Vysoké učení technické v Brně

Fakulta informačních technologií

Objektově orientované programování v Prologu

Ročníkový projekt

Brno květen 2002 Tomáš Kubíček

Čestné prohlášení

Prohlašuji, že jsem tento ročníkový projekt vypracoval samostatně pod vedením Ing. Vladimíra Janouška, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

Také bych rád na tomto místě poděkoval svému vedoucímu ročníkového projektu Ing. Vladimíru Janouškovi, Ph.D. za poskytnutí mnoha cenných rad a materiálů.

V Brně 17.5.2002 Tomáš Kubíček

Abstrakt

V této práci se zabývám možností propojení několika různých programovacích nástrojů a technik. Zejména jsem se zaměřil na spojení tak odlišných přístupů jako jsou nástroje využívané v oblasti objektově orientovaného programování a několika zástupců tradičního logického programování. Ve světě byla tato tématika v popředí zájmu „programátorské“ veřejnosti počátkem 90. let, kdy probíhalo mnoho diskusí, vzniklo několik zajímavých myšlenek na toto téma a malé procento přerostlo i do fáze implememntací. V současné době se tímto směrem ubírá pouze malá skupina specialistů. Jelikož však český Internet postrádá jakoukoli ucelenou informaci o této bezesporu zajímavé myšlence, sesbíral jsem několik podkladů a vytvořil stránky, které se této problematice věnují.

Klíčová slova:
Objektově orientované a logické programování (OOLP), třída, objekt, klauzule, predikát, slot, delegace, zapouzdření, dědičnost, PROLOG, PLUTO, LogTalk.

Obsah

1 Úvod **3**

2 Teorie [1] **4**
2.1 Jazyky, které spojují OOP a LP. 6
2.1.1 ESP:The Fifth Generation (Pátá generace) 6
2.1.2 ORIENT84/K: Souběžnost OOP a LP 6
2.1.3 OBJECT-PROLOG: V jednotě je síla 7
2.1.4 KSL/LOGIC: Reflexivní a rozšiřitelný 8

3 LogTalk: popis a ukázky použití [2] **9**
3.1 Architektura 10
3.2 Sloty 11
3.3 Události a monitory 12
3.4 Reprezentace vztahů mezi objekty 13
3.5 Ostatní důležité rysy 14
3.5.1 Dokumentování programů 14
3.5.2 Obsluha výjimek 14
3.5.3 Ladění programů 14
3.6 Implementace 15
3.7 Závěr 16

4 PLUTO: popis a ukázky použití [3] **17**
4.1 Typy a hodnoty 17
4.1.1 Definice atributů 18
4.1.2 Metody 20
4.1.3 Přetížení metod a polymorfismus 25
4.2 Hierarchie tříd a dědičnost 25
4.2.1 Dědičnost pomocí změny a vynechání atributů nebo metod . . 26
4.2.2 Vícenásobná dědičnost 27

4.3 Programování v Plutu 28
4.4 Závěr 29

5 Závěr **30**

Kapitola 1

Úvod

Cílem tohoto ročníkového projektu je vytvořit na českém Internetu webové stránky, které poskytují ucelenou informaci o problematice propojení objektově orientovaných programovacích nástrojů s technikami logického programování. Bylo nutné sesbírat podklady k této tématice v anglickém jazyce a z nich sestavit tento materiál.

OO filozofie definuje mocný nástroj vývoje životního cyklu kombinací abstrakce, zapouzdření a modularity. Avšak programování a ladění v OO jazycích v současnosti používaných je složité a klade spousty překážek při vývoji i ladění. Logické programování umožňuje řešení problémů popsaných deklarativním způsobem, kdy známe pouze cíl, kterého chceme dosáhnout, ale nemusíme znát konkrétní kroky řešení. Nástroje logického programování bohužel neposkytují silnou podporu pro tvorbu rozsáhlých aplikací. V minulosti myšlenka spojení OO a LP vzbudila velký ohlas a bylo navrženo mnoho takových programovacích jazyků. Ty však většinou ztroskotaly při implementaci základních OO nástrojů.

V této práci se snažím popsat několik takových pokusů ať už více či méně úspěšných. Zaměřil jsem se zejména na dva zástupce tohoto směru a to na Pluto a LogTalk. Popisuji, jak jsou v těchto konkrétních řešeních implementovány jednotlivé nástroje objektově orientovaného přístupu v prostředí logického programování, jako jsou např.: definice objektů a tříd, jejich atributů a metod, některé formy dědičnosti, zapouzdření, skrývání informací a polymorfismus.

První kapitolou je Úvod, jehož řádky právě pročítáte. Následuje kapitola nazvaná Teorie, která obsahuje veškeré získané myšlenky o zpracovávaném tématu. V dalších částech LogTalk a PLUTO rozebírám podrobně syntaxi a strukturu těchto programovacích jazyků. V další kapitole Implementace osvětluji, jak vznikaly webové stránky, které jsou výsledkem tohoto ročníkového projektu. V Závěru shrnuji celou práci, její přínos a možnosti navázání na ni.

Kapitola 2

Teorie [1]

Objektově orientované a logické programování jsou dvě nezávisle na sobě se rozvíjející odvětví počítačové vědy.

Nabízí lepší programovací techniky než tradiční procedurální jazyky. Logické programování se vyvinulo počátkem 70. let přímým vývojem z důkazů založených na automatech a umělé inteligence [12]. LP je založeno na prefixové logice formulované ve větách teorie důkazů a teorie modelování. Teorie důkazů poskytuje formální specifikaci pro správné vyjádření vlastních myšlenek, zatímco teorie modelování analyzuje jak vyjádřit obecná tvrzení s ohledem na množinu daných faktů. Prefixová logika nebyla používána až do uvedení jazyka PROLOG (jazyka pro PROgramování v LOGice). Prolog používá zúženou formu obecných technik důkazů a tím je efektivnější a lépe se v něm programuje. Umožňuje řešení problémů vyjádřené deklarativní formou, aniž bychom naprogramovali, jak má být problém řešen.

Od 60. let je OOP nejpopulárnější programovací technika. Definuje prostor jako množinu vzájemně se ovlivňujících objektů uchovávajících data a chování. OO filozofie definuje mocný nástroj vývoje životního cyklu kombinací abstrakce, zapouzdření a modularity. Abstrakce umožňuje vynechat detaily a zaměřit se pouze na důležité prvky. Zapouzdření odděluje vnější specifikaci od vnitřní implementace. Modularita umožňuje souhru, srozumitelnost a symetrii organizováním prostoru do skupin úzce souvisejících objektů.

Spojení OO a LP se koncem 80. let dostalo do popředí zájmu. Cílem bylo získat to nejlepší z obou přístupů. Několik postupů zahrnujících OOP v PROLOGu skončilo ve fázi diskusí [8, 15, 16], několik dalších jazyků jako např. OOLP [6], C-Logic [5], L&O [13], Prolog++ [14], COMPLEX [7], LIFE [4] bylo vymyšleno a některé z nich i implementovány. Žádný z nich však nepodporuje všechny důležité vlastnosti OOP.

Jedním z hlavních směrů, v nichž došlo k výše zmíněnému spojení obou přístupů

jsou mimo jiné také KR[1] úlohy a KBS[2].

Proniknutím do rodiny LISP (LOOPS, FLAVORS, CLOS) a dále pak do průmyslu (KEE, ART, NEXPERT-OBJECT) se OOP vydává svou vlastní cestou. Důležitou výhodou, kterou OOP vnáší do KBS je možnost implementovat veškeré komponenty KBS (báze znalostí, uživatelské rozhraní atd.) výhradně pomocí OOP.

Vynecháme-li hodnoty SW inženýrství, používá OOP prakticky onen stejný přístup k programování, který je znám např. u FORTRANu, PASCALu nebo C. Není sice vhodný pro KR úlohy, ale ostatní nástroje jako např. programování na základě pravidel, omezení a přístupů byly samozřejmě implementovány. Rodina LISP byla opět první, která experimentovala s nástroji OOP, nedosáhli však pronikavějších výsledků. Rodina AI[3] koketovala před cca dvaceti lety s něčím podobným jako OOP, ale až LOOPS a CLOS byli první, kteří použili OOP jednotně a systematicky a implementovali tak zásadní konstrukce jako dědičnost, zasílání zpráv a polymorfismus.

Další přístup, jehož aplikace v AI a KBS měly vždy své příznivce a odpůrce, je *Logické Programování* (LP). LP je z následujících důvodů výhodnější pro KR aplikace:

- LP je deklarativní: uživatel musí znát pouze hlavní body problému a jeho cíl, ale ne jednotlivé kroky řešení.
- LP jazyky mají čistou a snadno dosažitelnou logickou semantiku, která je obsažena v mnohých aplikacích v rozsahu od vyjádření složitostí algoritmů až po objasňování vlastností programů a jejich bází znalostí.

Oponenti LP tvrdí, že LP není vhodné pro AI aplikace, protože využití v jiných vědních oborech je mnohem podstatnější. Tento argument není dostatečně přesvědčující. Logika je pouze částí našeho chápání světa, avšak důležitou. Mnoho aplikací v PROLOGu vzniklo pod záštitou projektu Fifth Generation Computer Systems (FGCS) v Japonsku. Velmi brzy se však zjistilo, že PROLOG sám o sobě nemá dostatečné množství potřebných SW vybavení pro implementaci rozsáhlých KBS. Daniel Bobrow ve své práci „*Je-li PROLOG odpověď, jak zní otázka?*“ [17] uvádí, že PROLOG může být propojen jak s funkcionálním tak OO přístupem. K řešení různých problémů je dobré využívat různé přístupy. Je však důležité plně využít jejich potenciály při vyrovnaném propojení pokud jde o data a kontrolu provádění. V projektu FGCS se pracovalo na přizpůsobení PROLOGu KR úlohám a zároveň směřovali k současnému provádění. Ukázalo se, že je velmi jednoduché modelovat objekty pomocí současného provádění predikátů PROLOGu, které zasílají zprávy jeden druhému přes potencionálně neohraničenou frontu. V ní jsou zprávy částečně vyhodnocovány. Což

[1] Knowledge Representation = reprezentace (výklad) znalostí
[2] Knowledge-based Systems
[3] Artifitial Inteligence = Umělá inteligence

znamená, že predikát nebo objekt začne plnit požadavky, jakmile je ve frontě alespoň jedna zpráva a nečeká na úplné naplnění fronty. Po provedení požadavku zavolá objekt rekurzivně sám sebe s argumentem obsahujícím zbytek fronty zpráv. Proces dědičnosti může být zde dosažen pomocí zasílání požadavků. Existuje mnoho jazyků založených na tomto modelu: SPOOL, VULCAN, SCOOP, POLKA.

2.1 Jazyky, které spojují OOP a LP.

V této kapitole uvádím příklad několika programovacích jazyků, které spojují oba popisované přístupy.

2.1.1 ESP:The Fifth Generation (Pátá generace)

ESP (Extended Self-contained PROLOG) [18, 19, 22] nebyl původně navržen přímo pro KR; je to jazyk pro popis systému SIMPOS (Sequential Inferential Machine Programming and Operating System), operační systém PSI (Personal Sequential Inference), které tvoří jádro projektu FGCS. Byl vytvořen nad jazykem KL0 (Kernel Language version 0). Některé speciální příkazy a vestavěné predikáty KL0 byly vytvořeny konkrétně pro ESP. ESP je OO jazyk s proměnnými stavy, třídami objektů a hierarchickou vícenásobnou dědičností. Zjistilo se, že právě popsané složky ESP velmi zjednodušili implementaci systému SIMPOS a jsou také vhodné pro ostatní úlohy (zejména ty, které vyžadují hierarchickou reprezentaci znalostí).

Následující příklad ukazuje, definici třídy `ptak` s metodou `letat`. Druhá deklarace definuje třídu `tucnak`, která z třídy `ptak` zdědí všechny vlastnosti vyjma metody `letat`:

```
class ptak has instance
    :letat(ptak).
    ...
end.
class tucnak has nature ptak; instance
    :letat(tucnak) :- !, fail.
    ...
end.
```

2.1.2 Orient84/K: Souběžnost OOP a LP

Systém Orient84/K [20, 21]je množina spolupracujících KO[4] složených z několika částí (chování, znalosti, monitor). Část obsahující chování objektu poskytuje jako ve

[4]Knowledge Objects=Objekty reprezentující znalosti

SMALLTALKu imperativní metody, které podporují sekvenční kontrolu a provádění úloh. Jsou zde také metody pro přístup a modifikaci části znalosti. Část obsahující znalosti je stejně jako v PROLOGu lokální báze pravidel a faktů popisující vlastnosti objektů. Část monitor obsahuje tzv. démony, což jsou metody, které slouží k ošetření výjimek a dalších speciálních případů. Zmíněné tři části jsou pro OOP a LP nezbytné. V celkovém dojmu vypadá ESP jako rozšíření SMALLTALKu. Jeho implementace je založena více na OOP než na LP. Příklad ukazuje, že znalosti jsou umístěny uvnitř objektů.

```
DK0 subclass: #AClass
    instanceVariableNames: 'i1 i2'
    classVariableNames: 'c1 c2'
AClass methodsFor: 'metody instanci'
    <telo metody>
AClass knowledgeFor: 'instance KB'
    <Klauzule Horn>
AClass monitorsFor: 'monitory instanci'
    <tvrzeni (demon)>
```

2.1.3 OBJECT-PROLOG: V jednotě je síla

Struktury OBJECT-PROLOGu mají pouze jeden druh entit nazývaný worlds, který má mnoho významů: instance, třídy, metatřídy; mají jednoznačná jména. Každé „world“ obsahuje lokální množinu tzv. Horn klauzulí. „Worlds“ mohou být deklarovány staticky

```
world symfonie9.
    je-hudebni-dilo.
    slozil-kdo bethoven.
endworld.
```

nebo dynamicky:

```
?- createWorld(symfonie9),
    addKnowledge(je-hudebni-dilo) >> symfonie9,
    addKnowledge(slozil-kdo bethoven) >> symfonie9.
```

Následně je pak možné použít `deleteKnowledge`. Je velmi podstatné, že toto vytvoření „worldu“ je úplně zpětně krokovatelné. Operátor `>>` znamená zaslání zprávy „worldu“. OBJECT-PROLOG poskytuje uživateli všeobecný vzor pro vytvoření dědičnosti a umožňuje mu jej upravit podle vlastních potřeb. A toto zahrnuje dědičné vztahy mezi objektem a třídou, třídou a nadtřídou, třídou a metatřídou. Obecná forma dědičnosti v OBJECT-PROLOGu je následující:

```
world1 inherits predicate from world2 [:- condition].
```

Je-li splněna podmínka condition, zdědí `world1` predikáty, které odpovídají zadanému predicate a jsou-li pravdivé ve `world2`.

2.1.4 KSL/LOGIC: Reflexivní a rozšiřitelný

KSL/LOGIC je OOP jazyk KSL (Knowledge Specification Language) rozšířený prvky LP. Jednoznačným cílem při tvorbě KSL byla možnost sjednocení několika různých programovacích technik. KSL/LOGIC je tedy jedním z příkladů takového spojení. Aby toto spojení bylo možné, jsou jazykové konstrukce reprezentovány pomocí datových objektů. To způsobilo, že jazyk může prakticky spravovat sám sebe. Což je na jedné straně velmi užitečné pro implementaci takových nástrojů jako např. překladače a ladicí programy a navíc může být jazyk jednoduše rozšířen pouhým přidáním nových tříd. Těchto výhodných vlastností bylo dosaženo podobně jako v LISPu několika omezeními a sjednocením syntax jazyka. Vše v KSL (hlavičky metod, deklarace proměnných, příkazy atd.) je reprezentováno prakticky pomocí několika druhů seznamů.

Následující příklad, který vypočítá věk osoby, ukazuje strukturu jazyka:

```
(#Method
    (&Selector Vek)
    (&Class Osoba)
    (&ParmList (#List Self))
    (&VarList (#List Rok_narozeni Mesic_narozeni))
    (&ExprList (#List
        (Set Rok_narozeni (Rok (Datum_narozeni $Self)))
        (Set Mesic_narozeni (Mesic (Datum_narozeni $Self)))
        (When
            (#List (GreaterThan $Mesic_narozeni $Stavajici_mesic)
                (Subtrackt (Subtrackt $Stavajici_rok $Rok_narozeni) 1))
            (#List True
                (Subtrackt $Stavajici_rok $Rok_narozeni)
)))
```

Kapitola 3

LogTalk: popis a ukázky použití [2]

Logtalk je objektově orientovaná nadstavba programovacího jazyka Prolog. Je založen na reflexivní architektuře zaměřené na získání otevřeného systému, snadno uzpůsobitelnému potřebám uživatele. Je v něm implementována jednoduchá dědičnost a mechanismus delegace[1]. Logtalk umožňuje programátorům definovat nové typy slotů[2], každý s vlastní sémantikou, které mohou být použity libovolnými objekty. Logtalk obsahuje rysy programování založeném na událostech, umožňuje velmi dobře reprezentovat a udržovat vztahy, které omezují vnitřní stavy použitých objektů.

Programovací jazyk PROLOG [23] je znám svou schopností deklarativního způsobu popisu toho, co je pravdivé ve své oblasti (Prolog obsahuje vestavěný deduktivní mechanismus, který umožňuje najít důsledek nových skutečností našeho popisu oblasti). Na druhou stranu, objektově orientované programovací jazyky se nejlépe hodí pro popis struktury a chování objektů. Potřeba sloučit tyto dva různé přístupy vyústila v několika rozšířeních, které zjednodušují objektově orientované programování v Prologu [25, 29, 35]. Tato rozšíření mají společnou interpretaci teorie objektů, co se týče teorie, a princip zasílání zpráv jako důkaz stavby (v logickém smyslu). Naopak se liší úrovní strukturování objektů a v roli destruktivního přiřazování. Problémem většiny takovýchto rozšíření je absence otevřené architektury, která by umožnila experimentování s rozdílnými přístupy k objektově orientovanému programování. Logtalk se snaží toto vyřešit používáním reflexivní architektury.

Další otázkou, na kterou se Logtalk snaží najít odpověď je složitost jazyků jako Smalltalk [26] nebo C++ [34] co se týče reprezentace a údržby vztahů mezi objekty. I

[1] zasílání zpráv mezi objekty

[2] spojení pojmu atribut nebo metoda třídy

když definujeme vztahy pomocí tříd, je nezbytné přidat do objektů extra část kódu, aby vztahy byly upozorněny na změny, které mohou ovlivnit jejich konzistenci či platnost. Zde prezentované řešení je integrací schopností programování orientovaného na události v Logtalku.

3.1 Architektura

Reflexivní architektura Logtalku může být popsána následujícími požadavky:

1. Každý objekt je instancí nějaké třídy, ke které náleží.
2. Třída je objekt definovaný jinou třídou (metatřídou).
3. Třída může být popsána specializací nějaké jiné existující třídy (nadtřída)
4. Objekt se skládá z množiny slotů, definujících svůj stav a chování a z vnitřní základny.
5. Sloty objektů jsou deklarovány svojí třídou a všemi nadtřídami této třídy, tak jak to definuje mechanismus dědění.
6. Reprezentace a chování slotů je definováno objektem stejného typu jako příslušný slot.
7. Slot je brán delegací zamýšlené operace svým typem, při použití zásobníkového objektu jako klienta.
8. Každý výpočet je započat zasláním zprávy nějakému objektu.

Jak je vidět, objekt má strukturovanou část tvořenou sloty a nestrukturovanou část, svoji vnitřní základnu. Množina slotů je strukturovaná, protože každý je explicitně deklarován třídou. Obsah vnitřní báze není vázán žádným typem deklarace. Díky tomu je interní báze privátní a lokální ke každému objektu. Je normálně používána třídami k uchovávání pomocných definicí vyskytujících se v metodách (blok chování).

Delegace (na základě osmého požadavku) je spravována mechanismem zasílání zpráv, ve kterém objasníte klienta požadované operace. Protože sloty lze manipulovat pouze díky delegaci, zdá se tento mechanismus být elementárním způsobem zasílání zpráv. Delegace nemusí být používána pouze pro manipulaci se sloty, ale může být též použita pro zjištění odesílatele zprávy kdykoliv to potřebujete. Jednodušší zasílání zpráv, ve kterém není známo, kdo je odesílatel zprávy má výhodu respektování zapouzdření objektů tím, že se nezmiňuje o typu slotu.

Všimněte si, že v Logtalku je implementována jednoduchá dědičnost: každá třída může mít pouze jednu nadtřídu. Absence vícenásobné dědičnosti je částečně kompenzována mechanismem delegace. Ten prosazuje vytváření nových tříd kompozicí místo specializace.

Zavedení metatříd je pro reflexivní architekturu nezbytné. Nezapomeňme, že o systému říkáme že má reflexivní architekturu [28] pokud obsahuje reprezentaci sebe sama, spojenou příčinně se svým chováním. V případě programovacího jazyka to znamená, že je napsán v tom samém jazyce, který implementuje. V jazycích jako je PROLOG, kde není žádného rozdílu mezi procedurami a daty, je reflexivní architektura implementována kruhovým meta-překladačem[3]. Způsob, jakým je kruhovitý charakter porušen zavádí množinu rysů, které nemohou býti změněny uživatelem.

Tento koncept reflexivity proniká systém Logtalk. Nejdříve ve vztazích mezi základními třídami: `Objekt, AbstraktníTřída` a `Třída`. Objekt deklaruje standardní sloty pro všechny objekty. `AbstraktníTřída` specializuje `Objekt` sloty, které jsou společné pro všechny třídy. Je to standardní metatřída pro abstraktní třídy. `Třída` specializuje `AbstraktníTřídu` přidáváním slotů nezbytných pro zacházení s instancemi. Je to standardní metatřída pro třídy s bezprostředním přístupem. Všechny tři třídy jsou instancemi `Třídy`. Při prozkoumání této malé hierarchie je vidět, že každá třída má svůj vlastní slot a také slot ostatních dvou tříd, aniž by docházelo k cyklení. Za druhé, oproti většině jiných objektově orientovaných jazyků, co se týče reprezentace a správy slotů, mechanismus zasílání zpráv a přiřazovací operace nejsou primitiva, ale metody definované v objektech. Jsou přístupné k prohlédnutí a modifikaci uživatelem. Zaslání zprávy se skládá vlastně z poslání dvou odlišných zpráv: z jedné pro nalezení metody a druhé ke spuštění metody. Za každou z těchto dvou zpráv se skrývá identický proces hledání a spouštění. Abychom se vyhnuli nekonečným smyčkám, Logtalk definuje dva sloty korespondující se dvěma zprávami zabezpečujícími primitiva, když je proces hledání a spouštění přímo vykonán (bez opětovného třídění pro zaslání dalších zpráv).

3.2 Sloty

Slovo *slot* je v tomto textu použito k označení buďto stavového slotu (instance či třídy ve Smalltalku, data v C++) nebo slotu chování (metody ve SmallTalku, funkce v C++). Stejné jméno je použito ze dvou důvodů. Zaprvé, slotem může být jakýkoliv výraz v Prologu (ne proměnná), stejným způsobem jako bloku chování. Zadruhé, každý typ slotu je spravován delegací.

[3]Meta-překladač je program s možností modifikace a spouštění jiných programů napsaných ve stejném jazyce

Deklarace slotu obsahujícího dosud nedeklarované proměnné je stejná jako deklarování množiny slotů (potenciálně nekonečné) která je sjednocena jménem slotu. Například, pokud máme slot z názvem `generátor(Generator)`, dostaneme variantu slotu pokaždé, když vytvoříme proměnnou `Generator`. Jedná se o vhodné řešení, pokud počet variant slotů které aplikace využívá.

Sloty v Logtalku nejsou objekty (evidentně mohou obsahovat odkazy na objekty), ale mají sémantiku definovanou objektem nazývaným typem slotu. Tento objekt definuje množinu dostupných operací a dědičnost hodnot slotu. Např. pokud blok reprezentuje zprávu, jeho typ definuje metody (které mají být použity), hledání a spouštění korespondujících metod. Pokud blok reprezentuje proměnnou, jeho typ reprezentuje metody pro přiřazování hodnot. Tento způsob jak uživatel může definovat své vlastní typy slotů dobře vyhovuje potřebám aplikace. Sémantika slotů se nachází pouze na jednom místě, aniž by byla roztroušena v metodách, které blok používají.

Každá operace slotu je vykonána s použitím mechanismu delegace zpráv [27], kde klientem je objekt obsahující slotu. Například pokaždé když pošleme zprávu, cílový objekt deleguje hledání a spouštění korespondujících metod v typu slotu. Důvěrně známý příklad najdeme ve Smalltalku. V tomto jazyce máme několik typů slotů (instance, třída, pool proměnné, metody instancí a tříd), každý se svou vlastní sémantikou.

Chování je tradičně sdíleno všemi instancemi stejné třídy a stav je ke každé instanci lokální. V Logtalku libovolný slot sdílenou, lokální, nebo obojí. Můžeme mít metody definované lokálně, uvnitř instance, nebo sdílený stav definovaný třídou.

3.3 Události a monitory

V objektově orientovaném systému je každý výpočet započat zasláním zprávy. Je tedy přirozené říci, že jedinou událostí, která může v takovémto systému nastat je zaslání zprávy. Pokud pohlížíme na zpracování zpráv jako na nerozdělitelnou aktivitu, můžeme rozlišovat mezi dvěma událostmi: zaslání zprávy a navrácení řízení objektu, který zprávu vyslal. Tímto způsobem dosáhneme lepší kontroly dynamiky systému. V Logtalku jsou tyto dvě události pojmenované `before` a `after`. Tyto události jsou generovány automaticky při každém zpracovávání zprávy. V situacích, kdy si to nepřejeme, je možné zaslat zprávu tichým způsobem, tj. bez generování žádné události.

Představa pojmu monitor je velmi blízká myšlence události. Monitor je objekt, který je systémem automaticky aktivován pokaždé, když dojde k nějaké události. Proto každý monitor přísluší objektu zodpovědnému za správu událostí (je jeden pro každý druh událostí) - jaké zprávy, k jakým objektům určené - jsou ve hře. Mechanismus správy zpráv používá tuto informaci aby věděl, kdy upozornit monitor že došlo

ke sledované události. Toto upozornění se děje zasláním zprávy monitoru. Pro tento efekt třída `Object` deklaruje dvě zprávy pojmenované `before()` a `after()`, které jsou děděny všemi objekty (v Logtalku může být libovolný objekt monitor). K provedení nezbytných výpočtů každý monitor musí pouze definovat příslušné metody. Jsou dvě zprávy, které jsou monitoru zaslány tichým způsobem automaticky mechanismem správy zpráv. Je důležité uvědomit si roli monitoru jako objektu, stejně tak jako že monitorované události mohou být dynamicky změněny za běhu programu.

Používáním tohoto programového mechanismu založeného na událostech docházíme k několika úrovním poznání, s objekty pozorujícími a monitorujícími jiné objekty bez toho, aniž by jim zaslali zprávu, bez jejich vědomí.

Mezi nejobvyklejší použití událostí patří ladění a profilace programu (metody `before()` a `after()` mohou produkovat zápis zaslaných zpráv). Události mohou být také použity k jednoduchému přidání grafické reprezentace vnitřního běhu aplikace [33], bez nutnosti ji modifikovat.

3.4 Reprezentace vztahů mezi objekty

Normálně jsou vztahy mezi objekty implementovány s použitím slotů obsahujících odkazy na příbuzné objekty. Toto je příčinou různých problémů, včetně duplikace informace mezi zúčastněnými objekty, rozšiřování znalostí vztahů, a potřebu znát všechny vztahy objektu, atd. Jiným řešením je nahlížet na vztah jako na plně uzavřený objekt, uchovávající všechny významné informace [30, 31].

Vysvětleme to na jednoduchém příkladu. Představme si, že máme zásobník slotů, reprezentovaný třídou `Stack`. Bloky jsou reprezentovány jako instance třídy `Block`. `Stack` deklaruje dva sloty jako vrchol a podporu souvisejících objektů. `Blok` deklaruje slot, aby obsahoval koordinaci místa a zprávu pro posun bloků. Nyní mějme na zásobníku dva bloky, `A` a `B` (`A` je na vrcholu). Pokud pošleme `B` zprávu posunu, stejně se bude muset posunout i `A`. Přes toto, `B` vůbec o `A` neví (ani by neměl). Elegantním řešením je, aby instance třídy `Stack` vystupovali jako monitory událostí pohybu spřízněných objektů. V tomto případě, instance `Stack`u poté, co byl upozorněn, že `B` se posunul, musí zaslat zprávu posun bloku `A`, aby zabezpečil prostorový vztah. Takto tedy, pokaždé když nějaký objekt je zúčastněn ve vztahu, kde má dojít ke změně, relace může být automaticky upozorněna, aby byla schopna zajistit nezbytné akce k zachování konzistence.

Pro reprezentaci vztahů mezi objekty definuje Logtalku dvě třídy: `Relation` a `ConstrRelation`. První je užívána pro vztahy bez důsledků pro vnitřní stavy zúčastněných objektů. To znamená že kdykoliv bereme v úvahu nějaký vztah, nezajímají nás vnitřní stavy zúčastněných objektů. Druhá třída, `ConstrRelation`, umožňuje reprezentaci a správu vztahů vynucením vnitřních stavů zúčastněných objektů. Tak

jako v příkladě, relace informuje systém o událostech o kterých si přeje být informován. To budou přesně ty, které mohou měnit stav a hrozí ohrožení konzistence. Metody `before()` a `after()` ve třídě `ConstrRelation` implementují algoritmus který šíří změny do všech ohrožených skupin objektů. Tento algoritmus [24, 32] kontrolou změn předchází cyklení. V krátkosti: pro každou relaci souvisejících objektů obsahující změněný objekt jsou zaslány nezbytné zprávy jiným skupinám objektům. Všimněte si rekurzivní vlastnosti tohoto algoritmu.

3.5 Ostatní důležité rysy

3.5.1 Dokumentování programů

Součástí každého programu je nejen kód, ale důležitou část tvoří též dokumentace. Každý objekt vlastní množinu bloků, které udržují informaci o verzi, autorech, kategorii a cíli. Pro každý blok je zde rozšiřitelný slovník popisující cíl, signaturu, viditelnost, kategorii, počet řešení, možnosti opětovného definování a jiné informace. Možnost vyjádření všech těchto informací v kódu umožňuje vývoj zajímavých aplikací: grafické prohlížeče umožňující uživateli navigaci skrze systém hierarchie objektů, nástroje, které extrahují a formátují dokumentaci. Současná verze používá pro dokumentaci LaTeX2ε a HTML. S využitím deduktivních možností Prologu v budoucnu očekáváme komplexní dokumentaci vztahů mezi objekty (například závislosti mezi metodami různých tříd). Jiná možnost by mohla být používání vnořené dokumentace pro analýzu a ověřování správnosti programů.

3.5.2 Obsluha výjimek

Třída `Object` deklaruje zprávu (děděnou všemi objekty) která umožňuje signalizovat a zpracovat výjimky. Tato zpráva má dva argumenty. Prvním je výraz v PROLOGu popisující výjimku. Druhým argumentem je zpráva spuštění odpovídající metody, která zapříčinila výjimku. Je vidět že obsluha výjimek v Logtalku je založena na a podobné té, kterou můžeme nalézt ve Smalltalku-80. Všestrannost tohoto mechanismu pochází z možnosti vyjádřit detailní popis nastanuvší výjimky.

3.5.3 Ladění programů

Množina tříd a objektů byla definována tak, že s použitím událostí implementují možnost ladění programů v Logtalku. Tyto možnosti jsou podobné těm v PROLOGu. Můžeme si zvolit které objekty a které zprávy sledovat během běhu programu. Je možné sledovaným událostem přiřadit akce, např. zjištění stavu objektu před a po spuštění metody.

3.6 Implementace

Systém Logtalku byl implementován s ohledem na standardy obsažené v ISO Prologu. Snaží se využívat pouze možnosti společné pro většinu jazyků na bázi PROLOGu, ať už komerčních či akademických. Převzatá reprezentace objektů je velmi jednoduchá a efektivní, umožňuje přímou manipulaci bez použití kompilátoru pro překlad mezi definicemi objektů a kódem PROLOGu. Každý objekt je reprezentován jako množina klauzulí. Každá vnitřní báze je reprezentována 1-ární klauzulí. Každý slot je reprezentován 2-ární klauzulí nebo vyšší, kde první argument udává jméno bloku. Metody jsou běžně reprezentovány pravidly Prologu, kde vstupní a výstupní parametry jsou argumenty jména zprávy. Např.:

```
'Class'( legalObjectName(Jmeno) ) :- ...
'Class'( subClassOf,'Abstraktni_trida' ).
'Class'( instances(Instance), Self ):- ...
```

Proměnná `Self` je vytvořena se jménem objektu který obdržel zprávu na prvním místě. V případě delegace, proměnná nazvaná `Klient` udržuje jméno objektu delegace. Např.:

```
'Real'( set(Slot, Hodnota), Klient, Self ) :- ...
```

Zasílání a delegace zpráv je udělána takto:

```
send( :Zprava, Objekt )
delegate( :Zprava, Klient, Objekt )
```

V případě, že generování událostí je nechtěné, můžeme použít:

```
send( !Zprava, Objekt )
delegate( !Zprava, Klient, Objekt )
```

Např.:

```
send( !instance(Je), Trida )
delegate( !set(hmotnost,7.6), kniha, aReal )
```

Všimněte si, že „:“ a „!“ jsou metody ve třídách `send` a `delegate`. Logtalk disponuje několika způsoby jak zaslat zprávu množině objektů (broadcasting) a možností výběru podmnožiny objektů, které daná zpráva uspokojuje (s nebo bez sdílení argumentů). Spojení mezi zprávou a příslušnou metodou je vytvořeno dynamicky s použitím vyhledávacího algoritmu implementovaným v typu slotu. Sledované události jsou deklarované klauzulemi jako:

```
beforeMonitor(Objekt, Zprava, Monitor)
afterMonitor(Objekt, Zprava, Monitor)
```

Pokud jsou například zaslány následující zprávy:

```
delegate( :set(hmotnost,7.6), kniha, aReal )
```

a existuje klauzule:

```
afterMonitor( aReal, set(hmotnost,Hodnota), monitor)
```

systém zašle zprávu:

```
delegate( !after(set(hmotnost,7.6),aReal), kniha, monitor )
```

Pokud chcete využít výhod možností správy událostí, vybraný PROLOG umožňuje indexování prvního argumentu v dynamickém kódu. Takto tedy existence sledovaných událostí nemá vliv na výkon nemonitorovaných zpráv.

Je možné za běhu programu vytvářet, editovat a rušit třídy, objekty a sloty. Tato možnost, která potvrzuje dynamické vlastnosti Logtalku, nalézá využití na poli umělé inteligence.

Logtalk nabízí bohatou knihovnu tříd a typů slotů. Každému typu v PROLOGu odpovídá příslušná třída v Logtalku. Zahrnuty jsou různé třídy pro kolekce a slovníky. Jiné třídy umožňují správ vztahů mezi objekty, ladění a dokumentování programů. Je vyvíjen systém pro řešení problémů prohledávání stavového prostoru.

3.7 Závěr

Reflexivní architektura zavedená v Logtalku z něj dělá otevřený systém snadno přizpůsobitelný potřebám uživatele, který je dobrým základem pro experimentování s OO nástroji. Mechanismus správy událostí umožňuje přirozené rozdělení znalostí mezi objekty při řešení komplexních vztahů. Zajištění mechanismu delegace je zajímavou alternativou k použití vícenásobné dědičnosti. Představa slotu jako generalizace představ metod a proměnných umožňuje stručné a všestranné reprezentace. Možnost definování nových typů slotů umožňuje aplikacím mít pouze potřebnou úroveň složitosti.

Kapitola 4

PLUTO: popis a ukázky použití [3]

Zde si popíšeme Pluto - OOLP jazyk, který podporuje téměř všechny OO vlastnosti na základě logického programování např. identita objektu, definice tříd, zapouzdření dat a metod, skrývání informací, přetížení, pozdní vazba, polymorfismus, hierarchie tříd a veškeré druhy dědičnosti, odchytávání blokování a konfliktů.

Pluto převzalo mnoho charakteristických rysů z ROL2 [11] a Javy. Narozdíl od ROL2, který je spíše jazykem pro DB dotazy, Pluto je převážně programovací jazyk zahrnující mnoho programových struktur, které ROL2 neobsahuje jako např. V/V, konstruktory, nový mechanismus vytváření objektů, způsob přiřazení používaných v PROLOGu, tečkovou notaci a mnoho dalších z Javy a PROLOGu.

Má dva hlavní cíle:

1. Podpořit v základu založeném na klauzulích (clause-based) vše co poskytují OO jazyky, ale na vyšší deklarativní a konceptuální úrovni, čímž se programování usnadní a v důsledku tohoto přestanou být používány některé nižší programovací jazyky.

2. Podpořit v OO směru vše, co poskytují logické programovací jazyky.

4.1 Typy a hodnoty

V Plutu je typ prakticky množina hodnot. Existují čtyři druhy typů: *základní datové, třídy, seznamy, množiny.*

Základní datové typy jsou vestavěné a zahrnují - `integer, real, char, string, void`. Tyto typy nabývají následujících hodnot: `integer, real, char` a

`string` označují množiny celých i desetinných čísel znaků a řetězců, které mohou být součástí systému; `void` označuje jednoduchou množinu obsahující `nil`.

Typ třída označuje množinu objektů. Třídy musí být definovány pomocí definic, které jsou popsány v další kapitole. V Plutu můžeme definovat např. třídu - osoba, student, zaměstnanec. Objekty v množině, které třída vymezuje, nazýváme instance třídy. Tyto jsou vytvářeny `new` operátorem.

Typ množina označuje skupinu homogenních množin, tj. množiny stejného typu. Např. {`integer`} je typ množina obsahující množiny integerů; {`osoba`} je typ množina obsahující množiny osob, pokud je `osoba` definovaná třída.

Typ seznam označuje skupinu homogenních seznamů. Např.(`integer`), (`osoba`), jsou dva příklady typů seznam.

Pluto je OOLP. Různé třídy definujeme definicemi tříd, které vypadají takto:

`[abstract] class` c `[isa` $c_1, ..., c_n$`]`[$definice\_atributu$ $definice\_metod$]

kde c je třída, $c_1, ..., c_n$ pro $n \geq 0$ jsou existující třídy, definice atributů specifikují strukturu třídy a definice metod popisuje chování třídy. Vše, co je v [...] je volitelné. Význam klíčového slova `isa` bude vysvětlen.

4.1.1 Definice atributů

Objekty mohou mít atributy, kterými jsou provázány s ostatními. Atributy, které mají mít všechny objekty třídy, *nazýváme atributy* instancí a musejí být deklarovány ve třídě. Definice atributu instance má v Plutu následující tvar:

`[public|private][instance]` $a \Rightarrow \tau$ `[default` ν`];`

kde a je atribut, τ je typ a ν je hodnota. Toto specifikuje, že a je atribut třídy τ definovaný tak, že pokud tento atribut konkrétního objektu nenabývá žádných hodnot, je použita implicitní hodnota ν daná definicí.

Předpokládejme následující příklad:

```
class osoba [rok_narozeni ⇒ integer(1900..2000) default
1970; druh ⇒ osoba;
pohlavi ⇒ char({'M','F'}) default 'F'; rodice ⇒ {osoba}]
```

Takto deklarujeme atributy `rok_narozeni`, `pohlavi`, `druh` a `rodice` třídy `osoba` jako základní datové typy `integer(1900..2000)`, `char({'M','F'})`, sebe sama a typ množina {`osoba`} a pokud nejsou zadány hodnoty pro atributy `rok_narozeni` `pohlavi` jsou použity počáteční hodnoty.

Pokud jsou `bob` a `pam` jména dvou instancí třídy osoba, pak `bob.rok_narozeni` $\rightarrow$ `1960` a `pam.druh` $\rightarrow$ `bob` specifikují, že objekt určený názvem `bob` má hodnotu atributu `rok_narození 1960` a dále objekt určený názvem `pam` má hodnotu atributu `druh` naplněnou ukazatelem na objekt určený názvem `bob`. Tyto hodnoty jsou v rámci předchozí deklarace typu `osoba` zadány správně. Ale nesmí existovat zároveň `bob.rok_narozeni` $\rightarrow$ `1960 a bob.rok_narozeni` $\rightarrow$ `1950`. Jinak řečeno, hodnoty atributů musejí být unikátní. Připouští se také možnost, že hodnoty atributů některých objektů není nutné zadat. Atributy instance mohou mít pouze instance třídy.

Na hodnoty atributů instancí se můžeme dotázat následujícím způsobem:

```
?-pam.druh → S,  S.rok_narozeni → B.
```

Pluto také podporuje tradiční tečkovou notaci. Formálně zapsané $o.a_1.a_2..a_n \rightarrow X$ znamená $o.a_1 \rightarrow X_1, X_1.a_2 \rightarrow X_2, ..., X_{n-1}.a_n \rightarrow X$. Původní dotaz na hodnotu atributu instance může tedy vypadat v Plutu i takto:

```
?-pam.druh.rok_narozeni → B.
```

Má-li třída několik atributů stejného typu: $a_1 \Rightarrow \tau, ..., a_n \Rightarrow \tau$ můžeme toto deklaraci spojit do $a_1, ..., a_n \Rightarrow \tau$. Následuje příklad:

```
class bod[x,y ⇒ real]
```

Jednou ze základních vlastností OO přístupu je skrývání informací. To znamená, že můžeme omezit přístup k hodnotám atributů. V Plutu se používá modifikátor `private` k označení, že k definovanému atributu není možný přímý přístup, zatímco jeho vynechání nebo použití slova `public` značí že k atributu můžeme přistoupit přímo zvenku.

Předpokládejme následující příklad:

```
class zamestanec [jmeno ⇒ string; private plat → integer]
```

Pro instanci třídy je atribut `jmeno` přímo dosažitelné, zatímco atribut `plat` nikoliv. K privátním atributům můžeme nepřímo přistupovat také pomocí metod, což bude vysvětleno později.

V Plutu mají i třídy své atributy. Kvantifikátorem `static` stejně jako v Javě nebo v C++ označujeme, že definovaný atribut je *atributem třídy*. Je-li tento vynechán je definovaný atribut pouze atributem instance. Atribut třídy pak v Plutu definujeme:

```
[public|private] static a ⇒ τ [default ν];
```

kde a je jméno atributu třídy a τ je typ. Tato deklarace říká, že a je atributem třídy τ.

Předpokládejme následující příklad:

```
class osoba [jmeno ⇒ string; vek ⇒ integer;
static soucet ⇒ integer default 0;
static tento_rok ⇒ integer default 2000]
```

kde první dva atributy jsou atributy instancí a zbývající dva jsou atributy třídy. V Plutu se používají atributy tříd pro reprezentaci konstant důležitých pro třídu jako např.: `tento_rok`. Každá instance třídy má své vlastní atributy instance, zatímco atributy třídy jsou atributy konkrétní definované třídy a jsou dostupné pro instance této třídy. Stejně jako atributy instancí můžeme pomocí modifikátorů public a private ovlivnit dostupnost k atributům tříd.

Implicitně je nastaven `public`, pokud však nastavíme `private` znemožníme tím přímý přístup k danému atributu.

Pro přímý přístup k `public` atributům tříd můžeme využít přímo třídu. Sestavíme např. takovýto dotaz:

```
?- osoba.soucet → X.          ?- osoba.tento_rok → X.
```

Přímo přistoupit k atributům tříd můžeme také pomocí jednotlivých instancí tříd. Např. je-li `bob` objekt (instance) třídy `osoba`, pak mají následující dotazy stejný výsledek jako ty předchozí:

```
?-bob.soucet → X.             ?-bob.tento_rok → X.
```

4.1.2 Metody

K objektu jako k instanci dané třídy je možné přistoupit také pomocí metod. Pluto podporuje dva druhy metod: *instancí* a *tříd*.

Metody instance

Metody instancí se používají výhradně k manipulaci s objekty. Tyto musejí být definovány ve třídě a zapouzdřený v definici třídy. Metoda instance v Plutu sestává ze dvou částí: *definice typu* a *implementace*:

`[public|private]` $op(\tau_1, ..., \tau_n) \Rightarrow \tau\{definice\_pravidel\}$

kde $op(\tau_1, ..., \tau_n) \Rightarrow \tau$ je definice typu, která specifikuje *op*, což je jméno metody, typy argumentů metody $\tau_1, ..., \tau_n$ a typ výsledku metody τ. Podstatnou částí definice je $op(\tau_1, ..., \tau_n)$, která bývá nazývána podpisem (*signaturou*) metody a tato musí být unikátní v rámci třídy. Pokud je typ τ *void* můžeme jej při deklaraci vynechat a psát pouze $op(\tau_1, ..., \tau_n)$; nemá-li metoda žádné argumenty, musíme přesto psát $op() \Rightarrow \tau$, abychom tento zápis odlišili od definice atributů. V části definice pravidel je popsána imlementace jednotlivých metod.

Metody instancí můžeme rozdělit v Plutu na pět druhů: V/V, vyvozovací (deduction), aktualizace (updates), konstruktory a vytváření objektů.

V/V Pluto poskytuje dva základní V/V příkazy: `write` a `read` jako v PROLOGu. Použití ukazuje následující příklad:

```
class osoba[rok_narozeni ⇒ integer; pohlavi ⇒
char('M','F');
zobrazOsobu(){zobrazOsobu() :- rok_narozeni → B, pohlavi →
G,
write("Rok narozeni je" + B), write("Pohlavi je" + G)}]
```

kde + znamená spojení řetězců jako v Javě. Všimněte si, že jsme v metodě `zobrazOsobu` museli použít proměnné `B` a `G` k získání roku narození a pohlaví. Z programátorského hlediska to není příliš povedená konstrukce, proto Pluto uživatelům dovoluje použít přímo jména atributů. Metoda `zobrazOsobu` by pak vypadala:

```
zobrazOsobu() :- write("Rok narozeni je" + rok_narozeni),
write("Pohlavi je" + pohlavi)
```

Metody instancí mohou být volány pouze prostřednictvím instancí. Např. je-li `bob` objekt třídy `osoba`, pak následující dotaz ukazuje volání metody `zobrazOsobu`

```
?-bob.zobrazOsobu()
```

Pluto poskytuje mnoho dalších V/V metod ve třídě `system` např. `readInt, readReal, writeInt, writeChar`.

Dedukce Pluto je LP jazyk. Používá se hlavně k získání informací, které nejsou explicitně dostupné. Předpokládejme následující příklad:

```
class osoba[rok_narozeni ⇒ integer; rodice ⇒ osoba;
static tento_rok ⇒ integer default 2000;
vek() ⇒ integer{age() → A:- A = tento_rok - rok_narozeni
predci() ⇒ {osoba}predci() → ⟨X⟩:- rodice → ⟨X⟩;
predci() → ⟨X⟩:- rodice → ⟨P⟩, P.predci() → ⟨X⟩}]
```

První metoda získá věk osoby z data `rok_narozeni`. Druhá metoda získá předky z atributu `rodice`. Konstrukce $\langle X_1 ... X_n \rangle$ je převzata z ROL [9] a RLOG [10] pro manipulaci s množinami. Pracuje dvěma různými způsoby: je-li uvedena v těle pravidla, označuje podmnožinu; uvedena v hlavičce, seskupuje množiny.

Definice typu a implementace metody může být v Plutu provázána. Předchozí definice třídy může být psána podobně jako v Javě:

```
class osoba[rok_narozeni ⇒ integer;
rodice ⇒ {osoba}; static tento_rok ⇒ integer default 2000;
vek() ⇒ integer A {A = tento_rok - rok_narozeni}
predci() ⇒ {osoba} ⟨X⟩ {rodice -> ⟨X⟩; rodice → ⟨X⟩,
predci() → ⟨X⟩}]
```

Podobně jako v PROLOGu jsou i v Plutu pravidla vyhodnocována nedeterministicky. Pro ovlivnění vyhodnocení pravidel je použito `fail` a `!` (cut).

Aktualizace Hodnoty atributů objektu jsou v čase proměnné. V PROLOGu jsou pro aktualizaci použity metody `assert` a `retract`, Pluto používá `insert` a `delete`, jak ukazuje následující příklad:

```
class osoba[druh ⇒ osoba;
ozen(osoba) {ozen(S) :- not druh → _, insert druh → S,
insert S.druh → Self}
rozved(){rozved() :- delete druh → S, delete S.druh → _}]
```

Z předchozího příkladu je patrné, že použití operací `delete` a `insert` je příliš těžkopádné a pracné. V Plutu proto používáme operaci `is`. Předchozí dvě metody mohou být tak zapsány následujícím způsobem:

```
ozen(osoba S) { not druh → _, druh is S, S.druh is Self}
rozved() {druh.druh is null, druh is null}
```

Formálněji má zápis A `is` V následující významy:

1. Není-li A výskyt proměnné, znamená A `is` V vypočítej V a výsledek ulož do A, jinak
2. je-li V null, znamená A `is` V `delete` $A \rightarrow$ _, jinak
3. není-li A obsaženo ve V, znamená A `is` V `insert` $A \rightarrow V$, jinak
4. je-li A obsaženo ve V a zároveň V neobsahuje X, znamená A `is` V `delete` $A \rightarrow X$, `insert` $A \rightarrow V$', kde V' vznikne z V nahrazením A symbolem X.

Konstruktor Při vzniku objektu je obvykle vhodné definovat inicializační akci, která naplní jeho atributy hodnotami. Konstruktor je metoda mající stejné jméno jako třída. Předpokládejme následující příklad:

```
class osoba[rok_narozeni => integer;
pohlavi => char('M','F');
static soucet => integer default 0;
osoba(integer B) soucet is soucet + 1, rok_narozeni is B
osoba(integer B, char G) soucet is soucet + 1, rok_narozeni
is B, pohlavi je G]
```

Výše definovaná třída má dva konstruktory mající různé vstupní hodnoty. Narozdíl od ostatních metod jsou konstruktory prováděny při vzniku objektu třídy. Mějme dotaz:

```
?-liz = new osoba(1950).
```

Vznikne nový objekt třídy `osoba`, je mu přiřazeno jméno `liz` a odpovídajícímu atributu je nastavena hodnota 1950, dále je pak zvýšen součet objektů ve třídě zavoláním konstruktoru. Všimněme si, že atribut `pohlavi` žádnou hodnotu nezískal. Systém přiřadí tomuto atributu implicitní hodnotu `F`. Ke změně hodnoty tohoto atributu musíme později použít `delete` a `insert`.

Není-li takový konstruktor ve třídě definován, je nutné použít následující dotaz a následnou aktualizaci k dosažení stejného výsledku.

```
?-liz = new osoba().          ?-liz.count is liz.count+1.
```

Uvědomme si, že Pluto provádí typovou kontrolu a také kontrolu konzistence. Mějme následující příklad:

```
?-pam = new                   ?-liz.rok_narizeni is 1952
osoba(1952,"Female")
```

Systém zaprvé zahlásí, že `"Female"` není znak a zadruhé řekne, že `liz` má již atribut `rok_narozeni` naplněn.

Vytváření objektu V Plutu mohou být objekty vytvářeny dvěma různými způsoby. Buď používá metody pro vytvoření objektu a nebo dotazy nebo programy Pluta, jak již bylo ukázáno výše. Podrobnosti budou vysvětleny v kapitole 4.3.

Mějme následující příklad:

```
class datum[den ⇒ integer(1..31);mesic ⇒ integer(1..12);
rok ⇒ integer(1950..2050);
datum(integer D, integer M, integer R){den is D, mesic is M,
rok is R}
copy() ⇒ datum N{den → D, mesic → M, rok → R, N = new
datum(D, M, R)}]
```

v tomto příkladě je `datum` konstruktor a `copy` metoda pro vytvoření objektu.

Metody třídy

Stejně jako atributy mají i třídy svoje metody. Metoda třídy je v Plutu deklarována následujícím způsobem:

`static op(`τ_1`,...,`τ_n`)` $\Rightarrow$ τ $\{definice\_pravidel\}$

kde $op(\tau_1, ..., \tau_n) \Rightarrow \tau$ je *typ metody*, který specifikuje jméno *op* metody, typy argumentů $\tau_1, ..., \tau_n$ a typ výsledku metody τ. Podstatnou částí definice je $op(\tau_1, ..., \tau_n)$, která bývá nazývána *podpisem (signaturou)* metody a tato musí být unikátní v rámci třídy. Pokud je typ τ `void` můžeme jej při deklaraci vynechat.

Následuje příklad:

```
class osoba[jmeno ⇒ string;
static count ⇒ integer default 0;
static zobrazSoucet(){zobrazSoucet() :- soucet → C,
write(C)}]
```

Metody třídy mohou používat výhradně atributy třídy, nikoliv však atributy instancí v rámci třídy. Naopak metody instance smějí používat oba typy metod. Pro příklad sestavme následující dotaz:

```
?-osoba.zobrazSoucet().
```

Metody třídy mohou být volány i z instancí. Následující příklad dává stejný výsledek jako předchozí dotaz:

```
?-bob.zobrazSoucet().
```

4.1.3 Přetížení metod a polymorfismus

V Plutu může být v jedné třídě totéž jméno použito pro více metod s různou signaturou; což umožňuje přetížení tohoto jména. Přetížení metody je způsob, jak realizovat polymorfismus. Mějme následující deklaraci třídy:

```
class osoba [rodice ⇒ {osob};
deti() ⇒ {osoba} {deti() → ⟨C⟩:- C.rodice → ⟨Self⟩}
deti(osoba) ⇒ {osoba} {deti(P) → ⟨C⟩:- C.rodice → {Self
,P}}]
```

kde první metoda je použita k získání dětí osoby, zatímco druhá k získání společných dětí dvou osob, z nichž jedna je označena jako `Self` druhá `P`. Jméno `deti` je v těchto dvou metodách přetíženo. Stejně jako u atributů můžeme i u všech ostatních metod kromě konstruktorů kontrolovat jejich použití pomocí modifikátorů `public` a `private`. Implicitní hodnota je opět `public`, zatímco uvedení slova `private` zabraňuje přímému použití a je hlavně využíváno pro přechodné metody. Konstruktory jsou vždy `public`.

4.2 Hierarchie tříd a dědičnost

V Plutu existuje hierarchie tříd na jejímž vrcholu je třída `object` a na jejím dolním konci třída `none`. Vícenásobnou dědičností jsou z nadtříd odvozovány podtřídy. Podtřída v Plutu zdědí od svého předka veškeré atributy instance i s jejich implicitními hodnotami a metody přímých nadtříd s parametry jejich dostupnosti. Takto vzniklá třída si může pochopitelně dodefinovat vlastní atributy i s počátečními hodnotami i metody. Dále platí, že každá instance podtřídy je zároveň i nepřímou instancí všech jejích předků.

Mějme následující definici třídy:

```
class zamestanec isa osoba[plat ⇒ real default 5000;
zvyseni(real R){plat is plat + plat*R}]
```

Takováto definice říká, že `zamestanec` je podtřídou třídy `osoba`. A proto dědí `zamestanec` od `osoby` všechny atributy instancí, jejich implicitní hodnoty a metody. Navíc je však deklarován atribut `plat` s implicitní hodnotou a metoda `zvyseni`. Každá instance třídy `zamestanec` je tak zároveň nepřímou instancí třídy `osoba`.

Při definici přímých potomků třídy `object` můžeme toto slovo vynechat. Třída `osoba` tak může vzniknout dvěma různými definicemi:

```
class osoba isa object[...]     nebo     class osoba[...]
```

4.2.1 Dědičnost pomocí změny a vynechání atributů nebo metod

Pluto rovněž dovoluje úpravy a vynechání definic atributů, implicitních hodnot a metod, které třída zdědila od svého předka.

Mějme následující definice tříd:

```
class pacient[osetruje ⇒ lekar]
class alkoholik isa pacient[osetruje ⇒ psychiatr]
```

První definice říká, že o pacienty se starají lékaři. Druhá definice upřesňuje, že alkoholici jsou pacienti, ale v rukou psychiatrů. Třída `alkoholik` zavádí novou definici pro atribut `osetruje`, který vznikl změnou deklarace atributu `osetruje` zděděného ze třídy `pacient`.

Následující příklad ukazuje, jak změnit implicitní hodnoty atributů, aniž bychom měnili jejich definici.

```
class student isa osoba[rok_narozeni default 1975;
pohlavi default 'M'; navstevuje ⇒ {kurz};
prihlasit(kurz){prihlasit(C):- insert navstevuje → ⟨C⟩}
odhlasit(kurz){odhlasit(C):- delete navstevuje → ⟨C⟩}]
```

Třída `student` jako podtřída třídy `osoba` zdědila veškeré instance atributů a metody této třídy. Zděděné implicitní hodnoty byly změněny na hodnoty 1975 a M. Třída `student` má oproti svému předku ještě atribut `navstevuje` a metody `prihlasit` a `odhlasit`.

Následující příklad ukazuje, jak vynechat atributy a metody zděděné z nadtřídy:

```
class celibat isa osoba[druh ⇒ none; ozen(osoba) → none;
rozved() → none]
```

Vestavěná třída none umožňuje vynechat ze třídy `celibat` zděděné atributy a metody ze třídy `osoba`. V důsledku toho nemá instance třídy `celibat` atribut `druh` a nemůže používat metody `ozen` a `rozved`.

Protože Pluto podporuje přetížení metod, je na tomto místě kvůli „vynechávání“ nutné specifikovat signatury metod. Definice atributů a jejich implicitní hodnoty jsou svázány i při dědičnosti. Což znamená, že zdědí-li podtřída od svého předka definici atributů, pak zdědí i implicitní hodnoty (jsou-li nějaké definovány) odpovídajících atributů, pokud tyto nebyly procesem dědění změněny. Pokud byla definice atributu při dědění změněna nebo vynechána, je pochopitelně vynechána i implicitní hodnota tohoto atributu. Metody používající atributy a metody, které nebyly zděděny, mohou způsobit i run-time chybu, pokud tento zděděn byl.

System Pluto používá vztahy závislostí k určení toho, co může být zděděno automaticky.

4.2.2 Vícenásobná dědičnost

Pluto podporuje vícenásobnou dědičnost. Podtřída může děděním získat vlastnosti od více než jedné nadtřídy. Tím však mohou nastat konflikty v dědičnosti.

```
class zamestanec[...mistnost ⇒ string]
class student[...mistnost ⇒ integer default 123]
class wstudent isa zamestanec, student[]
```

Z uvedeného vyplývá, že `wstudent` je podtřída tříd `zamestnanec` a `student`. Protože dědí atributy a metody obou svých předků, nastává konflikt u atributu `mistnost`.

Narozdíl od Javy, která povoluje pouze jednoduchou dědičnost a C++, které nedovoluje zdědit konfliktní vlastnosti, poskytuje Pluto dva různé způsoby řešení tohoto problému: (1) uživatelem definovaná priorita a (2) výběr postupu dědění, přičemž uživatelem definovaná priorita je implicitní.

Uživatelem definovaná priorita Pluto při vytváření podtřídy prochází seznam uživatelem uvedených předků vždy zleva doprava a vyskytnou-li se nějaké konflikty, ať už ve jménech atributů nebo v signaturách děděných metod, jsou zděděny vždy ty nejlevější.

V uvedeném příkladě tak `wstudent` automaticky zdědí atribut `mistnost` ze třídy `zamestnanec` a implicitní hodnota ze třídy `student` nebude zděděna. Vyskytují-li se ve třídách předků metody se stejným jménem, ale rozdílnou signaturou, jsou zděděny všechny a v takto vzniklé podtřídě nastává přetížení metod tohoto jména.

Výběr postupu dědění Pokud potřebujeme děděním vytvořit podtřídu z několika nadtříd tak, že z každé nadtřídy potřebujeme jen některé její vlastnosti, není možné použít uživatelem definovanou prioritu, ale užijeme klíčové slovo `from`.

V návaznosti na předchozí příklad s `wstudentem`: předpokládejme, že třídy `zamestnanec` a `student` mají obě metodu `prijem` se stejnou signaturou, ale rozdílnou implementací. Chceme-li, aby `wstudent` zdědil atribut `mistnost` a metodu `prijem` ze třídy `student`, použijeme následující definici třídy:

```
class wstudent isa zamestanec, student [mistnost, prijem()
from student]
```

Uvědomme si, že třída `wstudent` získá děděním ze třídy `student` nejen atribut `mistnost`, ale také jeho implicitní hodnotu. Je-li metoda `prijem` ve třídě `wstudent` přetížena, bude předchozí definicí zděděna pouze metoda bez argumentů.

4.3 Programování v Plutu

Programovací jazyk Pluto je prakticky jako jiný OOPL: Nejdříve definujeme třídy potřebné pro aplikaci a poté vytváříme program, v němž tyto třídy používáme. Definované třídy je nutné přeložit do spustitelné podoby jako např. v Javě. Kompilace se vztahuje též k dědičnosti, proto třída ve spustitelné podobě obsahuje veškeré atributy, implicitní hodnoty a metody, ať už přímo definované nebo zděděné.

Následují dvě úplné definice tříd:

```
class bod[x, y ⇒ real;
bod(real X, Y) {x is X, y is Y}
presun(bod V) {x is x + V.x, y is y + V.y}
vzdalenost(bod V) ⇒ real D {X is x - V.x, Y is y - V.y, D =
sqrt(X*X + Y*Y)}]

clas obdelnik[v1, v2, v3, v4 ⇒ bod;
obdelnik(integer X1, Y1, X2, Y2, X3, Y3, X4, Y4) {v1 → new
bod(X1, Y1),v2 → new bod(X2, Y2),v3 → new bod(X3, Y3),v4
→ new bod(X4, Y4)}
delka() ⇒ real L{L is v1.vzdalenost(v2)}
sirka() ⇒ real W{W is v1.vzdalenost(v4)}
obsah() ⇒ real A{A is delka() * sirka()}
presun(bod V) {v1.presun(V), v2.presun(V), v3.presun(V),
v4.presun(V)}]
```

K systému Pluto přistupujeme pomocí interpretu příkazů Pluto jako např. v PROLOGu. Pomocí metody `load` z vestavěné třídy system nejprve načteme potřebné třídy do paměti. Nyní již můžeme pro práci s objekty používat příkazy systému Pluto. Existují čtyři druhy těchto příkazů: (1) `new` - metoda pro vytvoření objektu jisté třídy v paměti (2) `destroy` - metoda pro odstranění objektu z paměti a také uvolnění místa obsazeného objektem (3) `insert` a `delete` - pro práci s hodnotami atributů (4) dotazovací příkazy - získávání hodnot atributů a volání metod.

Mějme následující příklady:

```
?-system.load(bod).
?-p1 = new bod(0,0).
?-r1 = new
obdelnik(0,0,0,1,1,0,1,1).
?-r1.presun(p2).
```

```
?-system.load(obdelnik).
?-p2 = new bod(1,1).
?-write(r1.obsah()).
?-p1.destroy().
```

První dva dotazy načtou třídy do paměti. Dalšími třemi příkazy vytvoříme dva objekty třídy `bod` a jeden objekt třídy `obdelnik` a pojmenujeme je `p1, p2` a `r1`. Šestý a sedmý příkaz jsou dotazy. Pomocí šestého získáme obsah obdélníku `r1` a sedmý přesune obdélník `r1` v závislosti na bodu `p2`. Posledním příkazem odstraníme objekt `p1` z paměti.

Všimněme si, že některé objekty jména mají (`p1,p2,r1`) a jiné nikoliv jako např. body vzniklé zavoláním konstruktoru třídy obdélník. Pojmenované objekty zůstávají v systému, dokud nejsou explicitně odstraněny, ostatní existují pouze, je-li na ně odkudkoli odkazováno.

Příkazy v Plutu můžeme také sjednotit do pojmenovaného programu. Takto vypadá jeho struktura: `program` *jmeno* [*prikazy*]. Takto sepsaný program může být přímo spuštěn v příkazovém interpretu Pluto.

4.4 Závěr

V této kapitole byl popsán programovací jazyk Pluto, který poskytuje OO nástroje na podstatě logického programování. Ukázali jsme si, že i spojením dvou odlišných programovacích přístupů jsme schopni dosáhnout nejlepšího řešení. Dokázali jsme, že uživatel může porozumět OOP z hlediska LP a naopak.

Vývojovou sílu Pluta můžeme rozšířit přidáním různých tříd jako v Javě např.: grafika, applety a persistentní data.

Pluto se stále vyvíjí. Po testování a odstranění chyb, bude poskytnuto Pluto veřejnosti na Internetu. Cílem je vyvinout vyšší programovací jazyk rozšířený o další vlastnosti poskytující mnoho vestavěných tříd a tak vznikne opravdu všeobecný programovací jazyk.

Kapitola 5

Závěr

Při zadávání tohoto projektu jsem dostal úkol vytvořit stránky na českém Internetu, které by se blíže věnovaly problematice spojení několika programovacích přístupů zejména pak objektově orientovaných nástrojů s logickými programovacími jazyky. Vzhledem k tomu, že se v současné době tomuto tématu věnuje jen malá část programátorské obce, nebylo jednoduché sehnat příslušné materiály a pojednání, které by se tímto hlouběji zabývaly. Přesto se mi podařilo na Internetu vyhledat několik článků, které sepsali: V. Alexiev, Mengchi Liu a Paulo Moura. Z těchto jsem následně vyšel a sestavil myšlenky v nich obsažené do práce jejíž závěr právě pročítáte.

Podle mých informací jsou tyto stránky v době jejich vzniku jedinou ucelenou zmínkou o spojení OO a LP na našem Internetu, z čehož vyplývá jednoznačný přínos tohoto ročníkového projektu.

Na tomto místě bych rád uvedl možnosti dalšího pokračování a navázání na tuto práci. Myslím si, že zejména v kapitolách o jazycích LogTalk a Pluto jsem velmi dopodrobna rozebral syntaxi a uvedl několik ilustrativních příkladů použití. Z tohoto důvodu pokračování projektu spatřuji v možnosti nastudování způsobů programování a vytvoření konkrétní ukázky aplikace, která by dokládala smysluplné využití obou programovacích přístupů, tedy OO a LP.

Jelikož jsem příznivcem OO programování byl jsem velmi zvědav, jak je možné využít těchto technik v jiném než procedurálním jazyce. Velmi mě potěšilo zjištění, že je takové propojení možné, a proto jsem o tomto tématu vytvořil webové stránky, abyste se o tom dozvěděli i vy ostatní.

Literatura

[1] Vladimir Alexiev. *Object-Oriented and Logic-Based Knowledge Representation.* April 1993. 1, 4

[2] Paulo Moura, Ernesto Costa. *Logtalk: Object-Oriented pogramming in Prolog.* 30 September 1994. Domovská adresa jazyka LogTalk. 1, 9

[3] Mengchi Liu. *Pluto: An Object-Oriented Logic Programming Language.* In Proceding of the 39th International Conferences on Technology of Object-Oriented Languages & System (TOOLS USA 2001) Santa Barbara, USA, 29-August 2001. IEEE-CS Press. 1, 17

[4] H. Ait-Kaci and A. Podelski. Towards a Meaning of Life. *Journal of Logic Programming*, 16(3):195-234, 1993. 4

[5] W. Chen and D.S. Warren. C-Logic for Complex Objects. In *Proceedings of the ACM Sympo- sium on Principles of Database Systems*, pages 369-378, Philadelphia, Pennsylvania, 1989. 4

[6] M. Dalal and D. Gangopadhyay. OOLP: A Translation Approach to Object-Oriented Logic Programming. In W. Kim, J.M. Nicolas, and S. Nishio, editors, *Proceedings of the International Conference on Deductive and Object-Oriented Databases*, pages 593-606, Kyoto, Japan, 1989. North-Holland. 4

[7] S. Greco, N. Leone, and P. Rullo. COMPLEX: An Object-Oriented Logic Programming System. *IEEE Transactions on Knowledge and Data Engineering*, 4(4):344-359, 1992. 4

[8] K. Kahn, E.D. Tribble, M.S. Miller, and D. G. Bobrow. Objects in Concurrent Logic Programming Languages. In *OOPSLA '86 Proceedings*, pages 247-252. ACM New York, 1986. 4

[9] M. Liu. ROL: A Deductive Object Base Language. *Information Systems*, 21(5):431-457, 1996. 22

[10] M. Liu. Relationlog: A Typed Extension to Datalog with Sets and Tuples. *Journal of Logic Programming*, 36(3):271-299, 1998. 22

[11] M. Liu. Overview of the ROL2 Deductive Object-Oriented Database System. In *Proceedings of the 30th International Conference on Technology of Object-Oriented Languages & Systems (TOOLS USA '99)*, pages 63-72, Santa Barbara, CA, USA, August 1-5 1999. IEEE-CS Press. 17

[12] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2 edition, 1987. 4

[13] F. G. McCabe. *Logic and Objects*. Prentice Hall., 1992. 4

[14] C. Moss. *Prolog++*. Addison-Wesley, 1994. 4

[15] P. Schachte and G. Saab. Efficient object-oriented programing in prolog. In Christoph Beierle and Lutz Plumer, editors, *Logical Programming: Formal Methods and Practical Applications, Studies in Computer Science and Artificial Intelligence*. Elsevier Science, 1995. 4

[16] E. Shapiro and A. Takeuchi. Object oriented programming in concurrent Prolog. *New Generation Computing*, 1:25-48, 1983. 4

[17] Daniel Bobrow. If PROLOG is the answer, what is the question? In *International Conference on Fifth Generation Computer Systems (FGCS'84*, pages 138=145. ICOT, 1984. 5

[18] T.Chikayama. EPS-Extended Self-contained PROLOG-as a preliminary kernel language of Fifth Generation computers. *New Generation Computing*, 1:11-24, 1983. 6

[19] T.Chikayama. Unique features of ESP. In *International Conference on Fifth Generation Computer Systems*, pages 292-298, Tokyo, November 1984. 6

[20] Y. Ishikawa and M. Tokoro. Concurrent object-oriented knowledge representation language ORIENT84/K: Its features and implementation. In *OOPSLA'86*, Portland, OR, September 1986. 6

[21] Y. Ishikawa and M. Tokoro. ORIENT84/K: A language with multiple paradigms in the object framework. In *Nineteenth Annual Hawaii International Conference on System Siences*, volume II: Software Track, Honolulu, HI, Januar 1986. 6

[22] R. Iwanaga and O. Nakazawa. Development of the object-oriented logic programming language CESP. *Oki Technical Review*, 58(142):39-44, November 1991. 6

[23] Clocksin, W.F., Mellish, C.S.. *Programming in Prolog*, SpringerVerlag, New York 9

[24] Fornarino, M., Pinna, A.M.,Trousse, B.. *An Original ObjectOriented Approach for Relation Management*, Proceedings of the 4th Portuguese Conference on Artificial Intelligence(390):1326 14

[25] Fukunaga, K., Hirose, S.. *An Experience with a Prologbased ObjectOriented Language*, Proceedings OOPLSLA 86, 21(11):224231 9

[26] Goldberg, A., Robson, D.. *Smalltalk80 The language and its implementation*, AddisonWesley Series in Computer Science 9

[27] Lieberman, H.. *Using Prototypical Objects to Implement Shared Behaviour in Object Oriented Systems*, Proceedings OOPLSLA 86:189214 12

[28] Maes, P.. *Concepts and Experiments in Computational Reflection*, Proceedings OOPLSLA 87:147155 11

[29] McCabe, F. G.. *Logic and Objects*, Prentice Hall Series in Computer Science 9

[30] Razek, G.. *Combining Objects and Relations*, Communications of the ACM, 27(12):6670 10 13

[31] Rumbaugh, J.. *Relations as Semantic Constructs in an ObjectOriented Language*, Proceedings OOPLSLA 87:466481 13

[32] Rumbaugh, J.. *Controlling Propagation of Operations using Attributes on Relations*, Proceedings OOPLSLA 88:285296 14

[33] Stefik, M. J., Bobrow, D. G. , Kahn, K. M.. *Integrating AccessOriented Programming into a Multiparadigm Environment*, IEEE Software, January 1986:1018 13

[34] Stroustrup, B.. *The C++ Programming Language*, AddisonWesley 9

[35] Welsch, C., Barth, Gerhard. *Reasoning Objects with Dynamic Knowledge Bases*, Proceedings of the 4th Portuguese Conference on Artificial Intelligence (390):257268 9