Funkcionální a logické programování
FPR
Studijní opora
Funkcionální programování

Dušan Kolář
Ústav informačních systémů
Fakulta informačních technologií
VUT v Brně

Únor '06 – Říjen '06
Verze 1.0

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

Abstrakt

Funkcionální programovací jazyky vznikly jako reakce na potřebu efektivně zvládnout rozsáhlé softwarové systémy a také jako přirozený programovací nástroj vůbec. Během doby se řada rysů, která se prvně objevila u funkcionálních jazyků, dostala i do jazyků imperativních (někdy téže zvaných „ne čistě funkcionální“). Čistě dominantní funkcionální jazyk lze těžko hledat, ale jazyk Haskell představuje jeden z významných směrů. Na druhou stranu používání tabulkového procesoru k něčemu „vyššímu“, než jen tvorbě tabulek, je vlastně taktéž funkcionální programování, které navíc používá řada lidí díky přirozenému chápání věci. Ne čistě funkcionální jazyky se potom prosazují zejména díky jazyku ML a jeho derivátům.

Tato publikace shrnuje základní principy a obraty z užití jazyka Haskell a jeho formálního základu — λ-kalkulu. Jejím cílem je poskytnout dostatečný podklad pro hrubou orientaci v dané kategorii programovacích jazyků a poskytnout dostatečný podklad pro zvládnutí základních programovacích technik v jazyku Haskell, kterou čitatel plně rozvine ve vlastní praktické činnosti. Užití dalších jazyků ať čistě či ne čistě funkcionálních by potom neměl být problém.

Věnováno Mirkovi

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

Obsah

Notace a konvence použité v publikaci

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

Čas potřebný ke studiu: 2 hodiny 15 minut

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

Cíle kapitoly

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

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

Průvodce studiem

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

Za průvodcem bude vždy uveden obsah kapitoly.

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

Výklad

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

Definice!

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

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

Algoritmus!

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

Přeskoky v rámci jednoho oddílu však nedoporučujeme.

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

Řešený příklad

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

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

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

Některé informace mohou být vypíchnuty, či doplněny takto bokem.

Pojmy k zapamatování

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

Úlohy k procvičení:

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

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

Závěr

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

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

Klíč k řešení úloh

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

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

Další zdroje

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

Kapitola 1

Úvod

Čas potřebný ke studiu: 32 hodin 50 minut
Tento čas reprezentuje dobu pro studium celého modulu.

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

Cíle modulu

Cílem modulu je na konkrétním příkladu funkcionálního programovacího jazyka prezentovat základní obraty, možnosti a charakter práce v takovémto typu jazyka. V úvodu potom je zmíněna i teoretická báze všech funkcionálních jazyků λ-kalkul. Význam modulu je v prezentaci čistě deklarativního programovacího paradigmatu a jeho ovládnutí, neboť tak otevírá čitateli nové možnosti v přístupu k programovacím technikám vůbec.

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

Po ukončení studia modulu:

- budete schopni jednoduché práce v λ-kalkulu;
- budete mít dostatečnou znalost jazyka Haskell pro další rozvoj práce v něm i pro tvorbu středně náročných aplikací;
- budete ovládat některé knihovní funkce jazyka Haskell, funkce vyššího řádu;
- budete mít základní znalosti ze zpracování typů a typových tříd v jazyce Haskell.

Průvodce studiem

Modul začíná definicí formální báze funkcionálních programovacích jazyků, λ-kalkulu. Následuje úvod do jazyka Haskell, který představuje příklad deklarativního a čistě funkcionálního jazyka. V dalších kapitolách je rozvíjen koncept funkce až po funkce vyššího řádu. Na závěr modulu se potom zmíníme o datech, datových typech a práci s nimi. Vrcholem jsou typové třídy, jako prostředek pro další úroveň polymorfismu.
Pro studium modulu je důležité mít znalosti z oblasti programovacích technik a imperativních programovacích jazyků. Také znalosti algoritmizace. Výhodou je samozřejmě nějaká zkušenost s prací ve funkcionálních jazycích. Není však nezbytná. Vždy je však nutné počítat s tím, že postupy a koncepty užívané v imperativních jazycích se dají jen málokdy aplikovat do prostředí jazyků deklarativních.

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

Návaznost na předchozí znalosti

Pro studium tohoto modulu je nezbytné , aby studující měl znalosti z programovacích technik a technologií a rozvinutou praktickou zkušenost s programováním na vysoké úrovni. Zkušenost s deklarativními programovacími jazyky není nutná, ale je výhodou. Stejně tak znalosti z oblasti zpracování a překladu programovacích jazyků se mohou jevit jako výhoda.

1.1 Koncepce modulu

Následující kapitola definuje λ-kalkul a prezentuje nezbytné minimum pro jeho zvládnutí a práci v něm.

Další kapitola se zabývá úvodními koncepty a konstrukcemi programovacího jazyka Haskell. Prezentuje základní typy jazyka, způsoby definice funkce a zejména kombinace obojího pro zpracování a manipulaci s daty. U funkcí ukazuje varianty zápisu i užití pro typické konstrukce programů. Data jsou prezentována jak atomická, tak datové struktury vestavěné v jazyce.

V dalších kapitolách jsou rozvíjeny pojmy funkce, způsoby využití rekurze, zavádění funkcí vyššího řádu a pokročilá práce se seznamy. Dále je rozvinuta práce s daty pro typové třídy a uživatelské datové typy a jejich vzájemné propojení.

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

1.2 Potřebné vybavení

Pro studium a úspěšné zvládnutí tohoto modulu není třeba žádné speciální vybavení. Naprosto dostačující je běžné PC s operačním systémem řady windows či Linux. Na takovém PC je nutné mít nainstalován interpret či překladač jazyka Haskell (www.haskell.org). Pro práci začátečníka je interpret jednodušší, doporučujeme hugs/winhugs, případně ghci. Výhodou je přístup k Internetu, aby bylo možné znalosti průběžně aktualizovat a doplňovat z dalších zdrojů.

Další zdroje

Funkcionální paradigma nevyžaduje sice explicitně žádné znalosti, ale je dobré mít zvládnuté obecné techniky algoritmizace, algebry apod. Tento modul není úvodem do programování, ale úvodem do jistého druhu programování, proto není možné vysvětlovat základní obraty a je nutné je nastudovat jinde.
Řadu odkazů či přímo textů lze nalézt přímo na www.haskell.org. Z literatury tištěné lze doporučit:

- Češka, M., Motyčková, L., Hruška, T.: *Vyčíslitelnost a složitost*, s. 217, červen 1992, Vysoké učení technické v Brně.
- Thompson, S.: *Haskell, The Craft of Functional Programming*, ADDISON-WESLEY, 1999, ISBN 0-201-34275-8
- Jones, S.P.: *Haskell 98 Language and Libraries*, Cambridge University Press, 2003, p. 272, ISBN 0521826144.
- Bieliková, M., Návrat, P.: *Funkcionálne a logické programovanie*, Vydavateĺstvo STU, Vazovova 5, Bratislava, 2000.

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

Kapitola 2

λ-kalkul

Tato kapitola podává formální základ funkcionálních programovacích jazyků — λ-kalkul. Od něj se odvíjí zpracování a funkčnost funkcionálních programovacích jazyků.

Čas potřebný ke studiu: 20 hodin 30 minut.

Cíle kapitoly

Cílem kapitoly je zvládnout orientaci a jednoduchou práci se základními a důležitými pojmy z oblasti λ-kalkulu, jakožto i tvorbu výrazů a jejich zpracování. Osvojit si terminologii, která je s tímto formalismem spojena. Zvládnout důležité vlastnosti a způsoby zpracování uvedeného formalismu a zejména důsledků, které z nich vyplývají pro zpracování a vyhodnocení funkcionálních jazyků.
Kapitola je míněna jako nezbytné minimum, které je nutné v dané oblasti zcela ovládat. Doporučuji dále rozvinout znalosti λ-kalkulu pomocí dalších podkladů.

Průvodce studiem

Před tím, než se budeme zabývat funkcionálním programovacím jazykem Haskell, je třeba si vybudovat jistou taxonomii a názvosloví, aby potom bylo jasné, jaký termín se v daných souvislostech jak chápe. Krom toho je nutné si navyknout na nový způsob chápání pojmu funkce v programech, k čemuž je λ-kalkul ideální.
Ke studiu této kapitoly bude stačit papír a tužka, případně pomocná literatura ať klasická, či na Internetu (potom tedy potřebujete počítač s prohlížečem WWW stránek a připojení k Internetu). Jako u většiny ostatních se jedná o to si zapamatovat danou terminologii, zvládnout prezentované mechanismy a techniky a zapamatovat si důležité a podstatné věci v kapitole. Pro hlubší studium, nebo bližší objasnění termínů, je však vhodné využít informací na Internetu, či v další literatuře.

Obsah

Výklad

2.1 λ-kalkul — definice, konvence

λ-kalkul (nebo také lambda kalkul, anglicky λ-calculus) je teorie funkcí, kterou zpracoval ve dvacátých letech 20. století Alonzo Church, když jako základ použil teorii kombinátorů (Moses Schönfinkel). Ve třicátých letech téhož století bylo dokázáno, že λ-kalkul je ekvivalentní teorii kombinátorů (Haskell Curry) a že λ-kalkul je univerzální výpočetní systém (Kleene). Jak je vidět, tak vznik teorie se datuje před vynález počítače, přesto dnes slouží jako formální základ funkcionálních jazyků, kdy každý funkcionální jazyk lze přeložit na λ-kalkul a sémantiku každého imperativního jazyka v něm lze taktéž vyjádřit.

λ-kalkul — definice

Jazyk λ-kalkulu je jednoduchý, povoluje totiž pouze tři typy výrazů:

- **Proměnné**: proměnné jako každé jiné, definují vazbu s okolím.
- **Aplikace**: máme-li v λ-kalkulu dva výrazy, E_1 a E_2, potom, je-li to jinak možné, tak je možné jeden aplikovat na druhý. Pokud zvolíme aplikaci v tomto pořadí (E_1 E_2), tak výraz (též možné psát λ-výraz) E_1 je operátorem (rator) a E_2 je operandem (rand) v dané aplikaci.
- **Abstrakce**: reprezentují funkce s jednou vázanou proměnnou (hlavičkou abstrakce) a tělem, které je opět tvořeno λ-výrazem; pokud nějaká operace vyžaduje více parametrů, tak bezprostředním vnořením λ-abstrakcí dosáhneme výsledku.

Formálně lze jazyk λ-kalkulu pomocí BNF definovat takto:

Definice 2.1.1 Definice!

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

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

λ-kalkul — konvence

Z definice vyplývá, že kromě proměnné je každý jiný výraz oddělen závorkami. Jelikož by to v praxi vedlo na velký počet závorek a nepřehledné výrazy, tak existují jisté konvence pro jejich eliminaci. Následují definice tří konvencí.

Definice!

Definice 2.1.2 *Aplikace je vždy zleva asociativní, takže zápis* $((\dots(E_1\ E_2)\dots)E_n)$ *je možné zkrátit na* $E_1\ E_2 \dots E_n$.

Definice!

Definice 2.1.3 *Dosah hlavičky abstrakce „λV“ sahá tak daleko doprava, jak to je jen možné, takž zápis* $(\lambda V.(E_1\ E_2 \dots E_n))$ *je možné zkrátit na* $\lambda V.E_1\ E_2 \dots E_n$.

Definice!

Definice 2.1.4 *Bezprostředně vnořené λ-abstrakce je možné zřetězit, takže zápis* $(\lambda V_1.(\dots(\lambda V_n.E)\dots))$ *je možné zkrátit na* $\lambda V_1 \dots V_n.E$.

Jak již se text v úvodu zmínil, tak proměnné především zajišťují vazbu mezi výrazem a okolím. Vazbu proměnné definuje hlavička abstrakce, přičemž proměnná je vázána nejbližší příslušnou hlavičkou nalevo od svého výskytu.

Definice!

Definice 2.1.5 *Volné a vázané proměnné si definujme na následujících příkladech:*

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

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

2.2 Konverze

Výrazy v λ-kalkulu lze modifikovat oproti jejich původnímu zápisu podle určitých pravidel. Často tak činíme proto, abychom původní zápis zjednodušili, což často znamená eliminaci aplikací, ale nepředbíhejme. V současnosti jsou v λ-kalkulu tři pravidla pro zjednodušení, či *konverzi* nebo *redukci* výrazů. V průběhu takových redukcí dochází k nahrazování proměnných — jejich substituci — za jiné proměnné, či výrazy. Konverze potom specifikují pravidla těchto záměn.

konverze, redukce

Definice 2.2.1 *Konverzní pravidla v λ-kalkulu jsou:* **Definice!**

- *α-konverze: libovolná abstrakce tvaru $\lambda V.E$ může být redukována na abstrakci $\lambda V'.E[V'/V]$. Zápis $E[V'/V]$ označuje substituci proměnné V' za volné výskyty proměnné V ve výrazu E, přičemž substituce musí být* platná. *Pojmem platná rozumíme, že při substituci $E[E'/V]$ se žádná volná proměnná ve výrazu E' nestane vázanou.* substituce platnost

- *β-konverze: libovolná aplikace tvaru $(\lambda V.E_1)E_2$ může být redukována na $E_1[E_2/V]$, pokud je substituce platná.*

- *η-konverze: libovolná abstrakce tvaru $\lambda V.(EV)$, kde V není volné v E, může být redukována na E.*

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

- $E_1 \underset{\alpha}{\rightarrow} E_2$ pokud se redukce provedla pomocí α-konverze
- $E_1 \underset{\beta}{\rightarrow} E_2$ pokud se redukce provedla pomocí β-konverze
- $E_1 \underset{\eta}{\rightarrow} E_2$ pokud se redukce provedla pomocí η-konverze

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

Řešený příklad

Zadání: Pro zadané λ-výrazy proveďte předepsané redukce.
Řešení:

1. $\lambda x.x\, y \underset{\alpha}{\rightarrow} \lambda z.z\, y$
2. $(\lambda x\, y.x\, y)(u\, v) \underset{\beta}{\rightarrow} \lambda y.(u\, v)\, y$
3. $\lambda x.(u\, v)x \underset{\eta}{\rightarrow} (u\, v)$

2.3 Rovnost a relace $\to$

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

Definice!

Definice 2.3.1 *Dva λ-výrazy, E_1 a E_2 jsou identické, pokud jsou zapsány toutéž posloupností znaků, píšeme: $E_1 \equiv E_2$*

Definice!

Definice 2.3.2 *Jsou-li E a E' dva λ-výrazy, potom jsou si rovny, píšeme $E = E'$, pokud buďto $E \equiv E'$, nebo existují takové výrazy $E_1, \ldots, E_n$, že:*

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

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

Definice!

Definice 2.3.3 *Jsou-li E a E' dva λ-výrazy, potom $E \rightarrow E'$, pokud buďto $E \equiv E'$, nebo existují takové výrazy $E_1, \ldots, E_n$, že:*

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

2.4 Substituce

Doposud jsme při substituci $E[E'/V]$ nahrazovali volné výskyty V v E za E', přičemž jsme vyžadovali, aby byla platná — žádná volná proměnná z E' se v E po substituci za V nesměla stát vázanou. Abychom se nemuseli nadále zabývat platností substituce, tak zavedeme zobecněnou substituci, která bude platit pro všechny výrazy účastnící se substituce.

Definice!

Definice 2.4.1 *Zobecněnou substituci definujeme rekurzivně přes E ve výrazu $E[E'/V]$ takto:*

E		$E[E'/V]$
V		E'
V'	*(kde* $V \neq V'$*)*	V'
$E_1\,E_2$		$E_1[E'/V]\,E_2[E'/V]$
$\lambda V.E_1$		$\lambda V.E_1$
$\lambda V'.E_1$	*(kde* $V \neq V'$ *a* V' není *volné v* E'*)*	$\lambda V'.E_1[E'/V]$
$\lambda V'.E_1$	*(kde* $V \neq V'$ *a* V' je *volné v* E'*)*	$\lambda V''.E_1[V''/V'][E'/V]$ *(kde* V'' *je proměnná, která není volná v* E' *nebo* E_1*)*

Výklad

2.5 Reprezentace objektů

Abychom mohli v λ-kalkulu s výrazy lépe pracovat, tak zavedeme notaci, která umožní pojmenovat nějaký výraz a toto nové jméno potom užívat namísto takového výrazu. Zavedení nového pojmenování/označení pro λ výraz budeme psát takto:

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

kde $\sim$ reprezentuje nové označení. Abychom odlišili pojmenování výrazu od ostatních částí výrazu, tak budeme psát takové označení **tučně** či podtrženě.

Na příkladu si ukažme, jak to bude vypadat. Definujme například výrazy reprezentují pravdivostní hodnoty *true* a *false*.

$$\texttt{LET}\ \mathbf{true} = \lambda x\,y.x$$
$$\texttt{LET}\ \mathbf{false} = \lambda x\,y.y$$

Řešený příklad

Zadání: Pro definici pravdivostních hodnot definujte výraz *not* s významem negace pravdivostní hodnoty.

Řešení: Uvážíme-li, že **true** bere ze dvou argumentů, na které je aplikováno, první a vrací jako výsledek, zatímco **false** se chová opačně, tak toho můžeme využít tak, že **true** aplikujeme na **false** a něco, třeba proměnnou x. Potom **false** je třeba aplikovat na nějakou proměnnou, třeba y a dále **true**, aby vrátilo svoji negaci. Když to porovnáme, tak stačí nějakou pravdivostní hodnotu, řekněme p, aplikovat po řadě na **false** a **true** a dostaneme, co potřebujeme. Výsledný výraz tedy je

$$\texttt{LET}\ \mathbf{not} = \lambda p.p\ \mathbf{false}\ \mathbf{true}$$

Dále si budeme demonstrovat, pro vybraný případ, jak lze takové definice využít při konverzích. Ukažme postup vyhodnocení výrazu **not false**.

$$\begin{aligned}\textbf{not false} &= (\lambda p.p\ \textbf{false true})\ \textbf{false}\\ &= \textbf{false false true}\\ &= (\lambda x\, y.y)\ \textbf{false true}\\ &= (\lambda y.y)\ \textbf{true}\\ &= \textbf{true}\end{aligned}$$

Podobně jako je možné najít výrazy pro reprezentaci pravdivostních hodnot, tak je možné najít reprezentaci pro celá čísla i pro operace nad nimi, např. takovéto:

- **pre** $\underline{n} = \underline{n-1}$
- **add** $\underline{m}\ \underline{n} = \underline{m+n}$
- **iszero** $\underline{0} =$ **true**
- **iszero** $\underline{n} =$ **false**, pokud $\underline{n} \neq \underline{0}$

Je dokonce možné najít i operace podmíněného výrazu, podobně jako je ternární operátor v jazyce C. Můžeme jej značit například takto $(E\,?\,E_1 \mid E_2)$. A jeho definice je poměrně jednoduchá díky vlastnosti výrazů pro pravdivostní hodnoty:

$$\texttt{LET}\ (E\,?\,E_1 \mid E_2) = (E\ E_1\ E_2)$$

Už takováto malá sada operací a definice celých čísel a mohli bychom definovat výraz pro násobení dvou čísel. Intuitivně jistě budeme souhlasit s tímto rekurzivním vztahem:

$$\textbf{mult}\ \underline{m}\ \underline{n} = (\textbf{iszero}\ \underline{m}\ ?\ \underline{0} \mid \textbf{add}\ \underline{n}\ (\textbf{mult}\ (\textbf{pre}\ \underline{m})\ \underline{n}))$$

který vychází z rovností:

$$m * n = 0,\ \text{pokud } m = 0$$

$$m * n = n + ((m-1) * n),\ \text{pokud } m > 0$$

I když by se mohlo zdát, že je tedy možné definovat pojmenování pro výraz reprezentující násobení celých čísel, tak to tak ve skutečnosti není, protože bychom se na pravé straně definice odkazovali na výraz **mult**, který ale do té doby není známý (např. jako u makrodefinice v jazyce C).

2.6 Operátor pevného bodu

Situaci rekurzivních definic je možné řešit, pokud definujeme, přesněji budeme-li schopni definovat, λ-výraz **Y** s touto vlastností:

$$\mathbf{Y}\ E \;=\; E\ (\mathbf{Y}\ E)$$

Takovému výrazu říkáme *operátor pevného bodu*. Je to proto, že máme-li výraz E a k němu hodnotu X takovou, že $E\ X \;=\; X$, potom X je pevným bodem funkce (je-li dáno jako parametr, je i výsledkem). Pokud máme nějakou funkci, pojmenujme ji **FixIt!**, která takovou hodnotu z výrazu získá, potom **FixIt!** $E \;=\; X$. A jelikož platí $E\ X \;=\; X$, potom určitě platí i $X \;=\; E\ X$ a dále **FixIt**! $E \;=\; E$ (**FixIt**! E). Taková funkce se potom nazývá operátor pevného bodu.

operátor pevného bodu

V λ-kalkulu je možné naštěstí najít výraz, a ne jeden, který takové vlastnosti má. My uvádíme definici toho nejproslulejšího.

$$\texttt{LET}\ \mathbf{Y} \;=\; \lambda f.(\lambda x.f\,(x\,x))(\lambda x.f\,(x\,x))$$

Řešený příklad

Zadání: Ukažte, že operátor pevného bodu **Y** má požadované vlastnosti.

Řešení:

$$\begin{array}{lcll}
\mathbf{Y}\,E & = & (\lambda f.(\lambda x.f\,(x\,x))(\lambda x.f\,(x\,x)))\ E & (1)\\
 & = & (\lambda x.E\,(x\,x))\ (\lambda x.E\,(x\,x)) & (2)\\
 & = & E\ ((\lambda x.E\,(x\,x))\ (\lambda x.E\,(x\,x))) & (3)\ \text{za } E \text{ je (2), tj. } \mathbf{Y}\,E\\
 & = & E\ (\mathbf{Y}\,E) & (4)
\end{array}$$

Jelikož operátor pevného bodu **Y** má požadované vlastnosti, tak si ukážeme schéma, jak libovolnou rekurzivní definici převést na výraz v λ-kalkulu. Máme-li definici tvaru

$$\mathbf{f}\ x_1, \ldots, x_n = \ldots\ \mathbf{f}\ \ldots$$

kde $\ldots\ \mathbf{f}\ \ldots$ označuje libovolný lambda výraz, který obsahuje **f**, potom ji lze převést na výraz a pojmenovat ho, potom zápis provedeme takto:

$$\texttt{LET}\ \mathbf{f} = \mathbf{Y}\ (\lambda f\ x_1 \ldots x_n\ .\ \ldots\ f\ \ldots)$$

Na základě tohoto předpisu je již možné definovat λ-výraz pro násobení dvou celých čísel:

$$\texttt{LET}\ \mathbf{mult} = \mathbf{Y}\ (\lambda f\ m\ n\ .\ (\mathbf{iszero}\ m\ ?\ 0\ |\ \mathbf{add}\ n\ (f\ (\mathbf{pre}\ m)\ n)))$$

Řešený příklad

Zadání: Demonstrujte správnost výrazu pro násobení na číslech 2 a 3. U výrazů, kde neznáte tělo prostě vyhodnoťte.

Řešení:

$$
\begin{aligned}
\textbf{mult}\,2\,3 &= (\mathbf{Y}\,(\lambda f\,m\,n.(\textbf{iszero}\,m\,?\,0\,|\,\textbf{add}\,n\,(f\,(\textbf{pre}\,m)\,n))))\,2\,3\\
&= (\lambda f\,m\,n.(\textbf{iszero}\,m\,?\,0\,|\,\textbf{add}\,n\,(f\,(\textbf{pre}\,m)\,n)))\,\textbf{mult}\,2\,3\\
&= (\lambda m\,n.(\textbf{iszero}\,m\,?\,0\,|\,\textbf{add}\,n\,(\textbf{mult}\,(\textbf{pre}\,m)\,n)))\,2\,3\\
&= (\lambda n.(\textbf{iszero}\,2\,?\,0\,|\,\textbf{add}\,n\,(\textbf{mult}\,(\textbf{pre}\,2)\,n)))\,3\\
&= (\textbf{iszero}\,2\,?\,0\,|\,\textbf{add}\,3\,(\textbf{mult}\,(\textbf{pre}\,2)\,3))\\
&= \textbf{add}\,3\,(\textbf{mult}\,(\textbf{pre}\,2)\,3)\\
&= \textbf{add}\,3\,(\textbf{mult}\,1\,3)\\
&= \ldots =\\
&= \textbf{add}\,3\,(\textbf{add}\,3\,(\textbf{mult}\,0\,3))\\
&= \ldots =\\
&= \textbf{add}\,3\,(\textbf{add}\,3\,0)\\
&= \ldots =\\
&= 6
\end{aligned}
$$

2.7 Normální forma

Pozornému čtenáři asi neuniklo, že postup vyhodnocení a jeho zastavení se může odehrát mnoha způsoby. Určitě si klade otázku, jestli existuje obecně užitelný postup a tvar λ-výrazu, který by byl jistotou v nepřeberném množství možností.

Co se týká tvaru λ-výrazu, tak ten existuje, opět v několika variantách a my si uvedeme nejsilnější variantu, *normální formu*.

Definice!

Definice 2.7.1 *λ-výraz je v* normální formě*, pokud neobsahuje žádné β- a η-redexy. (Tzn. jediná redukce, kterou lze provést, je α-redukce.)*

S existencí normální formy do jisté míry souvisí i následující věta a její důsledky.

Teorém 2.7.1 Church-Rosserův teorém pro λ-kalkul
Pokud $E_1 = E_2$*, pak existuje* E *takové, že* $E_1 \to E$ *a* $E_2 \to E$*.*

Intuitivně lze tuto větu interpretovat tak, že E je výraz v normální formě a ostatní dva výrazy různými postupy na danou normální formu převedeme. Nicméně situace není tak jednoznačná a jednoduchá, jak naznačují důsledky předchozí věty.

Teorém 2.7.2 Důsledky Church-Rosserova teorému

- *Pokud E má normální formu, potom $E \to E'$ pro nějaké E' v normální formě.*
- *Pokud E má normální formu a $E = E'$, potom E' má normální formu.*

Pokud $E = E'$ a E i E' jsou oba v normální formě, potom E a E' jsou identické až na přejmenování vázaných proměnných (α-konverzi).

To, že existují λ-výrazy, které nemají normální formu, se lze přesvědčit poměrně jednoduše, neboť například operátor pevného bodu $\mathbf{Y}$ ji nemá. Stejně tak to, že ji nějaký výraz má ještě neznamená, že k ní lze dojít jakýmkoliv postupem konverzí. Např. výraz $(\lambda x.\mathbf{true})(\mathbf{Y}\,f)$ má normální formu **true**, i když

$$\begin{aligned}(\lambda x.\mathbf{true})(\mathbf{Y}\,f) &\to (\lambda x.\mathbf{true})(f\,(\mathbf{Y}\,f)) \to \\ &\to (\lambda x.\mathbf{true})(f\,(f\,(\mathbf{Y}\,f))) \to \\ &\to \ldots\end{aligned}$$

Jak tedy dosáhnout normální formy, pokud ji λ-výraz má? Na to odpovídá následující a poslední věta této kapitoly.

Teorém 2.7.3 *Pokud E má normální formu, potom opakovaná redukce nejlevějšího β- či η-redexu (po možné α-konverzi pro zabránění neplatné substituci) bude končit ve výrazu v normální formě.*

normalizační teorém

Pojmy k zapamatování

Klíčových pojmů této kapitoly je celá řada, jsou to: aplikace, abstrakce, λ-výraz, volná proměnná, vázaná proměnná, α-, β-, η-konverze, redukce, redex, substituce, platná substituce, zobecněná substituce, rovnost, identita, relace $\to$, notace `LET`, pevný bod, operátor pevného bodu, normální forma, normalizační teorém.
Krom pojmů samotných je však důležité zvládnout i práci v λ-kalkulu, tvorbu vlastních λ-výrazů.

Závěr

V této kapitole jste se seznámili s podstatou a základy λ-kalkulu. Pokud se orientujete ve vymezených pojmech a jste schopni nejen pasivně, ale i aktivně tvořit výrazy v λ-kalkulu, tak kapitola splnila svoji úlohu. Zcela jistě se jednalo o náročnou kapitolu, která si zřejmě vyžádala pomocnou literaturu (např. viz konec kapitoly). Vzhledem k tomu, že λ-kalkul je základem funkcionálních programovacích jazyků, tak je zvládnutí práce v něm důležité. Proto nepostupujte do další kapitoly, pokud jste alespoň tyto minimalizované základy zcela nezvládli. V případě úspěšného zvládnutí potom můžete přejít dále.

Úlohy k procvičení:

- Definujte výrazy, které navazují na pravdivostní hodnoty definované v kapitole a realizují operace **and**, **or**.
- Buďto sami, či na základě doporučené literatury, zvolte reprezentaci pro celá čísla. Realizujte operace uvedené v kapitole bez definice, dále doplňte systém o všechny běžné aritmetické operace nad celými čísly.
- Zamyslete se nad realizací vzájemně rekurzivních funkcí v λ-kalkulu.

Klíč k řešení úloh

V doporučené literatuře najdete řešení všech otázek.

Další zdroje

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

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

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

Kapitola 3

Úvod do jazyka Haskell

Tato kapitola seznamuje s absolutními základy jazyka Haskell — jaké jsou základní datové typy a konstrukce pro definici funkcí. Objeví se také ukázky a řešené úlohy v kódu jazyka Haskell.

Čas potřebný ke studiu: 3 hodiny 20 minut.

Cíle kapitoly

Cílem kapitoly je naučit čtenáře zvládnout práci se základními předdefinovanými typy v jazyce Haskell a vytváření vlastních funkcí a tím pádem i programů.
V příkladech bude prezentována zjednodušená syntaxe jazyka, která je závislá na poměrně jednoduchých pravidlech rozmístění textu programu. Její dodržení umožňuje rychlou a efektivní tvorbu programů, které jsou navíc čitelné a přehledné. Další kapitoly už se potom budou věnovat technikám a drobným rozšířením. Jedná se tedy o klíčovou kapitolu.

Průvodce studiem

Tato kapitola odhalí sice základní, nicméně nosné prvky jazyka Haskell, které se používají ve všech programech. Každou novou vědomost je nutné si okamžitě ověřit prakticky, aby si čtenář osvojil i způsob, *jakým vyhodnocení jazyka Haskell pracuje*.
Ke studiu této kapitoly je naprosto nezbytné mít k dispozici počítač s instalovaným interpretem jazyka Haskell. Je možné mít i kompilátor, ale práce v něm, zejména pro začátečníky, je mnohem, ale mnohem náročnější. Na stránkách jazyka (www.haskell.org) je možné najít interpret hugs i jeho klon čistě pro MS Windows, který se jmenuje winhugs.

Obsah

Výklad

3.1 Úvod k funkcionálnímu programování

Základním modelem funkcionálních jazyků je matematický pojem funkce aplikované na argumenty a vypočítávající jediný výsledek. Podobně jako v λ-kalkulu je tedy výsledek deterministický (není závislý na čase, místě, kde ho obdržíte).

Proto se někdy hovoří o jazycích čistě funkcionálních, když chceme zdůraznit, že funkce v době vyhodnocení nemá žádný vedlejší účinek. Přísně vzato však funkcionální jazyky jsou vždy čisté a ostatní, taktéž nazývané ne-čistě funkcionální, nebo hybridní patří do kategorie jazyků imperativních. Mezi (čistě) funkcionální jazyky řadíme například jazyky FP, Hope, Haskell, Miranda, Orwell. Do kategorie imperativních jazyků, které nabízejí vlastnosti funkcionálních potom spadají LISP, Scheme, ML a jeho deriváty.

První implementace některého ze zmiňovaných jazyků se objevila v roce 1958 a byl to LISP (Mac Carthy). V roce 1965 se objevuje jazyk ISWIM (Landin), což byl předchůdce jazyka ML. V roce 1978 Roger Milner navrhuje jazyk ML s typovou inferencí. V roce 1985 se objevuje Miranda, která jako první zavádí strategii vyhodnocení známou jako „lazy evaluation“. Jazyk Haskell se objevuje v roce 1987, taktéž užívá lazy evaluation. Jazyk Standard ML přichází v roce 1989 (Appel, Mac Queen), v roce 1998 přichází standard pro jazyk Haskell. Jazyk ML a jeho deriváty, stejně jako jazyk Haskell, se však neustále vyvíjejí (aspoň v době psaní těchto řádků to byla pravda).

3.2 Základní typy jazyka Haskell

Podobně jako řada dalších jazyků, i jazyk Haskell nabízí sadu víceméně známých a očekávatelných datových typů, které jsou dále nedělitelné. V následujícím výčtu uvádíme jméno typu a příklady některých jeho literálů. Jazyk Haskell je citlivý na velikost písmen, pro psaní prvního písmene v literálu má navíc pevná pravidla. Počáteční velké písmenu musí mít jména typů, typových tříd a datové konstruktory. Všechny ostatní literály začínající písmenem musejí začínat malým písmenem.

citlivý na velikost písma

- **Int**, celá čísla, např.: 45, 890
- **Char**, znaky, např.: ’a’, ’X’

- **Bool**, reprezentace logických hodnot, což jsou: True, False
  Pozn.: Vidíme, že pro vytvoření logické hodnoty se používá datový konstruktor, jak zjistíme dále, tak na rozdíl od ostatních typů zde vyjmenovaných, je možné tento typ definovat přímo v jazyce a v knihovnách jazyka tomu tak i je.
- **Float**, desetinná čísla, např.: 3.14159, 234.2, 6.5e-4

Seznamy

silně typovaný

seznam je homogenní

polymorfní

Kromě atomických typů nabízí jazyk Haskell i typy strukturované. Tím nejtypičtějším je zřejmě seznam. Jelikož jazyk Haskell je *silně typovaný*, což znamená, že změna typu z jednoho na jiný je možná pouze zavoláním převodní funkce a každá entita má přesně daný svůj typ, je seznam v jazyce Haskell homogenní datová struktura. Tedy, že všechny jeho prvky musejí být stejného typu. Na druhou stranu je možné vytvářet seznamy nad různými typy, tzn., že datové konstruktory pro typ seznam jsou polymorfní.

Pozorný čtenář si jistě říká, jak lze zajistit, aby seznam byl homogenní, ale zároveň mohly vznikat různé seznamy a to v prostředí silně typovaného jazyka. Důvodem je existence typových proměnných. Seznam celých čísel lze zkonstruovat např. takto:

$$1 : 2 : 3 : []$$

Konstruktorem je zde „:“ (dvojtečka) a pár prázdných hranatých závorek. Pokud bychom uvážili existenci pouze celých čísel, potom typ pro dvojtečku (konstruktor neprázdného seznamu) je:

$$(:) \ :: \ \texttt{Int} \rightarrow [\texttt{Int}] \rightarrow [\texttt{Int}]$$

Což znamená, že prvním argumentem, na který jej lze aplikovat musí být typu `Int`. Dalším argumentem musí být seznam nad typem `Int`. No a výsledkem je potom seznam nad typem `Int` s tím, že tento nový prvek je umístěn na začátek takto vzniklého seznamu. Jen pro úplnost, operátor je zprava asociativní.

Pokud bychom chtěli vytvořit seznam pravdivostních hodnot, tak potom příklad konstrukce a typ pro konstruktor neprázdného seznamu (dvojtečka) by byly tyto:

$$\texttt{False} : \texttt{True} :[]$$

$$(:) \ :: \ \texttt{Bool} \rightarrow [\texttt{Bool}] \rightarrow [\texttt{Bool}]$$

Abychom tyto dva uvedené ale i všechny ostatní typy mohli pojmout, tak skutečný typ pro konstruktor neprázdného typu je tento:

$$(:) \ :: \ \texttt{a} \rightarrow [\texttt{a}] \rightarrow [\texttt{a}]$$

kde `a` je typová proměnná. Jakmile operátor aplikujeme na první argument, řekněme typu `Int`, dojde k unifikaci typové proměnné s typem `Int`. Tedy aplikace dvojtečky na číslo 3 má tento typ:

$$((:)3) \;::\; [\mathtt{Int}] \rightarrow [\mathtt{Int}]$$

(První typ byl již „spotřebován“ aplikací na číslo 3, proto tedy o ten argument méně.)

Teď již zbývá, pro pořádek, dodat typ pro konstrukci prázdného seznamu.

$$[] \;::\; [a]$$

Kromě uvedených možností lze seznam zapsat i přirozenější formou, pro celá čísla např. takto:

$$[1, 2, 3]$$

Toto sice není čistý zápis v jazyku Haskell, ale je přehlednější a jednodušší a překladač ji přeloží na korektní zápis užitý výše:

$$1 : 2 : 3 : []$$

který sestává pouze z prvků jazyka.

Pozn.: To, že typ pro prázdný seznam je takový, jaký je, je častým zdrojem nepochopení. Při testování, řekněme řadicí funkce mySort, dáváme jako parametry různé seznamy a i seznam prázdný, ten je však typu `[a]`*. Abychom mohli úspěšně otestovat i tuto možnost, je možné typ kdykoliv zúžit, takže i pro typ prázdného seznamu, takže to lze vyřešit takto, kdy zúžíme typ parametru:*

$$\mathtt{mySort}\,([] :: [\mathtt{Int}])$$

Pokud bychom typ nezúžili korektně, systém (překladač či interpret) to odmítne.

Díky tvaru konstruktorů také můžeme dedukovat, jak je možné přistoupit k seznamu.

- Detekce prázdného seznamu (prázdné hranaté závorky).
- Přístup k prvním prvku seznamu (hlavičce) a jeho zbytku/konci/ocásku (užívá se operátor dvojtečka, zápis s proměnnými např. takto: `(x:xs)`).

N-tice

Dalším připraveným strukturovaným typem jsou n-tice. Na rozdíl od seznamu jsou n-tice seznamy heterogenní datové struktury. Tedy každý složka n-tice může být jiného typu. Konstruktorem n-tice je „.“

n-tice je heterogenní

(čárka), ve spojitosti s kulatými závorkami, které omezí zápis n-tice. Příkladem, i s typovou signaturou, mohou být tyto n-tice:

$$(1, 2) \ :: \ (\mathrm{Int}, \mathrm{Int})$$

$$(1, ['a', 'b']) :: (\mathrm{Int}, [\mathrm{Char}])$$

Potom typ pro konstruktory dvojic a trojic je:

$$(,) :: \mathrm{a} \to \mathrm{b} \to (\mathrm{a}, \mathrm{b})$$

$$(, ,) :: \mathrm{a} \to \mathrm{b} \to \mathrm{c} \to (\mathrm{a}, \mathrm{b}, \mathrm{c})$$

Podobně je možné rozšířit na trojice, čtveřice, ...

Knihovny jsou nejvíce nachystány na dvojice a trojice, ale doplnění manipulačních funkcí i na další n-tice je pro programátora však rutinou. Při přístupu ke složkám n-tice je přístup ke všem členům rovnocenný, nicméně závazný — dvojici není možné použít na místě trojice, stejné tak jako trojice celých čísel a trojice znaků není to samé.

Výklad

3.3 Funkce

Jak již vyplývá z typů datových konstruktorů, tak typ každé funkce je tvaru `f :: a -> b` (všimněte si, jak se zapisuje aplikace v typu — šipka vpravo — v textu programu). Operátor aplikace je taktéž zprava asociativní, takže např. pro konstruktor neprázdného seznamu, pokud

$$(:) \ :: \ a \to b \qquad (1)$$

a my víme, že

$$(:) \ :: \ \mathtt{a} \to [\mathtt{a}] \to [\mathtt{a}] \qquad (2)$$

potom

$$a_{(1)} \ = \ a_{(2)} \ \wedge \ b_{(1)} \ = \ ([a_{(2)}] \to [a_{(2)}])$$

(indexy jsou užity jen pro ujasnění, ale jinak se nepíší a `a` pro výraz (1) je jiné, než pro výraz (2). Pokud by vám to dělalo problémy, pojmenujte si typové proměnné na každé straně jinak a je po problému.

Pokud bychom pokračovali v rozvíjení typu pro konstrukci seznamu dále, tak máme

$$((:) \ x) \ :: \ a \to b$$

a konkrétně víme, že

$$((:) \ x) \ :: \ [\mathtt{c}] \to [\mathtt{c}]$$

a tedy že

$$a = [c] \wedge b = [c]$$

Řešený příklad

Zadání: Pro předepsané aplikace funkcí rozhodněte o výsledném typu, případně korektnosti aplikace.
Řešení:

- `f (1 :: Int) True`
  `f :: a -> b -> a`

  Výsledek: `(f 1 True) :: Int`
- `f (1 :: Int) (x :: Bool)`
  `f :: a -> a -> a`

  Výsledek: nelze aplikovat
- `f True [False,False]`
  `f :: a -> [a] -> [a]`

  Výsledek: `(f True [False,False]) :: [Bool]`
- `f (x :: [b]) (y :: [b])`
  `f :: a -> [a] -> a`

  Výsledek: nelze aplikovat

Pro definice funkcí v programu se používá zásadně dvou prvků:

- Rovnic
- Unifikace vzorů (pattern matching)

Hypotetickou funkci tedy lze zapsat takto:

```
f <vzor11> ... <vzor1N> = <pravá_strana1>
f <vzor21> ... <vzor2N> = <pravá_strana2>
  ...
f <vzorM1> ... <vzorMN> = <pravá_stranaM>
```

Jako vzor lze použít *proměnnou*. Potom definice funkcí pro sčítání a druhou mocninu budou vypadat takto: proměnná jako vzor

```
add x y = x + y
square x = x * x
```

Jelikož se jedná o unifikaci vzorů, tak především datové konstruktory lze uplatnit v definici funkcí. Nejjednodušším datovým konstruktorem jsou *konstanty*: konstanta jako vzor

```
not True = False
not False = True

sgn 0 = 0
sgn n = if n<0 then -1 else 1
```

Na definici funkce `sgn` je vidět, že je možné kombinovat různé typy vzorů pro jednotlivé definice téže funkce (pokud je lze logicky kombinovat). Krom toho spatřujeme i užití příkazu `if-then-else` v „tradičním“ pojetí. Na rozdíl od imperativních jazyků však tato konstrukce musí mít vždy obě části (then i else). V případě selhání podmínky by totiž neměla funkce definován výsledek.

Důležité je také vědět, že pořadí (shora dolů),v jakém jednotlivé definiční řádky pro funkci uvedeme, je potom použito při vyhledání vzoru pro unifikaci. Nejjednodušší pravidlo je takové, kdy jednotlivé předpisy pro vzory se vzájemně vylučují (definice funkce `not`). To však často není možné, nebo výhodné. Potom, pochopitelně, musíme nejobecnější vzor (takový, co pokrývá i další případ, či případy, jako u funkce `sgn`) dát až na konec. Případ, kdy naše funkce nemá být pro určitou skladbu definována vůbec, sice řeší překladač za nás tak, že sám dogeneruje tento chybějící vzor a potom k němu vyvolání chybového hlášení, nicméně je vhodné toto udělat ve vlastní režii, jakožto ukázku dobrých programátorských mravů.

seznam jako vzor

Dalším vzorem, z již zmíněných typů, je *seznam*. Pro něj je možné, krom kombinace s proměnnou, využít vzor jak pro prázdný seznam, tak pro neprázdný. Obojí demonstrujme na funkci pro délku seznamu:

```
length [] = 0
length (x:xs) = 1 + length xs
```

Na příkladu jsou zajímavé dvě věci. Vzor pro neprázdný seznam je uzavřen do závorek, které však nepatří k tomuto vzoru, ale jsou zde z důvodu priorit, aby vzor složily. Pokud by tam chyběly, tak by překladač místo jednoho parametru nalezl parametry 3 (proměnná, operátor, proměnná). Druhým důležitým prvkem je naznačení priorit. Aplikace funkce na parametr(y) má nejvyšší prioritu. Ostatní operátory vždy nižší. Proto také nejdříve dojde ze zjištění zbytku délky seznamu a teprve potom o zvýšení o 1.

Pozn.: Symboly pro pojmenování vzoru u neprázdného seznamu vycházejí z angličtiny, kdy množné číslo se z jednotného utvoří přidáním „s“ na konec slova.

n-tice jako vzor

N-tice jsou posledním typem, který jsme ještě nezmínili při jeho užití jako vzoru v definici funkce:

```
plus (x,y) = x+y
swap (x,y) = (y,x)
```

Kromě proměnných, konstant a/či datových konstruktorů umožňuje jazyk Haskell specifikovat ještě vzory typu $n + k$, kde n je proměnná a k celočíselná konstanta. Nejlépe vše opět demonstruje následující příklad:

vzor n+k

```
fact 0 = 1
fact (n+1) = (n+1) * fact n
```

Závorky opět hrají roli prioritní a nejsou samotnou součástí vzoru.

Pro zjednodušení práce a konečně i pro urychlení výpočtu je možné na levé straně definic funkcí, tj. v části vzorů ještě využít dvě zkratkové vlastnosti.

- pojmenování části vzoru
- anonymní proměnné

První z nich umožňuje manipulaci s jednotlivými částmi i celkem nějakého vzoru, druhá umožňuje užití jednoho a téhož označení nepoužitých proměnných v místě vzoru, kde musíme vyznačit přítomnost složky datové struktury, avšak s její hodnotou nehodláme na pravé straně definice pracovat — toto je zvláště výhodné jednak z hlediska softwarového inženýrství, jednak z toho důvodu, že v rámci jednoho definičního řádku se musejí jména proměnných na levé straně lišit. Obě tyto vlastnosti jsou patrné z následujících příkladů:

```
merge [] l2 = l2
merge l1 [] = l1
merge l1@(x:xs) l2@(y:ys) =
   if x<y then x:merge xs l2 else y:merge l1 ys

hd (x:_) = x

ziphd (x:_) (y:_) = (x,y)
```

Jednou z možností, jak větvit výpočet v jazyku Haskell, je užití strážených rovnic. Jeden charakter vzorů je dále doplněn o predikáty, které vybírají další podvarianty. Ukažme si to na již dříve definovaných funkcích:

strážené rovnice

```
fact n | n < 2     = 1
       | otherwise = n * fact (n-1)

sgn n | n < 0     = -1
      | n > 0     = 1
      | otherwise = 0
```

Všimněte si slova `otherwise`. Není to klíčové slovo jazyka, jak by se mohlo zdát, jak tomu je např. u `if`, `then` a `else`. Je to funkce, která je vždy pravdivá: `otherwise = True`. Pořadí jednotlivých stráží (guards) opět hraje klíčovou roli.

lokální funkce

I když doposud uvedený koncept je bohatý, tak se určitě neobejdeme bez lokálních funkcí. Pro jejich definici máme v zásadě dvě možnosti. Nejdříve uvádím tu, která by měla být preferována:

```
sumsqr x y = xx + yy
  where
    sqr a b = a*b
    xx = sqr x x
    yy = sqr y y
```

Klíčové slovo `where` uvozuje blok lokálních definic, je možno jej vnořovat. Odsazení a dodržení charakteru formátování je důležité, i když se nemusí přesně shodovat s uvedeným příkladem — `where` by mělo být odsazeno od definičního řádku funkce, lokální definice by měly být odsazeny v něm.

Druhou možností je užití tohoto zápisu:

```
sumsqr' x y =
  let
    sqr a b = a*b
    xx = sqr x x
    yy = sqr y y
  in
    xx + yy
```

Oba se do jisté míry liší, což lze zjistit při důkladném prozkoumání, ale to nechám na čtenáři a dodatečné literatuře. Pro naše účely je vysvětlení a způsob užití dostatečné.

Pozn.: Připojení apostrofu ke jménu funkce/identifikátoru je v jazyku Haskell naprosto korektní a běžné. Podobně jako v matematice, informatice, apod.

prefix, infix

Užití funkcí a operátorů je možné dvojím způsobem i když pro každý typ je jedna forma typická. Jedním způsobem je infixový a druhým prefixový zápis vzhledem k parametrům. Na další ukázce demonstrujme takové možné užití a záměny:

```
  mod :: Int -> Int -> Int
  (+) :: Int -> Int -> Int
Infix
    5 `mod` 3
    1 + 1
```

```
Prefix
    mod 5 3
    (+) 1 1
```

Pro infixový zápis funkce je třeba jméno vložit do zpětných apostrofů. Typicky je toto užití možné vidět u funkcí se dvěma parametry uvnitř výrazu, lze ho použít však i na funkce s více parametry, avšak tím již nic nezískáme.

Jak naznačuje již λ-kalkul a částečně to bylo zmíněno výše, tak funkci není nutné v její aplikaci saturovat — dodat ji tolik parametrů, kolik je jen schopná spotřebovat. Výsledkem *částečné aplikace* je tak nová funkce. Jazyk Haskell toto umožňuje i pro operátory v kombinaci s parametry z obou stran:

částečná aplikace

```
add x y = x+y

inc = add 1
inc' = (+) 1
inc'' = (1+)
```

A aby toho nebylo dosti, tak lze užít i λ-abstrakce, které se zapíší takto:

λ-abstrakce

```
inc''' = \x -> x+1
```

Přitom λ je nahrazena znakem „\" (zpětné lomítko, backslash) a tečka je nahrazena šipkou (dvojznak — mínus, je větší).

Pojmy k zapamatování

V této výkladové části jsme se seznámili s vybranými vestavěnými typy v jazyku Haskell a také s tím, jak definovat funkci. Většina pojmů se tedy pojí s těmito okruhy: celá čísla, logické hodnoty, desetinná čísla, znaky, seznamy a n-tice. Pro práci s funkcemi je důležité znát pojem unifikace vzorů, typy, unifikace typů, jak pracovat a co lze pro unifikaci vzorů použít, strážené rovnice, částečnou aplikaci a λ-abstrakce v Haskellu.

Kromě teoretických znalostí pojmů je zejména důležité je umět používat při tvorbě funkcionálních programů. K tomu by mělo vše i směřovat.

Závěr

Tato kapitola si kladla za cíl uvést vás do funkcionálního jazyka Haskell tak, abyste byli schopni psát jednoduché programy. Nyní byste měli vědět, jakým způsobem se tvoří program v jazyku Haskell, jak se definují funkce, jaké se k tomu používají notace, jaké jsou základní typy, se kterými můžete pracovat. Pokud jste si zkoušeli vše v nějakém interpretu, což doporučuji i zpětně, tak byste si také měli poradit s jednoduchými úlohami, kdy stačí výsledek funkce v podobě, jak jej zobrazí interpret jazyka Haskell.

Úlohy k procvičení:

1. Definujte funkci, která spočte n-tý člen Fibonacciho posloupnosti. Vhodně ošetřete nekorektní vstupy, zamyslete se nad časovou složitostí, navrhněte optimalizaci.
2. Vytvořte funkci `spoj :: [a] -> [a] -> [a]`, která za sebe spojí dva seznamy.
3. Vytvořte funkci `obrat :: [a] -> [a]`, která v zadaném seznamu obrátí pořadí jeho položek. *Pozn.: Využijte funkci* `spoj`*.*
4. Projděte si základní knihovnu jazyka Haskell, která se jmenuje Prelude.hs a je součástí běžných distribucí interpretů i překladačů jazyka (hugs, ghci, winhugs). Podívejte se na obraty, snažte se porozumět obratům. Které funkce/operátory odpovídají funkcím `spoj` a `obrat`?

Klíč k řešení úloh

1. Základní definici Fibonacciho funkce jistě najdete sami, co se týká optimalizace, tak se zamyslete, kolik členů v daný moment potřebujete na výpočet hodnoty následujícího a jestli tedy lze zamezit opakování výpočtů.
2. Uvažte, že je-li první seznam prázdný, tak druhý je výsledek. No a je-li první neprázdný, tak spojený bude určitě začínat jeho prvním prvkem, za který se připojí spojení prvního a celého druhého.
3. První prvek neprázdného seznamu se připojí za obrácený zbytek seznamu.

Další zdroje

Řadu informací podaných v této kapitole najdete v literatuře citované níže, nebo přímo v nápovědě, či doprovodné dokumentaci, či manuálech, které jsou na stránkách věnovaných jazyku Haskell.

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

Kapitola 4

Funkce vyššího řádu

V této kapitole se zaměříme hlouběji na možnosti a vlastnosti funkcí v jazyce Haskell, které jsme zavedli v kapitole předchozí. Bude se zejména jednat o práci s rekurzí a ukázek funkcí vyššího řádu.

Čas potřebný ke studiu: 3 hodiny 10 minut.

Cíle kapitoly

Cílem kapitoly je prezentovat další možnosti tvorby funkcí v jazyku Haskell, zejména funkcí vyššího řádu. Tvorba a užití takových funkcí patří k nutnostem funkcionálního návrhu a programování. Jedná se tedy také o klíčovou kapitolu.
V příkladech bude i nadále prezentována zjednodušená syntaxe jazyka, která je závislá na poměrně jednoduchých pravidlech rozmístění textu programu.

Průvodce studiem

Tato kapitola odhalí pokročilé a vysoce účinné „zbraně" jazyka Haskell. I když se jedná o pokročilé konstrukce, tak jde o nosné prvky jazyka, které se používají takřka ve všech programech. Neustále platí, že každou novou vědomost je nutné si okamžitě ověřit prakticky, aby si čtenář osvojil i způsob, *jakým vyhodnocení jazyka Haskell pracuje*.
Ke studiu této kapitoly je naprosto nezbytné mít k dispozici počítač s instalovaným interpretem jazyka Haskell. Je možné mít i kompilátor, ale práce v něm, zejména pro začátečníky, je mnohem, ale mnohem náročnější. Na stránkách jazyka (www.haskell.org) je možné najít interpret hugs i jeho klon čistě pro MS Windows, který se jmenuje winhugs.

Obsah

Výklad

4.1 Rekurze

Jedinou možností, jak vyjádřit v jazyku Haskell opakování, je užití rekurze. I když tento koncept by měl být čtenáři znám, tak určitě nezaškodí pojmy znovu připomenout, aby bylo zřejmé, jaký typ konstrukcí vede k nejefektivnějším programům v jazyku Haskell.

Definice 4.1.1 *Funkci nazýváme* zpětně rekurzivní, *pokud po návratu z rekurzivního volání ještě probíhá další výpočet pro zisk výsledku:* **Definice!**

$$f(x) = e(f(x'))$$

K této definici existuje, dle očekávání, i doplněk.

Definice 4.1.2 *Funkci nazýváme* dopředně rekurzivní, *pokud rekurzivní volání je poslední část výpočtu:* **Definice!**

$$f(x) = f(e(x'))$$

I když jsou obě definice zásadní, tak nic neříkají o vhodnosti užití té které strategie. Intuitivně sice můžeme chápat, která je lepší, ale pro to, abychom dospěli k jednoznačnému závěru, tak je třeba ještě dále zpřesnit formu rekurze.

Definice 4.1.3 *Rekurzivní funkci budeme nazývat* lineárně rekurzivní, *pokud je v rekurzivní části právě jedno rekurzivní volání.* **Definice!**

K této definici se vztahuje důležitá věta, kterou uvádím bez důkazu (důkaz je nad rámec publikace a je možné jej najít v teoretické literatuře pojící se s tématem, respektive překladem funkcionálních jazyků).

Teorém 4.1.1 *Každou lineárně rekurzivní funkci lze převést na cyklus.*

I když je tato věta velmi důležitá, bohužel nic neříká o tom, jak to provést. Případně je-li to realizovatelné nějakou strojovou a efektivní formou. Nicméně je to značně navádějící v tom, jak psát efektivní program. Povšimněte si, že věta nic neříká o tom, jestli je rekurze dopředná, nebo zpětná!

Následující definice a věta nás posune ke kýženému cíli.

Definice!

Definice 4.1.4 *Funkce, která je dopředně a lineárně rekurzivní nazýváme* koncově rekurzivní.

Věta je opět uvedena bez důkazu, ze stejných důvodů, jako věta 4.1.1.

Teorém 4.1.2 *Každou koncově rekurzivní funkci lze jednoduše přeložit na efektivní cyklus (není třeba uchovávat stav nedokončeného výpočtu).*

Takže teď už by mělo být jasné, jaké konstrukce jsou pro tvorbu efektivních funkcionálních programů nejlepší. Je to však vždy možné takovou funkci zapsat? Někdy ne a často to nejde bez obtíží, ale je třeba se o to vždy pokusit.

Řešený příklad

Zadání: Definujte funkci `fib :: Int -> Int`, která vypočítá n-tý člen Fibonacciho posloupnosti. Užijte koncové rekurze.
Řešení:

```
fib :: Int -> Int
fib n = if n<0
        then error "fib: Negative input"
        else fib' 0 1 n
  where fib' k l 0 = k
        fib' k l n = fib' l (k+l) (n-1)
```

U řešeného příkladu si povšimněme několika věcí. Jednak je to vestavěná funkce `error :: String -> a`, která má jako parametr textový řetězec (text uzavřený v uvozovkách) a výsledný typ se unifikuje s libovolným typem (typová proměnná `a`). Tato funkce, je-li vyvolána, zastaví výpočet a vypíše zadaný text. Její typické použití je pro ošetření chybových stavů, nekorektních hodnot parametrů, apod. O typu `String` se dozvíte více v následující kapitole, zatím je to plně dostačující takto.

Dále za zmínku stojí uvedení typové signatury pro funkci `fib`. Doposud jsme to nedělali a ani tady bychom to nemuseli dělat. Systém sám odvodí typ funkce. Udělá to i v tomto případě, potom to porovná s naším. Pokud je náš shodný, nebo zužující (zpřísňující), použije náš. (Více o typech viz další kapitolu).

Výklad

4.2 Funkce vyššího řádu

Výklad v této kapitole začneme dvěma řešenými příklady, abychom dostatečně motivovali další úvahy a počínání.

Řešený příklad

Zadání: Definujte funkci `sumlist :: [Int] -> Int`, která sečte hodnoty všech prvků v zadaném seznamu celých čísel. Pro prázdný seznam je součet definován jako nulový.
Řešení: Typ funkce je dán a určuje, co máme vytvořit. Je dán dokonce i součet prázdného seznamu, takž můžeme hned zapsat:

```
sumlist [] = 0
```

Součet neprázdného určíme tak, že sečteme prvky seznamu, ze kterého první prvek odebereme, a hodnotu odebraného prvku, takže můžeme zapsat:

```
sumlist (x:xs) = x + sumlist xs
```

Tím jsme vytvořili rekurzivní vztah, máme definovanou i podmínku zastavení, takže můžeme zapsat celý výsledek:

```
sumlist :: [Int] -> Int
sumlist [] = 0
sumlist (x:xs) = x + sumlist xs
```

V druhém řešeném příkladu již nebude proveden tak podrobný rozbor úlohy, ale postup by byl v zásadě obdobný — řešení podmínky zastavení a rekurzivního vztahu.

Řešený příklad

Zadání: Definujte funkci `propojlist :: [[a]] -> [a]`, která spojí za sebe všechny seznamy obsažené v seznamu předaném jako parametr.
Řešení:

```
propojlist :: [[a]] -> [a]
propojlist [] = []
propojlist (xs:xss) = spoj xs (propojlist xss)
```

V příkladu jsme využili funkci `spoj` z úloh řešených na konci předešlé kapitoly. V knihovně Prelude.hs, která je typickou součástí interpretů jazyka Haskell se tato funkce vyskytuje jako operátor `++`, který je zprava asociativní. Dále tedy budeme používat tento operátor.

Nyní se ale podívejme na obě vytvořené funkce. Když se na ně podíváme, tak zjistíme, že pracují podle shodného schématu, které můžeme zapsat např. takto:

```
proc [] = k
proc (z:zs) = f x (proc zs)
```

V případě funkce `sumlist` dosadíme za `k` konstantu 0 a za `f` funkci `(+)`. V případě funkce `propojlist` to budou prázdný seznam `[]` a operace pro spojení dvou seznamů `(++)` (či „naše“ `spoj`).

Jistě vás napadá, že by bylo možné nějak tuto společnou vlastnost zakomponovat do jediné funkce, která by se potom jen parametrizovala. Ano, možné to je a to díky funkcím vyššího řádu, konkrétně v tomto případě díky funkci `foldr`, kterou je možné definovat takto:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Prvním parametrem je funkce, která kombinuje dva parametry do výsledky odpovídajícímu typu druhého parametru. Druhý parametr funkce `foldr` je právě hodnota typu výsledku funkce vkládané jako první parametr (výsledek, pokud je seznam prázdný) a třetím je seznam hodnot nad typem, hodnoty kterého přijímá funkce předávaná jako první parametr taktéž na místě prvního parametru.

Zkráceně se tedy jedná o kýžené parametrizovatelné zobecnění našich funkcí `sumlist` a `propojlist`, které jsme navrhli ve funkci `proc`. S využitím funkce `foldr` tedy můžeme psát:

```
sumlist' xs = foldr (+) 0 xs
propojlist' xss = foldr (++) [] xss
```

Dokonce, díky částečné aplikaci a znalosti η-redukce z lambda kalkulu, můžeme psát:

```
sumlist'' = foldr (+) 0
propojlist'' = foldr (++) []
```

Jiný, ne zcela nepodobný případ, kdy je třeba vytvořit funkce pracující na stejné bázi, navozuje další řešený příklad.

Řešený příklad

Zadání: Definujte funkce `squareAll :: [Int] -> [Int]` a `lengthAll :: [[a]] -> [Int]`, kdy první nahradí celočíselné prvky seznamu jejich druhými mocninami a druhá nahradí v seznamu seznamů každý vnořený seznam jeho délkou.

Řešení:

```
squareAll :: [Int] -> [Int]
squareAll [] = []
squareAll (x:xs) = (x*x) : squareAll xs

lengthAll :: [[a]] -> [Int]
lengthAll [] = []
lengthAll (xs:xss) = (length xs) : lengthAll xss
```

Opět můžeme vysledovat určité schéma v daných funkcích. Pro přehlednost si opět zkusme takovou schématickou funkci definovat:

```
proc' [] = []
proc' (z:zs) = f z : proc' zs
```

V našem schématu potom stačí za `f` dosadit buďto funkci realizující umocnění na druhou (např. jako lambda abstrakci (`\x -> x*x`) anebo funkci pro zjištění délky seznamu `length`.

Tušíte, že opět směřuji k funkci vyššího řádu, která je, stejně jako funkce `foldr`, typickou součástí základních knihoven funkcionálních jazyků vyššího řádu, což je i jazyk Haskell. V tomto případě se však funkce jmenuje `map` a lze ji definovat např. takto:

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Předchozí příklad potom můžeme vyřešit, při maximální úspoře psaných znaků díky využití částečné aplikace, takto:

```
squareAll' = map (\x -> x*x)
lengthAll' = map length
```

Další typ abstrakce si budeme demonstrovat rovnou na výsledné funkci vyššího řádu, která je definována takto:

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs) | p x = x:xs'
                | otherwise = xs'
                  where xs' = filter p xs
```

Jejími parametry jsou predikát, přesněji funkce, která očekává jediný parametr a poté vrací hodnotu typu `Bool`, a seznam. Výsledkem je seznam, který obsahuje pouze ty hodnoty, pro které je predikát splněný, navíc v původním pořadí. Využití budeme demonstrovat na dalším řešeném příkladu.

Řešený příklad

Zadání: Definujte funkce `getOdd` a `getLessThen`, kdy první vybere ze seznamu jen lichá čísla (seznam je nad celými čísly) a druhá dostane jako parametr číslo a seznam (v uvedeném pořadí) a vrátí ze vstupního seznamu jen čísla menší než číslo zadané.

Řešení:

```
getOdd = filter odd
getLessThen x = filter (<x)
```

Řešení využívá jednak vestavěné funkce `odd` se zřejmou funkcí, druhak částečné aplikace v kombinaci s infixovým operátorem — povšimněte si pozice `x` ve výrazu `(<x)`, zkuste polohu změnit a otestovat funkci, podobně lze využít i další operátory.

Výklad

4.3 Generátory seznamů

Seznamy hrají ve funkcionálních jazycích poměrně důležitou roli, jak je ostatně patrné z dosavadního výkladu. Jazyk Haskell proto umožňuje velmi kompaktní zápis pro seznamy konstruované na základě jistých pravidel, která specifikují prvky daného seznamu. Jakýmsi předobrazem je specifikace množiny známá z matematiky, např. množina všech sudých přirozených čísel (operace „modulo" značí zbytek po dělení):

$$\{\ x \mid x \in \mathbf{N},\ x \text{ modulo } 2 = 0\ \}$$

V jazyku Haskell potom pro definici seznamu používáme šablony, která má podobný charakter:

$$[\ \text{výraz} \mid \text{kvalifikátory}\ ]$$

Mezi kvalifikátory počítáme:

- generátory — zápis tvaru `vzor <- výraz`, kdy `výraz` definuje seznam, z něhož jsou postupně vybírány prvky a unifikovány se vzorem `vzor`;

- filtry — booleovský výraz nad již definovanými proměnnými, predikát.

Nejjednodušším příkladem je zápis, který definuje znovu tentýž seznam (v praxi ho nikdy nevyužijeme):

```
[x | x <- xs]
```

takže platí $[x \,|\, x \leftarrow xs] \equiv xs$.

Řešený příklad

Zadání: S využitím generátoru seznamů najděte zápis, který odpovídá využití funkcí `map` a `filter`.
Řešení:

```
[f x | x <- xs]       ~  map f xs
[x | x <- xs, p x]    ~  filter p xs
```

Generátorů v generátorech seznamů, může být obecně více, stejně tak výraz, který definuje prvky seznamu, nemusí zdaleka záviset na všech (dokonce na žádné) proměnných definovaných v generátorech. Uvažme tyto příklady:

```
[(m,n) | m <- [1..3], n <- [4,5]] =
              [(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]

[0 | n <- [1..10]] = [0,0,0,0,0,0,0,0,0,0]
```

Z prvního je patrné, v jakém pořadí dochází, pokud zřetězíme generátory, k výběru položek z generátorů. Druhý ukazuje, že výraz definující prvky výsledného seznamu zdaleka nemusí být závislý na generátoru.

V ukázce jsme také použili doposud neznámý zápis pro seznam. Tento zápis využívá operátoru „..“ a vestavěných vlastností jazyka Haskell, které umožňují pracovat s posloupnostmi pro definici seznamů zkrácenou formou. Princip funkce by měly dostatečně vysvětlit následující ukázky a vaše vlastní experimenty.

```
[1..10] = [1,2,3,4,5,6,7,8,9,10]
[1,3..10] = [1,3,5,7,9]
[10..1] = []
[10,9..1] = [10,9,8,7,6,5,4,3,2,1]
```

Jelikož jazyk Haskell využívá tzv. lazy strategie vyhodnocení, je možné definovat i potenciálně nekonečné datové struktury. Tato strategie totiž uvádí do absolutní perfekce strategii vyhodnocení označovanou „vyhodnocení v případě potřeby“ (call-by-need). Na rozdíl od této

lazy strategie

strategie, kdy výraz, častěji parametr funkce, nemusí být vyhodnocen sice vůbec (není v těle jeho hodnota potřeba), ale může být vyhodnocen i vícekrát (hodnota je potřeba na více místech), strategie lazy vyhodnocení vyhodnocuje výraz maximálně jedenkrát. Jakmile je hodnota výrazu získána, zůstává uchována pro případné pozdější užití.

Pro definici potenciálně nekonečných seznamů je potom možné využít všechny známé funkce a zápisy, takže i zkrácený zápis posloupností:

```
[1..] = [1,2,3,4,5, ...
[2,4..] = [2,4,6,8, ...
[1,-1,..] = [1,-1,-3,-5, ...
```

I když je seznam nekonečný, tak není možné, zcela pochopitelně, všechny jeho prvky vyčíslit. Díky strategii vyhodnocení, pokud je náš program správně napsán, tak můžeme pracovat jen s prvními několika prvky seznamu a zbytek se nevyhodnotí, přitom se nemusíme dopředu omezovat na nějaký konkrétní strop pro délku seznamu,

V základní knihovně existuje funkce `take :: Int -> [a] -> [a]`, které ze seznamu vezme prvních n prvků, pokud jich tolik vůbec je, a vrátí je jako výsledek. Pomocí ní lze demonstrovat funkčnost strategie lazy vyhodnocení:

```
take 5 [1..] = [1,2,3,4,5]
take 10 [-1,-3..] =
        [-1,-3,-5,-7,-9,-11,-13,-15,-17,-19]
```

Nezbývá, než si to vyzkoušet na dalších příkladech.

Pojmy k zapamatování

V této výkladové části jsme se seznámili s pojmy užívanými ve spojitostí s rekurzí: dopředná, zpětná, lineární, koncová. Dále jsme zavedli pojem funkce vyššího řádu. V závěru jsme se potom věnovali generátorům seznamů a strategii lazy vyhodnocení.
Kromě znalosti těchto pojmů jako takových je zejména důležité je využít při tvorbě programů ve funkcionálních jazycích a na základě toho, co představují, umět konstruovat programy zejména v jazyku Haskell.

Závěr

Cílem kapitoly bylo rozvést základní a principiální koncepty funkcionálního programování tak, abyste byli schopní tvořit prakticky všechny náročnější programy a přitom využívali pokročilé koncepty. Měli byste tak zvládat prakticky všechny důležité techniky a pro dosažení nezbytné praktické zručnosti tak zbývá už jen jedna kapitola, která bude prezentovat definici a užití vlastních datových typů a užití tzv. typových tříd.

Úlohy k procvičení:

1. Definujte celou Fibonacciho posloupnost s využitím generátoru seznamů. Znovu se zamyslete nad optimálním řešením (využijte operátoru `(!!) :: [a] -> Int -> a`, který ze seznamu vybírá jeho $n+1$ prvek).
2. S využitím generátorů seznamů definujte funkci quicksort nad seznamy, jako medián užijte první prvek seznamu.
3. Vytvořte seznam všech prvočísel.
4. Zamyslete se na nad funkcí `foldr`, jakou rekurzi používá? Nešlo by definovat k ní doplňkovou, říkejme jí `foldl`, která využije opačný typ rekurze? Kdy je možné tyto funkce zaměnit v užití, kdy naopak ne?

Klíč k řešení úloh

1. Nápověda je shodná s tou v předchozí kapitole, řešením není užití efektivní funkce pro výpočet n-tého členu Fibonacciho posloupnosti. Prvních x členů musíte zadat explicitně, s tím se počítá. Kolik je to x?
2. Generátory seznamů užijte pro rozdělení seznamu na dva podseznamy, co se mají dále řadit.
3. První člen seznamu musíte zadat explicitně. Další prvočíslo vždy určíte na základě těch již vypočtených dle známých pravidel.
4. Nahlédněte do základní knihovny Prelude.hs.

Další zdroje

Řadu informací podaných v této kapitole najdete v literatuře citované níže, nebo přímo v nápovědě, či doprovodné dokumentaci, či manuálech, které jsou na stránkách věnovaných jazyku Haskell.

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

Kapitola 5

Datové typy

Tato kapitola završí přehled všech vlastností jazyka — zaměříme se na tvorbu vlastních uživatelských typů, zjistíme, co je typová třída a k čemu se využívá.

Čas potřebný ke studiu: 5 hodin 50 minut.

Cíle kapitoly

Cílem kapitoly je objasnit tvorbu uživatelských typů, typových synonym a zejména pojem typových tříd, které umožňují přetěžování operátorů a funkcí. Jelikož uživatelské datové typy jsou klíčovou vlastností v programování vůbec, tak se tedy jedná také o velmi důležitou kapitolu.
V příkladech bude i nadále prezentována zjednodušená syntaxe jazyka, která je závislá na poměrně jednoduchých pravidlech rozmístění textu programu.

Průvodce studiem

Tato kapitola odhalí způsob, jakým je řešené přetěžování u jazyka Haskell a nastíní tedy možnosti plného využití uživatelských datových typů. I když se jedná o pokročilé konstrukce, tak jde o nosné prvky jazyka, které se používají takřka ve všech programech. Neustále platí, že každou novou vědomost je nutné si okamžitě ověřit prakticky, aby si čtenář osvojil i způsob, *jakým vyhodnocení jazyka Haskell pracuje*.
Ke studiu této kapitoly je naprosto nezbytné mít k dispozici počítač s instalovaným interpretem jazyka Haskell. Je možné mít i kompilátor, ale práce v něm, zejména pro začátečníky, je mnohem, ale mnohem náročnější. Na stránkách jazyka (www.haskell.org) je možné najít interpret hugs i jeho klon čistě pro MS Windows, který se jmenuje winhugs.

Obsah

Výklad

5.1 Typové třídy

Doposud jsme užívali polymorfismus na úrovni vstupu různých seznamů do jedné funkce, např. funkce pro délku seznamu `length :: [a] -> Int`. Což ale třeba případ, kdy chceme otestovat, zda se v seznamu čísel nachází dané číslo? Takovou funkci můžeme definovat např. takto:

```
elemI :: Int -> [Int] -> Bool
elemI _ [] = False
elemI n (x:xs) = if x==n then True else elemI n xs
```

Co třeba ale stejná funkce pro vyhledání dvojice celé číslo a booleovská hodnota v seznamu stejných typů? Nabízí se napsat:

```
elemPIB :: (Int,Bool) -> [(Int,Bool)] -> Bool
elemPIB _ [] = False
elemPIB p (x:xs) = if x==p then True else elemPIB n xs
```

Jenže, potom by operátor `==` musel být schopen jednou pracovat s celými čísly a jednou s dvojicí a to ještě vystavěnou nad celým číslem a booleovskou hodnotou. Víme, že jazyk Haskell je silně typovaný, takže nějaký „trik“ či „podvod“ se nedá uplatnit.

Když se však podíváme do standardní základní knihovny jazyka Haskell (Prelude.hs), tak zjistíme, že funkce `elem` tam je a když ji vyzkoušíme, tak bude fungovat v obou prezentovaných případech. Když si však necháme vypsat její typ, tak zjistíme, že je typu `elem :: Eq a => a -> [a] -> Bool`. Text mezi `::` a `=>` říká, že typ a musí náležet do typové třídy `Eq` a potom je typ funkce `elem` signatury `a -> [a] -> Bool`.

Typová třída `Eq` je definována takto:

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

    x == y      = not (x/=y)
    x /= y      = not (x==y)
```

Definuje tedy dva operátory s příslušnou typovou signaturou, řekněme jakousi šablonou, či maskou, které musejí všechny instance této typové třídy odpovídat. Jedná se o operátory pro rovnost a nerovnost. Jakou součást typové třídy je i bázová definice funkce jednoho operátoru na základě jiného. Určitě je vám jasné, že tato „definice kruhem“ má nějaký háček. Ano má. Vždy, pro každou instancí typové třídy, je nutné alespoň jeden operátor definovat explicitně a pro druhý užít bázové definice.

Pro celá čísla je potom instance typové třídy Eq definována takto:

```
instance Eq Int where
    (==) = primEqInt
```

Jak vidíme, je definován pouze jeden operátor a jeho definice je navázána na vestavěnou funkci (vazba na základní primitiva, mimo rozsah této publikace, více na www.haskell.org). Mohla by však obsahovat svou vlastní definici.

Když bychom se podívali na definici operace „je rovno“ pro n-tice, tak zjistíme, že celý oddíl je v poznámce, protože se jedná o vazbu na základní primitiva. Tato definice by však mohla být učiněna explicitně např. takto:

```
instance (Eq a, Eq b) => Eq (a,b) where
    (x,y) == (xx,yy) = if x==xx then y==yy else False
```

Zde vidíme využití (hned dvojnásobného) odkazu na příslušnost pro nějaký typ do určité třídy (zde do stejné třídy Eq) a na základě této příslušnosti následuje rekurzivní definice pro operátor rovnosti pro další typ — n-tici.

Pro seznam je například v knihovně Prelude.hs uvedena tato definice rovnosti dvou seznamů:

```
instance Eq a => Eq [a] where
    []     == []     =  True
    (x:xs) == (y:ys) =  x==y && xs==ys
    _      == _      =  False
```

Jedinou neznámou by mohl být operátor &&, který reprezentuje logickou konjunkci (logické „a“).

5.2 Odvozené typové třídy

Existence přetížených operátorů pro rovnost a nerovnost je jistě pěkná věc, ale operátory jako „je větší než“ apod. by si určitě zasloužily stejnou péči. Proč nejsou přímo ve třídě Eq, když k sobě mají tak blízko? Ač je to spíše řečnická otázka, tak odpovídám proto, že umět něco porovnat na rovnost ještě neznamená, že něco uspořádám od nejmenšího k největšímu (např. body v rovině).

Nicméně opakovat definici rovnosti by nemuselo být nikterak výhodné ani praktické. Naštěstí existují *odvozené typové třídy*. Třída zahrnující operátory pro porovnání na uspořádání je definována takto:

odvozené typové třídy

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

    compare x y | x==y      = EQ
                | x<=y      = LT
                | otherwise = GT

    x <= y  = compare x y /= GT
    x <  y  = compare x y == LT
    x >= y  = compare x y /= LT
    x >  y  = compare x y == GT

    max x y | x <= y      = y
            | otherwise   = x
    min x y | x <= y      = x
            | otherwise   = y
```

Vidíme, že její definice je poměrně rozsáhlá a ač to není patrné (jen z dokumentace), tak stačí definovat implementaci pro operátor `<=`, nebo funkci `compare`, která vrací jako výsledek hodnoty typu, který přímo pojmenovává (anglické zkratky) vztah mezi hodnotami (je větší, je menší, ...).

My si však zejména budeme všímat odvození, kdy třída `Ord` je definovaná na základě existence třídy `Eq`. To platí i pro instance, tedy musíme nejdříve definovat pro nějaký typ rovnost a teprve potom uspořádatelnost jeho hodnot. V takovém případě můžeme hovořit o jednoduchém odvození, či, chcete-li, dědění vlastností. Pro instance již odvození z jiných tříd však neuvádíme.

jednoduchá dědičnost

Jak je asi zřejmé, tak existuje i násobné odvození při definici typové třídy. Jako ukázku uvádíme hlavičku takové definice z knihovny Prelude.hs:

vícenásobná dědičnost

```
class (Real a, Fractional a) => RealFrac a where
    ...
```

Počet předků není nijak limitován. Pochopitelně systém závislostí musí tvořit acyklický orientovaný graf (DAG). Při implementaci instance opět musí platit, že implementace instancí, z nichž odvozuji, jsou již dány. Také to, že v instanci závislosti již neuvádím platí i v tomto případě.

Výklad

5.3 Uživatelské datové typy

Síla typových tříd, jakožto i síla jakéhokoliv programovacího jazyka, je do značné míry určena možností tvorby a užití vlastních datových typů. Mezi nejjednodušší vlastní typy v jazyku Haskell patří *typová synonyma*. Vytvářejí se tak, že dáme nové jméno již existujícímu typu, či složenině, např.:

typová synonyma

```
type ComplexF = (Float, Float)
type Matrix a = [[a]]
```

Jak je vidět, tak lze pojmenovat zcela konkrétní typ (`ComplexF`), nebo je možné typ parametrizovat (`Matrix`). Parametrizace snese více jak jeden parametr.

V předchozích kapitolách se objevil literál pro textový řetězec. Bylo napsáno, že tento typ se jmenuje `String`. Nyní je možné uvést jeho definici:

```
type String = [Char]
```

Řetězec je tedy reprezentován jako seznam znaků a podle toho je možné s ním i zacházet.

Kromě typových synonym je možné vytvářet i jednoduché datové typy. Od definice typového synonyma se neliší nijak výrazně. Kromě klíčového slova je to nutnost datového konstruktoru na začátku popisu typu. Pokud upravíme předchozí ukázku, tak můžeme psát např.:

jednoduché datové typy

```
newtype ComplexC = ReIm (Float,Float)
newtype MatrixC a = Matrix [[a]]
```

Kromě rozdílů v definici a výsledku se liší i použití. Napíšeme-li `(3.5, 6.3)`, tak se jedná o dvojici desetinných čísel, přesněji, interpret jazyka Haskell odvodí tento typ `(3.5,6.3) :: (Fractional a, Fractional b) => (b,a)`. Pokud tedy chceme, aby nějaký literál měl námi předepsaný typ, tak to musíme explicitně vyjádřit, musíme napsat `(3.5,6.3) :: ComplexF`. Často tedy použijeme typové synonymum spíše pro explicitní otypování funkce, než u literálů. Naproti tomu jednoduché datové typy už mají svůj konstruktor, takže pokud napíšeme v našem programu `ReIm (3.5, 6.3)`, bude tento literál zcela jistě typu `ComplexC`. Není nutné explicitně typovou signaturu přidávat.

Pokud však napíšeme v příkazovém řádku interpretu zápis konstanty typu `ComplexC`, tak se maximálně dozvíme toto:

```
ERROR - Cannot find "show" function for:
*** Expression : ReIm (3.5,6.3)
*** Of type    : ComplexC
```

Interpret jazyka nám tím říká, že neví, jak takovou konstantu zobrazit. Nemá pro ni funkci `show`, která je z typové třídy `Show` a která stojí za zobrazováním hodnot všech typů, pokud to tedy je definováno.

Jednou z možností, jak instanci vytvořit, je udělat to plně ve vlastní režii. K tomu je třeba implementovat funkci `showsPrec :: Show a => Int -> a -> ShowS`, kde první parametr je údaj o prioritě (pro doplnění závorek) a druhý jsou zobrazovaná data. Výsledkem je opět funkce, jelikož typ `ShowS` je definován jako `type ShowS = [Char] -> [Char]`. Tato funkce by měla být za vzniknutý textový řetězec (vytvořený převodem hodnoty daného typu na text) připojit nějaký další text, který ho třeba obklopuje. Jedna možnost je tato:

```
instance Show ComplexC where
  showsPrec p (ReIm (f1,f2)) =
               (\r -> "ReIm " ++ sf1 ++ " " ++ sf2 ++ r)
    where sf1 = show f1
          sf2 = show f2
```

Potom, napíšeme-li na příkazový řádek interpretu `ReIm (1.1,2.2)`, tak nám opáčí `ReIm 1.1 2.2`.

Tato definice je však poměrně neefektivní a ne příliš „čistá“. Lepší by asi bylo použít tuto vlastní konstrukci se stejnými výsledky, ale efektivnějším vyhodnocením:

```
instance Show ComplexC where
  showsPrec p (ReIm (f1,f2)) =
                  ("ReIm " ++) . sf f1 . (' ':) . sf f2
    where sf = showsPrec 9
```

Pozn.: Definici operátoru `(.)` *najdete v Prelude.hs, ostatní potom v literatuře k jazyku Haskell, jelikož výklad této ukázky je mimo rozsah publikace.*

V řadě případů však postačí nejjednodušší řešení, kdy definujeme typ takto: deriving

```
  newtype ComplexC = ReIm (Float,Float)
    deriving Show
```

Pochopitelně, že potom je výstup přesně takový, jak je zadán v definici typu, nemůžeme jej dále ovlivnit, takže potom interpret jazyka Haskell na náš literál odpoví pro tento příklad takto:

```
Main> ReIm (3.5, 6.3)
ReIm (3.5,6.3)
```

Překladač sám odvodí, tou nejjednodušší cestou, implementaci pro třídu `Show`.

Tento postup lze zvolit pro typové třídy `Eq`, `Ord`, `Ix`, `Enum`, `Read`, `Show` a `Bounded`. O tom, kdy to má význam, či je možné si nechat toto začlenění vytvořit systémem, se dočtete v literatuře k jazyku Haskell, neboť to sahá za rámec této publikace. Pro náš případ stačí, že bychom mohli uvést:

```
newtype ComplexC = ReIm (Float,Float)
  deriving (Show, Read, Eq)
```

komplexní datové typy

Nejpokročilejší formou uživatelských datových typů jsou tzv. *komplexní datové typy*, které můžeme rozdělit na:

- výčtové typy,
- rozšířené typy,
- parametrické typy,
- rekurzivní datové typy.

Pro všechny však platí toto schéma:

```
data Jméno_typu a1 a2 ... an
     = Konstruktor_1
     | Konstruktor_2
       ...
     | Konstruktor_m
     deriving (...)
```

Přitom parametry typu a klauzule `deriving` jsou nepovinné.

Výčtové typy

Výčtové typy jsou nejjednodušší formou komplexních datových typů. Například typ reprezentující složky známého barevného modelu můžeme definovat takto:

```
data Color = Red | Green | Blue
```

Jak bylo řečeno na samém úvodu k jazyku Haskell, tak tento jazyk je citlivý na velikost písmen. Tedy prezentovaný případ není možné (u prvních písmen každého identifikátoru) zapsat jinak.

Každý takový typ lze potom užít ve funkci na místě pro unifikaci vzorů, např. takto:

```
isRed Red = True
isRed _   = False
```

Pokud si necháme interpretem jazyka Haskell vypsat typ takové funkce, tak dostaneme očekávaný výsledek `isRed :: Color -> Bool`.

Rozšířené typy

Za rozšířený typ budeme považovat takový výčtový typ, který navíc pro některé varianty používá již definované a pevně stanovené typy jako parametry. Trochu uměle můžeme navázat na předchozí případ a definovat nový typ `Color'`:

```
data Color' = Red | Green | Blue | Grayscale Int
```

Zde `Grayscale` představuje datový konstruktor, který podobně jako u jednoduchých typů, potřebuje hodnoty nějakého jiného typu, aby vytvořil hodnotu požadovaného typu. V tomto případě tedy `Grayscale :: Int -> Color'`. Pro vytvoření hodnoty typu `Color'` tedy musíme použít např. tento zápis: `Grayscale 4`.

Použití ve funkci pro unifikaci vzorů je potom shodné jako u výčtových typů s tím, že varianty s parametry se musejí závorkovat a správně doplňovat na počet parametrů:

```
getLevelOfGray (Grayscale n) = n
getLevelOfGray _ = 0
```

Parametrické typy

Tento typ uživatelských typů je dalším rozšířením předchozích variant a je prakticky vrcholem. Díky technice zavedené už u typových synonym, či jednoduchých typů umožňuje definovat parametry typů, takže je možné je konkretizovat dle potřeby až v momentě užití, či je nechat volné a definovat např. generický algoritmus, který je na takovém parametru nezávislý — podobně jako funkce `length` pro zjištění délky seznamu, která není závislá na typu, nad kterým je seznam vybudován.

Potom tedy můžeme náš typ s barvami rozvést např. do takovéto (i když asi ne úplně ideální, ale pro demonstrativní účely dostačující) podoby:

```
data RGBColor a = RGBc a a a

data CMYColor a = CMYc a a a

data Color a
```

```
= RGB (RGBColor a)
| CMY (CMYColor a)
| Grayscale a
```

Jak je patrné, tak parametry typů je možné předávat i jiným typům, které využíváme pro konstrukci našeho nového typu. Definice typů `RGBColor` a `CMYColor` není možná pomocí jednoduchého typu (`newtype`), neboť typ není jednoduchý, má tři komponenty, i když stejného typu a.

Při vytváření hodnot typů pak nejsme vázáni pevně předdefinovaným typem, který se užívá v bázi typu nadřazeného, takže zápis `RGBc 1.0 2.0 3.0` definuje hodnotu typu `RGBColor Double`. Zatímco zápis `RGBc 3 3 1` definuje hodnotu typu `RGBColor Integer`.

Díky parametrizaci typu a díky systému typových tříd i definice takovéto funkce:

```
rgb2grayscale (RGB (RGBc r g b)) =
                          Grayscale ((2*r+4*g+2*b)/8)
```

není zcela monomorfická. Její typ, jak říká sám interpret Haskellu, totiž je:

```
rgb2grayscale :: Fractional a => Color a -> Color a
```

V praxi to znamená, že se jedná o číslenou hodnotu, která přitom nemá vlastnosti celých čísel, ale čísel racionálních, či reálných (více viz typové třídy a jejich dokumentace na www.haskell.org).

Rekurzivní typy

Rekurzivní datové typy vlastně využívají naplno možností tvorby uživatelských typů v jazyku Haskell. V části parametrů konstruktorů se však již odkazují na sebe sama. Např. definice typu pro zásobník (ne nepodobná seznamu):

```
data Stack a = Top a (Stack a)
             | Bottom
```

Stejně jako v jiných případech je možné tento typ začlenit do příslušných typových tříd buďto ve vlastní režii a využít tak toho, že máme vše plně pod kontrolou, nebo nechat pracovat překladač jazyka Haskell.

V tomto případě, i když by to šlo využít i v jiných, by se nám však mohlo hodit, aby konstruktory typu pro zásobník nebyly jako funkce, ale jako operátory. Jazyk Haskell umožňuje definovat vlastní operátory jak pro funkce, tak pro datové konstruktory. U datových konstruktorů

vlastní operátory

však musí být obsažena vždy dvojtečka. Pro operátor (všeho druhu) je dobré definovat asociativitu a prioritu. Pro asociativitu máme 3 možnosti:

- `infixl` — levě asociativní operátor (např. +)
- `infixr` — pravě asociativní operátor (např. :)
- `infix` — operátor bez asociativity (např. ==)

Při deklaraci operátoru potom následuje nepovinný údaj o prioritě (0–9) a potom operátor (krom již definovaných, pochopitelně).

Naši definici typu pro zásobník bychom potom upravíme tak, aby byla stručnější pro zápis hodnot a definovala rovnost dvou zásobníků:

```
infixr 5 :>

data Stack' a
  = a :> (Stack a)
  | Bottom
  deriving Eq
```

Pro zobrazení použijeme vlastní definic, abychom dosáhli požadovaného efektu:

```
instance Show a => Show (Stack a) where
  showsPrec _ Bottom = (++) "Bottom"
  showsPrec p s@(_:>_) = ("[:" ++) . mkString s
    where
      mkString Bottom   = (++) ">]"
      mkString (t :> b) =
          if isBottom b
          then showsPrec 5 t . mkString b
          else showsPrec 5 t . ('-':) . mkString b
      isBottom Bottom = True
      isBottom _      = False
```

Důsledkem vlastní implementace je potom mnohem kompaktnější zápis. V případě využití vlastností jazyka, respektive překladače jazyka Haskell bychom pro zásobník s prvky 1 ... 4 dostali tento zápis `1 :> (2 :> (3 :> (4 :> Bottom)))`, zatímco naše implementace bude produkovat `[:1-2-3-4>]`. Co na tom, že při tvorbě literálů to bude jinak. Stejně budeme hodnoty často tvořit programově.

Pozn.: Kompilátor GHC i jeho interaktivní verze umožňují definici i bezparametrického konstruktoru jako operátoru:

```
data Stack' a
  = a :>> (Stack' a)
  | (:||)
  deriving (Eq, Show)
```

Zápis literálu je potom ale takovýto: `1 :>> 2 :>> 3 :>> (:||).`

Nyní budeme prezentovat jeden řešený příklad, který se točí kolem návrhu a užití rekurzivního datového typu.

Řešený příklad

Zadání: Vytvořte datový typ pro index-sekvenční vyhledávání hodnot dle klíčů. Dále vytvořte funkci pro inicializaci takové struktury na základě parametrů (počet pater, počet dělení na patře, celkový rozsah indexů), funkci pro vložení a vyhledání v takové struktuře.
Řešení: Pro index-sekvenční přístup se dobře hodí kombinace stromu a seznamu. Na vnitřních uzlech stromu budou pouze intervaly klíčů, na listové vrstvě i data. Na každé vrstvě budeme, pro případy dalšího užití, držet pokud možno co nejúplnější informaci. Začneme pomocnými typovými synonymy:

```
type Level = Int
type Division = Int
```

Následovat bude vlastní datový typ:

```
data ISstruct key value
  = Node Level Division key key
                    [(key,key,ISstruct key value)]
  | Leaf key key [(key,value)]
  deriving (Eq, Show)
```

V každém uzlu neseme informaci u patře, počtu intervalů, na které se dané patro dělí, spodní a horní mez intervalu, který je dělen, a seznam n-tic nesoucí pod-interval dělení a dále rekurzivně další část struktury. V listu je údaj o pokrývaném intervalu a seznam dvojic klíč a data.
Typ je poměrně rozsáhlý, díky udržované informaci. Mějme na paměti, že indexem nemusí být čísla, ale libovolný typ, který náleží třídě `Enum` (jde na základě spodní a horní meze určit všechny prvky uvnitř). Tvorba struktury bude probíhat rekurzivně. Nejdříve si nadefinujeme pomocnou funkci, která existující interval co nejrovnoměrněji rozdělí na pod-intervaly dle zadaného počtu dělení.

```
mkIntervals low high division = next division 0 add1to
  where
    interval = [low .. high]
    size = length interval
    singlesize = size `div` division
    add1to = size `rem` division
    next 1 start _ = [(interval !! start,high)]
    next toDiv start a1 =
                    int : next (toDiv-1) newstart (a1-1)
      where
        int = (interval !! start, interval !! last)
        is1 = if a1>0 then 1 else 0
        newstart = start+singlesize+is1
        last = newstart-1
```

První dva parametry této funkce definují spodní a horní mez rozkládaného intervalu. Třetím a posledním parametrem je počet pod-intervalů, na které chceme daný interval rozdělit. Lokální definice jsou potom tyto:

- `interval` — jelikož pracujeme s hodnotami, které nemusejí být číselné, tak si je takto všechny vyčíslíme pro další použití
- `size` — délka seznamu všech hodnot intervalu udává počet hodnot, které budeme dále dělit
- `singlesize` — udává základní délku jednoho intervalu
- `add1to` — jelikož délka intervalu nemusí být násobkem počtu pod-intervalů, tak některé budeme muset natáhnout o 1
- `next` — hlavní výkonná funkce

Funkce `next` má jako argumenty počet intervalů, které zbývá vydefinovat, dále index (číselný), od kterého se bude zadaný interval (`low` ...`high`) dělit na pod-interval, a potom čítač určující, kolik pod-intervalů zbývá ještě prodloužit o 1. Funkce jako výsledek poskytne seřazený seznam dvojic hodnot určujících dolní a horní mez pod-intervalů. *Detailní rozbor této funkce jakož i zjištění funkce vestavěného operátoru* `!!` *je ponechán na čtenáři.*
Teď již můžeme jednoduše nadefinovat funkci, která vytvoří základní datovou strukturu pro zadaný interval, počet úrovní a počet dělení v každé úrovni. Ve funkci se budeme snažit ošetřit i nevhodnou kombinaci vstupních hodnot.

```
initStruct low high levels division
  | levels<0 || high<low || division<2
               = error "Bad arguments"
  | size<levels*division
               = error "Bad geometry"
  | levels==0  = Leaf low high []
  | otherwise  = Node levels division low high recs
  where
    size = length [low .. high]
    ints = mkIntervals low high division
    subInt (l,h) =
                (l,h,initStruct l h (levels-1) division)
    recs = map subInt ints
```

Funkce nejprve testuje nepřípustné hodnoty, kdy odlišuje úplné nesmysly (záporná/malá čísla, špatně nastavený interval) a špatnou geometrii (přílišný počet dělení na úzký interval). Dále již pracuje na struktuře. V nulté úrovni je list s intervalem a prázdným seznamem dat. V ostatních úrovních se vytváří uzel a rekurzivně synovské uzly (až po listy). Definice `size` se užívá jen pro kontrolní účely. Jinak se vytvoří intervaly pomocí funkce `mkIntervals` a do takovéhoto seznamu se potom „namapují“ synovské uzly.
Nyní se už můžeme soustředit na vkládací a vyhledávací funkci. Začneme vyhledávací, neboť vkládací bude jen o něco těžší a bude vlastně jakousi kopií funkce vyhledávací. Pro výsledek, jelikož je možné, že pro daný klíč se žádná hodnota nebude vložena, použijeme předdefinovaný typ `Maybe a`, který má konstruktory `Just :: a -> Maybe a` a `Nothing :: Maybe a` se zřejmým významem.

```
  search key (Leaf l h vals) =
    if key<l || key>h then Nothing
    else result $ filter pred vals
      where pred (k,_) = k==key
            result [] = Nothing
            result [(_,val)] = Just val
  search key (Node _ _ l h sub) =
    if key<l || key>h then Nothing
    else search key (sel $ filter pred sub)
      where pred (l,h,_) = key<=h && key>=l
            sel [(_,_,tree)] = tree
```

Funkce pro vyhledání hodnoty v každé variantě nejprve kontroluje, že vyhledávaný klíč je v rozsahu užitých indexů. Poté se s využitím funkce `filter` vyhledá buďto příslušný pod-interval, nebo přímo hodnota. Výsledkem filtrování je vždy seznam. U vnitřních uzlů spoléháme na korektní konstrukci struktury, takže přímo vybíráme příslušný podstrom k dalšímu prohledávání. U listu je možné, že pro daný klíč nebyla vložena doposud žádná hodnota, proto ověříme, že po „filtrování“ je seznam neprázdný, jinak vracíme `Nothing` jako neúspěch.
Funkce pro vložení, kterou implementujeme s přepisem již existujících hodnot se stejným klíčem, je v podstatě velmi podobná té vyhledávací s tím, že nesmíme „zahazovat“ části struktury, které nechceme měnit, ale musíme je zachovat pro další užití.

```
insert key val lf@(Leaf l h vals) =
  if key<l || key>h then lf
  else Leaf l h ((key,val) : filter pred vals)
    where pred (k,_) = k/=key
insert key val nd@(Node x y l h sub) =
  if key<l || key>h then nd
  else Node x y l h (map chg sub)
    where chg s@(l,h,t) =
            if key<l || key>h then s
            else (l,h,insert key val t)
```

Varianta pro list pracuje tak, že odstraní ze seznamu hodnot s klíči případně ten, co chceme vložit a vkládaný uloží na začátek seznamu. Okolní části listu pochopitelně zachová. Pro vnitřní uzly potom vždy je výsledkem uzel s modifikovaným seznamem pod-uzlů. Modifikaci zajistí mapování funkce `chg` na všechny pod-uzly. Podle klíče se vybere ten, jenž má být modifikován a je na něj rekurzivně aplikována funkce `insert`.

Tím je úloha vyřešena.

To, že jsme řešení úlohy udrželi na dostatečné úrovni obecnosti je možné si ověřit výpisem typu v interpretu jazyka Haskell. Uvidíme, že pro typy sloužící pro indexy nejsou specifikovány jinak, než že musí být součástí třídy `Enum` a `Ord`, přičemž pro vyhledávání a vkládání stačí třída `Ord`. Ověřit si to můžete třeba definováním typu pro měsíce v roce, jejich začleněním do příslušných typových tříd a potom užitím při konstrukci struktury i ve hledání a vkládání. Sami jsme to udělali. A vám bych závěrem této kapitoly doporučuji to samé, vše si pořádně odzkoušet a osahat.

Pojmy k zapamatování

V této výkladové části jsme se seznámili s pojmy, které se pojí s uživatelskými datovými typy, typovými třídami a jejich instancí. Pojmů není mnoho, vlastně to jsou všechny zásadní. Více jak pojmy samotné je však nutné celé tématice dobře porozumět a umět vlastní typy vytvářet, a to všemi způsoby. Umět je začleňovat do typových tříd a umět s nimi dále pracovat.

Závěr

Tato kapitola prezentovala všechny koncepty, jak vytvořit uživatelské datové typy v jazyku Haskell (bez rozšíření jazyka apod.). Pokud jste tuto část opravdu dobře zvládli, tak byste, po prostudování knihoven nutných pro běžnou práci programátora (zejména vstupy a výstupy), měli být schopni vytvořit prakticky libovolný program v jazyku Haskell. A to i díky tomu, že jste zvládli tvorbu uživatelských datových typů.

Pozn.: Pro další samostudium doporučujeme zvládnutí typu `Monad` *a zejména jeho instance* `IO` *pro vstupy a výstupy. Dále potom paralelismus v jazyku Haskell — vícevláknové zpracování, které je, po zvládnutí konceptu monád, jednoduše použitelné.*

Úlohy k procvičení:

1. Definujte vlastní typ pro binární vyhledávací strom a implementujte operace pro vložení do stromu, vyhledání ve stromu, převodu stromu na seznam průchody in-order, pre-order, post-order, vyvážení stromu.

   Zamyslete se nad časovou složitostí funkcí převádějících strom na seznam, zkuste je urychlit.

2. Začleňte vaši stromovou strukturu do typové třídy Functor. Potom také vytvořte explicitní instanci pro třídu Show, kdy, s použitím vhodných znaků, strom opravdu vykreslíte jako strom (např. jak to dělá Průzkumník v MS Windows). *Pozn.: Se šířkou terminálu/okna se netrapte.*

3. Navrhněte datovou strukturu pro reprezentaci λ-kalkulu. Jako proměnné uvažte řetězce. Implementujte funkci realizující substituci.

4. Navrhněte datovou strukturu, která bude reprezentovat aritmetické výrazy (operace aditivní, multiplikativní, mocniny, logaritmy, exponenty, goniometrické funkce, konstanty, proměnné). Implementujte funkci, která se zadanými (vhodně, třeba prostřednictvím vašeho stromu) hodnotami proměnných vyhodnotí výraz. Implementujte funkci pro symbolickou derivaci podle zadané proměnné.

Klíč k řešení úloh

1. Pro optimalizaci se inspirujte u optimalizované implementace funkce `showsPrec` v textu kapitoly, i u typu samotné funkce `showsPrec`.

2. Jako parametr pro `fmap` předávejte klíč i hodnotu. U tisku tiskněte klíč i data na řádek, strom kreslete vlevo.

3. Nahlédněte zpět do kapitoly o λ-kalkulu.

4. Pro derivace nahlédněte na internet, ostatní by měla být rutina, nicméně: výraz bude taky jistý druh stromu, oddělte definice operátorů a funkcí od stromu pro výrazy.

Další zdroje

Řadu informací podaných v této kapitole najdete v literatuře citované níže, nebo přímo v nápovědě, či doprovodné dokumentaci, či manuálech, které jsou na stránkách věnovaných jazyku Haskell.

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

Kapitola 6

Závěr

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

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

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

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

Literatura

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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