraci predikátů exportovaných z modulu, apod. Tyto zjednodušené deklarace potom obsahují pouze jméno predikátu a počet parametrů. Počet parametrů je důležitý jednak pro funkci, druhak proto, že některé jazyky dovolují přetěžování jmen predikátů na úrovni počtu parametrů. Typová kontrola, díky povaze jazyků, nemá smysl.

řízení výpočtu
cíl
podcíl
uspívání
selhání

Řízení vyhodnocení logického programu je dáno *cílem* (goal) výpočtu, které ho chceme dosáhnout, ten se potom sestává z dílčích *podcílů* (subgoals). Pokud dospějeme k výsledku, tak říkáme, že cíl uspěl, nebo uspívá (satisfied), podobně to můžeme říci i o podcílech. V opačném případě cíl, či podcíle selhávají (fail) a nejsme tak schopni obdržet výsledek. Cíl, či podcíle mohou opakovaně uspívat.

Vlastní tok řízení/vyhodnocení programu je ovlivněn výběrem predikátů/klauzulí na základě unifikace vzorů (pattern matching) a na

základě uspívání/selhávání těla dané klauzule Tělo klauzule je opět specifikováno řadou predikátů. O výběru, v jakém pořadí jsou vyhodnoceny, rozhoduje strategie vyhodnocení. Typicky je to zleva doprava (Prolog) tak, jak je to uvedeno v textu. Typ operátorů, případně podmíněných vyhodnocení (if-then-else) může řídit lokální směr vyhodnocení v rámci jednoho těla klauzule.

Vezmeme-li do úvahy Hornovy klauzule, tak zápis v jazyku Prolog (schématicky) je tento:

```
a(...) :- b1(...), ..., bm(...).
```

Což znamená, v jazyky predikátové logiky:

$$\forall(a(\ldots) \leftarrow b_1(\ldots) \wedge \ldots \wedge b_m(\ldots))$$

Vidíme, že všechny klauzule jsou univerzálně kvantifikovány. Po jednoduchých úpravách se tedy dostaneme k:

$$\forall(a(\ldots) \vee \neg(b_1(\ldots) \wedge \ldots \wedge b_m(\ldots)))$$

$$\forall(a(\ldots) \vee \neg b_1(\ldots) \vee \ldots \vee \neg b_m(\ldots))$$

Výsledný tvar je potom možné již jednodušeji programově vyhodnotit, protože vyhodnocením každého z podcílů vlastně ubude z výrazu jeden člen (logický výraz „nepravda nebo výraz“ má hodnotu členu „výraz“).

Při vyhodnocení, dojde-li k selhání, systém nastolí režim zpětného navracení (backtracking), kdy se vrací ve výpočtu zpět a hledá jiné možnosti uspění již dříve uspívajících podcílů. Některé podcíle tedy jsou stavěny tak, aby znovu-uspívaly. Prakticky se tak realizuje opakování, nebo s výhodou i prohledávání stavového prostoru. I když je pro opakování možné použít rekurzi, tak zpětné navracení je metoda vlastní jazyku Prolog a proto se využívá častěji. Teprve až uspějí všechny podcíle, tedy zpětné navracení se zastaví na cíli, který znovu uspěl, a další podcíle v řadě již uspějí, uspívá i celý cíl. Pokud při zpětném navracení žádný podcíl znovu neuspěje, tak selhává celý cíl. *Pozn.: Vestavěné, mimo-logické predikáty velmi často nejsou schopny znovu uspívat.*

zpětné navracení

Návrh programu

Návrh a implementace programu může probíhat jak shora dolů, tak zdola nahoru. nebo kombinací obou přístupů. Díky absenci deklarací je jakákoliv modifikace programu později jednoduchá, neboť přidávat je možné kamkoliv do textu programu (kromě dovnitř jiných klauzulí, samozřejmě).

Text pro klauzule s jedním jménem je často umísťován do jednoho místa s tím, že pořadí deklarací klauzulí se uplatňuje při výběru klauzulí v době vyhodnocení a hraje roli při zpětném navracení. Proto je třeba na něj dbát.

Vestavěné predikáty často realizují operace, které přestavují vysoce optimalizovanou manipulaci s daty, která by sice byla možná pomocí uživatelských funkcí, ale délka jejich vyhodnocení by byla dlouhá. Kromě toho jsou to i systémově závislé operace pro vstup a výstup apod.

Velkou zvláštností jazyku Prolog, v porovnání s ostatními, je jeho schopnost měnit obsah programu (říkáme někdy, v případě jazyka Prolog, obsah databáze) za běhu. Má totiž vestavěné predikáty pro odstranění klauzulí z databáze i pro přidání nových. Je tak ojedinělým případem programovacího jazyka, který umožňuje za běhu měnit program!

Shrnutí vlastností

Logické programovací jazyky jsou v podstatě jedinou možností automatizované podpory důkazu. Nabízejí přímo ve vyhodnocovací strategii zpětné navracení, což je u jiných jazyků nevídané a je třeba to řešit jinými prostředky. U jazyka Prolog dokonce existuje standardizace. To jsou vše vlastnosti, které hovoří pro využití těchto jazyků v praxi. Krom toho jsou rozšířeny na řadu platforem a užívají sběrač smetí (garbage collector) pro správu paměti.

Na druhou stranu neexistuje čistá kompilace pro takové jazyky. Rychlost vyhodnocení je tak nižší a ještě klesá se zaplněním databáze klauzulí. I když se setkáme s modulární verzí těchto jazyků, tak je to, až na výjimky, modularita na úrovni zdrojových kódů. Tato negativa spolu s výjimečnými vlastnostmi však posouvají využití jazyka spíše do speciálních oblastí — simulace, verifikace, apod.

Výklad

5.4 Překlad, interpretace

překlad

Překlad logických jazyků do čisté binární formy je prakticky nerealizovatelný. Je to dáno možností modifikace databáze klauzulí za běhu, možnost vzniku takových klauzulí z dat, která jsou v době překladu neznámá. Pokud tedy je možné zdrojový kód překládat, tak výsledkem je kód, kdy jeho části jsou přímo vyhodnocovány, ale součástí je také plný interpret, který části interpretuje a případně supluje vyhodnocovací strategii pro klauzule utvořené za běhu programu.

Překlad se často děje do vyšších programovacích jazyků, či do speciálního mezikódu, přímo reflektující stavbu klauzulí, který je potom interpretován. Např, jazyk Gödel byl překládán do Prologu.

Úzká vazba na sběrač smetí (garbage collector) a implementace zpětného navracení je samozřejmostí.

interpretace

Interpretace je u logických jazyků mnohem častější. Přímá interpretace je ze zřejmých důvodů nemožná. Dochází tedy k transformaci do interní podoby (stromová struktura) a uložení do databáze (užití rozptylovacích funkcí — hash function — pro urychlení přístupu). Databáze má své praktické limity jak pro celkový obsah, tak pro efektivní vyhodnocení.

Interpret potom pracuje nad takovou interní reprezentací a přistupuje přímo do databáze. Opět platí využití sběrače smetí a zpětného navracení přímo v implementaci interpretu. Způsob vnitřního uložení je optimalizován pro zrychlení vyhodnocení.

SLD rezoluce

Vyhodnocovací strategií pro jazyk Prolog (a řadu dalších) je *SLD rezoluce*. Vstupem jsou jí Hornovy klauzule. Vyhodnocení probíhá podle tohoto jednoduchého schématu:

1. Klauzule jsou z databáze vybírány shora dolů, dle umístění ve vstupním textu.

2. Těla predikátů jsou zpracovávána zleva doprava.

Jedná se tedy o *prohledávání do hloubky* — dokud neuspěje první podcíl, ostatní zůstávají nedotčené.

Pro demonstraci uvažme tento příklad:

```
a(...) :- b(...), c(...), d(...).
a(...) :- d(...), e(...).

b(...) :- c(...), d(...), c(...).
b(...) :- e(...).

c(...) :- d(...).
c(...) :- e(...), d(...), c(...).

d(...).
d(...) :- e(...), d(...).

e(...).
```

A nechť naším cílem je predikát `a`. Vyhodnocení nechť se zachytí hned na první klauzuli, tedy naznačme jako:

```
a(...) -->> b(...), c(...), d(...).
```

Dále dochází k ověření, zda uspívá první podcíl, označený zeleně a pro něj nechť uspívá hned první klauzule, tedy:

```
b(...) -->> c(...), d(...), c(...).
```

Stejný nechť je postup s podcílem `c`, tedy:

```
c(...) -->> d(...).
```

Nyní je tedy ověřena první klauzule `d`, která je bez těla (říká se jí *fakt*). Ta nechť selže, tedy naznačme např. takto:

```
d(...).
```

Potom se tedy uplatní druhá klauzule pro predikát `d` v pořadí a v ní se v jejím těle opět vybere nejlevější podcíl pro vyhodnocení:

```
d(...) -->> e(...), d(...).
```

Tímto způsobem tedy probíhá vyhodnocení dál, dokud nedojde k uspění, či selhání (někdy také vyvrácení) cíle.

Pokud tedy tento naznačený postup shrneme do jediné ukázky, potom tedy:

```
a(...) -->> b(...), c(...), d(...).
               +->> c(...), d(...), c(...).
                       +->> d(...).
                               +->> d(...).
                               +->> e(...), d(...).
```

V průběhu vyhodnocení dochází k *unifikaci* mezi proměnnými a hodnotami, které se objevují na příslušných pozicích parametrů. Pokud dojde k uspění cíle, tak kromě oznámení úspěchu je vypsána substituce pro nejvyšší úroveň proměnných (proměnné specifikované v cíli).

unifikace

Pojem *unifikace* je v logických jazycích a tím pádem i v jazyku Prolog velmi důležitý. Unifikace probíhá při hledání hlavičky podcíle v databázi klauzulí, kdy klauzule očekává parametry jistého tvaru, nebo je přijímá, případně v době explicitního vynucení unifikace, kdy jsou proti sobě postaveny termy/proměnné.

Takovým způsobem dochází k vzájemnému provázání mezí proměnnými a dalšími entitami v celém programu. Unifikace probíhá vždy na úrovni termů a hledá se nejobecnější unifikátor (most general unifier, mgu). Nejobecnější unifikátor je takový, že neexistuje žádný jiný unifikátor, který by pro nějakou část unifikace nabízel obecnější řešení, přitom dva unifikátory jsou nejobecnější, pokud se liší pouze přejmenováním volných proměnných.

nejobecnější unifikátor

Mezi typické vlastnosti unifikace v jazyku Prolog (často nejen v něm) patří:

- Kontrola výskytu — ověřuje se, zda proti sobě stojí dvě identické proměnné, neboť to může v řadě případů vést k nekonečným termům, apod. (jazyk Prolog neprovádí). kontrola výskytu
- Hloubka unifikace — unifikace neprobíhá na libovolně rozsáhlých termech.
- Zpracováni cyklických a nekonečných struktur — často zakázáno a úzce souvisí s kontrolou výskytu.

Uvažme tento příklad, kdy máme dva termy, jenž chceme unifikovat:

```
tree(X, leaf, Y)
tree(128, Z, tree(W,U,leaf))
```

Jejich nejobecnějším unifikátorem je substituce, kdy za `X` budeme substituovat `128`, za `Z` atom `leaf`, za `Y` potom term `tree(W,U,leaf)`. Pokud provedeme tyto náhrady v obou termech, dostaneme:

```
tree(128, leaf, tree(W,U,leaf))
tree(123, leaf, tree(W,U,leaf))
```

což jsou identické termy. Substituci, jak byla naznačena, budeme zapisovat takto:

```
[128/X]
[leaf/Z]
[tree(W,U,leaf)/Y]
```

Jelikož se tyto substituce aplikují po řadě všechny na oba termy, tak zřetězení substitucí budeme značit kolečkem, tedy: $[\texttt{128/X}] \circ [\texttt{leaf/Z}] \circ [\texttt{tree(W,U,leaf)/Y}]$

Jiná možná náhrada:

```
[128/X]
[leaf/Z]
[tree(W,U,leaf)/Y]
[34/W]
```

je sice unifikátorem, protože dostaneme

```
tree(128, leaf, tree(34,U,leaf))
tree(123, leaf, tree(34,U,leaf))
```

ale není nejobecnějším unifikátorem, neboť proměnnou `W` jsme nahradili konkrétní hodnotou `34`, čímž jsme snížili možnosti další unifikace, tedy míru obecnosti. Nejedná se tedy o nejobecnější unifikátor.

Pro dvojici termů:

```
tree(X, leaf, Y)
tree(9, Z, tree(W,Y,leaf))
```

však nejobecnější unifikátor nenalezneme, pokud nepovolíme nekonečné termy, neboť dvojici `Y` a `tree(W,Y,leaf)` nelze unifikovat, neboť se proměnná `Y` vyskytuje na obou stranách v různých úrovních, tj. nikoliv proti sobě.

Robinsonův algoritmus

Unifikačních algoritmů je celá řada, uveďme si Robinsonův unifikační algoritmus, který se svým způsobem s jazykem Prolog pojí, i když v současnosti existují i další unifikační algoritmy.

Algoritmus!

Algoritmus 5.4.1 *Robinsonův 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* {
    skonči selháním unifikace
  } *jinak* {
    nechť $p = (x, t)$, *kde* x *je proměnná*
    pokud se x *vyskytuje v* t { – kontrola výskytu!!
      skonči selháním unifikace
    } *jinak* {
      nastav $\mu = \mu \circ [t/x]$
    }
  }
}
vrať μ

Ukažme si činnost algoritmu na našem příkladu. Množina nechť sestává ze dvou termů:

```
t1 = tree(X, leaf, Y)
t2 = tree(128, Z, tree(W,U,leaf))
```

První nesouhlasný pár je `<X, 128>`, Z algoritmu plyne, že přidáme substituci `S1 = [128/X]`. Po aplikaci substitucí (teď jediné) dostáváme:

```
t1 = tree(128, leaf, Y)
t2 = tree(128, Z, tree(W,U,leaf))
```

a hledáme další nesouhlasný pár, kterým je `<leaf, Z>`. Z algoritmu plyne, že budeme vytvářet další substituci `S2 = [leaf/Z]`. Po aplikaci substitucí (`S1` $\circ$ `S2`) dostáváme:

```
t1 = tree(128, leaf, Y)
t2 = tree(128, leaf, tree(W,U,leaf))
```

a hledáme další nesouhlasný pár, kterým je `<Y, tree(W, U, leaf)>`. Substituce je `S3 = [tree(W, U, leaf)/Y]`. Po aplikaci substitucí (`S1` ∘ `S2` ∘ `S3`) dostáváme:

```
t1 = tree(128, leaf, tree(W,U,leaf))
t2 = tree(128, leaf, tree(W,U,leaf))
```

Což jsou identické termy a tedy výsledná substituce je:

$$\mu = \texttt{S1} \circ \texttt{S2} \circ \texttt{S3} = [\texttt{128/X}] \circ [\texttt{leaf/Z}] \circ [\texttt{tree(W, U, leaf)/Y}]$$

Další strategie

Jiné strategie vyhodnocení jsou například u systémů CLP, kdy výsledkem není seznam substitucí, ale omezení pro proměnné volné v cíli, které vymezují jejich hodnoty. Tato omezení jsou typicky reprezentována jako množina rovnic, či nerovnic, nebo dalším způsobem.

Zcela ojedinělá strategie potom byla uplatněna v jazyku Gödel, kdy nebyla uplatněna strategie prohledávání do hloubky, ale zejména v pravé straně klauzule byl vždy vybrán právě nejvhodnější podcíl pro vyhodnocení. Jeho chování tak bylo mnohem více deklarativní a často mnohem efektivnější. Kromě toho jazyk neobsahuje různé háčky (jako jazyk Prolog, např. operátor řezu) a navíc zavádí operátory jinde nevídané (univerzální a existenční kvantifikátory, apod.).

5.5 Na závěr

Zpracování typů

Jak již bylo zmíněno, tak jazyky logické jsou většinou netypované, takže typová informace se začne zpracovávat v momentě, kdy nějaká operace vyžaduje operandy specifického typu. Počet základních typů je většinou nízký a jen ojediněle jej může programátor obohatit o svoje typy (uměl to např. Gödel). Nicméně síla termů a struktur, které lze z nich vytvořit, je dostatečná pro jakoukoliv práci.

Formální verifikace

Formální verifikace programů v logických jazycích typicky neprovádíme. Otázkou totiž je, co bychom chtěli kontrolovat a hlavně jak. Jedná se, *de facto*, o formule v predikátové logice. Takže jsme vlastně limitování predikátovou logikou jako takovou.

Pro algoritmy strukturální (manipulace rekurzivních datových struktur, typicky seznamy) je možné použít strukturální indukci. Jelikož jádro logických programů takové operace typicky netvoří, tak jen zvolit strategii a postup pro důkaz by bylo hodně těžké.

Možnosti a vhodnosti využití

Asi si kladete otázku, jestli jsou tyto jazyky vhodné pro větší projekty. Ano, ale... Tyto jazyky jsou určeny pro specifický okruh problémů, vazba mezi moduly není stejná, jako u imperativních jazyků. Kromě toho i poměrně rozsáhlé programy mohou být vytvořeny jedním člověkem. Tedy jednoznačná odpověď není.

Při tvorbě programů je však třeba mít na paměti, jakým způsobem se vybírá klauzule pro další vyhodnocení (její hlavička), stejně tak to, že systém má vestavěnou strategii zpětného navracení. Proto je třeba bedlivě promyslet, jestli vyhodnocení bude probíhat opravdu jen těmi částmi programu, jak chceme my a případně dodat explicitní blokaci pro některé výpočty (operátor řezu, další podmínky na parametry, v těle klauzule, apod.).

Pro odlišení jednotlivých variant téhož predikátu je ideálně využít odlišení již na úrovni struktury parametrů, kdy se vybírá hlavička na základě unifikace vzorů (pattern matching). Jinak by podmínky měly být na začátku, to jak aritmetické, tak testy na charakter parametrů (volná proměnná, vázaná proměnná, atom, atd.).

Zejména v jazyku Prolog (a jemu příbuzných) je třeba si dávat pozor na efektivitu a optimálnost vašeho programu, neboť vyhodnocovací mechanismus uplatňuje „slepou“ strategii výpočtu, kterou řídí zejména program, žádné zefektivnění není (jako např. u CLP a jazyku Gödel).

Pojmy k zapamatování

Pojmů k zapamatování je v této kapitole celá řada. Uveďme ty nejdůležitější z nich: logické paradigma, automatizace důkazů, predikátová logika (a vše, co k ní patří, zejména jazyky, interpretace, model, splnitelnost, platnost), klauzule, hlavička, tělo, atom, term, predikát, cíl, podcíl, uspívání, selhání, zpětné navracení, unifikace vzorů, unifikace, SLD rezoluce, Robinsonův algoritmus, nejobecnější unifikátor, kontrola výskytu.

Závěr

Cílem kapitoly bylo bližší seznámení s logickými programovacími jazyky. Ukázat jejich klíčové vlastnosti, způsob zpracování, pokusit se srovnat některé jejich vlastnosti s jazyky imperativními. Měli byste nyní znát tyto věci a být schopni správně takový jazyk rozpoznat, určit jeho vlastnosti a případnou vhodnost, či nevhodnost pro řešení nějaké úlohy. Klíčovou je zejména znalost vlastností, které nejsou v imperativních jazycích běžné, protože je možné je, jistým způsobem, zúročit i při programování právě v imperativních jazycích.

Úlohy k procvičení:

1. Zvolte si nějaký imperativní jazyk a v něm implementujte datové struktury pro reprezentací termů a klauzulí v jazyku Prolog.
2. Implementujte funkci, která nad termy realizuje substituci.
3. Implementujte nad termy Robinsonův unifikační algoritmus.

Klíč k řešení úloh

1. Jedná se o rekurzivní strukturu, kdy základem je atom a proměnná.
2. Viz text kapitoly pro inspiraci.
3. Viz text kapitoly.

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, ...). Predikátovou logiku najdete buďto na Internetu, nebo ve vědecké knihovně.

- 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 6

Ostatní deklarativní jazyky

V této kapitole se stručně seznámíme s charakterem a vlastnostmi programovacích jazyků pro definici a manipulaci dat, jazyků s grafickým rozhraním a srovnáme deklarativní jazyky vzájemně.

Čas potřebný ke studiu: 1 hodina 45 minut.

Cíle kapitoly

Cílem kapitoly je stručně obeznámit čtenáře se základními a charakteristickými vlastnostmi jazyků pro práci s daty a jazyků s grafickým programátorským rozhraním. Nebudeme zabírat do podrobností, neboť to je často náplní jiných modulů, které se takové problematice věnují více.
Na konci kapitoly by mělo být zřejmé, co lze od takových jazyků očekávat. Zejména by mělo vyplynout na povrch, kde se s takovými jazyky můžeme setkat, na jaké typy úloh se používají. Deklarativní programování by tak mělo dostat jasný ráz.

Průvodce studiem

Tato kapitola předpokládá dobrou orientaci v základních pojmech z klasifikace a popisu imperativních programovacích jazyků a terminologií s tím spjatou. Dále předpokládá znalost deklarativního paradigmatu a znalost předchozích kapitol.
Ke studiu této kapitoly není třeba žádného speciálního vybavení. Jako u většiny ostatních se jedná o to si zapamatovat danou terminologii a pochopit klíčové principy předkládaných konceptů. Pro hlubší studium, nebo bližší objasnění prezentovaných skutečností je vhodné využít informací na Internetu a vlastní praktické ověření vlastností na konkrétním jazyce.

Obsah

Výklad

6.1 Jazyky pro definici a manipulaci dat

Jazyky pro definici a manipulaci dat, označujeme dle anglických zkratek:

- DDL — Data Definition Language
- DML — Data Manipulation Language

Mezi jejich typické vlastnosti patří, že mají velmi často nižší vyjadřovací sílu, než Turingovy stroje (není možné jimi pospat libovolný algoritmus). Dále lze zpozorovat, že i když existuje formální základ jejich syntaxe, tak často jsou velice blízké přirozeným jazykům, respektive angličtině. Proto je také možné to, co popisují, celkem jednoduše přečíst a pochopit, jen se slabou znalostí jazyka, ale tvorba takových konstrukcí je již o hodně složitější.

nižší vyjadřovací síla

blízko přirozeným jazykům

Vytvářené akce/operace/funkce/dotazy/příkazy typicky nemají pojmenování a není možné se tedy na ně odvolávat a takto strukturovat. Často se to obchází tím, že jsou součástí skriptovacího jazyka, kde je toto nějakým způsobem umožněno, nicméně vzájemná vazba možná není, pouze je možné si něco pojmenovat a to potom přímo použít. Pojem procedury, či funkce znám z imperativních jazyků tu nemá smysl. Jedinou možností jsou jazyky skriptovací, či podobné, které nějaká DML či DDL jazyk obalují a dodávají jim tak jiný ráz.

pojmenovávání

Často tyto jazyky obsahují velké množství vestavěných klíčových slov a funkcí, či operátorů. Kromě toho, ač existují standardy (např. SQL99), tak pro různé databázové systémy jsou různě rozšiřovány a upravovány, aby poskytly vlastnosti vycházející z implementace systému řízení báze dat a nabídly tak nějakou výhodu oproti ostatním implementacím.

Při zpracování takových jazyků rozhoduje fakt, zda existuje gramatika. Potom je zpracování poměrně jednoduché, neboť syntaktická analýza se dá zvládnout běžnými prostředky. Sémantická analýza je však závislá na okamžitém stavu systému řízení báze dat (kontrola, zda lze tabulku smazat, se odvíjí od toho, zda taková tabulka existuje, zda k ní uživatel má přístup, má práva apod.). Pokud se jedná o dotaz, který čte z báze dat, tak výsledkem sémantické analýzy a přípravy vykonání dotazu je i postup, v jakém se jednotlivé kroky mají vykonat. Tento postup se opět může měnit na základě stavu báze dat, protože jisté pořadí může vyhovovat pro daný stav dat, ale pro jiný už ne.

sémantická analýza datově závislá

Požadavky uživatele jsou vyhodnocovány okamžitě s tím, že požadavek je interpretován, po kompletní analýze a převodu na postup

interpretace

vykonání základních operací. V průběhu vykonání požadavku vznikají různé mezivýsledky, které vstupují do dalších operací, než dostaneme požadovaný výsledek. Právě optimalizace zmíněné výše vedou k tomu, aby takové mezivýsledky byly co nejmenší a vyhodnocení požadavku co nejrychlejší. Pro tyto optimalizace si systém řízení báze dat udržuje různé informace.

distribuované databáze

Různé systémy nabízejí různá specifika. Např. *distribuované databázové systémy* vyžadují opravdu perfektní přípravu dotazu, neboť přesun velkého množství dat po síti by byl pochopitelně nežádoucí, zvláště pokud by to bylo zbytečné. Kromě klasických optimalizačních postupů se nasazují postupy pro paralelní vyhodnocení (jednotlivé kroky dotazu mohou být vyhodnoceny na různých uzlech systému). V potaz se při optimalizacích bere i to, že doba přesunu dat mezi jednotlivými uzly trvá nějakou dobu.

databáze pro speciální typ dat

Prostorové, temporální, multimediální (a další) databázové systémy zase disponují vestavěnými operacemi pro manipulaci specifických dat, se kterými se u databází pro obecné nasazení nesetkáme. Kromě specializovaných indexačních algoritmů jsou to třeba operace z analytické geometrie, temporální logiky, či zpracování bitmapových obrazů. Správná volba databázového systému potom může ušetřit spoustu času při programování aplikační části!

uložené operace

Poslední skupinou databázových jazyků tvoří jazyky pro manipulaci objektových databází, či databází aktivních. U takovýchto systémů je uložen kód metod, či funkcí pro odezvu na události v systému uložen přímo v databázi. Často je tak uložen ve dvojí formě. Jednak ve formě zdrojové, pro případnou změnu od uživatele, jednak ve formě připravené k vykonání interpretačním strojem systému řízení báze dat. Předzpracování takového kódu může snížit čas potřebný pro analýzu, avšak optimalizace dotazu nemusí reflektovat okamžitý stav dat v systému. Najít správné vyvážení je proto obtížné. Jazyky DDL a DML jsou v takových případech obaleny nějakým skriptovacím jazykem, nebo má dotazovací jazyk zcela speciální formu, kdy je výpočetně úplný. Potom je ale často imperativní.

6.2 Jazyky s grafickým rozhraním

Typickými představiteli těchto jazyků jsou digramy datových či signálových toků, což anglicky zapisujeme a zkracujeme takto:

- DFD — Data Flow Diagrams
- SFD — Signal Flow Diagrams

Někdy je možné do této třídy jazyků přiřadit i stavové diagramy, či konečné automaty, ale díky jejich přirozené sekvenční povaze a možnosti přirozené implementace my tyto diagramy sem řadit nebudeme.

tvorba a uložení programu

Tvorba programu v těchto systémech se hodně podobá kreslení v programu pro vektorovou grafiku (CAD systémy, apod.). Často jsou manipulovány i podobné objekty, dají se otáčet, propojovat, apod. Tento zápis se potom ukládá do formy vhodné pro další zpracování, jak editaci, tak převod na cílový kód. I když to může být i textová forma, tak je často nevhodná pro přímý zápis programu (např. XML). Stejně tak se ale může jednat o vhodnou binární formu.

simulace

Pro odladění takových programů je typickou součástí vývojového prostředí simulátor, který nám umožní testování nanečisto. Cílová platforma je často odlišná od té, kde program vyvíjíme. Díky tomu, že simulace pracuje s jiným hardware, může docházet k simulaci bloků, které v cílové platformě jsou ve formě hardware. Simulace potom může být velmi náročná, a to nejen výpočetně, ale zejména vzhledem k věrnosti chování cílové platformy.

zpracování

Analýza takových jazyků částečně probíhá už v době tvorby diagramu (způsoby propojení, chybějící propojeni, apod.). Před simulací, či generováním cílového kódu však proběhne kompletní analýza, která zkontroluje korektnost diagramu. Potom se vstupuje do části, kdy dochází k serializaci operací. Tato by nemohla proběhnout bez náročné a důkladné analýzy. Po takové přípravě je možné spustit interpret, zejména pro simulační a ladící účely. Nebo se přistupuje ke generování cílového kódu. Takový kód však nemusí být vhodný pro danou architekturu a časové požadavky. Často tedy můžeme volit z různých strategií generování cílového kódu, nebo musíme zvolit jinou cílovou platformu, která má nadbytek výpočetního výkonu pro zpracování programu. Díky své povaze potom takovéto jazyky a vývojové systémy nacházejí uplatnění v nejnáročnějších aplikacích pracujících v reálném čase, např. leteckého průmyslu.

zpětný převod takřka nemožný

I když u podobných systémů dochází ke generování cílového kódu do imperativních programovacích jazyků, tak se tato reprezentace nepoužívá pro přímý vstup programátora. Je to dáno tím, že zpětná rekonstrukce diagramu nemusí být možná, nebo jednoznačná. Snahou však je, aby drobné změny v *uživatelských částech kódu* byly promítány zpět do diagramu.

diskrétní zpracování spojitých systémů

I když takovéto diagramy popisují spojité chování a spojité systémy, tak ke zpracování dochází na diskrétní bázi. Může se tedy často stát, že díky tomuto rozporu se právě v časové rovině může objevit problém. Někdy pomůže zrychlení cílové platformy, ale někdy je třeba změna algoritmu.

6.3 Srovnání deklarativních jazyků

Pro deklarativní jazyky je možné vypozorovat některé společné a typické vlastnosti, které nalezneme u všech, jsou to zejména:

- před vyhodnocením, či generováním kódu dochází k serializaci kódu, která je řešena kompilátorem, či interpretem;
- pokud je porovnáme s jazyky imperativními, tak nám vždy nabízejí „něco navíc“;
- úroveň abstrakce je často mnohem vyšší (hardware je často úplně schován a není možné na něj nijak přistoupit;
- naprostá většina těchto jazyků má jasnou a často i přímou vazbu na nějakou formální bázi, na které je postavena.

Mezi jednotlivými typy deklarativních jazyků však můžeme najít i řadu odlišností:

- složitost analýzy je různá (Prolog versus Haskell versus SQL);
- práce s typy a jejich zpracování je různé (Prolog versus Haskell);
- implicitní možnosti paralelizace či serializace se různí (Prolog versus SQL);
- výpočetní síla (SQL versus Haskell);
- úroveň deklarativity (Haskell versus Prolog);
- typická aplikační doména.

Proč bychom měli tyto jazyky využívat? Někdy nemáme ani jinou možnost (např. SQL). Nebo se jedná o specifickou doménu. Případně potřebujeme verifikovat algoritmus. Nejčastěji by to ale mělo být především pro vlastnosti, co tyto jazyky nabízejí, a tak umožňují mnohem efektivnější tvorbu programů na vyšší úrovni abstrakce.

Jaké jsou důvody proti jejich využití? Může to být pomalejší běh na klasických architekturách, zejména díky interpretaci, ale tento vliv už často s kompilátory mizí. Vážnějším důvodem může být penetrace těchto systému, podpora vývojářům apod. V případě komerčních produktů však ani toto není pravda.

Co napsat závěrem? Role deklarativních jazyků a jejich nezastupitelnost je zřejmá, i když nejsou tak rozšířené. Vyšší úroveň abstrakce a větší síla pro programátora je vykoupena náročnou analýzou a překladem. Velice často však v takových jazycích najdeme nejnovější koncepty z informačních technologií. Na druhou stranu architektura, která by byla vhodná pro přímý překlad těchto jazyků chybí, i když v minulosti takové pokusy byly (např. pro funkcionální jazyky).

Pojmy k zapamatování

Pojmy v této kapitole se točí kolem dvou typů deklarativních jazyků. Ty zajímavé jsou vytaženy na okraj textu a to jsou také ty, které byste měli plně ovládat. Pojmy však nevytrhávejte z kontextu a snažte se zvládnout celou problematiku jako celek. Jedině tak vám jednotlivé pojmy vytvoří celkový obraz.

Závěr

Cílem kapitoly bylo základní seznámení s jazyky pro definici a manipulaci dat a s jazyky s grafickým programátorským rozhraním. V závěru kapitoly jsme potom srovnali deklarativní jazyky jako celek. Měli byste tak být schopni takové jazyky nejen rozpoznat, ale správně určovat jejich možnosti a tak správně rozhodnout, zda jejich užití může být přínosné, případně co od jejich užití lze očekávat.

Úlohy k procvičení:

1. Prostudujte jazyk UML. Které jeho části mají operační charakter a šly by transformovat do proveditelného kódu např. nějakého imperativního jazyka.
2. Jaká rozšíření je třeba dodat k vybraným částem UML, abyste mohli vytvořit kompletní programy pro libovolný algoritmus?
3. V jazyku SQL definujte takový dotaz, na kterém budete demonstrovat, že lze docílit stejného výsledku při jeho vyhodnocení různými kroky.

Další zdroje

Řadu informací k této kapitole najdete v popisech či tutoriálech k daným typům jazyků. Spoustu obecných pojmů a rozšíření lze nalézt pochopitelně na Internetu.

Kapitola 7

Závěr

Publikace představuje v několika kapitolách typické rysy několika základních tříd deklarativních jazyků. Soustřeďuje se na zvládnutí těchto kategorií jednak z hlediska vlastního užití těchto jazyků, jednak z hlediska zpracování těchto jazyků, přitom si vybírá jen některé třídy pro zevrubný průzkum a ostatní zmiňuje jen informativně.

Po absolvování by student měl být schopen v rámci prezentovaných kategorií rozpoznat a začlenit různé deklarativní programovací jazyky. Měl by zvládnout pochopení jejich vyhodnocení a tím i výhodnost pro užití v určitých případech, či možnost širšího uplatnění vůbec. Výklad popisující zpracování by měl umožnit absolventovi lépe se orientovat při vývoji programového vybavení v jazyku s danými vlastnostmi.

Modul je posledním modulem a navazuje na moduly, které pojednávají šířeji o imperativních jazycích a o objektově orientovaných jazycích. Pro doplnění znalostí jsou vhodné moduly pojící se s výkladem nějakého konkrétního programovacího jazyka, které ale leží mimo tento studovaný předmět.

Modul obsahuje pouze nejnutnější informace a nástiny problematiky. Pro úplné zvládnutí problematiky je vhodné rozšířit si vědomosti nějakou vhodnou doporučenou literaturou jak z oblasti teorie programovacích jazyků, tak třeba z okrajových témat.

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.