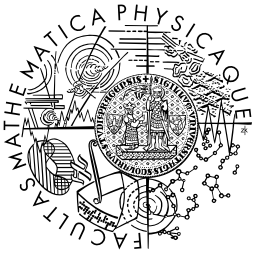


MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

DIPLOMOVÁ PRÁCE

Bc. Jana Rapavá

Doménově specifické jazyky ve funkcionálním programování

Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce:	RNDr. Jan Hric
Studijní program:	Informatika
Studijní obor:	Teoretická informatika

Praha 2018

Prohlašuji, že jsem tuto diplomovou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ..... Podpis autora

Ďakujem RNDr. Janovi Hricovi za vedenie práce a svojej rodine a priateľom za podporu pri jej písaní.

Název práce: Doménově specifické jazyky ve funkcionálním programování

Autor: Bc. Jana Rapavá

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: RNDr. Jan Hric, Katedra teoretické informatiky a matematické logiky

Abstrakt: V umelej inteligencii a zvlášť v programovaní s obmedzujúcimi podmienkami je populárne navrhovať rozličné modelovacie jazyky, ktoré umožňujú riešiť problémy na úrovni domény a prostredníctvom doménových abstrakcií. Pri tom je často užitočné používať techniky známe z oblasti doménovo špecifických jazykov. Funkcionálne programovacie jazyky poskytujú nové prostriedky pre návrh týchto jazykov, obzvlášť v prípade vnorených doménovo špecifických jazykov. Táto práca skúma výhody a nevýhody využitia techník funkcionálneho programovania pri návrhu a implementácii vnoreného doménovo špecifického jazyka pre problémy prehľadávania stavových priestorov.

Klíčová slova: umelá inteligencia prehľadávanie stavových priestorov doménovo špecifické jazyky vnorené doménovo špecifické jazyky funkcionálne programovanie

Title: Domain Specific Languages in Functional Programming

Author: Bc. Jana Rapavá

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: RNDr. Jan Hric, Department of Theoretical Computer Science and Mathematical Logic

Abstract: In Artificial Intelligence, especially in area of constraint programming, it's popular to design various modeling languages which allow solving problems on domain level and by using domain specific abstractions. Techniques known from research on Domain-Specific Languages are often useful in this effort. Functional programming languages offer new tools for designing such languages, particularly Domain-Specific Embedded Languages. This work investigates the advantages and disadvantages of using functional programming for designing and implementing a Domain-Specific Embedded Language for state space search problems.

Keywords: Artificial Intelligence state space search Domain-Specific Languages Domain-Specific Embedded Languages functional programming

Obsah

Úvod **3**

1 Doménovo špecifické jazyky **4**
1.1 Charakteristika a využitie 4
1.2 Vnorené doménovo špecifické jazyky 5

2 Od imperatívneho programovania k funkcionálnemu **8**
2.1 Typový systém jazyka Haskell 9
2.1.1 Konštrukcia nových typov a porovnávanie so vzorom na typoch 9
2.1.2 Záznamy 12
2.2 Moduly 13
2.3 Definícia operátorov 14
2.4 Práca s funkciami 14
2.4.1 Funkcie vyšších rádov 14
2.4.2 Čiastočná aplikácia funkcií 15
2.4.3 Lambda funkcie 16
2.4.4 Skladanie funkcií 17
2.4.5 Aplikácia funkcií pomocou $ 17
2.5 Lazy evaluation 18

3 Využitie funkcionálneho programovania pri implementácii DSEL a prehľadávaní stavových priestorov **19**
3.1 Typové triedy 19
3.2 Typové triedy, ktoré budeme potrebovať 21
3.2.1 Typová trieda Eq 21
3.2.2 Typová trieda Ord 21
3.2.3 Typová trieda Show 22
3.2.4 Typová trieda Read 22
3.2.5 Typová trieda Functor 23
3.3 Monády 25
3.3.1 Monáda State 29
3.3.2 Monáda List 31
3.3.3 Monáda Reader 32
3.3.4 Monáda Cont 32
3.4 Typové triedy s viacerými parametrami a funkcionálne závislosti . 33
3.5 Skladanie monád 35
3.5.1 Formalizácia konceptu 35
3.5.2 Obmedzenia skladania monád 36
3.5.3 Monadické transformátory 38

4 Použitie jazyka Haskell pri implementácii DSEL pre prehľadávanie stavových priestorov **40**
4.1 Prehľad známych metód 40
4.2 Popis jazyka Haggle 43

5 Imperatívne techniky návrhu DSEL **45**
5.1 Postupnosť funkcií 46
5.2 Reťazenie metód 46
5.3 Vnorené funkcie 47
5.4 Zhrnutie 48

6 Prehľadávanie stavových priestorov **49**
6.1 Neinformované (slepé) prehľadávanie 50
6.1.1 Prehľadávanie do hĺbky 50
6.1.2 Prehľadávanie do šírky 51
6.1.3 Prehľadávanie s obmedzenou hĺbkou 51
6.1.4 Iteratívne prehlbovanie 52
6.1.5 Obojsmerné prehľadávanie 53
6.2 Informované prehľadávanie a heuristiky 53
6.2.1 Prehľadávanie s výberom najlepšieho 53
6.2.2 Prehľadávanie s obmedzeným počtom diskrepancií 54
6.2.3 Vlastnosti heuristík 54
6.2.4 A* 55
6.2.5 IDA* 55
6.2.6 SMA* 56

7 Implementačné detaily jazyka Haggle **58**
7.1 Popis návrhu 58
7.2 Podrobný popis finálnej implementácie 60
7.2.1 Jadro jazyka Haggle 60
7.2.2 Implementované prehľadávacie algoritmy 62
7.2.3 Testy 63

Záver **65**

Zoznam použitej literatúry **66**

Úvod

Prvotným impulzom pre napísanie tejto práce bola prednáška „Folding Domain-Specific Languages: Deep and Shallow Embeddings“ (Gibbons a Wu, 2014), ktorá zaznela na International Conference of Functional Programming v roku 2014. To viedlo k prieskumu literatúry, ktorá sa zaoberá používaním funkcionálnych paradigiem pri návrhu a implementácii doménovo špecifických jazykov. Záujem autorky o umelú inteligenciu jeho rozsah zúžil na jazyky, ktoré sa usilovali riešiť problémy z umelej inteligencie - zvlášť prehľadávania stavových priestorov.

Zistili sme, že prevažná väčšina existujúcich návrhov je buď zameraná na riešenie jedného konkrétneho problému alebo triedy problémov (ako napríklad programovanie s obmedzujúcimi podmienkami), alebo na techniky lokálneho prehľadávania. Preto sme navrhli jazyk Haggle, ktorého cieľom je umožniť rýchle a jednoduché testovanie kombinovaných algoritmov pre globálne prehľadávanie stavových priestorov.

Predpokladaným čitateľom tejto práce je programátor, ktorý je oboznámený s tým, ako sa pri návrhu a implementácii doménovo špecifických jazykov využívajú imperatívne paradigmy, a rád by spoznal funkcionálny pohľad na túto problematiku. Imperatívne techniky preto rozoberieme len stručne a skôr pre pripomenutie. Na druhej strane však predpokladáme len základnú znalosť princípov funkcionálneho programovania a prakticky žiadnu znalosť algoritmov pre prehľadávanie stavových priestorov.

Vo všetkých príkladoch uvedených v texte práce sme dávali prednosť zrozumiteľnosti pre človeka prichádzajúceho z imperatívneho prostredia. Skúseným používateľom Haskellu sa preto môžu zdať neprehľadné a nemotorné - týchto čitateľov prosíme o trpezlivosť, a pravdepodobne o nič neprídu, ak kapitoly 2 a 3 proste preskočia.

V prvej kapitole si definujeme pojem doménovo špecifického jazyka a vnoreného doménovo špecifického jazyka, popíšeme ich výhody a nevýhody a uvedieme nekoľko príkladov týchto jazykov.

Druhá kapitola je všeobecným úvodom do funkcionálneho programovania pre programátorov zvyknutých na imperatívne, objektovo orientované jazyky.

Tretia kapitola na ňu nadväzuje a popisuje konkrétne techniky funkcionálneho programovania, ktoré sú užitočné pri návrhu doménovo špecifických jazykov a pri riešení problémov prehľadávania stavových priestorov.

Štvrtá kapitola poskytuje prehľad existujúcich jazykov pre všeobecné prehľadávanie stavových priestorov, na ktorý nadviažeme základným popisom jazyka Haggle.

Piata kapitola porovnáva návrh vnorených doménovo špecifických jazykov v imperatívnych hostiteľských jazykoch s funkcionálnym prístupom.

Šiesta kapitola vysvetľuje základné algoritmy pre informované a neinformované prehľadávanie stavových priestorov.

Siedma kapitola podrobne popisuje implementáciu jazyka Haggle.

Záver poskytuje zhrnutie výsledkov dosiahnutých v práci a porovnáva jazyk Haggle s ostanými vnorenými doménovo špecifickými jazykmi pre prehľadávanie stavových priestorov.

1. Doménovo špecifické jazyky

1.1 Charakteristika a využitie

Podľa autora programovacieho jazyka Perl Larryho Walla sú pre dobrého programátora charakteristické tri základné cnosti: lenivosť, netrpezlivosť a arogancia (Wall a Schwartz, 1991). Spolu s vzrastajúcou výkonnosťou hardvéru, ktorá umožňuje efektívnejšie využívanie abstraktných konceptov v programovaní, toto tvrdenie zdôvodňuje existenciu - a objasňuje vzrastajúcu popularitu - doménovo špecifických jazykov (Domain Specific Languages; DSL).

- Dobrý lenivý programátor radšej navrhne a implementuje nový jazyk, než by mal písať tisíce riadkov opakujúceho sa kódu, alebo znovu a znovu upravovať zložitú funkciu tak, aby spĺňala zmenené požiadavky zákazníka. Tento prístup je živnou pôdou nielen pre vznik klasických DSL, ktoré umožňujú stručne popísať riešenia problémov z konkrétnej oblasti, ale aj pre presun logiky z programu samotného do konfiguračných súborov, ktoré svojou syntaxou viac a viac pripomínajú DSL. (Ako príklady môžeme uviesť sendmail a iptables.)

- Netrpezlivosť vedie dobrého programátora k hľadaniu jazyka, v ktorom je možné rýchlo prototypovať a tým odhaľovať chyby a nedostatky v špecifikácii alebo riešení problému. Aj v tejto oblasti predstavujú DSL výhodu - popis riešenia problému, v ktorom sú zreteľne vidieť doménové aspekty, umožňuje efektívnejšiu komunikáciu medzi programátorom a doménovým expertom. Vývoj prebieha rýchlejšie a v programe sa vyskytuje menej chýb. Niektoré DSL dokonca umožňujú doménovému expertovi riešiť problémy bez znalosti programovania.

- Programátorská arogancia vedie k hľadaniu elegantných, udržovateľných a ľahko rozšíriteľných riešení. Dobre navrhnutý DSL umožnuje vytváranie abstrakcií špecifických pre danú oblasť, ktoré majú všetky spomínané vlastnosti. Na rozdiel od všeobecne zameraných programovacích jazykov, ktoré obsahujú nízkoúrovňové, doménovo nezávislé abstrakcie (ako je napríklad pojem objektu, triedy, rozhrania, vlákna. . .), z ktorých programátor vytvára ďalšie abstrakcie vrstvu po vrstve, až kým sa nedostane k doménovým pojmom, sú programy v DSL priamočiare a napísané (viac, alebo menej) v jazyku domény.

Použitie DSL však nie je čarovným prútikom, ktorý vyrieši problémy každého softvérového projektu. Vstupné náklady sú nezanedbateľné - návrh jazyka, implementácia prekladača a jeho integrácia s ostatnými komponentami projektu si vyžadujú čas a námahu. Ich prínos z pohľadu zákazníka navyše nie je na prvý pohľad zjavný, čo môže spôsobiť problémy u komerčných projektov. U nich je potrebné počítať aj s opakujúcimi sa „vyššími nákladmi“ pre každého nového programátora začínajúceho na projekte. Ale aj tam, kde naše rozhodnutia tieto faktory neovplyvňujú, môže implementácia DSL viesť až k absurdnému skomplikovaniu kódu.

Pokiaľ nemáme presne vyhranenú doménu, môžeme skončiť s re-implementáciou veľkej časti abstrakcií hostiteľského jazyka. Tento problém sa často popisuje v kolekciách návrhových antivzorov ako „efekt vnútornej platformy“ (inner platform effect).

Problémové sú aj DSL, ktoré sa spoliehajú na generovanie kódu - je náročné nájsť dostatočne dobrý kompromis medzi stručnosťou popisu požadovaného kódu a vyjadrovacou silou generátoru. Ako v prípade zprojectu[1], nakoniec zistíme, že je omnoho jednoduchšie podporovať integráciu ručne napísaných fragmentov než pridávať nové a nové parametre do generátoru kódu - a aj to je pomerne zložité.

Existujú však oblasti, v ktorých je používanie DSL prakticky štandartom. Spomeňme napríklad manipuláciu s textom (klasické unixové nástroje sed a AWK; často sa takýmto spôsobom používa aj Perl, na ktorý sme sa odkazovali na začiatku kapitoly, aj keď ide o všeobecne zameraný programovací jazyk), získavanie dát z databáz pomocou rozličných dialektov SQL, alebo nepreberné množstvo nástrojov, ktoré sa usilujú zjednodušiť - niekedy dokonca zautomatizovať - niektoré aspekty vývoja softvéru. Spektrum riešených problémov je naozaj široké, od prekladu a zostavenia programu (make, ant, autoconf, automake), cez generovanie dokumentácie (doxygen), projektových metadát (gsl a na ňom založený zproject) až po samotné generovanie kódu pre určitú doménu - lexikálnu analýzu (flex), parsovanie (bison, ANTLR) alebo generovanie kódu v prekladačoch (LLVM).

1.2 Vnorené doménovo špecifické jazyky

Všetky uvedené „ štandartné“ príklady patrili do triedy DSL, ktorých autori museli navrhnúť od základu nový programovací jazyk a implementovať preň kompletný prekladač - od parseru až po interpret alebo generátor kódu. O tejto triede jazykov budeme v ďalšom texte hovoriť ako o externých DSL.

Môžeme sa však rozhodnúť vybudovať DSL nad nejakým už existujúcim programovacím jazykom, ako knižnicu, ktorej rozhranie má ďalej od klasického API a bližšie k programovaciemu jazyku. Touto triedou DSL - vnorenými doménovo špecifickými jazykmi (Domain Specific Embedded Languages; DSEL) - sa budeme v tejto práci primárne zaoberať.

Hlavným rozdielom medzi DSEL a klasickým API je, akým spôsobom medzi sebou interagujú rozličné funkcie aplikovateľné na ten istý objekt. Funkcie klasického API sú striktne rozdelené na príkazy (menia vnútorný stav objektu, nemali by vracať hodnotu) a dopyty (vracajú hodnotu, nemali by meniť stav objektu). Pri implementácii DSEL však chceme, aby každá funkcia vrátila objekt, s ktorým pracovala, bez ohľadu na to, či zmenila alebo nezmenila jeho vnútorný stav. Rozhranie implementované takýmto spôsobom nám umožní reťaziť volania funkcií, ktoré budú vyzerať ako sekvencia príkazov a vyvolajú pocit, že pracujeme s programovacím jazykom.(Fowler, 2010)

Druhý veľký rozdiel je v pomenovávaní funkcií. Zatiaľ čo mená v API musia dávať zmysel bez ohľadu na kontext, mená funkcií v DSEL musia dávať zmysel v kontexte, v ktorom sa budú používať. Zatiaľ čo volanie funkcie z API má tvar `onTable (table1)`, ekvivalentné volanie v DSEL môže mať napríklad tvar `table1().on()`. (Fowler, 2010)

[1] https://github.com/zeromq/zproject

Po tomto všeobecnom úvode si podrobnejšie rozoberieme výhody a nevýhody vnorených doménovo špecifických jazykov.

Použitie DSEL si od programátora vyžaduje, aby sa do veľkej miery vzdal kontroly nad jeho syntaxou (tá bude výrazne ovplyvnená hostiteľským jazykom), ale toto rozhodnutie so sebou prináša mnohé výhody. Počiatočné náklady na návrh a vývoj jazyka sú nižšie; samotný návrh je nutne zameraný na sémantiku (čo projektu pravdepodobne pomôže, pretože, ako hovorí Wadlerov zákon[2], návrhári jazykov majú tendenciu prehnane sa sústrediť na syntax).

DSEL dedí od hostiteľského jazyka mnoho užitočných implementačných detailov, ako napríklad typový systém alebo automatickú správu pamäte. Tieto vlastnosti robia programovacie jazyky omnoho produktívnejšími a príjemnejšími na používanie, ale je náročné ich správne navrhnúť a implementovať.

Pri vývoji v DSEL je možné používať vývojové prostredie, debugger, profiler a podobné nástroje z hostiteľského jazyka. Nevýhodou však je, že tieto nástroje sú určené na prácu s všeobecne zameraným programovacím jazykom - nepodporujú teda doménové abstrakcie. DSEL sa navyše často vyznačujú programovacím štýlom odlišným od hostiteľského jazyka. Používanie takýchto vývojových nástrojov si väčšinou vyžaduje, aby sme opustili premýšľanie v kontexte DSEL a zostúpili na úroveň jeho implementácie. Aj keď pre základné úlohy tieto nástroje postačia, pokiaľ chceme, aby sa DSEL rozšíril a stal sa populárnym nástrojom, musíme preň implementovať vývojové nástroje, ktoré pracujú na jeho úrovni.

V DSEL, pokiaľ dôverujeme jeho implementácii, je tiež jednoduchšie dokazovať správnosť programu formálnymi metódami - môžeme pracovať na úrovni domény.

Kde však oproti externým DSL jednoznačne vedú, je komunikácia s okolitým svetom - bez námahy získame interoperabilitu s komponentami alebo knižnicami hostiteľského jazyka, čo je často nezanedbateľná výhoda.(Hudak, 1996)

Kľúčovou otázkou pri návrhu DSEL je výber hostiteľského jazyka. Na základe prieskumu literatúry, funkcionálne jazyky s voľnejšou syntaxou (Haskell, Scala,...) sa ukazujú ako lepšia voľba než imperatívne jazyky s prísnejšou syntaxou (ako napríklad C). Vo všeobecnosti, dobrý hostiteľský jazyk pre DSEL by mal mať nasledovné vlastnosti (Caithaml, 2009):

- výrazy namiesto príkazov
- jednoduchá podpora rekurzívnych štruktúr (termy)
- funkcie vyšších rádov
- silné statické typovanie (typová kontrola spojená s prehľadnou syntaxou)
- flexibilná notácia (preťažovanie operátorov; definovanie doménovo-špecifických operátorov)
- automatická správa pamäte
- lazy evaluation (možnosť redefinovať primitíva kontroly toku programu a definovať nové, špecifické pre danú doménu)

[2]https://web.archive.org/web/20110607102628/http://www.cse.unsw.edu.au/ dons/haskell-1990-2006/msg00737.html

- modularita

Tieto pojmy si podrobnejšie rozoberieme v nasledujúcej kapitole, kde sa pokúsime vysvetliť základné koncepty funkcionálneho programovania programátorom zvyknutým na imperatívne jazyky.

Na záver len poznamenáme, že za hostiteľský jazyk v tejto práci sme si zvolili Haskell - nielen kvôli týmto vlastnostiam, ale aj preto, že práce, z ktorých budeme vychádzať, sú založené na tomto jazyku.

2. Od imperatívneho programovania k funkcionálnemu

V tejto kapitole sa pokusíme popísať vlasnosti jazyka Haskell, ktoré budeme používať vo zvyšku práce, a to tak, aby dávali zmysel programátorovi prichádzajúcemu z imperatívneho prostredia. Vysvetlenia konceptov vychádzajú z učebníc Haskellu (Lipovaca, 2011) a (Hutton, 2007) a z tutoriálu „Learn Haskell in 10 minutes“ (Haskell tutorial), pokúšame sa však trochu viac stavať na známych imperatívnych idiomoch. Pre experimentovanie s Haskellom je užitočný interaktívny interpret GHCi, ktorý je dostupný pre všetky široko používané operačné systémy[1].

Haskell je funkcionálny jazyk, čo znamená, že štandartné idiomy imperatívneho programovania - priradzovací príkaz, podmienka, cyklus, deklarácia dátového typu ako objektu - buď nie sú implementované, alebo sa správajú inak, ako je človek prichádzajúci z imperatívneho prostredia zvyknutý. Asi jedinými výnimkami sú zabudované dátové typy (`Bool`, `Int`, `Char`, `String`, ...) a aritmetické operátory.

Základnou poskytovanou abstrakciou je funkcia. Funkcie sú „občanmi prvej triedy“ (first-class citizens) - je možné ich odovzdávať do funkcií, vracať z funkcií a ukladať do zvyčajných dátových typov.

Pokiaľ nepožadujeme inak, všetky funkcie v Haskelli sú funkcie v matematickom slova zmysle. Vo funkcionálnom názvosloví sa takéto funkcie označujú ako čisté. Niekedy sa tiež hovorí, že nemajú vedľajšie efekty - prostredie pred a po spustení funkcie je rovnaké; nedošlo k zmenám stavu, k výpisu textu na konzolu, k zápisu dát na disk, k nadviazaniu sieťovej komunikácie atď.

Programátor prichádzajúci z imperatívneho prostredia sa prirodzene pýta, ako môže takýto program komunikovať so svojím užívateľom. Odpoveďou je, že v predchádzajúcom odstavci je kľúčové to počiatočné „pokiaľ nepožadujeme inak“. Zatiaľ čo v imperatívnych jazykoch sú defaultne všetky vedľajšie efekty dovolené a my sa rozličnými technikami (od extrahovania kódu do knižníc až po jazyky s automatickou správou pamäte) snažíme dosiahnuť, aby mal program len tie želané vedľajšie efekty, v čistých funkcionálnych jazykoch postupujeme opačne. Každej funkcii dovolíme vykonávať len tie vedľajšie efekty, o ktoré si explicitne požiada (Hughes, 1989).

Po tomto vysvetlení prístupu k vedľajším efektom vo funkcionálnom programovaní sa vrátime k syntaktickým detailom.

Definícia funkcie má tvar <názov> <parametre> = <definujúci výraz>. Pri definícii funkcie je však možné použiť aj viacero definičných rovností. Ako príklad s jednou rovnosťou si uvedieme funkciu, ktorá počíta súčet druhých mocnín:

```
sumSquares x y = x*x + y*y
```

Pri volaní funkcie sa názov funkcie od parametrov, ako aj parametre od seba oddeľujú medzerou:

[1] **Poznámka:** Príklady v texte práce, ako aj priložený zdrojový kód boli vyvíjané a testované s použitím GHC (Glasgow Haskell Compiler) verzie 7, s ktorým vyššie verzie nie sú stopercentne kompatibilné.

```
>> sumSquares 3 4
25
```

Symbol `>>` budeme v priebehu práce používať na označenie promptu GHCi. Prácu s interaktívnym interpretrom budeme uzatvárať do boxov (tak ako v poslednom uvedenom príklade), zatiaľ čo zdrojové kódy ukladané do súborov len oddelíme prázdnym riadkom od ostatného textu. V oboch prípadoch budeme zdrojový kód sádzať `takýmto` fontom.

2.1 Typový systém jazyka Haskell

Ak definujeme funkciu, Haskell si je schopný (až na výnimočné prípady) odvodiť jej typ automaticky. U zložitejších funkcií je však zvykom typy uvádzať.

Typ funkcie nemusí byť jednoznačný, jeden z jej typov však vždy bude najvšeobecnejší možný. Ako príklad si otypujeme funkciu `sumSquares`:

```
sumSquares :: Int -> Int -> Int
```

`X :: Y` je zápis pre „X má typ Y“. Vidíme, že `sumSquares` prijíma dva parametre typu `Int` a vracia výsledok typu `Int`.

Toto však nie je jediné možné otypovanie funkcie `sumSquares`. V jej definícii nikde nehovoríme, že parametre musia byť celé čísla - stačí, aby na nich boli definované operácie sčítania a násobenia. Ďalší prípustný typ pre `sumSquares` teda je:

```
sumSquares :: (Num a) => a -> a -> a
```

V tomto otypovaní je `a` takzvaná typová premenná a `(Num a) =>` je typová podmienka/typový kontext, ktorý špecifikuje, aké vlastnosti má mať typ, aby sme ho mohli dosadiť za premennú `a`.

Typový kontext `(Num a)` predpisuje, že pre typ dosadený za premennú `a` musia byť definované základné aritmetické operácie. Čo sa v Haskelli považuje za základné aritmetické operácie a ako zariadime, aby boli pre daný typ definované, si dopodrobna popíšeme v kapitole o typových triedach.

Drobná poznámka na záver: špecifikácia jazyka Haskell definuje, že premenné (z hľadiska pomenovávania považujeme za premenné aj funkcie) a typové premnené začínajú malým písmenom alebo `_` (podtržítkom). Všetky ostatné identifikátory - typy, ale aj moduly, dátové konštruktory, typové konštruktory a typové triedy, ktoré si všetky popíšeme neskôr - začínajú veľkým písmenom (Marlow, 2010).

2.1.1 Konštrukcia nových typov a porovnávanie so vzorom na typoch

Z imperatívnych programovacích jazykov sme zvyknutí, že okrem zabudovaných typov (`Bool`, `Int`, `Char`, `String`, ...) nám štandartná knižnica jazyka poskytuje aj nejaké „štandartné“ kontajnery, do ktorých je možné tieto typy ukladať, a tak vytvárať zložitejšie dátové štruktúry.

Programovací jazyk Haskell poskytuje takéto „štandartné“ kontajnery dva: zoznamy a n-tice (tuples).

Zoznamy

Do zoznamov sa ukladajú hodnoty toho istého typu. Píšu sa do hranatých zátvoriek:

```
>> [1,2,3] :: [Int]
[1,2,3]
>> [] :: [Int]
[]
```

Konštrukcia `[]` značí prázdny zoznam.

Prvok sa na začiatok zoznamu pridáva pomocou sprava asociatívneho binárneho operátora `(:)`:

```
>> 1:[2,3]
[1,2,3]
>> 2:5:[1]
[2,5,1]
```

(Podrobnejšiemu popisu operátorov v jazyku Haskell je venovaná nasledujúca podkapitola.)

So zoznamami ako parametrami funkcií sa väčšinou pracuje pomocou rozboru prípadov a takzvaného porovnávania so vzorom (pattern matching):

```
f []     = <spracuj prázdny zoznam>
f (x:xs) = <spracuj prvý prvok zoznamu x
         rekurzívne sa zavolaj na zvyšok zoznamu xs>
```

N-tice

N-tice slúžia na uloženie pevného počtu hodnôt, ktoré môžu mať rozličné typy, a píšu sa do okrúhlych zátvoriek:

```
>> (1, "foo") :: (Int, String)
(1, "foo")
>> (True, "bar", 2) :: (Bool, String, Int)
(True, "bar", 2)
>> zip [0, 1] [False, True]
[(0, False), (1, True)]
```

Funkcia `zip` vytvorí dvojice z prvkov, ktoré sú v dvoch zoznamoch na rovnakom indexe.

Knižnice jazyka Haskell poskytujú funkcie na prácu s n-ticami, ktoré majú 2 až 32 položiek.

Existujú funkcie na získavanie prvkov z n-tíc podľa ich indexu. My si však uvedieme len najčastejšie používaný prípad - dvojice:

```
>> fst (1, 2)
1
>> snd (1, 2)
2
```

Aj na parametre funkcií, ktorých typy sú n-tice, môžeme použiť porovnávanie so vzorom:

```
f (x, y, z) = <spracuj x, y a z uložené v trojici>
```

Konštrukcie data a newtype

Ešte zložitejšie dátové štruktúry sa definujú pomocou konštrukcie `data`. Môžeme napríklad zadefinovať dátový typ, ktorý rozširuje štandartný typ `Int` o (kladné) nekonečno:

```
data InfInt = Fin Int | Inf
```

V tejto definícií sa `Fin` a `Inf` nazývajú dátové konštruktory. Hodnotu „zabalenú“ v dátovom konštruktore môžeme získať pomocou porovnávania so vzorom, ktoré funguje obdobne ako u zoznamov a n-tíc.

Napríklad: pokiaľ je hodnota typu `InfInt` odovzdaná funkcii ako parameter, môžeme správanie tejto funkcie na rozličných dátových konštruktoroch definovať rozborom prípadov:

```
f (Fin x) = ...
f Inf     = ...
```

Konštrukciu `newtype` používame, keď chceme zadefinovať rozličné užívateľské typy, ktorých vnútorná reprezentácia zodpovedá tomu istému štandartnému typu jazyka Haskell. Použijeme ju napríklad v prípade, keď chceme definovať samostatný typ pre mená a samostatný typ pre adresy, ale oba typy budú vlastne `String`.

Druhým prípadom využitia `newtype` je ukrytie vnútornej reprezentácie typu pred jeho užívateľmi. Ide vlastne o funkcionálny ekvivalent zapuzdrenia v objektovo orientovanom programovaní. Pokiaľ máme užívateľský typ pre maticu, ktorý je vlastne zoznam dvojíc, a nechceme, aby sa programy využívajúce tento typ spoliehali na jeho vnútornú reprezentáciu, vložíme tento typ do `newtype`.

Pokiaľ máme program, ktorý má počítať priemerné rýchlosti, môžeme pomocou `newtype` rozlíšiť kilometre a hodiny, a tak implementovať základnú kontrolu dimenzionality:

```
newtype Km  = Km Float
newtype Hour = H Float
newtype Speed = S Float
```

Pri použití týchto typov v kóde potom môžeme priamo získať hodnoty z parametrov pomocou porovnávania so vzorom:

```
avgSpeed :: Km -> Hour -> Speed
avgSpeed (Km dist) (H time) = S (dist / time)
```

Definície nových dátových štruktúr môžu byť parametrizované a rekurzívne. Ako príklad si pre zaujímavosť definujeme binárny strom s hodnotami v listoch. Nejedná sa o klasický binárny vyhľadávací strom, ale o dátovú štruktúru, ktorá sa využíva napríklad pre reprezentáciu Huffmanovho kódovania:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

Túto definíciu čítame nasledovne: „strom typu a je buď list typu a, alebo uzol, ktorý má ľavý a pravý podstrom s prvkami typu a“. Do takéhoto stromu je možné uložiť hodnoty ľubovoľného typu, všetky uložené hodnoty však musia mať rovnaký typ.

Akákoľvek funkcia využívajúca typ `Tree a` môže používať porovnávanie so vzorom na jeho typových konštruktoroch:

```
f (Leaf x)          = ...
f (Node left right) = ...
```

Na záver sa ešte na chvíľu vrátime k zabudovanému dátovému typu zoznamov. Teraz už vidíme, že z hľadiska konštrukcie a porovnávania so vzorom sa zoznamy správajú, ako keby boli definované rekurzívnym parametrizovaným dátovým typom:

```
data  [a]  =  [] | a : [a]
```

A aj keď ide o zabudovaný dátový typ, ktorý je vnútorne implementovaný inak, takáto definícia zoznamu sa skutočne vyskytuje v špecifikácii jazyka Haskell (Marlow, 2010).

2.1.2 Záznamy

Záznamy (records) sú alternatívnou formou zápisu, ktorú je možné používať pri vytváraní typov pomocou konštrukcie `data` alebo `newtype`.

Pokiaľ dátovú štruktúru deklarujeme ako záznam, automaticky tým získame funkcie, ktoré nám umožnia pristupovať k jej jednotlivým prvkom.

Ako príklad si uvedieme adresár, kde chceme mať pre každého človeka jeho meno, priezvisko, zoznam jeho telefónnych čísel a zoznam jeho e-mailov. Príslušný dátový typ by bol:

```
data Person = Person String String [String] [String]
```

Takáto definícia je však extrémne neprehľadná. Pomôže, keď použijeme `newtype`:

```
newtype Name        = Name String
newtype Surname     = Surname String
newtype PhoneNumber = PhoneNumber String
newtype EMail       = EMail String

data Person = Person Name Surname  [PhoneNumber] [EMail]
```

Musíme si však ručne napísať get-funkcie pre všetky položky uložené v type Person:

```
getName (Person (Name name) _ _ _) = name
```

a obdobné funkcie pre všetky ostatné položky.

(V tejto funkcii sme pri porovnávaní so vzorom využili novú konštrukciu `_`, ktorá má význam „na tomto mieste môže byť ľubovoľná hodnota, a nikdy ju nebudeme používať“.)

Nebolo by možné tieto get-funkcie generovať automaticky z definície dátového typu? Presne to sa stane, ak použijeme záznamovú syntax (Lipovaca, 2011). Jednotlivé položky tak dostanú mená, čo má dve výhody. Prvou je sprehľadnenie definície bez nutnosti zavádzať pomocné typy:

```
data Person = Person { name         :: String
                     , surname      :: String
                     , phoneNumbers :: [String]
                     , eMails       :: [String]
                     }
```

Druhou výhodou je, že automaticky dostaneme funkcie, ktoré umožňujú prístup k jednotlivým položkám záznamu:

```
name         :: Person -> String
surname      :: Person -> String
phoneNumbers :: Person -> [String]
eMails       :: Person -> [String]
```

2.2 Moduly

Väčšina moderných programovacích jazykov umožňuje rozdeliť kód programu do modulov, a Haskell nie je výnimkou. Moduly sa používajú na organizáciu kódu v štandartných knižniciach a použili sme ich na organizáciu kódu aj v jazyku Haggle, preto si vysvetlíme aspoň základy práce s nimi.

Deklarácia modulu sa musí vyskytovať v zdrojovom súbore pred všetkými ostatnými deklaráciami a má tvar:

```
module Module where
```

Názov modulu musí začínať veľkým písmenom.

Ak chceme využívať funkcie z daného modulu, použijeme konštrukciu `import`, ktorá sa musí vyskytovať pred prvou definíciou funkcie:

```
import Data.List
```

Pokiaľ importovaná knižnica obsahuje nejaké symboly, ktoré kolidujú so symbolmi v našom programe, môžeme použiť kvalifikovaný import:

```
import qualified Data.List as L
```

V prípade, že by kolidujúcim symbolom bola funkcia `find`, môžeme po kvalifikovanom importe volať funkciu `find` z modulu `Data.List` ako `L.find`.

V jazyku Haskell (na rozdiel od napríklad #include direktív v C) moduly nie sú importované rekurzívne. V každom zdrojovom súbore sú viditeľné iba symboly z tých modulov, o ktoré si explicitne zažiada konštrukciou `import`.

Pokiaľ nešpecifikujeme inak, všetky symboly definované v module sú globálne a je možné ich využívať, keď modul importujeme. Jazyk Haskell umožňuje symboly z modulu explicitne exportovať (všetky neexportované symboly sa potom stanú lokálnymi), to však v tejto práci nebudeme používať.

2.3 Definícia operátorov

Na rozdiel od napríklad C++ nie sú v jazyku Haskell operátormi len preddefinované reťazce. Pokiaľ definujeme funkciu, ktorej meno je reťazec špeciálnych symbolov (napríklad `>>=`), táto funkcia sa bude defaultne používať v infixovej notácii. Medzi príklady zo štandartnej knižnice jazyka Haskell patria bežné aritmetické operátory, alebo pridávanie na začiatok zoznamu pomocou `(:)`.

Pri definovaní operátoru v Haskelli je možné definovať aj jeho prioritu, to však v tejto práci nebudeme potrebovať. Je však užitočné vedieť, že aplikácia funkcie viaže viac než akýkoľvek operátor, takže výraz `f x + 1` sa spracuje ako `(f x) + 1`.

Operátory nie sú v Haskelli kľúčovou súčasťou jazyka. Slúžia výhradne na sprehľadnenie syntaxe a v prvom kroku prekladu sa odstránia.

Na rozdiel od mnohých imperatívnych jazykov sú v Haskelli len binárne operátory.

Každý operátor je možné použiť ako funkciu (v prefixovej notácii) tým, že ho uzavrieme do okrúhlych zátvoriek.

```
>> (+) 2 3
5
```

Každú funkciu je možné použiť ako operátor tým, že jej názov uzavrieme do spätných apostrofov.

```
>> [0, 1] `zip` [False, True]
[(0, False),(1,True)]
```

To je pre účely tejto práce všetko, čo budeme potrebovať vedieť o operátoroch.

2.4 Práca s funkciami

2.4.1 Funkcie vyšších rádov

Funkcie, ktoré prijímajú ako parametry iné funkcie, sa nazývajú funkcie vyšších rádov. Ide o jednu z tých konštrukcií funkcionálneho programovania, ktoré už prenikli aj do tradične imperatívnych jazykov, ako je napríklad C++. Funkcie ako `std::for_each` a `std::accumulate` majú v jazyku Haskell svoje ekvivalenty. Štandartná knižnica jazyka Haskell však poskytuje aj mnoho iných funkcií vyšších rádov.

Ako príklad si implementujeme funkciu `zipWith`, ktorá je rozšírením funkcie `zip`. Zatiaľ čo funkcia `zip` dva prvky s rovnakým indexom vždy spojí do dvojice, `zipWith` dostane spájaciu funkciu ako prvý parameter:

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith _ [] _          = []
zipWith _ _ []          = []
zipWith f (x:xs) (y:ys) = (f x y):(zipWith f xs ys)
```

2.4.2 Čiastočná aplikácia funkcií

Na všetky funkcie je možné pozerať sa ako na funkcie s jedným parametrom. Ak sa bližšie pozrieme na typ funkcie `zipWith`, ktorú sme si definovali v časti o funkciách vyšších rádov

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

môžeme o nej uvažovať:

- ako o funkcii, ktorá prijíma funkciu, zoznam typu `[a]` a zoznam typu `[b]`, a vracia zoznam typu `[c]`; napríklad:

  ```
  zipWith (+) [1,2] [3,4] :: [Int]
  ```

- ako o funkcii, ktorá prijíma funkciu a zoznam typu `[a]` a vracia funkciu typu `[b] -> [c]`; napríklad:

  ```
  zipWith (+) [1,2] :: [Int] -> [Int]
  ```

- ako o funkcii, ktorá prijíma funkciu typu `(a -> b -> c)` a vracia funkciu typu `[a] -> [b] -> [c]`; napríklad:

  ```
  zipWith (+) :: [Int] -> [Int] -> [Int]
  ```

Takéto oddelenie parametrov sa vo funkcionálnom názvosloví nazýva currying.

Veľmi často sa používa posledný pohľad, a to z dvoch dôvodov. Tým menej dôležitým je možnosť vynechať parametre pri definícii funkcie. Ak si chceme definovať pomocnú funkciu, ktorá sčíta dva zoznamy prvok po prvku, môžeme namiesto:

```
sumlists xs ys = zipWith (+) xs ys
```

písať:

```
sumlists = zipWith (+)
```

Tým dôležitejším je možnosť odovzdávať čiastočne aplikované funkcie ako parametre funkciám vyšších rádov. Napríklad vnorené zoznamy môžeme sčitať prvok po prvku nasledovne:

```
>> zipWith (zipWith (+)) [[1,2],[3,4]] [[5,6],[7,8]]
[[6,8],[10,12]]
```

Haskell umožňuje čiastočne aplikovať aj viac ako jeden parameter funkcie - musí však ísť o prvých n za sebou idúcich parametrov. Pokiaľ by sme v prechádzajúcom príklade chceli prvý parameter (funkciu `f`) vynechať a čiastočne aplikovať len ten druhý (zoznam `xs`), použijeme pomocnú funkciu `flip`, ktorá obráti poradie prvých dvoch parametrov funkcie:

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip f x y = f y x
```

Potom môžeme písať nielen:

```
with01 f ys = zipWith f [0,1] ys
```

Ale aj s využitím čiastočnej aplikácie:

```
with01 = (flip zipWith) [0,1]
```

2.4.3 Lambda funkcie

Pri používaní funkcií vyšších rádov je často užitočné mať možnosť vytvárať anonymné (nepomenované) funkcie. Tieto funkcie sa okamžite použijú tam, kde sa vyskytujú, a často sa používajú ako jednoúčelové parametre funkcií vyšších rádov. Ich podpora zvykne ísť ruka v ruka s podporou funkcií vyšších rádov - väčšina programovacích jazykov buď poskytuje obidve tieto abstrakcie, alebo ani jednu z nich. V dnešnej dobe však už nie sú charakteristické len pre funkcionálne jazyky.

V C++ môžeme implementovať lambda funkciu, ktorá inkrementuje každý prvok kontajnera, a tú zavolať prostredníctvom `std::for_each`. Výsledný kód bude vyzerať takto:

```
std::vector<int> nums{1,2,3,4};

std::for_each(nums.begin(), nums.end(),
[](int &n){ n++; } // toto je lambda funkcia
);

<nums obsahuje [2,3,4,5]>
```

Ekvivalentný kód používajúci rovnaké konštrukcie môžeme napísať aj v jazyku Haskell, a bude vyzerať takto:

```
>> let nums = [1,2,3,4]
>> map (\n -> n+1) nums
[2,3,4,5]
```

(Funkcia `map` je ekvivalentom `std::for_each` pre zoznamy v Haskelli a podrobnejšie sa jej budeme venovať v kapitole o monádach.)

Lambda funkcia v C++ pozostáva z troch častí:

- zoznam mien premenných zachytených z rozsahu platnosti, v ktorom lambda funkciu definujeme, v hranatých zátvorkách
- zoznam parametrov lambda funkcie v okrúhlych zátvorkách
- štandartné telo funkcie v jazyku C++

Lambda funkcia v Haskelli je výraz. V prípade potreby sa píše do okrúhlych zátvoriek, ktoré však nie sú súčasťou jej definície. Na rozdiel od C++ nemá zoznam zachytených premenných, pretože ak je lambda funkcia definovaná ako súčasť funkcie (nie na najvyššej úrovni), zachytáva všetky viditeľné mená premenných. Samotná funkcia pozostáva z:

- obráteného lomítka `\`
- zoznamu parametrov (oddelených medzerami)
- indikátoru tela funkcie `->`
- výrazu, ktorý je telom funkcie

2.4.4 Skladanie funkcií

V úvode kapitoly sme konštantovali, že pokiaľ nepožadujeme inak, funkcie v Haskelli sú funkcie v matematickom slova zmysle. Z matematiky poznáme koncept skladania funkcií:

$$(f \cdot g)x = f(gx) \tag{2.1}$$

Je možné definovať v Haskelli skladanie funkcií tak, aby malo túto vlastnosť? Odpoveď znie áno, a navyše na to nepotrebujeme žiadnu špeciálnu podporu v jazyku. Úplne nám bude stačiť, keď implementujeme takýto operátor (Lipovaca, 2011):

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = \x -> f (g x)
```

Operátor skladania funkcií je teda príkladom použitia lambda funkcie. Jeho definíciu môžeme chápať ako „funkcia f . g prijíma jeden parameter x a vráti f(g x)“.

Na záver spomenieme, že operátor `(.)` umožňuje skladať len funkcie s jedným parametrom. Pokiaľ chceme skladať funkcie s väčším počtom parametrov, musíme ich čiastočne aplikovať, alebo definovať iný operátor.

Ak chceme napríklad definovať funkciu, ktorá svoj parameter vynásobí dvomi a zmení znamienko výsledku, môžeme ju implementovať ako:

```
doubleflip = negate . (\x -> x*2)
```

Tú istú funkciu je však možné zapísať ako:

```
doubleflip x = negate (x*2)
```

2.4.5 Aplikácia funkcií pomocou $

Ako sme si povedali na začiatku kapitoly, aplikácia funkcie na argument sa v Haskelli značí medzerou:

```
>> sumSquares 3 4
25
```

A ako sme si povedali v kapitole o operátoroch, medzera, ktorá značí aplikáciu funkcie, je operátor a viaže tesnejšie než všetky ostatné operátory:

```
>> sumSquares 3 (3+1)
25
```

Nutnosť uzátvorkovania parametrov, ktoré sú samotné aplikáciami funkcií, však môže viesť k neprehľadnému kódu. Jazyk Haskell preto poskytuje alternatívny operátor aplikácie funkcie `$`, ktorý viaže slabšie než všetky ostatné operátory, a preto umožňuje zátvorky odstrániť:

```
>> sumSquares 3 $ 3+1
25
```

2.5 Lazy evaluation

Prevažná väčšina imperatívnych programovacích jazykov používa takzvanú eager evaluation - parametre funkcie sa vyhodnotia predtým, ako dôjde k jej zavolaniu.

Jazyk Haskell, spolu s niekoľkými ďalšími funkcionálnymi jazykmi, používa iný prístup. Parametre sa odovzdávajú do funkcie nevyhodnotené a k ich vyhodnoteniu dôjde, až keď je to potrebné (napríklad pri žiadosti o vypísanie výsledku na konzolu).

Výraz, ktorý ešte nebol vyhodnotený, je uložený ako „thunk“. Jeho uloženie je pamäťovo náročnejšie než uloženie hodnoty výrazu, a pamäťová náročnosť rastie so zložitosťou thunku (O'Sullivan a kol., 2008).

Vyhodnotené thunky si GHC zapamätá a používa ich, keď sú thunky zdieľané. Tento prístup sa nazýva memoizácia. Vo väčšine prípadov zlepšuje časovú zložitosť programu na úkor len o trochu väčšej spotreby pamäte, ale sú aj situácie (napríklad pri implementácii iteratívneho prehlbovania), kedy memoizáciu musíme nejakým programátorským trikom zakázať (Kiselyov, 2012).

3. Využitie funkcionálneho programovania pri implementácii DSEL a prehľadávaní stavových priestorov

V tejto kapitole si podrobnejšie popíšeme vlastnosti, knižnice a rozšírenia Haskellu, ktoré sú užitočné pri implementácii DSEL pre prehľadávanie stavových priestorov.

3.1 Typové triedy

Typová trieda na prvý pohľad pripomína koncept rozhrania (interface) v objektovo orientovaných jazykoch. Podobne ako rozličné triedy môžu implementovať zadané rozhranie, a tak umožnia programátorovi implementovať funkciu, ktorá sa správa odlišne pre odlišné typy bez toho, aby tieto typy explicitne skúmala, rozličné dátové typy v jazyku Haskell sa môžu stať inštanciami zadanej typovej triedy, a umožnia tak programátorovi písať kód využívajúci funkcie z typovej triedy, a pritom nezávislý na konkrétnom type. V oboch prípadoch ide vlastne o to, aké syntaktické prostriedky pre implementáciu polymorfných funkcií jazyk poskytuje.

Typové triedy zodpovedajú implementácii polymorfizmu skrze preťaženie (overloading) v objektovo orientovaných programovacích jazykoch. Okrem neho máme vo funkcionálnych jazykoch ešte takzvaný parametrický polymorfizmus. Ako príklad môžeme implementovať funkciu:

```
id :: a -> a
id x = x
```

ktorá je polymorfná (`a` je typová premenná, za ktorú môžeme dosadiť ľubovoľný typ) a pritom nevyužíva typové triedy. V prípade parametrického polymorfizmu máme jednu definíciu funkcie, ktorá slúži pre všetky typy.

Typová trieda je definovaná svojím názvom a kolekciou funkcií, ktoré musí každý typ v nej implementovať. Pokiaľ definícia funkcie pre daný typ nedáva zmysel a nemala by sa používať, postačí definovať ju formálne (napríklad pomocou `undefined`).

Ako príklad si uvedieme triedu pre numerické dátové typy zo štandardnej knižnice jazyka Haskell (Marlow, 2010):

```
class Num a where
        (+)         :: a -> a -> a
        (-)         :: a -> a -> a
        (*)         :: a -> a -> a
        negate      :: a -> a
        abs         :: a -> a
        signum      :: a -> a
        fromInteger :: Integer -> a
```

Ak si chceme zadefinovať typ `InfInt`, ktorý rozširuje štandartný typ pre celé čísla o kladné nekonečno, môžeme to (ako sme už videli v predchádzajúcej kapitole) urobiť nasledovne:

```
data InfInt = Fin Int | Inf
```

Zdravá dávka programátorskej lenivosti nás potom privedie k otázke: potrebujeme naozaj implementovať všetkých 7 funkcií z `Num`, aby sme mohli `InfInt` využívať v aritmetických výrazoch? Zoznam vyzerá byť redundatný; za akých okolností by sme mohli chcieť definovať `a - b` inak ako `a + (negate b)`?

Naša intuícia je správna - v dokumentácii k typovej triede `Num` nájdeme sekciu „minimálna úplná definícia“ (minimal complete definition), ktorá obsahuje všetky neredundatné funkcie z tejto typovej triedy. Všetky ostatné funkcie z typovej triedy sú štandartným spôsobom odvodené z funkcií z jej minimálnej úplnej definície. Aj keď by sme ju mohli prepísať, funkciu `(-)` naozaj implementovať nemusíme.

Aby sme teda typ `InfInt` mohli používať vo všetkých číselných výrazoch, definujeme, že je inštanciou typovej triedy `Num`. Následne implementujeme všetky funkcie z jeho minimálnej úplnej definície:

```
instance Num InfInt where
        (Fin x) + (Fin y) = Fin (x + y)
        Inf + _           = Inf
        _ + Inf           = Inf

        (Fin x) * (Fin y) = Fin (x * y)
        Inf * _           = Inf
        _ * Inf           = Inf

        negate (Fin x) = Fin (negate x)
        negate Inf     = Inf

        abs (Fin x) = Fin (abs x)
        abs Inf     = Inf

        signum (Fin x) = Fin (signum x)
        signum Inf     = 1

        fromInteger x = Fin x
```

Neskôr, keď sa dostaneme k monádam, zistíme, že v typových triedach štandartnej knižnice Haskellu je redundancia poskytovaných funkcií bežná. Uvidíme aj, že takýto popis typových tried nie je samoúčelný. Typové triedy často popisujú matematické štruktúry, pre ktoré sú odvodené operácie alebo funkcie obvyklé. Dáva tiež zmysel umožniť užívateľovi, aby mohol v rozličných kontextoch využívať rôzne funkcie. Takáto redundancia nám navyše umožňuje o kóde jednoduchšie uvažovať a dokazovať jeho správnosť.

3.2 Typové triedy, ktoré budeme potrebovať

3.2.1 Typová trieda Eq

Väčšina objektovo orientovaných jazykov poskytuje štandartné rozhranie, ktoré musia objekty implementovať, aby sme ich mohli porovnávať na rovnosť. Typovú triedu `Eq` môžeme chápať ako haskellovský ekvivalent takéhoto rozhrania.

Jej popis je, ako by sme od Haskellu očakávali, redundantný:

```
class Eq a where
    (==)  :: a -> a -> Bool
    (/==) :: a -> a -> Bool
```

Minimálna úplná definícia je buď `(==)`, alebo `(/==)`.

Ak by sme chceli čísla reprezentované dátovým typom `InfInt` porovnávať na rovnosť, stačí príslušný dátový typ definovať ako:

```
data InfInt = Fin Int | Inf deriving (Eq)
```

Prekladač automaticky vygeneruje kód pre funkciu `(==)`, ktorý bude spĺňať:

- `Inf == Inf`
- `Fin x == Fin y` $\leftrightarrow$ `x == y`

Ak by sme v jazyku Haskell chceli ručne implementovať ako inštanciu triedy `Eq` príklad s `newtype` z predchádzajúcej kapitoly, urobíme to nasledovne:

```
newtype Name = Name String

instance Eq Name where
        (Name x) == (Name y) = x == y
```

Ak chceme použiť pri definovaní inštancií klauzulu `deriving`, musíme zapnúť rozšírenie jazyka Haskell `GeneralizedNewtypeDeriving`. Potom môžeme aj pre dátový typ odvodený pomocou `newtype` písať:

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

newtype Name = Name String deriving (Eq)
```

(Konkrétne inštanciu typovej triedy `Eq` by sme mohli odvodiť pomocou `deriving` aj bez tohto rozšírenia, vo všeobecnosti ho však potrebujeme.)

3.2.2 Typová trieda Ord

Typová trieda `Ord` má rovnakú sémantiku ako komparátor v objektovo orientovaných programovacích jazykoch. Ak chceme na prvkoch dátového typu definovať úplné usporiadanie, urobíme ho inštanciou typovej triedy `Ord`.

Na to stačí, aby sme implementovali funkciu

```
compare :: a -> a -> Ordering
```

Dátový typ Ordering je definovaný ako:

```
data Ordering = LT | EQ | GT
```

Ide teda len o prehľadnejší a typovo bezpečnejší zápis $\{-1, 0, 1\}$ konvencie pre návratové hodnoty, ktorú používaju komparátory v objektovo orientovaných jazykoch.

Jej definícia poskytuje veľké množstvo pomocných funkcií súvisiacich s porovnávaním, vrátane preťažených operátorov:

```
class Ord a where
    compare :: a -> a -> Ordering
    (<)     :: a -> a -> Bool
    (<=)    :: a -> a -> Bool
    (>)     :: a -> a -> Bool
    (>=)    :: a -> a -> Bool
    max     :: a -> a -> a
    min     :: a -> a -> a
```

Ako príklady inštancií použijeme dátový typ, ktorý sme použili ako príklad pre `Eq`:

```
newtype Name = Name String

instance Ord Name where
        (Name x) `compare` (Name y) = x `compare` y
```

3.2.3 Typová trieda Show

Typová trieda `Show` špecifikuje, ako sa daný dátový typ zobrazí na konzoli.

Inštancie tejto typovej triedy väčšinou definujeme pomocou už spomínanej klazuly

```
deriving (Show)
```

V tomto prípade prekladač sám vygeneruje defaultné definície funkcií v tejto typovej triede. Tie nám vo väčšine prípadov bohato stačia a funkcie typovej triedy `Show` teda ručne implementovať nemusíme.

3.2.4 Typová trieda Read

Typová trieda `Read` má na starosti funkcionalitu inverznú k `Show` - parsovanie vstupu do príslušných dátových typov. Reťazec vypísaný pomocou `Show` pre príslušný dátový typ by malo byť vždy možné prečítať pomocou funkcií z `Read` pre príslušný dátový typ. Implementácia defaultne vygenerovaná prekladačom túto konvenciu dodržiava, a vo väčšine prípadov si (podobne ako u `Show`) s ňou vystačíme.

3.2.5 Typová trieda Functor

Aby sme vysvetlili koncept funktoru, pripomenieme si, čo sme si povedali v predchádzajúcej kapitole o funkciách vyšších rádov - to sú funkcie, ktoré prijímajú ako parametre ďalšie funkcie.

Tento koncept sa rozšíril už aj do moderných imperatívnych jazykov, a väčšina z nich poskytuje v štandardnej knižnici funkciu `fmap`, ktorá dostane ako jeden parameter funkciu `f`, ako druhý parameter nejaký štandartný kontajner, a aplikuje funkciu `f` na každý prvok kontajneru. (Napríklad v C++ sa táto funkcia nazýva `std::for_each`).

Pravdepodobne všetky imperatívne programovacie jazyky, ktoré poskytujú funkciu `fmap` alebo jej ekvivalent, obsahujú nejaký štandartný dátový typ pre zoznamy, na ktorý je túto funkciu možné použiť.

Z druhej kapitoly vieme, že Haskell tiež obsahuje dátový typ pre zoznamy. Pre jednoduchosť si znova zopakujeme, ako vyzerá:

```
data  [a]  =  [] | a : [a]
```

(**Toto nie je platný kód v Haskelli!** Ide však o kanonickú definíciu, ktorá sa vyskytuje v špecifikácii jazyka (Marlow, 2010).)

Ako by sme teda implementovali ekvivalent funkcie `fmap` pre haskellovské zoznamy? Pri programovaní funkcie v Haskelli je vždy dobrý nápad začať s jej typovou signatúrou. Tá vyzerá v našom prípade takto:

```
fmap :: (a -> b) -> ([a] -> [b])
```

Túto typovú signatúru čítame ako:

- funkcia `fmap` má jeden paramter
- týmto paramterom je funkcia z `a` do `b`
- `fmap` vracia funkciu, ktorú je možné aplikovať na zoznam, ktorý obsahuje prvky typu `a`
- táto funkcia vráti zoznam, ktorý obsahuje prvky typu `b`.

Samotná implementácia je potom priamočiarym využitím porovnávania so vzorom a rekurzie:

```
fmap f []     = []
fmap f (x:xs) = f x : (fmap f xs)
```

Z imperatívnych jazykov však vieme, že štandartné zoznamy nie sú jediným kontajnerom, na ktorý je možné použiť funkciu `fmap`. V C++, o ktorom málokedy uvažujeme ako o funkcionálnom jazyku, môžeme použiť funkciu `std::for_each` na ľubovoľný iterátor, na ktorého prvkoch je definovaná rovnosť. Ako je na tom z tohto hľadiska Haskell?

Aby sme mali s čím experimentovať, zopakujeme si definíciu binárneho stromu z druhej kapitoly:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

(Pokiaľ táto definícia nie je jasná, pravdepodobne ju objasní návrat k časti o konštrukcii dátových typov v druhej kapitole.)

Funkcia `fmap` pre binárne stromy bude mať nasledovnú typovú signatúru:

```
fmap :: (a -> b) -> (Tree a -> Tree b)
```

Jej implementácia je priamočiara. Presne ako v prípade zoznamu použijeme porovnávanie so vzorom na dátových konštruktoroch a rekurzívne volanie v rekurzívnej časti definície:

```
fmap f (Leaf x)     = Leaf (f x)
fmap f (Node lx rx) = Node (fmap f lx) (fmap f rx)
```

Dáva preto zmysel deklarovať typovú triedu pre „všetko, čo má fmap“. Túto typovú triedu nazveme Functor:

```
class  Functor f  where
    fmap :: (a -> b) -> f a -> f b
```

Zo zoznamov a binárnych stromov môžeme urobiť inštancie triedy `Functor` tým, že pre oba implementujeme `fmap` - čo sme vlastne už urobili, takže kód, ktorý máme, len skopírujeme do definícií inštancií:

```
instance Functor [] where
    fmap f []     = []
    fmap f (x:xs) = f x : (fmap f xs)

instance Functor Tree where
    fmap f (Leaf x)     = Leaf (f x)
    fmap f (Node lx rx) = Node (fmap f lx) (fmap f rx)
```

Tu sa na chvíľu zastavíme a všimneme si niečo dôležité. Typová trieda `Functor` sa v jednom ohľade lísi od typových tried `Eq`, `Ord`, `Show` a `Read`, ktoré sme videli predtým. Jej prvkami nie sú `Tree Int`, `Tree Char`, atď., ale proste `Tree`. Inými slovami, jej prvkami nie sú typy, ale funkcie, ktoré berú ako parameter typ a vracajú iný typ - typové konštruktory. Akú významnú úlohu zohrávajú typové konštruktory a ich triedy v jazyku Haskell, uvidíme neskôr v kapitole o monádach.

Na záver ešte zostáva dodať, že samotná implementácia funkcie, ktorá vyhovuje typovej signatúre pre `fmap`, nestačí na to, aby sme mohli typový konštruktor považovať za funktor. Od funkcie `fmap` navyše požadujeme, aby mala určité vlastnosti. Na ich vyjadrenie použijeme operátor skladania funkcií `(.)`, ktorý sme si popísali v druhej kapitole, a funkciu `id`, ktorá je definovaná nasledovne:

```
id :: a -> a
id x = x
```

Listing 3.1: Funkcia id

Požadované vlastnosti funkcie `fmap` potom sú:

Vlastnosť 1. $\text{fmap}\, id = id$

Vlastnosť 2. $\text{fmap}\, (g.f) = \text{fmap}\, g \,.\, \text{fmap}\, f$

3.3 Monády

Monády sa často považujú za komplikovaný a veľmi abstraktný koncept, čo vedie k tendenciám vysvetľovať ich pomocou analógii. `Monad` je však typová trieda ako každá iná.

Podobne ako u triedy `Functor` ide o typovú triedu s jedným parametrom - jej prvkami teda nie sú typy, ale typové konštruktory.

Asi najjednoduchšou bežne používanou monádou je monáda pre typový konštruktor `Maybe` (Lipovaca, 2011):

```
data Maybe a = Nothing | Just a
```

Pre zopakovanie: typový konštruktor je vlastne funkcia, ktorá dostane typ a vráti typ. Ak dáme typovému konštruktoru `Maybe` ako parameter `Int`, vytvoríme tak typ `Maybe Int`. `Maybe Int` je typ kontajneru, do ktorého je možné uložiť buď informáciu o neexistencii hodnoty, alebo práve jednu hodnotu typu `Int`.

Pre typ `Maybe Int`, ale aj pre každý iný typ skonštruovaný z `Maybe`, môžeme definovať tri funkcie (Wadler, 1992):

```
mapMaybe  :: (a -> b) -> Maybe a -> Maybe b
unitMaybe :: a -> Maybe a
joinMaybe :: Maybe (Maybe a) -> Maybe a
```

Tu sa na chvíľu zastavíme a využijeme príležitosť pozrieť sa na monády z hľadiska konceptov, ktoré už poznáme. Pripomenieme si typovú signatúru funkcie `fmap`, ktorú sme použili pri popise typovej triedy `Functor`:

```
fmap :: (a -> b) -> (f a -> f b)
```

To vyzerá ako funkcia `mapMaybe` - je teda `Maybe` aj `Functor`? Zistíme, že áno - ak chceme zachovať sémantiku „aplikovania funkcie na kontajner“ pre `Maybe`, musíme `mapMaybe` definovať takto:

```
mapMaybe f Nothing  = Nothing
mapMaybe f (Just x) = Just (f x)
```

Listing 3.2: Funkcia mapMaybe

Aby sme získali lepšiu predstavu o tom, ako sa dokazuje, že nejaká implementácia funkcie spĺňa príslušné axiómy, dokážeme si, že takto definovaný `mapMaybe` spĺňa axiómy funktoru.

Budeme kombinovať rozbor prípadov z definície `Maybe` s technikou, ktorá sa nazýva „usudzovanie z rovností“ (equational reasoning). O definíciach z funkcie `mapMaybe` budeme uvažovať ako o rovnostiach, o ktorých vieme, že platia v oboch smeroch.

Dôkaz. [Prvá vlastnosť funktoru]

$$\begin{aligned} \text{mapMaybe}\, id\ Nothing &= Nothing && mapMaybe, 3.2 \\ &= id\ Nothing && id, 3.1 \end{aligned}$$

$$\begin{aligned}
\text{mapMaybe}\, id\ (Just\ x) &= Just\ (id\ x) && mapMaybe,3.2\\
&= Just\ x && id,3.1\\
&= id\ (Just\ x) && id,3.1
\end{aligned}$$

□

Dôkaz. [Druhá vlastnosť funktoru]

$$\begin{aligned}
\text{mapMaybe}\ (g.f)\ Nothing &= Nothing && mapMaybe,3.2\\
&= \text{mapMaybe}\, g\ Nothing && mapMaybe,3.2\\
&= \text{mapMaybe}\, g\ (\text{mapMaybe}\, f\ Nothing) && mapMaybe,3.2\\
&= (\text{mapMaybe}\, g\,.\,\text{mapMaybe}\, f)\ Nothing && 2.1
\end{aligned}$$

$$\begin{aligned}
\text{mapMaybe}\ (g.f)\ (Just\ x) &= Just\ ((g.f)\ x) && mapMaybe,3.2\\
&= Just\ (g(f\ x)) && 2.1\\
&= \text{mapMaybe}\, g\ (Just\ (f\ x)) && mapMaybe,3.2\\
&= \text{mapMaybe}\, g\ (\text{mapMaybe}\, f\ (Just\ x)) && mapMaybe,3.2\\
&= (\text{mapMaybe}\, g\,.\,\text{mapMaybe}\, f)\ (Just\ x) && 2.1
\end{aligned}$$

□

Obdobnú úvahu môžeme podniknúť pre ľubovoľnú monádu. Funkciu `mapMaybe` teda môžeme použiť ako `fmap` pri definícii `Maybe` ako inštancie `Functor`. Ide teda o špeciálny prípad `fmap` a v ďalšom texte ich budeme používať ako navzájom zameniteľné..

Čím presne sa teda monáda pre typový konštruktor `Maybe` líši od funktoru pre tento typový konštruktor? Je to len otázkou existencie funkcií s typovými signatúrami zodpovedajúcimi `unitMaybe` a `joinMaybe` ? Odpoveď nájdeme napríklad v článku Philipa Wadlera „Comprehending Monads“ (Wadler, 1992) - funkcie, ktoré sú súčasťou typovej triedy `Monad`, musia mať nasledovné vlastnosti (ako sme už spomínali pri vlastnostiach funktoru, `id` je identita, `(.)` značí operátor skladania funkcií) :

Vlastnosť 3. $\text{mapMaybe}\, f\,.\,\text{unitMaybe} = \text{unitMaybe}\ .\ f$

Vlastnosť 4. $\text{mapMaybe}\, f\,.\,\text{joinMaybe} = \text{joinMaybe}\,.\,\text{mapMaybe}\ (\text{mapMaybe}\, f)$

Vlastnosť 5. $\text{joinMaybe}\,.\,\text{unitMaybe} = id$

Vlastnosť 6. $\text{joinMaybe}\,.\,\text{mapMaybe}\ \ \text{unitMaybe} = id$

Vlastnosť 7. $\text{joinMaybe}\,.\ \text{joinMaybe} = \text{joinMaybe}\,.\,\text{mapMaybe}\ \ \text{joinMaybe}$

(To je už trochu príliš veľa vlastností na to, aby sme ich platnosť overovali ručným dokazovaním. Keď nejakú monádu implementujeme, môžeme získať dobrý odhad toho, či je to naozaj monáda, tým, že platnosť axiómov otestujeme napríklad pomocou QuickChecku (Claessen a Hughes, 2000).)

Keď sa však pozrieme na definíciu typovej triedy `Monad` v štandardnej knižnici jazyka Haskell, uvidíme niečo diametrálne odlišné:

```
class Monad m where
    return :: a -> m a
    join   :: m (m a) -> m a
    (>>=)  :: m a -> (a -> m b) -> m b
    fail   :: String -> m a
```

Ak si za typovú premennú `m` dosadíme `Maybe`, zistíme, že funkciu `join` poznáme z predchádzajúcej definície monády (Wadler, 1992) a `return` ma rovnakú typovú signatúru ako `unitMaybe` - ale odkiaľ sa vzali zvyšné dve funkcie?

V tomto prípade nám situáciu neobjasní ani minimálna úplná definícia - tou sú funkcie `return` a `(>>=)`.

Položíme si teda otázku, ktorá logicky nasleduje: dokážeme na základe typových signatúr povedať, ako by mali vyzerať funkcie `return` a `(>>=)` pre typový konštruktor `Maybe`? Pri ich definícii si musíme uvedomiť, že (ako sme si povedali v časti o operátoroch), `(>>=)` je operátor, a teda sa defaultne používa v infixovej notácii:

```
return :: a -> Maybe a
return x = Just x

(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
Nothing  >>= _ = Nothing
(Just x) >>= f = f x
```

Na tomto mieste poznamenáme, že funkcia `(>>=)` sa niekedy nazýva aj `bind`.)

Ďalšou rozumnou otázkou by mohlo byť: je možné implementovať `fmap` pomocou funkcií `return` a `(>>=)`? Vysvitne, že to možné je, a dokonca pomerne jednoduché:

```
fmap f mx = mx >>= (return . f)
```

Aby sme skonštruovali túto funkciu, pripomenieme si, že v Haskelli je z funkcie typu `f: a -> b -> c` možné vytvoriť funkciu `g: b -> a -> c` (s obráteným poradím argumentov) pomocou funkcie `flip`:

```
g = flip f
```

Funkcia `fmap` má typ `(a -> b) -> (m a -> m b)`, funkcia `bind` má typ `m a -> (a -> m b) -> m b`.

Funkcia 'flip bind' má teda typ veľmi podobný `fmap`: `(a -> m b) -> (m a -> m b)`. Jediné, čo potrebujeme, je vytvoriť funkciu, ktorá závolá `f` a „zabalí“ výsledok do monády - ale presne to dosiahneme zložením `f` s monádovým `return`. Máme teda funkciu

```
fmap f mx = flip (>>=) (return . f) mx
```

čo můžeme zrozumiteľnejšie zapísať ako:

```
fmap f mx = mx >>= (return . f)
```

Získali sme viac, než sme čakali; implementácia vôbec nezávisí na tom, v akej monáde je „zabalený“ argument funkcie `fmap` - táto implementácia bude fungovať pre každú monádu! A navyše je možné dokázať, že spĺňa požadované vlastnosti funktoru.

Sme teda schopní previesť funkcie, ktoré poskytuje typová trieda `Monad` v Haskelli, na tie z Wadlerovej definície monády. V presvedčení, že ide o ekvivalentné definície, nás utvrdí, keď medzi pomocnými funkciami, ktoré implementuje každá monáda v Haskelli, nájdeme zovšeobecnenie `mapMaybe` - je len ukryté pod názvom `liftM`.

Neoprekvapí nás preto, že pre `fmap` definovaný pomocou `(>>=)` vieme dokázať, že v spojení s našimi starými známymi funkciami `unitMaybe` a `joinMaybe` spĺňa vlastnosti 3 až 7.

Ekvivalenciu zavŕši fakt, že požadované vlastnosti monády je možné sformulovať pomocou funkcií `return` a `(>>=)`, a to veľmi prehľadným spôsobom - `return` je ľavá a pravá jednotka pre `bind` a `bind` je asociatívny.

Takáto formulácia vlastností je dokázateľne ekvivalentná tej, ktorú sme si uviedli vyššie, a jej formálny zápis je nasledovný (Yorgey, 2011):

Vlastnosť 8. $\text{return}\, a \;>\!>\!=\; k = k\, a$

Vlastnosť 9. $m \;>\!>\!=\; \text{return} = m$

Vlastnosť 10. $m \;>\!>\!=\; (\backslash x \to k\, x \;>\!>\!=\; h) = (m \;>\!>\!=\; k) \;>\!>\!=\; h$

Tieto axiómy sú pre nás zaujímavé aj preto, pretože poskytujú priestor pre optimalizujúce transformácie kódu, ktorý používa funkcie z typovej triedy `Monad` - obe strany rovnosti vrátia ten istý výsledok, ale rozdiely v dĺžke výpočtu alebo pamäťovej náročnosti môžu byť značné. (Piponi, 2007)

Zostáva nám ešte funkcia `fail`, ktorá sa ani v jednej sade axiómov nevyskytuje. Je to preto, že ide o syntaktické pozlátko, ktoré nie je pre definíciu monády nutné - ide proste o funkciu, ktorá sa zavolá, keď v kóde, ktorý využíva monády, dôjde k chybe pri porovnávaní so vzorom.

Je potrebné spomenúť, že Haskell nám však nebráni definovať ako inštanciu typovej triedy `Monad` aj typový konštruktor, ktorý tieto axiómy nespĺňa. Skontrolovať, že daná implementácia typového koštruktoru spĺňa monádove axiómy, je zodpovednosťou programátora.

Štandardná knižnica jazyka Haskell používa typovú triedu `Monad` ako zastrešujúci koncept pre nepreberné množstvo abstrakcií - nedeterministické výpočty, meniteľný stav, komunikácia s vonkajším svetom (vstup a výstup), práca s konfiguráciou programu, logovanie atď. Každá z týchto monád je definovaná konkrétnym typom a okrem vyššie uvedených spoločných funkcií má aj špecifické funkcie, ktoré realizujú vlastnú činnosť monády.

Ešte posledná poznámka na záver: pre zjednodušenie kódu, ktorý beží v monadickom kontexte, je možné použiť takzvanú do-notáciu. Do-notácia „vyzerá“

ako integrovaný haskellovský DSEL a je preto pre našu prácu kľúčová. Nejde však o základnú vlastnosť jazyka, ale o niečo, čo sa interne kompiluje do kódu používajúceho funkcie z typovej triedy `Monad`.

Do-notáciu si podrobnejšie vysvetlíme na príklade.

Kód

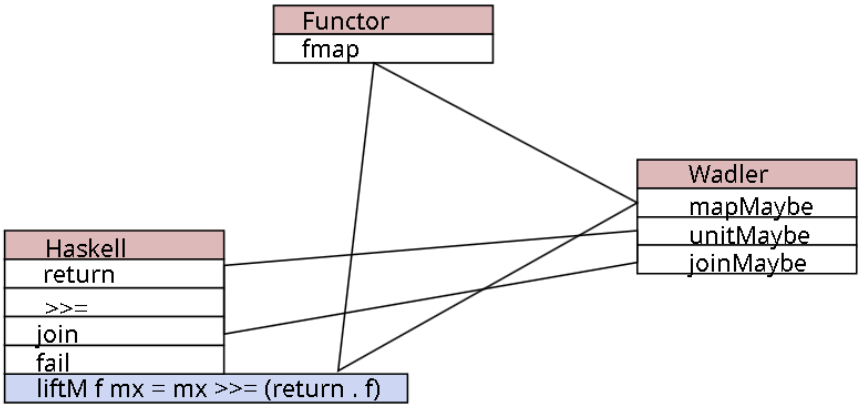
```
do
  put "x: "
  l <- get
  return (head l)
```

sa v prvom kroku prekladu prevedie do kódu (Marlow, 2010):

```
put "x: " >>= \_ ->
get       >>= \l ->
return (head l)
```

Volanie `put "x:"` je funkcia `State` monády, ktorá nevracia hodnotu. Naň nadväzujúca lambda funkcia teda svoj argument zahodí a zavolá funkciu `get`, ktorá vráti hodnotu uloženú v `State` monáde, čo je `"x:"`. Táto hodnota sa naviaže na argument `l` ďalšej lambda funkcie, ktorej telo zavolá `head l`. Výsledkom tohto volania je `"x"`. Funkcia `return` „zabalí“ hodnotu `x` do typového konštruktora prislúchajúceho `State` monáde.

Podrobnejšie si funkcie `put` a `get` popíšeme v nasledujúcej podkapitole, ktorá bude o stavovej monáde.



Obr. 3.1: Porovnanie typovej triedy Functor, Wadlerovej definície monády a definície monády v Haskelli. Hrany spájajú funkcie, ktoré si navzájom zodpovedajú.

3.3.1 Monáda State

Jednou zo základných abstrakcií imperatívneho programovania je koncept meniteľného stavu. Medzi najčastejšie používané príkazy patrí príkaz priradenia, ktorým meníme stav premennej.

Zatiaľ čo imperatívne jazyky stavajú na možnosti meniť stav kdekoľvek a kedykoľvek, a používajú rozličné techniky - lokálne premenné, funkcie, kompilačné

jednotky, extrahovanie kódu do knižníc - aby túto možnosť obmedzili, funkcionálne jazyky používajú opačný prístup. Funkcie možnosť udržovať si a meniť nejaký stav nemajú, pokiaľ o ňu explicitne nepožiadajú. Jednou z možností, ako v jazyku Haskell reprezentovať a spravovať meniteľný stav, je monáda `State`.

Rozhranie k stavu uloženému v tejto monáde zabezpečujú tri funkcie:

```
get         :: m s
put         :: s -> m ()
state         :: (s -> (a, s)) -> m a
```

Sémantika týchto funkcií je zjavná z ich typových signatúr. Funkcia `get` slúži na získanie stavu z monády, `put` na uloženie nového stavu a `state` vykoná jeden krok stavového výpočtu - prečíta stav, aplikuje funkciu odovzdanú parametrom, uloží nový stav a vráti výsledok.

Okrem týchto funkcií je v monáde `State` dôležitá ešte funkcia

```
runState:: State s a -> s -> (a, s)
```

na ktorú je možné sa pozerať dvomi rozličnými spôsobmi:

- ako na funkciu, ktorá z monadického stavového výpočtu extrahuje funkciu vykonávajúcu jeden krok stavového výpočtu
- ako na funkciu, ktorá dostane monadický stavový výpočet a počiatočný stav, a vráti výsledok výpočtu a finálny stav.

Vzhľadom k úlohe, akú zohráva `State` monáda v tejto práci, si uvedieme podrobnejší príklad jej použitia. Ukážeme si, ako je pomocou State monády možné očíslovať prvky v binárnom strome, ktorý má hodnoty len v listoch. Tento binárny strom sme už videli v druhej kapitole a v kapitole o funktoroch. V záujme prehľadnosti si však uvedieme úplný zdrojový kód príkladu.

```
import Control.Monad.State

data Tree a = Leaf a | Node (Tree a) (Tree a) deriving (Show)

label :: Tree a -> State Int (Tree (a, Int))
label (Leaf x) = do n <- state (\n -> (n,n+1))
                    return (Leaf (x,n))
label (Node lx rx) = do l' <- label lx
                        r' <- label rx
                        return (Node l' r')
```

Funkcia `label` vykonáva porovnávanie so vzorom na parametri typu `Tree a`.

Ak je paramater list, pomocou `state` získame číslo uložené v `State` monáde, inkrementujeme uloženú hodnotu o 1 a vrátime očíslovaný list.

Pretože vo vnútorných uzloch stromu nie sú uložené hodnoty, k nijakému číslovaniu tam nedochádza a funkciu len rekurzívne zavoláme na oba podstromy.

Funkcia `label` je vlastne monadický výpočet. Ak chceme získať jeho výsledok, musíme ho pomocou `runState` spustiť s nejakými konkrétnymi dátami.

Pomocou príkazu `:l <súbor>` môžeme vyššie uvedený zdrojový súbor načítať do interpreteru. Výsledok potom získame zavolaním:

```
>> fst $ runState (label (Node (Leaf 'a') (Leaf 'b'))) 0
Node (Leaf ('a',0)) (Leaf ('b',1))
```

Ako parametre funkcii `runState` dáme `label` so stromom, ktorý chceme očíslovať, a číslo 0 - počiatočný stav, ktorý chceme na začiatku uložiť do `State` monády.

Ako sme si už povedali, funkcia `runState` vracia dvojicu (výsledok výpočtu, finálny stav). Nás zaujíma len výsledok výpočtu, preto použijeme `fst`. Túto funkciu aplikujeme pomocou `$`, aby sme nemuseli pridávať ďalšiu úroveň zátvoriek.

3.3.2 Monáda List

`List` je monáda, ktorá je pre účely tejto práce skoro tak dôležitá ako `State`. Dopodrobna si preto rozoberieme jej správanie a použitie. (Po technickej stránke ide vlastne o definíciu typového konštruktoru `[]` ako inštancie `Monad` - v záujme prehľadnosti a konzistencie s dokumentáciou budeme ale hovoriť o `List` monáde.)

Dátový typ pre zoznam a jeho konštruktory už poznáme z druhej kapitoly. Monáda `List` tiež na rozdiel od väčšiny ostatných monád nemá špecifické funkcie, ktoré by zabezpečovali jej funkcionalitu.

Prejdeme preto rovno k jej definícii ako inštancie typovej triedy `Monad`:

```
instance Monad [] where
        return x = [x]

        xs >>= f = concatMap f xs
```

Pomocné funkcie sú implementované nasledovne:

```
concat :: [[a]] -> [a]
concat []       = []
concat (xs:xss) = xs ++ (concat xss)

concatMap = concat . fmap
```

Operátor `(++)` je štandartný operátor spájania zoznamov. Funkciu `fmap` sme implementovali, keď sme dokazovali, že zoznam je funktor. Pre zaujímavosť ešte spomenieme, že `concat` je vlastne `join` a táto definícia je teda aplikáciou monadického axiómu 6 na definíciu `fmap` pomocou `return` a `(>>=)`.

Ďalej si zopakujeme, že pre zápis monadického kódu je možné použiť donotáciu. Výsledkom je, že môžeme písať kód:

```
f = do
    x <- [1,2,3]
    y <- [4,5,6]
    return (x+y)
```

namiesto kódu, ktorý vyzerá takto:

```
f = [1,2,3] >>= \x ->
    [4,5,6] >>= \y ->
    return (x + y)
```

Operátor ← má sémantiku „vybalenia“ hodnoty z monadického kontextu, aby sme ju mohli poskytnúť čistej funkcii.

Vyššie uvedený kód vráti `[5,6,7,6,7,8,7,8,9]`. Vidíme, že výsledkom je zoznam všetkých možných súčtov medzi prvkami z prvého a prvkami z druhého zoznamu - presne ako keby sme pričítali k 1 najprv 4, potom 5, potom 6, a potom, keď sa nám minuli prvky druhého zoznamu, sme „backtrackovali“ do prvého zoznamu a vybrali z neho nasledujúci prvok. Monáda `List` nám umožnila implementovať nedeterministický súčet dvoch zoznamov!

Tiež stojí za povšimnutie, že naša implementácia nedeterministického súčtu zoznamov vyzerá viac ako pseudokód než ako Haskell. Význam operátora ← je intuitívne zrejmý. Z tohto dôvodu sa do-notácia v Haskelli často používa ako odrazový mostík k implementácii vnorených doménovo špecifických jazykov, a my ju budeme používať rovnako.

Táto jednoduchá implementácia nedeterministických algoritmov je dôvodom, prečo sa `List` často nazýva monádou nedeterminizmu - „vybaliť“ hodnotu z kontextu monády `List` vlastne znamená vykonať operáciu nedeterministického výberu. Presne takúto operáciu potrebujeme, aby sme mohli implementovať základný algoritmus prehľadávania stavových priestorov - prehľadávanie s návratom (backtracking).

Na záver ešte poznamenáme, že monáda `List` obsahuje nepreberné množstvo pomocných funkcií, ktoré uľahčujú prácu so zoznamami.

3.3.3 Monáda Reader

Z objektovo orientovaných jazykov poznáme triedy, ktoré nám umožňujú pristupovať ku globálnej konfigurácii programu - napríklad parametrom uvedeným na príkazovom riadku. Podobnému účelu slúži monáda `Reader` - môžeme do nej uložiť globálne parametre, aby sme ich nemuseli odovzdávať do funkcií explicitne. Niekedy sa preto nazýva aj monádou prostredia.

Základné funkcie špecifické pre túto monádu sú tri:

```
ask    :: m r
local  :: (r -> r) -> m a -> m a
reader :: (r -> a)        -> m a
```

Funkcia `ask` získa prostredie uložené v monáde.

Funkcia `local` vykoná lokálnu modifikáciu prostredia využitím funkcie, ktorú dostane ako prvý paramater.

Funkcia `reader` získa z prostredia jednu z hodnôt, ktoré sú v ňom uložené.

3.3.4 Monáda Cont

(Hlavným dôvodom existencie tejto podkapitoly je vyjasnenie niektorých detailov ohľadne alternatívnych implementácii monády zoznamov. Pokiaľ čitateľa táto problematika nezaujíma, môže túto podkapitolu bez obáv preskočiť.)

Ak čitateľ tejto práce niekedy skúmal do hĺbky štandartnú knižnicu jazyka C, pravdepodobne pozná rozhranie pre takzvané nelokálne skoky - funkcie `setjmp()` a `longjmp()`.

Funkcia `setjmp()` uloží stav programu v momente svojho zavolania do špeciálnej dátovej štruktúry. Ak zavoláme funkciu `longjmp()` a dáme jej túto štruktúru ako parameter, program obnoví svoj stav do bodu, v ktorom sa zavolala funkcia `setjmp()`, a pokračuje vo vykonávaní kódu od miesta zavolania funkcie `setjmp()`.(Kerrisk, 2017)

Oblasť použiteľnosti týchto dvoch funkcií je však pomerne obmedzená. Vyskočenie z funkcie pomocou `longjmp()` môže viesť k únikom pamäte alebo súborových deskriptorov, ktoré spravovala táto funkcia. Nevykonáva sa štandartné obnovenie stavu zo zásobníku, ku ktorému dochádza pri klasickom návrate z funkcie. Uložený stav programu je navyše platný len do okamihu návratu z funkcie, ktorá volala `setjmp()`.

Mnoho funkcionálnych jazykov používa obdobné konštrukcie; iné stratégie spravovania stavu však umožňujú implemenetovať ich prehľadnejšie a bezpečnejšie. Vo funkcionálnom názvosloví sa ekvivalent stavu programu nazýva „kontinuácia“ a ekvivalent navrátu k uloženému stavu „volanie/aplikácia kontinuácie“.

V programovacom jazyku Haskell sú kontinuácie implementované pomocou monády `Cont`. Rozhraním pre túto monádu je funkcia:

```
callCC :: ((a -> m b) -> m a) -> m a
```

Táto funkcia zavolá zadanú funkciu s „prostredím“ (kontinuáciou), ktorá je momentálne uložená v monáde, a vráti výsledok zadanej funkcie.

Na záver ešte poznamenáme, že vo funkcionálnych jazykoch sa kontinuácie niekedy používajú aj na reťazenie volaní funkcií. Tento idiom sa nazýva „odovzdávanie kontinuácií“ (continuation passing style, CPS). Uvedieme príklad, ktorý sa vyskytuje v dokumentácii k monáde `Cont` (Newbern a kol., 2007):

```
calculateLength :: [a] -> Cont r Int
calculateLength l = return (length l)

double :: Int -> Cont r Int
double n = return (n * 2)

main = do
  runCont (calculateLength "123" >>= double) print
```

Tento kód vráti `6`.

Už tento jednoduchý príklad naznačuje, že kontinuácie by mohli byť užitočné pri návrhu a implementácií funkcionálnych programovacích jazykov. Prieskum literátury dosvedčí, že je tomu naozaj tak - spomeňme napríklad (Friedman a Wand, 2008) a (Appel, 2007). Pretože sa však zaoberajú všeobecne zameranými funkcionálnymi programovacími jazykmi, ich hlbší rozbor je mimo rozsah tejto práce.

3.4 Typové triedy s viacerými parametrami a funkcionálne závislosti

(Tieto koncepty sú potrebné pre pochopenie implementácie monadických transformátorov, ktoré umožňujú skladanie monád v jazyku Haskell. Pokiaľ čitateľ nechce skúmať detaily ich fungovania, môže túto kapitolu bez obáv preskočiť.)

Všetky typové triedy, s ktorými sme sa zatiaľ stretli, prijímali len jeden parameter - v niektorých prípadoch typ (ako `Eq` a `Ord`), v iných prípadoch typový konštruktor (ako `Functor` a `Monad`). Vždy však šlo len o jeden parameter. Nič nám však nebráni definovať typové triedy, ktoré majú parametrov viacero.

Predstavme si napríklad typovú triedu, ktoré poskytuje abstrakciu nad rozhraním monády `Reader`:

```
class Monad m => MonadReader r m where
<funkcie Reader monády>
```

Podobne ako u typových signatúr funkcií, `Monad m =>` je typový kontext. Pomocou neho vyjadrujeme, že typ `MonadReader r m` je definovaný len v prípade, keď `m` je inštanciou typovej triedy `Monad`.

Výraz `MonadReader r m` je ekvivalentný výrazu `(MonadReader r) m`. Hlavička deklarácie typovej triedy teda vyjadruje, že ak máme monádu `m`, môžeme ju urobiť inštanciou typovej triedy `MonadReader r` bez ohľadu na hodnotu `r` - to znamená, bez ohľadu na typ prostredia, ktoré je uložené v `Reader` monáde.

Ak chceme používať typové triedy s viacerými parametrami, musíme povoliť rozšírenie jazyka Haskell `MultiParameterTypeClasses`.

Ak by sme však typovú triedu `MonadReader` chceli implementovať a používať presne tak, ako sme si ju definoval vyššie, náš program by neprešiel typovou kontrolou. Aby mal prekladač všetky potrebné informácie, potrebujeme nadefinovať takzvanú funkcionálnu závislosť (functional dependency; fundep).

Deklarácia typovej triedy `MonadReader` s funkcionálnou závislosťou vyzerá takto:

```
{-# LANGUAGE MultiParameterTypeClasses #-}
{-# LANGUAGE FunctionalDependencies #-}

class Monad m => MonadReader r m | m -> r where
<funkcie Reader monády>
```

`LANGUAGE` direktívy zapínajú potrebné rozšírenia jazyka Haskell (`FunctionalDependencies` umožňuje používať funkcionálne závislosti). Deklaráciu hlavičky typovej triedy `MonadReader` môžeme čítať nasledovne: „Za predpokladu, že m je monáda, MonadReader r m je typová trieda a m jednoznačne určuje r.“

Poďme sa ešte bližšie pozrieť na to, kde presne typová kontrola bez funkcionálnych závislostí zlyhá. Skúsime urobiť klasickú `Reader` monádu inštanciou našej typovej triedy `MonadReader`:

```
instance MonadReader r (Reader r) where
<implementácia funkcií z MonadReader>
```

Bez funkcionálnych závislostí nie je prekladač schopný odvodiť, že `r` ako parameter pre `Reader` a `r` ako prvý parameter `MonadReader` je ten istý typ, a preto ohlási chybu (O'Sullivan a kol., 2008).

3.5 Skladanie monád

Pri prehľadávani stavových priestorov je užitočné definovať typ, ktorý má vlastnosti viacerých monád súčasne. Napríklad môžeme chcieť v tom istom kóde využívať monádu `Reader` pre prístup ku globálnej konfigurácii prehľadávania (ako je obmedzenie na množstvo využitej pamäte alebo celkova hĺbka prehľadávania) a monádu `State` pre aktuálny stav.

V tejto kapitole si skladanie monád popíšeme formálne a rozoberieme jeho obmedzenia. Na záver si popíšeme iný spôsob, ktorý pre kombinovanie monád využíva jazyk Haskell - monadické transformátory.

3.5.1 Formalizácia konceptu

Definícia 1. *Nech N a M sú monády. Zloženie monád M a N si definujeme ako typový konštruktor, ktorý z každého typu* `a` *vytvorí typ* `(M (N a))`.

Aby sme lepšie videli, kde presne môže pri skladaní monád nastať problém, použijeme v nasledujúcom texte definíciu monády z (Wadler, 1992). V predchádzajúcej podkapitole sme pracovali s jej špeciálnym prípadom pre `Maybe` monádu.

Aby sme mohli z nášho typového konštruktoru urobiť monádu, potrebujeme nasledovné funkcie:

```
mapMN :: (a -> b) -> (M (N a) -> M (N b))
unitMN :: a -> M (N a)
joinMN :: M (N (M (N a))) -> M (N a)
```

A požadované vlastnosti sú:

Vlastnosť 11. $\text{mapMN}\, f \,.\, \text{unitMN} = \text{unitMN} \,.\, f$

Vlastnosť 12. $\text{mapMN}\, f \,.\, \text{joinMN} = \text{joinMN} \,.\, \text{mapMN}\ (\text{mapMN}\, f)$

Vlastnosť 13. $\text{joinMN} \,.\, \text{unitMN} = id$

Vlastnosť 14. $\text{joinMN} \,.\, \text{mapMN}\ \ \text{unitMN} = id$

Vlastnosť 15. $\text{joinMN} \,.\, \text{joinMN} = \text{joinMN} \,.\, \text{mapMN}\ \ \text{joinMN}$

Funkcie z monád, ktoré skladáme, budeme rozlišovať príponami M a N. Na ich základe jednoducho definujeme funkcie `mapMN` a `unitMN` (Jones a Duponcheel, 1993):

```
mapMN  = mapM . mapN
unitMN = unitM . unitN
```

Funkciu `joinMN` však takýmto spôsobom nedostaneme - funkcia `joinM` · `joinN` nie je dokonca ani typovo správna. Funkcia `joinN` prijíma argument typu `(N (N a))`; výsledok bude mať typ `N a`, ale `joinM` očakáva argument typu `(M (M b))` (Jones a Duponcheel, 1993).

Je možné dokázať, že neexistuje funkcia `joinMN`, ktorej definícia by používala len funkcie z definícii monád N a M (Jones a Duponcheel, 1993). Monády teda

nie sú uzavreté na operáciu skladania. Tento výsledok je pre nás natoľko dôležitý, že jeho implikáciam venujeme celú ďalšiu podkapitolu.

Teraz sa však poďme zamyslieť nad tým, akú podmienku musia spĺňať monády M a N, aby bola výsledkom ich zloženia zase monáda. Pravdepodobne by pomohlo, keby sme mali k dispozícii funkciu, ktorá prehodí druhý a tretí typový parameter v argumente `joinMN`:

```
swap23 :: M (N (M (N a))) -> M (M (N (N a)))
```

Na jej výsledok by sme mohli aplikovať `joinM` a dostať `M (N (N a))`, a aplikáciou `mapM joinN` by sme už dostali požadovaný výsledok - `M (N a)`. Ako však túto funkciu implementovať? Cez prvý výskyt M nás dostane `mapM`, takže jediné, čo potrebujeme, je funkcia:

```
swap :: N (M a) -> M (N a)
```

Požadovanú funkciu `joinMN` potom môžeme definovať ako (Jones a Duponcheel, 1993):

```
joinMN = mapM joinN . joinM . mapM swap
```

Len samotná existencia funkcie s typovou signatúrou zodpovedajúcou `swap` však nestačí na to, aby zložením M a N vznikla monáda. Potrebujeme, aby funkcia `swap` spĺňala určité pravidlá.

Pre zjednodušenie ich formulácie si zadefinujeme dve pomocné funkcie:

```
prod = mapM joinN . swap
dorp = joinM . mapM swap
```

Požadované vlastnosti funkcie `swap` potom sformulujeme nasledovne (Jones a Duponcheel, 1993):

Vlastnosť 16. $\mathrm{swap} \,.\, \mathrm{mapN}\ (\mathrm{mapM}\ f) = \mathrm{mapM}\ (\mathrm{mapN}\ f) \,.\, \mathrm{swap}$

Vlastnosť 17. $\mathrm{swap} \,.\, \mathrm{unitN} = \mathrm{mapM}\ \mathrm{unitN}$

Vlastnosť 18. $\mathrm{swap} \,.\, \mathrm{mapN}\ \mathrm{unitM} = \mathrm{unitM}$

Vlastnosť 19. $\mathrm{prod} \,.\, \mathrm{mapN}\ \mathrm{dorp} = \mathrm{dorp} \,.\, \mathrm{prod}$

3.5.2 Obmedzenia skladania monád

V tomto bode je prirodzené položiť si otázku, ako sú na tom monády zo štandartnej knižnice jazyka Haskell z hľadiska skladania.

Pre ďalšiu analýzu budeme potrebovať jednu definíciu:

Definícia 2. *(Jones a Duponcheel, 1993) Monáda M je* `komutatívna` *práve vtedy, keď platí, že výraz*

```
do
   x <- a
   y <- b
   f x y
```

je ekvivalentný

```
do
   y <- b
   x <- a
   f x y
```

Intuitívne to môžeme chápať ako „nezávislosť na poradí vedľajších efektov“ monadického výpočtu.

Z monád, ktoré poznáme, medzi komutatívne patria `Maybe` a `Reader`. Naopak monády `State`, `List` a `Cont` komutatívne nie sú.

Ďalej môžeme dokázať, že platí:

Veta 20. *(Jones a Duponcheel, 1993) Nech M a N sú monády a M (N a) typový konštruktor, ktorý vznikne ich zložením. Ak je monáda N komutatívna, M (N a) je monáda pre ľubovoľné M.*

Pre účely tejto práce nás primárne zaujíma, ako sa správa pri skladaní monáda `List`. Odpoveď nájdeme v článku (King a Wadler, 1993), ktorý skúma typové konštruktory v tvare `M (List a)`, a v jeho analýze v (Jones a Duponcheel, 1993).

Tvrdenie 21. *Monáda* `List` *nie je komutatívna.*

Dôkaz. Protipríklad sformulujeme pomocou kartézskeho súčinu (King a Wadler, 1993):

```
f = do
     x <- [1,2]
     y <- [3,4]
     return (x,y)
```

vráti `[(1,3),(1,4),(2,3),(2,4)]`,
ale:

```
g = do
     y <- [3,4]
     x <- [1,2]
     return (x,y)
```

vráti `[(1,3),(2,3),(1,4),(2,4)]`.
Monáda `List` teda nie je komutatívna.

□

Na základe 20 a 21 teda prídeme k záveru, na ktorý sa budeme priebežne v práci odkazovať:

Tvrdenie 22. *Typový konštruktor* `List (List a)` *nie je monáda.*

Ktoré z axiómov monády 8 až 10 však `List (List a)` porušuje? Po chvíli premýšľania zistíme, že len 10 - pravidlo asociativity. V ďalších kapitolách uvidíme, že aj takáto „kvázimonáda“ má dostatočnú štruktúru na to, aby bola užitočná v programátorskej praxi.

3.5.3 Monadické transformátory

Ako sme videli v predchádzajúcich dvoch podkapitolách, skladanie monád nie je ani zďaleka triviálny problém. Články (Jones a Duponcheel, 1993) a (King a Wadler, 1993) popisujú techniky, ktoré môžeme použiť, ak máme zafixovanú monádu na ľavej alebo pravej strane operátora skladania - žiadna však nie je dostatočne všeobecná, aby umožnila zložiť každú monádu v štandardnej knižnici s každou.

Navrhári jazyka Haskell však prišli ešte s treťou možnosťou, ktorá generalizuje omnoho lepšie - s konceptom monadických transformátorov.

V Haskelli má prakticky každá monáda v štandartnej knižnici príslušnú typovú triedu, ktorá obsahuje pre ňu špecifické funkcie. Aby sme implementovali monadický transformátor pre konkrétnu monádu, definujeme typový konštruktor, ktorý:

- je inštanciou typovej triedy, ktorá prislúcha tejto monáde
- je inštanciou typovej triedy `Monad`
- je inštanciou typovej triedy `MonadTrans`, ktorá je definovaná nasledovne (Yorgey, 2011):

```
class MonadTrans t where
  lift :: Monad m => m a -> t m a
```

Intuitívne si môžeme predstaviť, že `lift` je pre monadické kontexty to, čo je `fmap` pre funkcie.

Ako príklad si implementujeme monadický transformátor `MaybeT` (O'Sullivan a kol., 2008). Z technických dôvodov použijeme definíciu pomocou `newtype`:

```
newtype MaybeT m a = MaybeT {
      runMaybeT :: m (Maybe a)
}
```

Typovy konštruktor `Maybe` nemá žiadnu preň špecifickú typovú triedu, stačí nám teda implementovať inštancie pre `Monad` a `MonadTrans`:

```
returnMT :: (Monad m) => a -> MaybeT m a
returnMT a = MaybeT (return (Just a))

maybe :: b -> (a -> b) -> Maybe a -> b
maybe dfl _ Nothing = dfl
maybe _ f (Just x)  = f x

bindMT :: (Monad m) => MaybeT m a -> (a -> MaybeT m b) -> MaybeT m b
x `bindMT` f =
    MaybeT (runMaybeT x >>= maybe (return Nothing) (runMaybeT . f))

failMT :: (Monad m) => t -> MaybeT m a
failMT _ = MaybeT (return Nothing)
```

```
instance (Monad m) => Monad (MaybeT m) where
  return = returnMT
  (>>=)  = bindMT
  fail   = failMT

instance MonadTrans MaybeT where
    lift m = MaybeT (fmap Just m)
```

(Funkciu `fmap` sme si definovali v kapitole 3.2.5. V kapitole 3.3 sme odovodili z monadických funkciií `return` a `bind` definíciu funkcie `fmap`, ktorá má pre každú monádu požadované vlastnosti.)

Funkcie s príponou MT sú funkcie monadického transformátoru, funkcie z definície monády, ktoré voláme, patria monáde pre typový konštruktor `m`.

Štandartné knižnice jazyka Haskell poskytujú monadické transformátory pre všetky bežne používané monády. Od monád sa líšia pripojeným T za názvom monády - takže monadický transformátor pre `State` monadu nesie názov `StateT`.

Deklarácia hypotetického dátového typu, ktorý sme popísali ako motiváciu, prečo sa skladaním monád vôbec zaoberať, by teda bola:

```
data Searchspace a conf  = StateT (Reader conf) a
```

Pretože v predchádzajúcej podkapitole sme videli, že monáda `List` zložená so sebou samou nám bez ohľadu na použité techniky nikdy nedá monádu, vzniká prirodzene otázka, či má zmysel usilovať sa až o takúto mieru všeobecnosti. Monadické transformátory umožňujú zložiť ľubovoľné dve monády, ale je zodpovednosťou programátora si overiť, či je výsledok vôbec monáda.

Konkrétne v prípade zoznamov je však táto miera všeobecnosti určite na mieste. Pripomeňme si, že problém s monádou zoznamov zoznamov spočíval v tom, že nespĺňala poslednú vlastnosť monády - zákon asociativity 10.

Ak si porovnáme skladanie s monádou `List` z článku (Jones a Duponcheel, 1993) s implementáciou monadického transformátoru `ListT` v Haskelli, uvidíme, že oba prístupy trpia rovnakým problémom.

Asociativita sa do veľkej miery spolieha na lazy evaluation - a `ListT` tento predpoklad porušuje. Ak sú v zozname uložené monadické výpočty, ich vedľajšie efekty sa môžu vykonať aj pre prvky zoznamu, s ktorými sme nepracovali.

Existuje však alternatívna implementácia monády `List` s veľmi zaujímavou sémantikou - ak na začiatku výpočtu zvolíme, že x má mať hodnotu 1 a neskôr zistíme, že potrebujeme, aby x malo hodnotu 2, výpočet sa bez akejkoľvek našej intervencie „vráti“ na začiatok, dosadí za x hodnotu 2 a znovu sa „vráti“ do bodu, kde zistil nevyhovujúcu hodnotu. To nie je mágia, ide len o šikovné využitie kontinuácií, ktoré sme popísali v časti o monáde `Cont`. Táto alternatívna monáda nedeterministického výberu je dostupná v štandartných knižniciach Haskellu pod názvom `Amb` (ambiguity; nejednoznačnosť).

4. Použitie jazyka Haskell pri implementácii DSEL pre prehľadávanie stavových priestorov

4.1 Prehľad známych metód

Jednou z inšpirácií pre túto prácu boli články Toma Schrijversa et. al (Schrijvers a kol., 2009) a (Schrijvers a kol., 2013) o využití monád pri implementácii prehľadávacích programov[1] pre programovanie s obmedzujúcimi podmienkami.

Článok (Schrijvers a kol., 2009) popisuje prehľadávací program pre programovanie s obmedzujúcimi podmienkami, ktorý má dve časti - jedna rieši spĺňanie podmienok, druhá implementuje prehľadávanie. Riešič podmienok je postavený nad stavovou monádou a zakomponovaný do stromu prehľadávania, ktorý je tiež monáda.

Implementácia prehľadávania pozostáva z nasledovných komponent: špecifikácia stromu prehľadávania, základná prehľadávacia stratégia (DFS, BFS) a „prehľadávacia transformácia“, ktorá mení buď prehľadávací strom, alebo základnú stratégiu prehľadávania (obmedzuje napríklad hĺbku alebo počet navštívených uzlov).

V kóde, ktorý generuje a potom vyhodnocuje strom priradení možných hodnôt (prehľadávací strom pre problém spĺňania obmedzujúcich podmienok), autori parametrizovali základnú prehľadávaciu stratégiu tak, že definovali zoznam uzlov, ktoré je potrebné navštíviť, ako typovú triedu.

Pre stav vyhodnotenia a stav prehľadávacieho stromu existuje ďalšia typová trieda. Prostredníctvom nej môžu základné prehľadávacie stratégie meniť svoje správanie podľa aktuálneho stavu. Táto adaptabilita umožňuje implementáciu prehľadávacích transformácií.

Posledným problémom, ktorý je v článku riešený, je skladanie prehľadávacích transformácii. Je modelované pomocou ukladania na zásobník, ktorý má naspodu základnú prehľadávaciu stratégiu. Každá prehľadávacia transformácia bude teda namiesto funkcií z určitej základnej stratégie volať funkcie z vnorenej transformácie, ktorá je na zásobníku bezprostredne pod ňou. Pokiaľ je pod prehľadávacou transformáciou už len základná stratégia, použijú sa funkcie z tejto stratégie. Takéto správanie je zabezpečené prostredníctvom využitia kontinuácií.

V Search Combinators (Schrijvers a kol., 2013) je prístup založený na základných prehľadávacích stratégiách a ich transformáciách rozšírený na prehľadávanie so zložitejšími heuristikami. Na základných prehľadávacích stratégiách sú namiesto transformácií postavené kombinátory, ktoré je potom možné ďalej skladať do ďalších prehľadávacích stratégií. Medzi kombinátory patria:

- vykonávaj, kým platí podmienka

[1] constraint solvers

- vykonaj raz
- prehľadávaj s obmedzenou diskrepanciu
- prehľadávaj pomocou iterovaného prehlbovania atď.

Článok končí návrhom protokolu popisujúcim interakcie medzi jednotlivými kombinátormi v jednom uzle prehľadávacieho stromu.

Aj keď syntax jazyka Haggle je inšpirovaná Monadic Constraint Programming (Schrijvers a kol., 2009), jeho vnútorná implementácia vychádza zo série článkov od Dana Piponiho o prehľadávaní stavových priestorov v Haskelli pomocou monád[2]. Dôvodom pre toto rozhodnutie bolo, že generalizovaný prístup v Search Combinators (Schrijvers a kol., 2013) bol príliš zložitý a jeho vzorová implementácia bola v C++, zatiaľ čo v Monadic Constraint Programming (Schrijvers a kol., 2009) bolo ťažké oddeliť prehľadávacie abstrakcie od kódu, ktorý zabezpečoval spĺňanie obmedzujúcich podmienok.

Ako sme si povedali v predchádzajúcej kapitole, zoznamy je možné chápať ako monádu:

```
concat :: [[a]] -> [a]
concat []       = []
concat (xs:xss) = xs ++ concat xss

instance Monad [] where
    return x = [x]

    xs >>= f = (concat . fmap) f xs
```

Túto monádu je potom možné jednoducho použiť na implementáciu nedeterministických výpočtov - necháme nedeterministickú funkciu proste vrátiť zoznam všetkých možných výsledkov.

Ďalej si pripomenieme, že s použitím do-notácie naše príklady vyzerali viac ako pseudokód než ako Haskell - čo je dôvod, prečo je do-notácia základným stavebným kameňom pre implementáciu vnorených doménovo špecifických jazykov v Haskelli.

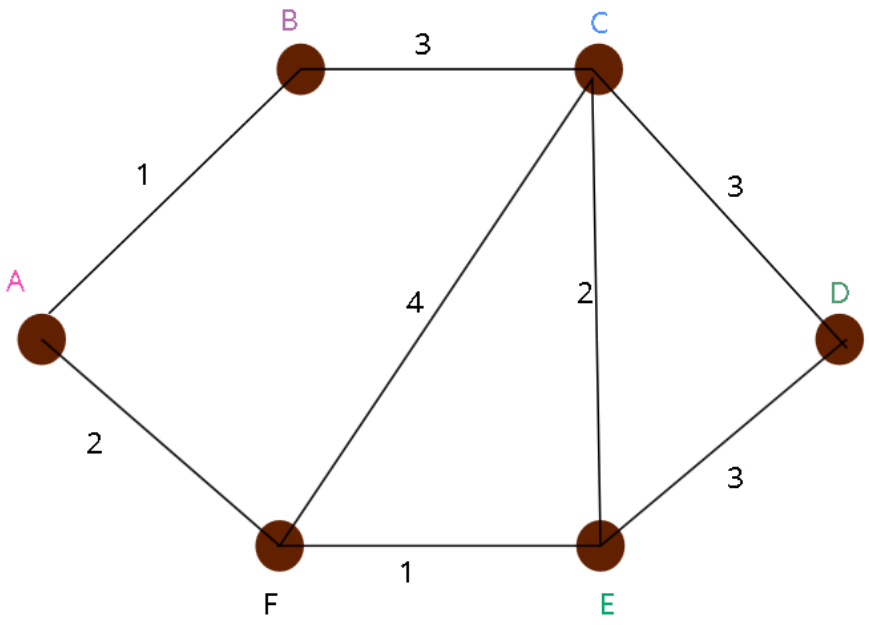
Ďalej budeme pokračovať kľúčovým zistením z článku, ktoré si v nasledujúcich odstavcoch popíšeme podrobnejšie:

> „Keď používame monádu zoznamov, prvky zoznamu môžeme interpretovať ako kandidátov na nasledujúci stav počas prehľadávania. Každému z už vygenerovaných stavov prislúcha zoznam jeho susedov v stavovom priestore. [...] Je pre nás dôležité, že sa používa lazy evaluation, takže kandidáti sú generovaní postupne.“ (Piponi, 2009a)

Citovaný článok uvažuje stavové priestory s ocenenými prechodmi medzi stavmi (príklad takého stavového priestoru je na obrázku 4.1), a usiluje sa nájsť cestu z počiatočného do koncového stavu s minimálnou váhou. Pretože technika skúšania všetkých ciest má exponenciálnu časovú zložitosť, hľadá lepší algoritmus.

Pri jeho hľadaní je dôležité uvedomiť si, že takýto stavový priestor je vlastne ohodnotený graf - jediným problémom je, že niektoré hrany môžu mať nekonečnú

[2]http://blog.sigfpe.com



Obr. 4.1: Príklad stavového priestoru, ktorý je možné popísať orientovaným grafom - A je počiatocný stav, D koncový stav. Najlacnejšia cesta z A do D vedie cez vrcholy F a E a má cenu 6.

váhu. S tým si však Dijkstrov algoritmus poradí bez problémov . Potrebujeme ho len implementovať ho nad polokruhom $(R \cup \{\infty\}, \min, +)$.

Moduly (ktoré si môžeme predstaviť ako ekvivalent vektorových priestorov nad štruktúrou s dvomi operáciami, ktorá nespĺňa axiómy telesa) nad týmto polokruhom tvoria monádu - takže jeden krok prehľadávania je možné implementovať ako monadický bind.(Piponi, 2007) To je posledná ingrediencia potrebná na to, aby sme pomocou monád mohli implementovať algoritmus na hľadanie cesty s minimálnou váhou, ktorý má polynomiálnu časovú zložitosť.

Tento prístup však funguje len v špeciálnom prípade stavových priestorov, v ktorých máme už na začiatku prehľadávania dostupné všetky informácie o váhach hrán. To sa pri bežnom prehľadávaní stavových priestorov nestáva - tam máme k dispozícii prinajlepšom heuristický odhad ceny najlacnejšej cesty do cieľa. Do monád, ktoré Piponi použil pri hľadaní najlacnejšej cesty v grafe, je však možné zakomponovať heuristiky, čo umožní použiť ich pri bežnom prehľadávaní stavových priestorov.

Dan Piponi v (Piponi, 2009a) ukazuje variantu predchádzajúcej monády, ktorá sa dá použiť na takéto heuristické prehľadávanie. Heuristika je implementovaná pomocou penalizácie neperspektívnych vetiev prehľadávacieho stromu. Monáda penalizovaných zoznamov je zoznam zoznamov. Pri bežnom prehľadávaní by sa v každom zozname nachádzali susedia jedného z už objavených stavov. Tento algoritmus však využíva vnorené zoznamy na združenie kandidátov s rovnakým stupňom penalizácie. Táto monáda nám, podobne ako tá z (Piponi, 2007), umožňuje prehľadávať nekonečné stavové priestory bez zacyklenia (to jednoduchá monáda `List` nedokáže). Ako príklad autor implementuje nedeterministický parser aritmetických výrazov s toleranciou chýb.

V (Piponi, 2009b) Piponi implementuje všeobecnú monádu pre heuristické prehľadávanie stavových priestorov, ktorá minimalizuje súčet váh na hranách na ľubovoľnej ceste do listu. Použitou dátovou štruktúrou je 'skew tree' (podobný Okasakiho leftist halde (Okasaki, 1998), ktorá je štandartnou implementáciou prioritnej fronty vo funkcionálnych jazykoch). Aby sme dokázali prehľadávať nekonečné priestory, v každom kroku musíme vrátiť buď výsledok (pre list) alebo aktualizáciu odhadu ceny cesty (pre vnútorný uzol). Ako príklad je implementované parsovanie jednoduchých nejednoznačných gramatík, v ktorom môžeme vidieť náznaky funkcionálnych techník spracovania prirodzeného jazyka.

4.2 Popis jazyka Haggle

Teraz, po zhrnuti článkov, na ktorých sme založili návrh nášho jazyka, popíšeme tento návrh samotný. Konkrétne prehľadávacie algoritmy, ktoré náš prístup umožňuje implementovať, popíšeme v kapitole 6.

Program v jazyku Haggle má tvar:

```
import Haggle

<import modulov pre používané algoritmy>

<import ostatných modulov, napríklad reprezentácie riešeného problému>

main = <selektor riešení> $ do
               with <data>
               <prehľadávací algoritmus 1> <parametre>
               <prehľadávací algoritmus 2> <parametre>
                          ...
               <prehľadávací algoritmus n> <parametre>
```

Importovaním modulu `Haggle` vyjadrujeme, že chceme používať tento DSEL. Každému podporovanému prehľadávaciemu algoritmu tiež zodpovedá modul, ktorý potrebujeme importovať, ak chceme tento algoritmus používať.

Selektor riešení je funkcia, ktorá nám umožňuje vybrať podmnožinu z nájdených riešení. Haggle poskytuje tri základné selektory riešení - `getOneSolution`, `getManySolutions` (ktoré berie ako parameter požadovaný počet riešení) a `getAllSolutions`. Nie je však problémom implementovať si aj vlastný selektor riešení.

Operátor `$` je haskellovský alternatívny operátor aplikácie funkcie, ktorý sme si popísali v druhej kapitole.

Funkcia `with` očakáva ako svoj parameter, v popise označený <data>, počiatočný uzol prehľadávacieho stromu.

Po tomto stručnom zhrnutí sa pozrieme na to, do akej miery môžeme využiť prístupy z existujúcej literatúry pri implementácii takéhoto jazyka.

Dátové typy z prvých dvoch Piponiho článkov sa vyznačujú jednoduchosťou, ale naším účelom nepostačujú. Reprezentácia stavového priestoru ručne napísaným grafom je vhodná len pre veľmi malé príklady. Monáda penalizovaných zoznamov škáluje lepšie, ale poskytuje informáciu len o relatívnom usporiadaní

kandidátov - pre spájanie výsledkov dvoch rozličných heuristických prehľadávacích algoritmov potrebujeme úplné usporiadanie na kandidátoch.

Prioritná fronta z článku (Piponi, 2009b) vyzerá omnoho sľubnejšie. Predstavuje však pre nás problém, že je implementovaná nad jednoduchým rekurzívnym stromom, ktorý nám umožňuje ľahko sa presúvať len v smere z koreňa do listov. Pre naše potreby musíme umožniť aj jednoduché prechádzanie ciest v opačnom smere - od listu (respektíve posledného objaveného vrcholu) do koreňa.

Použitie 'skew tree' je určite dobrá myšlienka, aj keď pravdepodobne na prvý prototyp príliš zložitá. Je však pre nás rozhodne motiváciou, aby sme stavový priestor reprezentovali abstraktným dátovým typom, ktorý môžeme jednoducho meniť a experimentovať tak s rozličnými dátovými štruktúrami.

Jednotlivé prehľadávacie algoritmy budeme spúšťať v spoločnej monáde, ktorá zabezpečí komunikáciu medzi nimi. Pre prechod z jedného prehľadávacieho algoritmu do druhého použijeme monadický bind. Takáto implementácia nám navyše umožňuje pre zápis programov použiť klasickú haskellovskú do-notáciu.

Podrobnému popisu návrhu a implementácie jazyka Haggle je venovaná kapitola 7.

5. Imperatívne techniky návrhu DSEL

Predtým ako prejdeme k technickým detailom implementácie doménovo špecifického jazyka, ktorý sme navrhli, by sme radi čitateľom zvyknutým na imperatívny, objektovo orientovaný štýl programovania poskytli nejaký referenčný rámec. Popíšeme si preto základné techniky návrhu a implementácie DSEL v imperatívnych programovacích jazykoch. Pretože sa však nejedná o hlavnú tému našej práce, pôjde len o stručný prehľad. Pre podrobnejšie informácie doporučujeme knihu (Fowler, 2010), z ktorej táto kapitola čerpá.

Naša analýza sa bude oproti tomuto zdroju špecializovať na vnorené doménovo špecifické jazyky a obzvlášť na porovnanie prístupov k implementácii DSEL v imperatívnych a funkcionálnych jazykoch.

Bez ohľadu na používaný hostiteľský jazyk riešime pri návrhu DSEL dve základné zložky: samotné funkcie poskytované DSEL a integráciu DSEL do zvyšku programu. Tieto problémy sa v imperatívnych a vo funkcionálnych jazykoch riešia diametrálne odlišnými spôsobmi.

Pri návrhu DSEL v imperatívnych jazykoch väčšinou vychádzame z existujúceho objektového modelu, pre ktorý už máme implementované klasické API, a usilujeme sa pre tento model poskytnúť rozhranie, ktoré bude zrozumiteľné doménovému expertovi a zakryje pred ním syntaktické požiadavky a obmedzenia hostiteľského jazyka.

Funkcie poskytované DSEL sa implementujú buď pomocou klasického API, alebo priamo nad objektovým modelom.

Kľúčovou súčasťou návrhu jazyka, ktorej sa venuje najväčšia pozornosť, je integrácia DSEL do zvyšku jazyka. Usilujeme sa preto o to, aby mal DSEL na najvyššej úrovni určitú pevnú štruktúru, ideálne zodpovedajúcu nejakému zo štandartných návrhových vzorov.

Na rozdiel od tohto prístupu sa vo funkcionálnych jazykoch usilujeme navrhnúť API tak, aby ho DSEL mohol priamo používať - nemáme samostatné klasické API.

Integrácia DSEL do hostiteľského jazyka je vo funkcionálnych jazykoch jednoduchšia a priamočiarejšia. Funkcionálne DSEL majú často podobu kombinátorových knižníc, ktoré sú ako akékoľvek iné knižnice proste kolekciami funkcií.

(Kombinátor je návrhový vzor vo funkcionálnom programovaní. Jeho základným princípom je, že pri riešení problému definujeme sadu jednoduchých „primitív“ a sadu „kombinátorov“ - funkcií, ktoré umožňujú vytvárať z týchto primitív zložitejšie štruktúry. (combinator) Príkladom kombinátorovej knižnice je Parsec[1] - knižnica na vytváranie parserov v jazyku Haskell.)

Takéto knižnice neobsahujú nijaký kód, ktorý by mal na starosti integráciu DSEL do hostiteľského jazyka.

Pokiaľ vo funkcionálnych jazykoch integráciu DSEL zabezpečujeme explicitne, má často podobu reimplementácie primitív kontroly toku programu, prípadne poskytnutia nových primitív. Tento prístup by bol v imperatívnych jazykoch veľmi náročný až nemožný.

[1] https://hackage.haskell.org/package/parsec

To isté platí aj o manipulácii s abstraktným syntaktickým stromom programu, ktorú umožňujú niektoré funkcionálne programovacie jazyky - jazyky z rodiny Lispu natívne, Haskell pomocou rozšírenia Template Haskell.

Vo zvyšku kapitoly si popíšeme tri návrhové vzory, ktoré sa používajú pri návrhu DSEL v imperatívnych jazykoch, aby zjednodušili jeho integráciu s hostiteľským jazykom.

5.1 Postupnosť funkcií

Postupnosť funkcií (function sequence) je návrhový vzor pre DSEL, ktorý využíva postupnosť volaní funkcií, ktoré medzi sebou na úrovni DSEL nekomunikujú - nedochádza medzi nimi k nijakému toku dát.

Tento návrhový vzor môžeme využiť dvomi rozličnými spôsobmi.

Jedna možnosť je navrhnúť knižnicu vhodne pomenovaných funkcií. Druhá možnosť je reprezentovať a pamätať si stav práve spracovávaného DSEL výrazu alebo skriptu a pracovať s týmto stavom.

Pri prvej možnosti ide vlastne len o zastrešenie funkcií z klasického API tak, aby poskytovali rozhranie zmyslupné z pohľadu doménového experta. Nebudeme vykonávať nijakú explicitnú integráciu do hostiteľského jazyka. Implementujeme teda vlastne ekvivalent kombinátorovej knižnice zo sveta funkcionálnych DSEL.

Pokiaľ sa však rozhodneme pre tento prístup, bude nám stáť v ceste fakt, že výpočty v imperatívnych jazykoch sa zakladajú na manipulácii so stavom. Takáto knižnica buď nesmie navonok meniť stav (povedané vo funkcionálnom názvosloví, funkcie nesmú mať vedľajšie efekty), alebo musí zariadiť, aby všetky funkcie fungovali vo všetkých možných stavoch - pokiaľ nejaká funkcia očakáva, že objekt nebude NULL, nijaká iná funkcia ho nesmie na NULL nastaviť, pretože nemáme nijakú záruku, že tieto funkcie nebudú z DSEL zavolané tesne po sebe.

Tieto podmienky však môžu byť splnené na najvyššej úrovni hierarchického DSEL alebo pri implementácii podpory homogénnych kontajnerov, a za týchto okolností sa tento návrhový vzor vyplatí použiť.

Pokiaľ chceme použiť tento návrhový vzor na DSEL s väčšou vyjadrovacou silou, zostáva už len druhá možnosť: práca s explicitne vyjadreným stavom práve spracovávaného DSEL výrazu alebo skriptu. Pre komplikovanejšie DSEL však môže byť tento stav veľký a komplikovaný a zodpovedať skoro až stavu parsera spracovávajúceho daný jazyk, čím z praktického hľadiska strácame slovo „vnorený“ z termínu „vnorený doménovo špecifický jazyk“.

5.2 Reťazenie metód

Aby sme mohli popísať tento návrhový vzor, musíme sa vrátiť ku konceptu rozhrania schopného reťazenia, ktorý sme spomenuli v prvej kapitole ako hlavnú odlišnosť medzi DSEL a klasickými API.

Reťazenie metód (method chaining) nevyhnutne potrebuje, aby všetky funkcie používané v DSEL vrátili ako svoju návratovú hodnotu objekt, s ktorým pracovali. Na tomto objekte potom môžeme zavolať ďalšiu funkciu, na jej návratovej hodnote ďalšiu, ... a takýmto spôsobom môžeme pokračovať tak dlho, ako potrebujeme.

Nie je jednoduché nájsť funkcionálnu analógiu k reťazeniu metód. Nejde o ekvivalent skladania funkcií vo funkcionálnom programovaní, pretože tam ku zmenám stavu nedochádza. Najpodobnejšie sú asi DSEL implementované pomocou monád, kde na reťazenie namiesto zavolania funkcie slúži monadický bind. Monadický spôsob implementácie má však väčšiu vyjadrovaciu silu (asi by bolo možné simulovať bind v Cont monáde pomocou reťazenia metód, ale implementácia by bola omnoho zložitejšia), preto tieto prístupy nepovažujeme za ekvivalentné.

Podobne ako postupnosť funkcií, aj tento návrhový vzor potrebuje reprezentáciu stavu spracovávaného DSEL výrazu alebo skriptu. Pretože však veľká časť zložitosti interakcií medzi jednotlivými funkciami z DSEL je ukrytá do objektov, s ktorými tieto funkcie pracujú, samotný stav je jednoduchší.

Tento prístup má však aj niekoľko problémov.

Nie je jasné, kde reťazec metód skončí a kedy zastaviť jeho spracovávanie - niekedy sa pre to používa explicitná metóda End.

Reťazenie metód nie je vhodné pre DSEL s hierarchickou štruktúrou. Musíme si ukladať umiestnenie spracovávaného výrazu v hierarchii do kontextových premenných alebo použiť nejaký iný návrhový vzor pre vytváranie podštruktúr.

Posledným problémom sú povinné volania funkcií v reťazci - potrebujeme hostiteľský jazyk so silným typovým systémom, ktorý nám umožní garantovať, že len metóda, ktorá je v danom mieste reťazca povinná, vráti návratovú hodnotu požadovaného typu.

Tento návrhový vzor sa používa napríklad pre implementáciu kontajnerov s meniacim sa počtom prvkov (ekvivalent `std::vector` v C++).

5.3 Vnorené funkcie

Tento prístup využíva na štrukturovanie kódu DSEL volania funkcií - argumentom funkcie je volanie nejakej inej funkcie. Hierarchia vnorených funkcií (nested functions) odráža štruktúru syntaktického stromu DSL.

Ide o návrhový vzor, ktorý je možné prakticky bezo zmeny previesť do funkcionálneho sveta. Tam sa však častejšie využívajú iné prístupy.

Výhodami tejto techniky sú jednoduchosť implementácie hierarchických štruktúr a fakt, že nepotrebuje reprezentovať stav spracovávaného DSEL výrazu alebo skriptu. Trpí však aj niekoľkými problémami.

Argumenty funkcií sú identifikované pozíciou, nie menom, čo môže zhoršiť prehľadnosť kódu, ktorý implementuje podporu pre takýto DSEL. Tento problém sa dá obísť niekoľkými spôsobmi, od použitia hostiteľského jazyka s pomenovanými parametrami cez wrappery až po vhodné pomenovanie návratových hodnôt z vnorených funkcií.

Je zložité zariadiť, aby jazyk podporoval voliteľné argumenty. V hostiteľskom jazyku musíme pri spracovávaní takýchto funkcií buď použiť defaultné hodnoty argumentov, alebo používať pomocné funkcie. Prípadne môžeme definovať samostatnú funkciu pre každú povolenú kombináciu argumentov, ak možností nie je príliš veľa.

Štruktúra DSEL využívajúceho vnorené funkcie môže mať problémy so zrozumiteľnosťou - pri vyhodnocovaní sa postupuje od najvnútornejšej funkcie k najvonkajšej, čo môže byť v niektorých prípadoch kontraintuitívne.

V niektorých hostiteľských jazykoch môže byť nevýhodou syntaktický šum - všetky indikátory volania funkcií a oddeľovače argumentov pre hostiteľský jazyk musia byť na svojich miestach, čo môže DSEL skomplikovať.

Posledným faktorom, ktorý môže byť v niektorých prípadoch nevýhodou, je nemožnosť využiť lazy evaluation. Tento návrhový vzor si vyžaduje, aby boli argumenty funkcie vyhodnotené ešte predtým, ako sa zavolá funkcia samotná.

Technika vnorených funkcií sa často používa na implementáciu kontajnerov s pevným počtom prvkov (ekvivalent poľa v C++).

5.4 Zhrnutie

Dve zložky návrhu a implementácie DSEL - návrh samotných funkcií a integrácia do hostiteľského jazyka - sa v imperatívnych a funkcionálnych jazykoch riešia odlišne a prikladá sa im rozličná dôležitosť. Môžeme však skonštatovať, že je jednoduchšie implementovať DSEL vo funkcionálnych jazykoch než v imperatívnych, a to hlavne kvôli spôsobu, akým funkcionálne jazyky pracujú so stavom. Sú však situácie, kedy dáva zmysel implementovať DSEL v imperatívnych jazykoch, napríklad keď využívajú služby doménovo špecifickej knižnice, ktorá neposkytuje rozhranie použiteľné z funkcionálnych programovacích jazykov (aj keď vďaka FFI (Foreign Function Interface) je asi takáto situácia pomerne zriedkavá).

6. Prehľadávanie stavových priestorov

V umelej inteligencii často potrebujeme skúmať, aké rozhodnutie je z pohľadu racionálneho agenta v danej situácii najlepšie. Jedným z možných prístupov, ako hľadať odpoveď na túto otázku, je modelovať si rozhodovací problém ako stavový priestor a skúmať, aká postupnosť rozhodnutí nás dovedie do želaného stavu.

Model úlohy pomocou stavového priestoru pozostáva z:

- počiatočného stavu
- funkcie, ktorá pre každý stav vráti množinu možných nasledujúcich stavov (nasledovníkov)
- testu cieľa (napr. množiny úspešných koncových stavov)
- ceny cesty (cesta je postupnosť po sebe nasledujúcich stavov)

Všetky možné postupnosti po sebe nasledujúcich stavov, ktoré začínajú v počiatočnom stave, tvoria prehľadávací strom (pokiaľ sa stavy neopakujú) alebo prehľadávací graf (ak sa stavy môžu opakovať). Vyriešiť problém prehľadávania stavového priestoru znamená nájsť (akúkoľvek) cestu z počiatočného do ľubovoľného z cieľových stavov. Často ešte navyše požadujeme, aby toto riešenie bolo optimálne - zo všetkých ciest, ktoré sú riešeniami, hľadáme tú s najmenšou cenou.

Z tohto popisu problému už dokážeme odvodiť základnú kostru algoritmu pre hľadanie riešenia:

```
1. začneme v počiatočnom stave
2. otestujeme, či aktuálný stav nie je cieľový
3. ak nie, uskutočníme expanziu stavu
(vygenerujeme preň množinu nasledujúcich stavov)
4. pomocou prehľadávacej stratégie vyberieme ďalší stav,
s ktorým budeme pokračovať krokom 2
```

Táto základná kostra algoritmu zostáva rovnaká pre stromové aj pre grafové prehľadávanie. Pokiaľ si to štruktúra stavového priestoru vyžaduje, môžeme algoritmus obohatiť o kontrolou cyklov tým, že budeme skúmať, či sa vybraný ďalší stav nevyskytuje na aktuálnej ceste. Pri grafovom prehľadávaní sa navyše musíme vysporiadať s faktom, že cesty v prehľadávanom stavovom priestore môžu byť redundatné (medzi dvomi stavmi existuje viacero možných ciest). Tento problém väčšinou riešime zapamätaním si už expandovaných uzlov.

Prvá dátová štruktúra, ktorú budeme potrebovať pri implementácii algoritmov prehľadávajúcich stavové priestory, je teda reprezentácia uzla prehľadávacieho stromu alebo grafu. Ďalej potrebujeme reprezentovať množinu listov, ktoré môžeme expandovať - na to sa vo väčšine prípadov používa fronta. Pri grafovom prehľadávaní potrebujeme navyše efektívne zistiť, či sme daný uzol už expandovali. Na to sa väčšinou používa hashovacia tabuľka.

Po tomto všeobecnom úvode budeme pokračovať popisom rozličných konkrétnych algoritmov, ktoré sa používajú na riešenie problému prehľadávania stavového priestoru. Hlavným cieľom tejto kapitoly je pomôcť čitateľovi vytvoriť si myšlienkový model týchto algoritmov, nie vysvetľovať ich implementáciu či už vo funkcionálnych, alebo v imperatívnych jazykoch. Preto v situáciach, kedy sa rozhodneme pri popise algoritmu použiť pseudokód, budeme používať konštrukcie známe z imperatívnych jazykov.

Náš výklad bude založený na učebnici „Artificial Intelligence: A Modern Approach“ (Russell a Norvig, 2009), budeme sa však usilovať prepojiť tieto algoritmy s dostupnou literatúrou, ktorá skúma použitie doménovo-špecifických jazykov v umelej inteligencii.

Skúmané prehľadávacie algoritmy budeme medzi sebou porovnávať po stránke úplnosti (ak existuje riešenie, nájde ho?), optimality (nájde optimálne riešenie?) a časovej a pamäťovej zložitosti (väčšinou vyjadrenej v závislosti od maximálneho počtu nasledovníkov (faktor vetvenia) b, hĺbky najmenej zanoreného cieľového uzla d a dĺžky maximálnej cesty m).

Algoritmy pre prehľadávanie do šírky a prehľadávanie s výberom najlepšieho budeme integrovať do jazyka Haggle, principiálne je však možné doňho bez akýchkoľvek zmien v návrhu integrovať všetky popísané algoritmy.

6.1 Neinformované (slepé) prehľadávanie

6.1.1 Prehľadávanie do hĺbky

Základným princípom prehľadávania do hĺbky (depth-first search; DFS) je vždy expandovať najhlbší uzol v hranici (množine listov, ktoré môžeme expandovať), a pokiaľ je tento uzol koncový, vrátiť sa k najhlbšiemu uzlu, ktorý je ešte možné expandovať. To zodpovedá ukladaniu vygenerovaných uzlov do dátovej štruktúry zásobník.

Pri prehľadávaní grafov do hĺbky môžeme donekonečna expandovať nekonečnú cestu, ktorá nevedie k cieľu - tento algoritmus je teda úplný len pre konečné stavové priestory. Pokiaľ použijeme stromové prehľadávanie a teda nebudeme detekovať redundantné cesty, algoritmus bude neúplný.

V oboch prípadoch prehľadávanie do hĺbky vracia prvý nájdený cieľ, čo znamená, že nie je optimálne.

Časová zložitosť stromového prehľadávania do hĺbky je $O(b^m)$, pamäťová zložitosť $O(bm)$. Pokiaľ však použijeme backtracking a budeme ďalšieho možného nasledovníka generovať až pri návrate do rodičovského uzla, dokážeme znížiť pamäťovú zložitosť na $O(m)$. (Pripomeňme si, že b značí faktor vetvenia a m dĺžku maximálnej cesty.)

Časová a pamäťová zložitosť grafového prehľadávania do hĺbky závisia na veľkosti prehľadávaného stavového priestoru.

Prakticky všetky doménovo špecifické jazyky, ktoré modelujú problémy programovania s obmedzujúcimi podmienkami, interne implementujú DFS. Ako príklady môžeme uviesť Oz (Smolka, 1996), Zinc (de la Banda a kol., 2006) a OPL (Van Hentenryck a kol., 2000).

Implementáciu DFS pre všeobecné prehľadávacie problémy využíva napríklad článok Search Combinators (Schrijvers a kol., 2013).

6.1.2 Prehľadávanie do šírky

Prehľadávanie do šírky (breadth-first search; BFS) expanduje všetky uzly na danej úrovni predtým, ako expanduje prvý uzol na ďalšej úrovni. Vždy teda expanduje uzol z hranice s najmenšou hĺbkou, čo zodpovedá ukladaniu vygenerovaných uzlov do dátovej štruktúry fronta.

Tento algoritmus je úplný pre stavové priestory s konečným faktorom vetvenia. Je optimálny, ak je cena cesty neklesajúcou funkciou hĺbky (čo v praktických problémoch často platí). Pokiaľ ceny hrán nie sú jednotkové, ide vlastne o variantu prehľadávania s výberom najlepšieho, ktoré je popísané v ďalšej časti kapitoly.

Jeho časová zložitosť je $O(b^{d+1})$, je však možné ju znížiť na $O(b^d)$, ak vykonávame test na cieľ už pri generovaní uzla. Pamäťová zložitosť je tiež $O(b^d)$, čo môže spôsobiť problémy u veľkých stavových priestorov.

Spomedzi doménovo špecifických jazykov pre programovanie s obmedzujúcimi podmienkami BFS implementujú napríklad Oz (Smolka, 1996) a OPL (Van Hentenryck a kol., 2000).

6.1.3 Prehľadávanie s obmedzenou hĺbkou

Pri prehľadávaní do hĺbky môžeme zamedziť problémom s nekonečnými cestami, ak budeme prehľadávať len do pevnej hĺbky l a uzly v hĺbke l budeme považovať za uzly bez nasledovníkov. Na rozdiel od predchádzajúcich algoritmov, ktoré keď skončia, nás vždy informujú o existencii alebo neexistencii riešenia, tento algoritmus môže vrátiť „riešenie možno existuje vo väčšej hĺbke než l“.

Prehľadávanie s obmedzenou hĺbkou (depth-limited search; DLS) si popíšeme prostredníctvom pseudokódu:

```
FUNCTION DEPTH-LIMITED-SEARCH (problem, limit)
RETURNS solution/FAIL/CUTOFF
============================
RETURN RECURSIVE-DLS (MAKE-NODE (problem, INIT=STATE),
                      problem,
                      limit)

FUNCTION RECURSIVE-DLS (node, problem, limit)
RETURNS solution/FAIL/CUTOFF
============================
IF (problem.GOAL-TEST (node.STATE)) THEN
    RETURN solution (node)
ELSE
    IF (limit == 0)
        RETURN CUTOFF
    ELSE
        cutoff_occured? = FALSE
        FOREACH action IN problem.ACTIONS (node.STATE) DO
            child = CHILD-NODE (problem, node, action)
            result = RECURSIVE-DLS (child, problem, limit-1)
            IF (result == CUTOFF) THEN
                cutoff_occured? = TRUE
            ELSE
                IF (result != FAIL)
```

```
                RETURN result
        DONE
    IF cutoff_occured? THEN
        RETURN CUTOFF
    ELSE
        RETURN FAIL
```

Prehľadávanie s obmedzenou hĺbkou je neúplné (pre $l < d$) a neoptimálne; má časovú zložitosť $O(b^l)$ a pamäťovú zložitosť $O(bl)$. Najvhodnejšia voľba l závisí na konkrétnom stavovom priestore.

Medzi existujúcimi modelovacími jazykmi pre problémy programovania s obmedzujúcimi podmienkami nájdeme implementáciu prehľadávania s obmedzenou hĺbkou napríklad v ECLiPSe [1], ktoré však leží už na hranici medzi doménovo špecifickými jazykmi a všeobecne zameranými programovacími jazykmi s podporou logického programovania.

6.1.4 Iteratívne prehlbovanie

Problémom pri prehľadávaní so obmedzenou hĺbkou je, že nevieme, akú hodnotu l máme zvoliť. Príliš nízke l spôsobí, že prehľadávací algoritmus vráti CUTOFF a my sa nič užitočné nedozvieme. Príliš vysoká hodnota l zase zvyšuje časovú a pamäťovú zložitosť algoritmu, aj keď už vo väčšej hĺbke žiadne lepšie riešenie neexistuje.

Dáva teda zmysel položiť si otázku: ako veľmi by algoritmus spomalilo, keby sme opakovane spúšťali prehľadávanie s obmedzenou hĺbkou so stále sa zvyšujúcou hodnotou l, až kým nenájdeme riešenie?

Odpoveďou je, že k žiadnemu významnému spomaleniu algoritmu nedôjde. Prieskum l-tej hladiny prehľadávacieho stromu s faktorom vetvenia b má časovú zložitost $O(b^l)$. Časová zložitosť preskúmania prvých $l-1$ hladín v takomto prehľadávacom strome je však tiež $O(b^l)$.

Pretože d je hĺbka najmenej zanoreného cieľového uzla a aj ten najlepší neinformovaný algoritmus musí v najhoršom prípade preskúmať minimálne d-tú hladinu, najlepšia dosiahnuteľná časová zložitosť je $O(b^d)$.

Z predchádzajúceho odstavca však plynie, že prieskum prvých $d-1$ hladín má tiež časovú zložitosť $O(b^d)$. Ak budeme teda pri opakovanom spúšťaní stále prehľadávať už preskúmané hladiny, asymptotickú časovú zložitosť algoritmu to nijako neovplyvní.

Celková časová zložitosť iteratívneho prehlbovania (iterative deepening search; IDS) je teda $O(b^d)$.

Aj tento algoritmus si popíšeme pomocou pseudokódu:

```
FUNCTION ITERATIVE-DEEPENING-SEARCH (problem)
RETURNS solution/FAIL
=====================
FOR depth = 0 to INF DO
    result = DEPTH-LIMITED-SEARCH (problem, depth)
    IF result != CUTOFF THEN
        RETURN result
```

[1] http://eclipseclp.org/

Tento algoritmus je navyše pre stavové priestory s konečným vetvením úplný. Jeho podmienka optimality je ekvivalentná tej pre prehľadávanie do šírky. Jeho pamäťová zložitosť je tiež pomerne nízka - len $O(bd)$. Ide preto o preferovaný spôsob, akým prehľadávať veľké stavové priestory s neobmedzenou hĺbkou riešení, pokiaľ nemáme dodatočné informácie.

Nejedná sa o populárny algoritmus pre riešenia problémov programovania s obmedzujúcimi podmienkami, pretože máme k dispozícii výkonnejšie alternatívy. Je však implementovaný v doménovo špecifickom jazyku zameranom na všeobecné prehľadávanie, ktorý je popísaný v Search Combinators (Schrijvers a kol., 2013).

6.1.5 Obojsmerné prehľadávanie

Algoritmus kombinuje prehľadávanie od počiatku s prehľadávaním od cieľa, kým sa niekde nestretnú. Táto stratégia vedie k omnoho nižšej časovej a pamäťovej zložitosti: $b^{d/2} + b^{d/2} << b^d$.

Test na cieľ musíme nahradiť kontrolou, či majú hranice oboch prehľadávaní neprázdny prienik. Prvé takéto riešenie však nemusí byť optimálne a musíme vykonať dodatočné kontroly. Pokiaľ ich vykonáme a obe prehľadávania používajú BFS, výsledný algoritmus je úplný a optimálny a jeho časová a pamäťová zložitosť sú $O(b^{d/2})$.

Prehľadávanie od cieľa však môže byť komplikované, pretože potrebujeme určovať predchodcov. Problém môže nastať aj pri určovaní, čo vlastne je množina cieľových stavov, pokiaľ nie je zadaná explicitne, ale napríklad testom na nejakú vlastnosť.

6.2 Informované prehľadávanie a heuristiky

Do kategórie informovaného prehľadávania zaraďujeme algoritmy stromového alebo grafového prehľadávania, kde vždy expandujeme uzol s najnižšou hodnotou evaluačnej funkcie $f(n)$. Hodnota evaluačnej funkcie je odhad dĺžky najlacnejšej cesty z počiatočného uzla do cieľa za podmienky, že táto cesta prechádza uzlom n.

Jednou zo zložiek evaluačnej funkcie je heuristická funkcia $h(n)$, ktorá odhaduje cenu najlacnejšej cesty zo stavu v uzle n do cieľového stavu. Platí:

- $\forall n : h(n) \geq 0$
- $h(n) = 0 \Leftrightarrow$ n je cieľový uzol

6.2.1 Prehľadávanie s výberom najlepšieho

Prehľadávanie s výberom najlepšieho (best-first search, BeFS) vždy expanduje uzol, ktorý je podľa heuristickej funkcie najbližšie ku cieľu. Jedinou komponentou evaluačnej funkcie je teda heuristická funkcia.

Podmienky pre optimálnosť a úplnosť tohto algoritmu sú rovnaké ako u DFS. Jeho časová i pamäťová zložitosť sú $O(b^m)$; použitie dobrej heuristiky však obe dokáže výrazne redukovať.

Spomedzi doménovo špecifických jazykov pre programovanie s obmedzujúcimi podmienkami BeFS implementujú napríklad Oz (Smolka, 1996) a OPL (Van Hentenryck a kol., 2000).

6.2.2 Prehľadávanie s obmedzeným počtom diskrepancií

Algoritmus prehľadávania s výberom najlepšieho sa pri hľadaní riešenia bezvýhradne spolieha na heuristiku. Heuristiky sa však môžu mýliť, a v mnohých prehľadávacích stromoch nájdeme riešenie rýchlejšie, ak v niektorých krokoch expandujeme iný uzol, než nám odporúča heuristika. Túto expanziu neodporúčaného uzlu budeme nazývať porušenie heuristiky alebo diskrepancia.

Motiváciou pre tento algoritmus je pozorovanie, že počet porušení heuristiky na ceste z počiatočného do cieľového stavu je malý. Vo všetkých ostatných uzloch dáva zmysel sa podľa nej rozhodnúť.

Vo všeobecnosti teda nájdeme riešenie skôr, keď sa pokúsime uzly s porušením heuristiky nájsť, než keď sa budeme bezvýhradne spoliehať na heuristiku a používať (hoci informované) prehľadávanie s návratom, keď nás heuristika k riešeniu nedovedie. Dáva preto zmysel relaxovať podmienku pre výber najlepšieho a dovoliť určitý pevný počet diskrepancií. (Harvey a Ginsberg, 1995)

Algoritmus pre prehľadávanie s obmedzeným počtom diskrepancií (limited discrepancy search; LDS) je implemenáciou týchto myšlienok - môžeme sa naň pozerať ako na prehľadávanie s „metaheuristikou“, ktorá hovorí, kedy a akým spôsobom býva na úspešných cestách porušená heuristika pre riešený problém.

V prípade, keď so zadaným povoleným počtom diskrepancií riešenie nenájdeme, zvýšime povolený počet diskrepancií o 1 a prehľadávanie reštartujeme. Ide o podobný prístup ako u iteratívneho prehlbovania, len namiesto hĺbky obmedzujeme počet diskrepancií.

Tento algoritmus implementujú doménovo špecifické jazyky pre programovanie s obmedzujúcimi podmienkami Oz (Smolka, 1996) a OPL (Van Hentenryck a kol., 2000).

Implementáciu LDS pre všeobecné prehľadávacie problémy využíva napríklad článok Search Combinators (Schrijvers a kol., 2013).

6.2.3 Vlastnosti heuristík

V tejto sekcii si definujeme niekoľko vlastností heuristík, ktoré budeme potrebovať pri rozbore ďalších algoritmov pre informované prehľadávanie.

Definícia 3. *Heuristika $h(n)$ sa nazýva* `prípustná` *práve vtedy, keď $h(n)$ vždy podhodnocuje cenu cesty z n do cieľového uzla.*

Definícia 4. *Buď n uzol prehľadávacieho stromu, n' ľubovoľný jeho nasledovník a $c(n,n')$ cena za prechod z n do n'. Heuristika $h(n)$ sa nazýva* `monotónna` *alebo* `konzistená` *práve vtedy, keď $\forall n \forall n' : h(n) \leq c(n,n') + h(n')$.*

Tvrdenie 23. *Každá monotónna heuristika je prípustná.*

Dôkaz. Nech je $n_1, \ldots, n_k$ optimálna cesta z n_1 do cieľového n_k.

Potom pre každú dvojicu susediacich uzlov (ktoré majú indexy i a i+1) z monotónnosti platí, že

$h(n_i) - h(n_{i+1}) \leq c(n_i,n_{i+1})$.
Z toho plynie, že
$h(n_1) \leq \sum_{i=1}^{k-1} c(n_i,n_{i+1})$.

□

6.2.4 A*

Algoritmus A* prehľadáva stavový priestor tak, že vždy expanduje uzol s najmenšou hodnotou evaluačnej funkcie

$$f(n) = g(n) + h(n)$$

kde $g(n)$ je cena cesty z koreňa do uzla n a $h(n)$ je heuristický odhad ceny najlacnejšej cesty z n do cieľa.

Ak je C* cena cesty k optimálnemu riešeniu a existuje len konečný počet uzlov s cenou $\leq$ C*, algoritmus je úplný.

Stromový A* je optimálny, ak používa prípustnú heuristickú funkciu $h(n)$.

Grafový A* je optimálny, ak používa monotónnu heuristickú funkciu $h(n)$.

Tento algoritmus je pre $h(n)$ monotónnu tiež optimálne efektívny - ak nepočítame uzly, pre ktore $f(n) =$ C*, žiadny optimálny algoritmus nemôže expandovať menej uzlov ako A*.

Jeho časová zložitosť závisí na presnosti heuristiky; pri nevhodnej heuristike môže dôjsť k expanzii exponenciálneho počtu uzlov.

Pamäťová zložitosť algoritmu A* je exponenciálna - pamätá si všetky vygenerované uzly.

6.2.5 IDA*

Algoritmus BFS mal dobré vlastnosti po stránke úplnosti a optimálnosti, ale príliš veľké pamäťové nároky. Pokúsili sme sa preto sformulovať algoritmus, ktorý by bol úplný a optimálny v situáciach, kedy je BFS úplný a optimálny, ale mal by nižšiu pamäťovú zložitosť. Tak sme dostali algoritmus iterätívneho prehlbovania.

S algoritmom A* sme teraz v podobnej situácii - máme rozumné podmienky na úplnosť a optimálnosť, ale príliš veľkú pamäťovú zložitosť. Ak sa pokúsime na A* aplikovať stratégiu opakovaných reštartov, dostaneme A* s iteratívnym prehlbovaním (iterative deepening A*; IDA*).

Namiesto obmedzovania hĺbky budeme obmedzovať hodnotu evaluačnej funkcie $f(n)$, a namiesto zvyšovania o 1 použijeme pri reštarte najmenšie $f(n)$, ktoré v predchádzajúcej iterácii limit prekročilo. (Tento prístup sa používa pre celočíselné $f(n)$. Pokiaľ evaluačná funkcia môže nadobúdať racionálne alebo reálne hodnoty, prírastky môžu byť príliš malé.)

Algoritmus si podrobnejšie popíšeme pomocou pseudokódu:

```
FUNCTION IDA* (problem)
RETURNS solution-sequence/FAIL
============================
STATIC f-limit, root
root = MAKE-NODE (INIT-STATE (problem))
f-limit = f-COST (root)
```

```
WHILE TRUE DO
    (solution, f-limit) = DFS-CONTOUR (root, f-limit)
    IF solution != NULL THEN
        RETURN solution
    IF f-limit == INF THEN
        RETURN FAIL
DONE

FUNCTION DFS-CONTOUR (node, f-limit)
RETURNS solution-sequence, new-f-limit
=======================================
STATIC next-f        //f-COST limit pre ďalšiu vrstevnicu
IF f-COST (node) > f-imit THEN
   RETURN (NULL, f-COST (node))
IF GOAL-TEST (problem.STATE (node)) THEN
    RETURN (node,f-limit)
FOREACH node s IN SUCCESSORS(node) DO
    (solution, new-f) = DFS-CONTOUR (s,f-limit)
    IF solution != NULL THEN
        RETURN (solution, f-limit)
    next-f = MIN (next-f, new-f)
DONE
RETURN (NULL, next-f)
```

6.2.6 SMA*

Algoritmus A* využíva príliš veľa pamäte, jeho varianta IDA* by zase mohla rýchlejšie nájsť lepšie riešenie, keby využila viac pamäte. Zjednodušený pamäťovo obmedzený A* (simplified memory-bounded A*; SMA*) je varianta algoritmu A*, ktorá využíva všetku dostupnú pamäť - keď dôjde k vyčerpaniu pamäte, vyhodíme list s najväčšou hodnotou $f(n)$ a cenu vyhodeného listu si zapamätáme u rodiča. Tento list a jeho podstrom znovu vygenerujeme, ak zistíme, že všetky ostatné cesty sú horšie.

Aj tento algoritmus si podrobnejšie popíšeme pomocou pseudokódu:

```
FUNCTION SMA* (problem)
RETURNS solution-sequence/FAIL
==============================
STATIC queue //fronta uzlov utriedená podľa hodnoty evaluačnej funkcie
queue = MAKE-QUEUE (MAKE-NODE (INIT-STATE (problem)))
WHILE TRUE DO
    IF queue.EMPTY THEN
        RETURN FAIL
    n = najhlbší uzol zo všetkých uzlov
        s najmenšou hodnotou evaluačnej funkcie v queue
    IF GOAL-TEST(n) THEN
        RETURN n
    s = NEXT-SUCCESSOR(n)
    IF (s nie je cieľ a jeho pridanie by vyčerpalo dostupnú pamäť) THEN
        f(s) = INF
    ELSE
```

```
        f(s) = MAX(f(n), g(s)  + h(s))
    IF (všetci nasledovníci n sú vygenerovaní) THEN
        aktualizuj hodnoty evaluačnej funkcie f u n a jeho predchodcov
    IF SUCCESSORS(n) všetci v pamäti THEN
        odstráň n z queue
    IF pamäť je plná THEN
        zmaž najplytší uzol s najväčšou hodnotou f z queue
        odstráň ho zo zoznamu následníkov rodiča
        vlož jeho rodičov do queue, ak treba
    vlož s do queue // môže vyvolať ďalšie mazanie
DONE
```

Tento algoritmus je úplný, ak existuje dosiahnuteľmé riešenie. Je aj optimálny, pokiaľ je optimálne riešenie dosiahnuteľné - inak vráti najlepšie dosiahnuteľné riešenie.

Ak musíme často prepínať medzi mnohými cestami, ktoré sa všetky nevojdú do pamäte, problém sa môže stať po stránke časovej zložitosti pomocou algoritmu SMA* neriešiteľným, aj keď A* s neobmedzenou pamäťou by ho dokázal vyriešiť.

7. Implementačné detaily jazyka Haggle

7.1 Popis návrhu

Prvým krokom návrhu je popísať, čo je naším cieľom - chceme navhrhnúť DSEL v jazyku Haskell, ktorý nám umožní jednoducho modelovať prehľadávacie algoritmy vzniknuté zložením základných prehľadávacích algoritmov (napríklad tých popísaných v kapitole 6). Tieto modely majú byť dostatočné abstraktné na to, aby sa bez vačších zmien dali použiť na širokú škálu prehľadávacích problémov. Chceme podporovať heuristické prehľadávanie. A našou poslednou požiadavkou je, aby bol systém modulárny a jednoducho rozšíriteľný - pridanie nového prehľadavácieho algoritmu by malo spočívať len v jeho implementácii, bez nutnosti zásahu do ostatných častí DSEL.

Druhým krokom návrhu je výber štruktúr, ktoré budú reprezentovať základné zložky navrhovaného jazyka: prehľadávacie algoritmy a spôsob ich spájania. Usúdili sme, že táto abstrakcia funkcií s vedľajšími efektami a ich vzájomnej komunikácie je dosť všeobecná na to, aby pre ňu štandartné knižnice jazyka Haskell poskytovali nejakú podporu, a nemá preto zmysel prichádzať s vlastným modelom - a nemýlili sme sa. Pre podporu tejto najvyššsej úrovne abstrakcie sme našli a zvážili typové triedy, ktoré popíšeme v nasledujúcich odstavcoch.

`Applicative` je rozšírenie typovej triedy `Functor`. Zatiaľ čo `Functor` umožňuje aplikovať čistú funkciu na kontajner, `Applicative` umožňuje aplikovať funkciu v kontajneri na kontajner toho istého typu. Takáto aplikácia je však asociatívna, a spájanie prehľadavacích algoritmov asociatívne nie je, čo v nás vzbudilo obavy, že `Applicative` nemá dostatočnú vyjadrovaciu silu. Navyše Haskell neposkytuje pre `Applicative` ekvivalent monadickej do-notácie, čo by sťažilo návrh jednoduchého a prehľadného DSEL. Posledným faktom, ktorý sme zvážili, bolo, že každá monáda je `Applicative`, a monády sú celkovo v jazyku lepšie podporované - majú do-notáciu, ich skladanie je jednoduché vďaka monadickým transformátorom,...

To isté platí aj o typovej triede `Alternative`, ktorú je možné považovať za rozšírenie `Applicative` pre „stavové“ inštancie, u ktorých je možné, že opakovaná aplikácia tej istej funkcie niekoľkokrát uspeje a potom zlyhá. (Kanonickým príkladom sú parsery.)

Veľmi sľubne vyzerala typová trieda `MonadPlus`, o ktorej by sa dalo povedať, že reprezentuje monadické akcie, ktoré je možné „spájať“ - nie v zmysle monadického bindu, ale v zmysle spájania zoznamov. Táto typová trieda sa často používa v dostupnej literatúre pre riešenie jedného konkrétneho problému pomocou prehľadávania, a používa ju aj Piponi v (Piponi, 2007). Nakoniec sme ju odložili bokom s tým, že ju použijeme, ak nám `Monad` nebude stačiť, alebo budeme vidieť, že by signifikantne zjednodušila kód. Táto situácia však nenastala.

Ďalšou zvažovanou možnosťou bola sada typových tried `Arrow`/`ArrowPlus`/`ArrowChoice`. Ide o zovšeobecnenie `Monad` pre funkcie, ktoré nielenže majú vedľajšie efekty, ale aj prijímajú ako vstup funkcie, ktoré majú vedľajšie efekty. Prišli sme však k záveru, že v porovnaní s typovou triedou `Monad` neposkytujú nič, čo

by pre nás mohlo byť užitočné.

Trochu iný pohľad na problém nám poskytla typová trieda `Traversable`. Jej inštanciami sú kontajnery, ktoré je možné prejsť a spracovať tak dáta v nich uložené. Toto spracovávanie však môže mať vedľajšie efekty, ktoré sú vyjadrené inštanciou `Applicative`. Nakoniec sme však narazili na rovnaké problémy, ako u typovej triedy `Applicative`, a rozhodli sme sa tento prístup opustiť.

Nakoniec sme teda za základnú štruktúru, ktorá bude reprezentovať prehľadávací algoritmus, zvolili `Monad`. Viedol nás k tomu hlavne fakt, že ide o zastrešujúci koncept pre mnoho užitočných abstrakcií, ktoré takto budeme môcť využívať, jednoduché vytváranie zložitejších monád pomocou monadických transformátorov a široká používanosť v literatúre, ktorá rieši podobné problémy.

Zostávalo ešte zvoliť spôsob, akým budú prehľadávacie algoritmy medzi sebou komunikovať. Implementujeme všetky v jednej a tej istej monáde, čo nám umožní využiť na komunikáciu monadický bind, alebo ponecháme každému prehľadávaciemu algoritmu jeho vlastnú monádu a komunikáciu budeme riešiť iným spôsobom? Spočiatku sme uvažovali o druhej možnosti a pokúsili sme sa navrhnúť komunikáciu medzi rozličnými monádami pomocou typovej triedy `Category`, ktorá zovšeobecňuje pojem skladania funkcií, ktorý sme si popísali v kapitole 2.4.4 , okrem iného aj na monadické funkcie. Nakoniec sme však prišli k záveru, že za opustenie možnosti používať do-notáciu to nestojí, a keď sú všetky skladané funkcie prehľadávacie algoritmy, mali by mať dosť spoločného na to, aby dokázali bežať v spoločnej monáde. Finálna implementácia teda používa prvý prístup.

Keď sme sa rozhodli, že každý implementovaný prehľadávací algoritmus budeme reprezentovať ako funkciu v spoločnej monáde, nasledoval tretí krok návrhu: ako bude táto monáda vyzerať?

Jednou z prvých myšlienok bolo použiť kontinuácie. Prieskum literatúry nás však presvedčil, že keď sú kontinuácie dosť silné na implementáciu plnohodnotných funkcionálnych jazykov ((Friedman a Wand, 2008) a (Appel, 2007)), použiť ich pri implementácii DSEL s relatívne jednoduchou syntaxou a sémantikou by viedlo k zbytočnému skomplikovaniu. V tomto názore nás na záver utvrdil fakt, že implementácia bola možná aj pomocou jednoduchších techník.

Že implementácia bola možná aj pomocou jednoduchších techník, platí aj o monáde `Amb`, ktorú sme popísali na konci tretej kapitoly. Rozhodne by však bolo zaujímave implementovať DSEL pre prehľadávanie stavových priestorov pomocou tejto monády a porovnať implementáciu s tou popisovanou v tejto práci.

Za túto všeobecnú monádu sme teda nakoniec zvolili zoznam ciest vložený do stavovej monády. Toto návrhové rozhodnutie už však bolo ovplyvnené aj štvrtým krokom návrhu: implementáciou samotných prehľadávacích algoritmov.

Podpora heuristických algoritmov si vyžaduje, aby sme si pre každý objavený uzol boli schopní uložiť hodnotu jeho evaluačnej funkcie. Nutnosť integrovať tieto algoritmy s algoritmami pre neinformované prehľadávanie zasa naznačuje, že by možno stálo za zváženie prísť aj pre neinformované algoritmy s funkciami, ktoré budú popisovať želané poradie prehľadávania a budú teda zastávať miesto „evaluačných funkcií“. Tieto „evaluačné funkcie“ však nezávisia len od objaveného uzla, ale od celej cesty - príkladom je dĺžka cesty u BFS. Tento fakt bol našou hlavnou motiváciou, prečo objavené cesty reprezentovať explicitne cestami, a nie implicitne prehľadávacím stromom.

Náš návrh vlastne vôbec neobsahuje explicitnú reprezentáciu prehľadávacieho

stromu - ten reprezentujeme pomocou funkcie generujúcej nasledovníkov. Tento prístup využíva lazy evaluation - pri objavení vrchola je zavolaná funkcia generujúca všetkých nasledovníkov, ku ktorých vyhodnoteniu dôjde až v prípade, keď ich budeme potrebovať.

Aby sme sa vrátili k cestám: cesty sú uložené v zozname a utriedené podľa ohodnotení. Pretože jediné operácie, ktoré potrebujeme, sú získanie cesty s najnižším ohodnotením a zlievanie dvoch utriedených zoznamov ciest, nemá zmysel používať pole, alebo dokonca lenses. Bolo by zmysluplné použiť (napríklad leftist) haldu ako v Piponi (2009b), ale dostupné balíčky sú pre verziu GHC 8, čo znemožnuje ich použitie.

Samotné cesty sú reprezentované dvojicami, kde prvý prvok je zoznam uzlov a druhý prvok je ohodnotenie. Ohodnotenia ciest sú kladné čísla (podobne ako v (Piponi, 2009b)), čo umožňuje jednoduché porovnávanie ohodnotení z rozličných prehľadávacích algoritmov. Pretože jediné požadované operácie sú pridanie na začiatok a aktualizácia ohodnotenia, táto reprezentácia plne postačuje.

Spôsob, akým reprezentujeme uložený stav, zodpovedá teda „kvázimonáde“ zoznamov zoznamov, ktorú sme popisovali v druhej kapitole. (Strate asociativity by sa dalo predísť, keby sme použili monádu `Amb` spomínanú na konci tretej kapitoly. Poznáme však mnoho príkladov „kvázimonád“, ktoré sú v programátorskej praxi užitočné aj napriek strate asociativity (Jones a Duponcheel, 1993), preto sme zvolili jednoduchšiu implementáciu.)

Uzly na cestách sú reprezentované typovou triedou `Expandable`, ktorú popíšeme v ďalšej sekcii.

Skončili sme teda s návrhom, v ktorom sú prehľadávacie algoritmy a reprezentácie riešených problémov navzájom ortogonálne. Implementované prehľadávacie algoritmy by mali fungovať na ľubovoľný problém, ktorého dátová reprezentácia implementuje funkcie z typovej triedy `Expandable`. A naopak, každý nový pridaný prehľadávací algoritmus by mal byť samostatný modul, ktorý sa do zvyšku systému zaintegruje tým, že bude bežať v monáde `State (Backlog a) [[a]]` - žiadne ďalšie zmeny nebudú potrebné.

7.2 Podrobný popis finálnej implementácie

V tejto kapitole si popíšeme implementáciu jazyka Haggle - v prvej časti jadro jazyka, v druhej prehľadávacie algoritmy a v tretej spôsob, akým sme implementáciu testovali.

7.2.1 Jadro jazyka Haggle

Modul Haggle

Tento modul implementuje základnú funkcionalitu jazyka Haggle. Ak sa vrátime naspäť k popisu syntaxe tohto jazyka v kapitole 4.2, uvidíme, že niektoré konštrukcie sú prevzané z Haskellu, ako napríklad import modulov alebo donotácia, ale o niektoré sme jazyk Haskell rozšírili.

Tieto rozširujúce funkcie sú implementované práve v module Haggle. Medzi najdôležitejšie z nich patrí funkcia `with`, ktorá počiatočný uzol prehľadávacieho

priestoru konvertuje do potrebného formátu a uloží do monády, v ktorej sú spúšťané prehľadávacie funkcie.

V tomto module sú implementované aj selektory výsledkov. Základným princípom je, že postupnosť volaní prehľadávacích algoritmov má typ monadického výpočtu v monáde `State`. Každý selektor výsledkov volá na tento monadický výpočet `runState` a potom funkciu, ktorá (pretože z jazyka Haskell dedíme lazy evaluation) vynúti vyhodnocovanie prehľadávacieho stromu, až kým nedostane požadovaný počet výsledkov. Haggle poskytuje nasledovné selektory:

- selektor `getOneSolution` požaduje vyhodnotenie jedného riešenia
- selektor `getManySolutions` požaduje vyhodnotenie zadaného počtu riešení
- selektor `getAllSolutions` požaduje vyhodnotenie všetkých riešení.

Modul Expandable

Modul `Expandable` implementuje definíciu typovej triedy `Expandable` spolu s niekoľkými pomocnými funkciami.

Typová trieda `Expandable` reprezentuje uzol prehľadávacieho stromu. Pretože počiatočný uzol je vlastne zadanie problému, ide aj o spôsob, akým sa v jazyku Haggle reprezentuje zadanie prehľadávacieho problému.

Jej návrh sme teda nutne museli založiť na určitých predpokladoch o prehľadávacích úlohách, ktorých riešenie budeme podporovať. Tieto predpoklady sú nasledovné:

- každý koncový uzol môžeme jednoznačne označiť za úspešný alebo za neúspešný
- úspešný vrchol sme schopní detekovať na základe dát, ktoré sú v ňom uložené - nepotrebujeme nijaký ďalší kontext
- pre každý uzol sme schopní vygenerovať zoznam jeho susedov
- lazy evaluation nám umožní od seba oddeliť generovanie susedov a samotné prehľadávanie bez straty efektivity
- „evaluačnú funkciu“ pre danú cestu chceme byť schopní spočítať aj na základe celej cesty alebo jej ľubovoľne dlhého úseku
- chceme zabrániť opakovanému prehľadávaniu vrcholov
- pokiaľ si to používaný algoritmus nevyžaduje, nechceme klásť nijaké podmienky na monotónnosť alebo prípustnosť používanej heuristiky (tieto pojmy sú definované v kapitole 6)

Za týchto predpokladov budeme potrebovať funkcie, ktoré:

- detekujú úspešný koncový vrchol
- detekujú neúspešný koncový vrchol

- spočítajú „evaluačnú funkciu“ cesty zo zadanej množiny vrcholov, ktoré sa na ceste vyskytujú
- vygenerujú zoznam susedov
- odstránia zo zoznamu susedov vrcholy, ktoré sme už prehľadali

Najskôr si zadefinujeme pomocný dátový typ:

```
data Result a = Fail | Success a | Sons [a]
```

Výsledná typová trieda má potom tvar:

```
class Eq a => Expandable a where
        -- predikát "detekuj úspešný koncový vrchol"
        stopSuccess :: a -> Bool
        -- predikát "detekuj neúspešný koncový vrchol"
        stopFail    :: [a] -> Bool
        -- vráť ohodnotenie cesty
        rank        :: (a -> Int) -> [a] -> Int
        -- vráť nasledujúce stavy k zadanému stavu
        generateNbs :: a -> [a]
        -- odstráň zo zoznamu potenciálnych kandidátov tých,
        -- ktorých sme už preskúmali
        prune       :: Result a -> [a] -> Result a
```

Modul Path

Modul Path obsahuje dátový typ reprezentujúci cestu v prehľadávacom strome a pomocné funkcie.

Vidíme, že implementácia jadra jazyka je pomerne jednoduchá. Rozhodnutie reprezentovať prehľadávacie algoritmy ako monadické výpočty a ich komunikáciu ako monadický bind sa naozaj vyplatilo.

Vo zvyšku kapitoly budeme pokračovať podrobným popisom prehľadávacích algoritmov, ktoré sme implementovali.

7.2.2 Implementované prehľadávacie algoritmy

Algoritmus BeFS

Funkcia implementujúca prehľadávací algoritmus BeFS má typovú signatúru:

```
befs :: (Expandable a) => (a -> Int) -> State (Backlog a) [[a]]
```

Parameter `(a -> Int)` je heuristická funkcia, ktorá sa má použiť ako parameter funkcie `rank` pri počítaní ohodnotenia cesty.

Samotná implementácia je priamočiara: z dátovej štruktúry `Backlog`, v ktorej sú uložené známe cesty, vyberieme cestu s najlacnejším ohodnotením. Toto je vždy prvá cesta, pretože `Backlog` udržujeme usporiadaný.

Z tejto cesty vezmeme posledný uzol, ktorý expandujeme. Výsledok expanzie je reprezentovaný dátovým typom `Result` popísaným v kapitole 7.2.1. Na ňom uskutočníme porovnávanie so vzorom:

- v prípade dátového konštruktoru `Fail` sa rekurzívne zavoláme na zvyšok `Backlogu` a vrátime výsledok tohto rekurzívneho volania
- v prípade dátového konštruktoru `Success` sa rekurzívne zavoláme na zvyšok `Backlogu` a vrátime výsledok tohto rekurzívneho volania, na ktorého začiatok pridáme aktuálnu cestu
- v prípade dátového konštruktoru `Sons` vygenerujeme všetky možné predĺženia súčasnej cesty, zlejeme ich s `Backlogom` a zavoláme sa rekurzívne na aktualizovaný `Backlog`.

Algoritmus BFS

Funkcia implementujúca prehľadávací algoritmus BFS má typovú signatúru:

```
bfs :: (Expandable a) => InfInt -> State (Backlog a) [[a]]
```

Parameter je typu `InfInt` - Int rozšírený o (kladné) nekonečno - ktorý sme si popísali na začiatku tretej kapitoly. Jeho hodnota reprezentuje hĺbku, v ktorej chceme zastaviť prehľadávanie.

Kvôli možnosti lepšie manipulovať s hĺbkou prehľadávania sme sa rozhodli implementovať variantu BFS, ktorá v jednom kroku expanduje všetky uzly s tou istou hĺbkou, namiesto varianty BFS, ktorá si ukladá uzly do fronty a expanduje v jednom kroku len jeden uzol.

Jedno volanie funkcie teda rozdelí `Backlog` na dve časti - všetky cesty s minimálnym ohodnotením a všetky ostatné cesty, a pracuje len s cestami s minimálnym ohodnotením. Pre tieto cesty vygenerujeme všetky ich možné predĺženia. Odstránime tie cesty, ktoré nebolo možné predĺžiť. Cesty, ktoré sa podarilo predĺžiť, rozdelíme na úspešné (ukončené cieľovým vrcholom) a ostatné. Ostatné cesty zlejeme s `Backlogom` a zavoláme sa rekurzívne na aktualizovaný `Backlog`, pričom o 1 zvýšime súčasnú hĺbku prehľadávania. Úspešné cesty spojíme s cestami, ktoré sme našli v rekurzívnom volaní.

7.2.3 Testy

Za konkrétny problém, ktorý budeme v testoch riešiť a na príklade ktorého nakoniec prezentujeme použitie jazyka Haggle, sme si zvolili hlavolam Lloydova pätnástka[1]. Tento problém sa navyše štandartne využíva na testovanie výkonnosti prehľadávacích algoritmov a heuristík, takže dáva zmysel, aby ho modelovací jazyk pre prehľadávanie stavových priestorov implementoval ako príklad, na ktorý sa môžu užívatelia jazyka odkazovať pri riešení realistickejších problémov.

Aby sme mohli definovať inštancie triedy `Expandable`, ktoré budeme potrebovať pre testovanie, musíme použiť rozšírenie jazyka Haskell `FlexibleInstances`.

Pre testovanie samotné sme použili haskellovský framework na unit testy s názvom HUnit[2]. Tento framework je inšpirovaný JUnit, čo je štandartný framework pre unit testovanie v jazyku Java. Zatiaľ podporuje len testovanie z príkazového riadka, ale to našim potrebám plne postačuje.

[1] http://mathworld.wolfram.com/15Puzzle.html
[2] http://hackage.haskell.org/package/HUnit

V HUnit je každý unit test funkcia typu `Test`. Testovanie sa vykonáva zavolaním testu z funkcie `main` v monáde poskytovanej testovacím frameworkom.

Aby sme implementovali skutočne unit testy, každý modul s prehľadávacou funkciou definuje pomocnú funkciu `test<prehľadávacia funkcia>`. Táto funkcia sa potom používa v unit testoch.

Okrem testov sa v zdrojovom kóde priloženom k práci nachádzajú aj príklady použitia jazyka Haggle. Obsah týchto súborov presne zodpovedá popisu na začiatku kapitoly 4.2.

Záver

Návrh DSEL vo funkcionálnom hostiteľskom jazyku je často používaný prístup ku problémom v niektorých oblastiach umelej inteligencie, napríklad v programovaní so obmedzujúcimi podmienkami. V tejto práci sme dokázali, že ide o zmysluplnú stratégiu. Zvlášť dobre si po tejto stránke vedie jazyk Haskell, a to hlavne kvôli spôsobu, akým sú v ňom implementované monády, a podpore do-notácie.

V práci sme poskytli prehľad techník, ktoré sa používajú pri návrhu a implementácii DSEL vo funkcionálnych programovacích jazykoch. Pre porovnanie sme popísali aj techniky využívané v imperatívnych programovacích jazykoch.

Podarilo sa nám v Haskelli navrhnúť prehľadný, intuitívny a ľahko rozšíriteľný DSEL pre všeobecné úlohy prehľadávania stavových priestorov. Jazyk Haggle podporuje používanie heuristických algoritmov a umožňuje jednoduché testovanie rozličných heuristík.

Na rozdiel od jazykov predstavených v článkoch Monadic Constraint Programming (Schrijvers a kol., 2009) a Search Combinators (Schrijvers a kol., 2013) sme opustili koncept prehľadávacích transformácii. To znamená, že ak by sme sme potrebovali implementovať napríklad BeFS s obmedzením na počet expandovaných uzlov, musíme namiesto pridania transformácie k už existujúcemu BeFS implementovať nový prehľadávací algoritmus.

Implemetácia jazyka Haggle je však o niekoľko rádov jednoduchšia. Nepotrebovali sme ani kontinuácie, ani monadické transformátory - úplne nám stačila `State` monáda. Celá implementácia jadra jazyka má asi 100 riadkov kódu, čo robí potenciálne možnými mieru testovania alebo časovej a pamäťovej optimalizácie, ktoré by u zložitejšieho jazyka možné neboli.

Úspešne sme nadviazali na prácu Piponiho - (Piponi, 2007),(Piponi, 2009a) a (Piponi, 2009b) - a dokázali sme, že sila do-notácie sa dá využiť aj pri implementácii zložitejších DSEL.

Určite však ostáva priestor aj pre ďalšie vylepšenia, či už implementáciu väčšieho počtu prehľadávacích algoritmov (najvýznamnejšie z ktorých sme taktiež popísali v texte práce), výkonnostné optimalizácie, alebo preskúmanie, ako by sa jazyk správal, keby sme pre reprezentáciu nedeterminizmu namiesto `List` monády využili monádu `Amb`.

Zoznam použitej literatúry

Appel, A. W. (2007). *Compiling with Continuations*. Cambridge University Press, New York, NY, USA. ISBN 052103311X.

Caithaml, T. (2009). Doménově specifické jazyky. *Diplomová práca (v češtine), vedúci RNDr.Jan Hric*. URL `https://is.cuni.cz/webapps/zzp/detail/65635`. Navštívená 2.5.2018.

Claessen, K. a Hughes, J. (2000). Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM. ISBN 1-58113-202-6. doi: 10.1145/351240.351266. URL `http://doi.acm.org/10.1145/351240.351266`. Navštívená 2.5.2018.

combinator (2007). Combinator pattern. URL `https://wiki.haskell.org/Combinator_pattern`. Navštívená 2.5.2018.

de la Banda, M. G., Marriott, K., Rafeh, R. a Wallace, M. (2006). The modelling language Zinc. In Benhamou, F., editor, *Principles and Practice of Constraint Programming - CP 2006*, pages 700–705, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-46268-2.

Fowler, M. (2010). *Domain Specific Languages*. Addison-Wesley Professional, 1st edition. ISBN 0321712943, 9780321712943.

Friedman, D. P. a Wand, M. (2008). *Essentials of Programming Languages, 3rd Edition*. The MIT Press, 3 edition. ISBN 0262062798, 9780262062794.

Gibbons, J. a Wu, N. (2014). Folding domain-specific languages: Deep and shallow embeddings (functional pearl). In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 339–347, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2873-9. doi: 10.1145/2628136.2628138. URL `http://doi.acm.org/10.1145/2628136.2628138`. Navštívená 2.5.2018.

Harvey, W. D. a Ginsberg, M. L. (1995). Limited discrepancy search. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'95, pages 607–613, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc. ISBN 1-55860-363-8, 978-1-558-60363-9. URL `http://dl.acm.org/citation.cfm?id=1625855.1625935`. Navštívená 2.5.2018.

Haskell tutorial (2016). Learn Haskell in 10 minutes. URL `https://wiki.haskell.org/Learn_Haskell_in_10_minutes`. Navštívená 2.5.2018.

Hudak, P. (1996). Building domain-specific embedded languages. *ACM Comput. Surv.*, **28**(4es), 196. doi: 10.1145/242224.242477. URL `http://doi.acm.org/10.1145/242224.242477`. Navštívená 2.5.2018.

Hughes, J. (1989). Why functional programming matters. *Comput. J.*, **32**(2), 98–107. ISSN 0010-4620. doi: 10.1093/comjnl/32.2.98. URL `http://dx.doi.org/10.1093/comjnl/32.2.98`. Navštívená 2.5.2018.

Hutton, G. (2007). *Programming in Haskell*. Cambridge University Press, New York, NY, USA. ISBN 0521871727, 9780521871723.

Jones, M. P. a Duponcheel, L. (1993). Composing monads. Technical report, Yale University, New Haven, Connecticut, USA. URL `http://web.cecs.pdx.edu/~mpj/pubs/RR-1004.pdf`. Navštívená 2.5.2018.

Kerrisk, M. (2017). *setjmp(3) Linux Programmer's Manual*, 4.16 edition.

King, D. J. a Wadler, P. (1993). Combining monads. In Launchbury, J. a Sansom, P., editors, *Functional Programming, Glasgow 1992*, pages 134–143, London, 1993. Springer London. ISBN 978-1-4471-3215-8.

Kiselyov, O. (2012). Preventing memoization in (AI) search problems. *Haskell Programming (osobná stránka)*. URL `http://www.okmij.org/ftp/Haskell/index.html#memo-off`. Navštívená 2.5.2018.

Lipovaca, M. (2011). *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press, San Francisco, CA, USA, 1st edition. ISBN 1593272839, 9781593272838.

Marlow, S. (2010). Haskell 2010 language report. URL `https://www.haskell.org/onlinereport/haskell2010/`. Navštívená 2.5.2018.

Newbern, J., Palamarchuk, A. a of Glasgow, T. U. (2007). *Control.Monad.Cont*, 2.2.2 edition. URL `http://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-Cont.html`.

Okasaki, C. (1998). *Purely Functional Data Structures*. Cambridge University Press, New York, NY, USA. ISBN 0-521-63124-6.

O'Sullivan, B., Goerzen, J. a Stewart, D. (2008). *Real World Haskell*. O'Reilly Media, Inc., 1st edition. ISBN 0596514980, 9780596514983.

Piponi, D. (2007). How to write tolerably efficient optimzation code without really trying... *A Neighborhood of Infinity (blog)*. URL `http://blog.sigfpe.com/2007/06/how-to-write-tolerably-efficient.html`. Navštívená 2.5.2018.

Piponi, D. (2009a). A monad for combinatorial search with heuristics. *A Neighborhood of Infinity (blog)*. URL `http://blog.sigfpe.com/2009/07/monad-for-combinatorial-search-with.html`. Navštívená 2.5.2018.

Piponi, D. (2009b). More parsing with best first search. *A Neighborhood of Infinity (blog)*. URL `http://blog.sigfpe.com/2009/09/language-nomonomorphismrestrictiongener.html`. Navštívená 2.5.2018.

Russell, S. a Norvig, P. (2009). *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition. ISBN 0136042597, 9780136042594.

Schrijvers, T., Stuckey, P. J. a Wadler, P. (2009). Monadic constraint programming. *J. Funct. Program.*, **19**(6), 663–697. doi: 10.1017/S0956796809990086. URL https://doi.org/10.1017/S0956796809990086. Navštívená 2.5.2018.

Schrijvers, T., Tack, G., Wuille, P., Samulowitz, H. a Stuckey, P. J. (2013). Search combinators. *Constraints*, **18**(2), 269–305. ISSN 1572-9354. doi: 10.1007/s10601-012-9137-8. URL https://doi.org/10.1007/s10601-012-9137-8. Navštívená 2.5.2018.

Smolka, G. (1996). The Oz programming model. In *Logics in Artificial Intelligence, European Workshop, JELIA '96, Évora, Portugal, September 30 - October 3, 1996, Proceedings*, page 251. doi: 10.1007/3-540-61630-6_17. URL https://doi.org/10.1007/3-540-61630-6_17.

Van Hentenryck, P., Perron, L. a Puget, J.-F. (2000). Search and strategies in OPL. *ACM Trans. Comput. Logic*, **1**(2), 285–320. ISSN 1529-3785. doi: 10.1145/359496.359529. URL http://doi.acm.org/10.1145/359496.359529.

Wadler, P. (1992). Comprehending monads. *Mathematical Structures in Computer Science*, **2**(4), 461–493. doi: 10.1017/S0960129500001560. URL https://doi.org/10.1017/S0960129500001560. Navštívená 2.5.2018.

Wall, L. a Schwartz, R. (1991). *Programming Perl*. A Nutshell handbook. O'Reilly & Associates. ISBN 9780937175644.

Yorgey, B. (2011). Typeclassopedia. *Haskell Wiki (oficiálna komunitná stránka o jazyku Haskell)*. URL https://wiki.haskell.org/Typeclassopedia. Navštívená 2.5.2018.