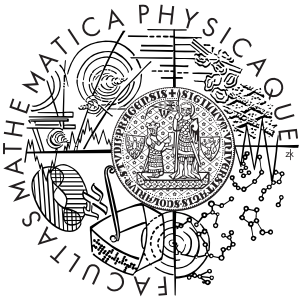
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Tomáš Caithaml

Doménově specifické jazyky

Katedra teoretické informatiky a matematické logiky

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

2009

Rád bych na tomto místě poděkoval RNDr. Janu Hricovi za trpělivé vedení mé diplomové práce a Honzovi Benešovi za přívaly připomínek.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne Tomáš Caithaml

Obsah

Název práce: Doménově specifické jazyky
Autor: Tomáš Caithaml
Katedra: Katedra teoretické informatiky a matematické logiky
Vedoucí diplomové práce: RNDr. Jan Hric
e-mail vedoucího: Jan.Hric@mff.cuni.cz

Abstrakt: Doménově specifické jazyky (DSL) jsou jazyky navržené pro určitou problémovou oblast. Jejich syntaxe a základní primitiva jsou přizpůsobeny třídě problémů, které mají řešit. Díky této specializaci jsou programy v nich napsané kratší a srozumitelnější než jejich protějšky zapsané v běžných programovacích jazycích. Velmi efektivní metodou vytváření DSL je jejich implementace uvnitř obecného programovacího jazyka. Vznikají tak vnořené doménově specifické jazyky (DSEL). Cílem této práce je prozkoumat techniky používané při návrhu DSEL ve staticky typovaných funkcionálních jazycích a analyzovat souvislost mezi vlastnostmi DSEL a vlastnostmi hostitelského jazyka. Za implementační jazyk jsme si vybrali Haskell.
Klíčová slova: DSL, DSEL, Haskell

Title: Domain Specific Languages
Author: Tomáš Caithaml
Department: Dpt. of Theoretical Computer Science and Mathematical Logic
Supervisor: RNDr. Jan Hric
Supervisor's e-mail address: Jan.Hric@mff.cuni.cz

Abstract: Domain specific languages (DSL) are languages designed for one particular area. Their syntax and basic primitives are carefully chosen to fit the class of problems they are meant to solve. Because of this specialisation, programs written using DSL are usually more concise and coherent than their counterparts written in common programming languages. There is a very effective way to implement DSL – to define them inside a full-fledged language. This leads to so called domain specific embedded languages (DSEL). Purpose of this thesis is to investigate techniques used in implementation of DSEL and to analyse connection between features of host and embedded language. We focus on statically typed functional languages as host languages. We have choosen Haskell as an implementation language.
Keywords: DSL, DSEL, Haskell

Kapitola 1

Úvod

Doménově specifické jazyky (DSL) jsou jazyky navržené pro jednu určitou problémovou oblast. Jejich syntaxe a základní primitiva jsou přizpůsobeny třídě problémů, které mají řešit. Díky této specializaci jsou programy v nich napsané kratší a srozumitelnější než jejich protějšky zapsané v běžných programovacích jazycích. Někdy dokonce natolik, že jim rozumí nebo je i píší doménoví experti – neprogramátoři, kteří se vyznají v dané oblasti.

Použití DSL může být velmi efektivní, je však třeba počítat s tím, že zavedení nového DSL do softwarového projektu znamená práci navíc:

- práci s návrhem jazyka,
- práci s jeho implementací,
- práci s integrací jazyka se zbytkem systému.

Velmi efektivní metodou vytváření DSL je jejich vnoření do obecného programovacího jazyka. Při tomto způsobu implementace hostitelský jazyk silně ovlivňuje návrh DSL a do značné míry určuje jeho vlastnosti. Funkcionální jazyky jsou obzvláště vhodnými kandidáty na roli hostitelských jazyků.

Cílem této práce je prozkoumat techniky používané při návrhu vnořených DSL a analyzovat vztah mezi vlastnostmi DSL a hostitelského jazyka s důrazem na čistě funkcionální staticky typované jazyky.

Celá práce je rozdělena do pěti kapitol. Úvod obsahuje přehled problematiky, zavádí pojem doménově specifických jazyků a studuje jejich jednotlivé varianty. V druhé kapitole už se zaměříme výhradně na vnořené doménově specifické programovací jazyky. V třetí kapitole představíme pokročilejší techniky používané při jejich implementaci.

Nevýhodou klasického způsobu vytváření vnořených DSL je určitá neefektivita výsledné implementace, která může být ve výpočetně náročných doménách neakceptovatelná. Ve čtvrté kapitole se proto zaměříme na postupy, kterými lze vnořené jazyky implementovat s ohledem na efektivitu provádění programů v nich napsaných. V poslední, páté kapitole pak shrneme výsledky celé diplomové práce.

Ukázky zdrojových kódu v této práci jsou v jazyce Haskell (důvody této volby jsou uvedeny dále v práci). V příloze je proto krátký přehled syntaxe tohoto jazyka, aby mohl čtenář kódu alespoň povrchně rozumět. Znalost Haskellu není pro porozumění práci nezbytně nutná, ale je samozřejmě velkou výhodou stejně jako povědomí o funkcionálním programování.

1.1 Proč DSL?

Ruku v ruce se vznikem počítačů přišla zároveň i potřeba mít odpovídající prostředky k jejich ovládání. První exempláře byly obrovské neohrabané stroje s nepříliš pohodlným rozhraním – naprogramování určité úlohy mohlo sestávat ze změny zapojení drátů, či (v lepším případě) z vyděrování štítků s instrukcemi přímo nahrávanými do operační paměti.

S růstem výpočetní síly počítačů rostla i složitost úloh, k nimž byly používány. To si vynutilo i postupnou evoluci způsobu jejich programování. Od strojového kódu se programovací jazyky postupně vyvinuly k vyšším abstrakcím. Během této cesty se zájem přesunul od programování počítačů k programování úloh. Kdy a jak se vykonává dříve zmíněný strojový kód se stalo nedůležitým detailem.

V současné době jsou mainstreamem tzv. jazyky třetí generace. Tyto obecné programovací jazyky (angl. General Purpose Language, GPL), jako například C/C++ nebo Java, dávají uživatelům k dispozici prostředky ke konstrukci softwaru (téměř) nezávislého na stroji, na kterém běží. Jak naznačuje jejich název, jde o jazyky uzpůsobené k popisu velmi rozsáhlé třídy úloh. Tato flexibilita je zároveň i slabinou těchto jazyků. Základními jednotkami v GPL jazycích jsou stále ještě kroky výpočtu. To sice umožňuje snadné vyjádření všech implementačních detailů softwarového díla, vývoj velkých aplikací ovšem odhaluje sémantickou propast mezi současnými nízkoúrovňovými vývojovými prostředky a vysokoúrovňovými zadáními.

Přechod od zadání k hotovému softwaru je náročný a riskantní proces. Obecně se věří, že neexistuje žádné univerzální řešení, které by zaručovalo jeho úspěch [**?**]. Přesto existují principy a postupy, které se snaží situaci vylepšit a snížit

riziko neúspěchu. Jedním takovým postupem je vytváření doménově specifických jazyků.

Doménově specifický jazyk je jazyk přizpůsobený určitému druhu úkolů nebo aplikační doméně. Hlavní myšlenka schovávající se za DSL je, že pokud programátor může zapsat program způsobem, který je blízko tomu, jak si jej představuje, může to být velký přínosem v oblastech:

- **Kvality**. Abstrahováním od nízkoúrovňových detailů se zmenšuje potenciál pro dělání chyb. Méně řádek kódu znamená méně chyb. Navíc může kompilátor DSL provádět doménově-specifické sémantické kontroly.
- **Produktivity**. Omezením se na určitou doménu se vývoj stává rychlejší a vyžaduje méně úsilí.
- **Čistoty**. Mapování myšlenek a požadavků do implementace se stává jednodušší. Dále jsou rozhodnutí v průběhu implementace lépe viditelná ve zdrojovém kódu (DSL) než v dokumentaci a komentářích.
- **Optimalizace**. Optimalizace mohou být delegovány na implementaci DSL, která využívá doménově specifické znalosti. Například mohou být aplikovány optimalizace, které jsou založeny na předpokladech, které sice obecně neplatí, ale jsou splněny v dané omezené doméně.
- **Uživatelské přívětivosti**. DSL může být natolik intuitivní, že ho zvládnou i neprogramátoři. Části aplikací tedy mohou psát experti v dané oblasti – analytici, manažeři, inženýři. V takovém případě odpadá náročná komunikace mezi expertem a programátorem.

1.2 Přehled DSL

Neexistuje jediná správná odpověď na otázku, jak by doménově specifické jazyky měly vypadat. Dokonce není vždy ani jasné, co je a co není DSL. Konfigurační soubor s určitou strukturou představuje DSL pro jedny, zatímco druzí používají tento termín jen pro plnohodnotné jazyky s vlastním překladačem.

Doménově specifické jazyky také nejsou žádná převratná novinka. Jazyky pro speciální účely existovaly vždy. Jedním z klasický příkladů je svět UNIXu, kde jsou známé pod názvem „Little Languages“ [**?**] nebo také „minilanguages“ [**?**].

To, že jsou DSL zaměřené na jednu úzkou oblast neznamená, že mají také úzké použití. Porovnávání textu s daným vzorem je velmi úzká oblast, přesto

představují regulární výrazy jedno z nejrozšířenějších DSL – existují v několika variantách, lze s nimi pracovat nejen pomocí samostatných programů (grep), ale jsou dostupné i ve většině programovacích jazyků, hlavně těch skriptovacích. Perl je dokonce zakomponoval do vlastní syntaxe. Bývají také častou součástí jiných DSL.

Dalším notoricky známým zástupcem DSL je dotazovací jazyk SQL. Málokterá větší aplikace, která pracuje s perzistentními daty, se obejde bez jeho použití.

Oba dva jazyky jsou výbornými příklady dobrých abstrakcí a DSL obecně. Umožňují uživateli deklarativní popis toho, co chce, nikoli způsob jakým to získat. Na pozadí obou jazyků je silný teoretický aparát, který umožňuje jejich efektivní implementaci (konečné automaty, relační algebra).

Vytvořit takovou implementaci ovšem není triviální. Zvláště u SQL je to patrné – implementací je vlastně celý databázový server. Vzhledem k počtu nasazení se ovšem úsilí vložené do implementace bohatě vyplatí a samotné uživatele to nemusí zajímat – k používání těchto jazyků nemusí mít žádné povědomí o tom, jak jsou vykonávány.

Každopádně použití těchto nástrojů je nesrovnatelně jednodušší a efektivnější, než se pokoušet tyto úlohy řešit přímo pomocí standardních prostředků GPL (porovnávání řetězců, základní operace se soubory).

Rozšířené a „standardní“ DSL mají ještě tu výhodu, že nevytváří vazbu na konkrétní software – díky SQL lze většinou snadno přejít od jedné databáze k druhé; vzor, který si otestujeme na příkazové řádce, můžeme poté klidně vložit do skriptu.

I když je to tak trochu paradox, DSL nemusí být používáno k psaní programů – PostScript je toho dokladem. V jeho případě spočíval úspěch DSL přístupu v tom, že programovací jazyk má větší vyjadřovací sílu než běžný datový formát. Implementace interpretu jednoduchého jazyka v tiskárnách umožnila jednotný a flexibilní přístup k tisku – otevřela se tak cesta k revoluci DTP v polovině 80. let[1]. PostScript je tak znám spíše jako datový formát pro elektronickou publikaci, než jako programovací jazyk.

Hlavní doménou unixových minijazyků je zpracování textu. Nejznámějšími zástupci takových jazyků jsou AWK a sed. Programy psané v těchto jazycích jsou většinou jednoduché několika řádkové skripty, ale mohou být i dosti složité.

K podobným účelům jako AWK a sed se občas používají skriptovací jazyky jako Python, Perl nebo Bash. Ty ovšem nepatří mezi DSL, i když jsou také

[1] V současnosti význam PostScriptu už spíše odpadá, levné tiskárny už vypouštějí jeho podporu

přizpůsobeny určitým druhům úloh a poskytují pro ně příhodnou syntax (platí hlavně o Perlu). Hlavní rozdíl je především v tom, že skriptovací jazyky jsou univerzální a mají přístup k nízkoúrovňovým funkcím jako je práce se soubory, s vlákny nebo se sítí, možnost použít systémová volání. Jak ale bylo řečeno v úvodu této kapitoly, přesná definice DSL neexistuje a hranice mezi skriptovacími jazyky a DSL není ostrá.

DSL mají velké použití i při samotné tvorbě softwaru. Klasickým příkladem jsou generátory parserů jako Bison nebo ANTLR. Jde o DSL, v nichž programátor zapíše gramatiku jazyka pomocí formalismu podobnému BNF. Výsledkem zpracování takového vstupu jsou zdrojové kódy parseru pro tuto gramatiku v nějakém obecném programovacím jazyce (ANTLR dokonce podporuje více cílových jazyků).

Pro DSL je typická volná syntax, která umožňuje co nejpřirozenější zápis programů v daném kontextu. Použití generátorů parserů podstatně usnadňuje zpracování takové syntaxe. S boomem XML se ovšem objevili i jazyky, které mají svojí syntax založenou na XML – Ant, XSLT a jiné. Takový přístup usnadňuje strojové zpracování zdrojového kódu na úkor přehlednosti a stručnosti, což je proti filosofii DSL.

1.3 Knihovny a frameworky

Ekonomika použití DSL je jednoduchá: implementace a návrh DSL vyžadují nemalé počáteční náklady, které se postupně vrací tím, že pozdější aplikace vyžadují daleko méně úsilí. Užitečnost DSL je tedy dána poměrem mezi náročností jeho implementace a rozsahem jeho nasazení.

Cena DSL nevyplývá jenom z počtu řádků kódu, která jsou potřeba na jeho implementaci. Významným faktorem (možná i důležitějším) je jeho návrh. Nalézt ty správné abstrakce, které umožní snadno popsat problémy řešené v dané doméně, není vždy jednoduché.

Klíčovým slovem v tomto ohledu je *znovupoužitelnost*. Obtíže spojené s návrhem vycházejí z problematiky pochopení dané domény. Jakmile je tato znalost jednou podchycena a implementována, mohou z ní uživatelé DSL těžit a využívat předpřipravená řešení.

V tomto ohledu se DSL podobají knihovnám z běžných programovacích jazyků. Ty také zachycují doménové znalosti a jsou určeny primárně k znovupoužití (externími programy).

DSL oproti knihovně umožňuje přístup ke své funkcionalitě jen za pomoci

textové editoru, což podstatně snižuje bariéry. Nezřídka kdy jsou DSL vybudovány okolo knihovny jako prostředek k jejímu (nepřímému) volání.

Objektově orientované frameworky rozvíjejí myšlenku knihoven. Klasické knihovny mají plochou strukturu, bývají zaměřeny na jednu službu a jsou volány aplikacemi. V případě frameworků je situace obrácená. Framework je zachycením architektury programů pro danou doménu – jakousi jejich esencí. Jde o obecnou aplikaci, která umožňuje programátorovi umístit do ní svůj kód a přizpůsobit si ji tak svým potřebám. Framework řídí běh aplikace a pouze volá kód dodaný programátorem.

Většina Java a .NET frameworků spoléhá na externí konfigurační soubory – v naprosté většině ve formátu XML. Tyto konfigurace jsou natolik složité, že bývají občas pejorativně označovány za DSL.

Problémem frameworků bývá to, že jsou rozsáhlé a nemotorné, doménové koncepty jsou zastíněny implementačními detaily. Existuje snaha řešit tyto problémy právě využitím DSL uvnitř frameworků.

1.4 Návrh DSL

V průběhu návrhu DSL je třeba provést netriviální rozhodnutí, z nichž každé může ovlivnit použitelnost a dokonce i životaschopnost vytvářeného DSL. Někdy jsou důsledky těchto rozhodnutí zřejmé, často však nikoli. Najít ty správné abstrakce pro návrh DSL je v současné době spíše umění než zaběhlá inženýrská rutina.

Klasický postup při vývoji DSL spočívá v navržení syntaxe jazyka a napsání parseru, který převede textový vstup na abstraktní syntaktický strom. Následuje zpracování tohoto stromu, které může zahrnovat:

1. další, sémantické kontroly;
2. transformace, optimalizace;
3. interpretace nebo generování kódu.

Oproti běžným GPL kompilátorům implementace DSL typicky neobsahuje nízkoúrovňové operace na úrovni instrukcí. Odpovídá tedy spíše front-endu klasického kompilátoru. Existují sice nástroje, které výrazně usnadňují implementaci kompilátoru (hlavně parsování), ale i tak jde o netriviální úlohu.

Výhodou tohoto přístupu je, že programátor má absolutní kontrolu nad všemi částmi DSL a může tak vytvořit jazyk přesně podle svých představ. Na druhou

stranu nedostane nic zadarmo – každá vlastnost nového jazyka vyžaduje implementaci. Vzniká tak pnutí mezi návrhem jazyka a snadností jeho realizace. Typickým řešením bývá mírně slevit z dokonalosti jazyka výměnnou za snazší implementaci. Nesmí se však zbytečně zavádět omezení, která znesnadňují jeho používání, a musíme mít na zřeteli záměr jazyka.

Návrh doménově specifických jazyků je založen na dvou důležitých principech: abstrakce a restrikce. Obecný programovací jazyk dovoluje uživateli definovat libovolný program – ovšem za vzrůstající cenu. Složitost dosažení výsledku roste se složitostí úkolu. Doménově specifické jazyky se snaží zvýšit úroveň abstrakce a tím snížit náročnost daných úloh. Na druhou stranu jiné úlohy může být těžší nebo přímo nemožné řešit pomocí daného DSL.

Další oblast, které je potřeba věnovat pozornost je komunikace DSL s okolím. Z definice jsou DSL úzce zaměřené a proto nemohou stát sami o sobě. Zde můžeme identifikovat několik vzorů:

- Nejjednodušší je to u jazyků jako LaTeX, PIC, troff, dot a podobně. Tyto jazyky dostanou na vstupu zadání a jejich výsledkem je samostatný artefakt – dokument nebo obrázek. Není tedy potřeba žádná významná interakce s okolím. Někdy je užitečné mít možnost vkládat do takových dokumentů neinterpretovaný kód. Tím se otevírá cesta ke zřetězení více jazyků za sebou.

- Další skupinou jsou jazyky, které popisují obecnou strukturu nebo často se opakující činnost, jejíž detaily se ovšem případ od případu velmi liší. Obvyklé řešení v takovém případě je, že DSL popisuje jen onu kostru problému a zbytek zapíše uživatel pomocí fragmentů neinterpretovaného kódu. Výhodou je univerzálnost takového řešení. Nevýhoda spočívá v tom, že DSL takovému kódu nerozumí a nemůže ho ověřit, jenom předat externímu programu. Kód je spouštěn nepřímo a případné ladění může být pro uživatele obtížné. Chyba navíc nemusí být jen na úrovni samotného fragmentu, ale i na úrovni interakce mezi daným fragmentem a tím, jak ho DSL používá.

  Klasickým příkladem takových jazyků je make. Jeho úkolem je sledovat závislosti mezi soubory a v případě potřeby přegenerovat ty, které už zastaraly. Konkrétní kroky, kterými jsou soubory generovány, už jsou ale nad jeho rámec – zde nastupují shellové skripty.

  Ještě markantnější příklad jsou generátory parserů, kde jednotlivá pravidla bývají oanotována kódem popisujícím výsledek naparsování. Tyto

fragmenty pak skončí ve výsledném parseru spolu s kódem pro samotné rozpoznávání jazyka.

- DSL jazyky také mohou být použity jako rozhraní – způsob, jakým se dostat k požadované funkcionalitě. Řekněme, že potřebujeme vykreslit nějaký jednoduchý graf do daného datového formátu. Nejefektivnější cestou, jak to zařídit, může být vygenerovat z grafu jeho popis ve vhodném DSL a následné vyvolat program, který takové DSL zpracovává.

  Slabinou tohoto přístupu je obtížná kontrola správnosti generování programu. Ošetřit všechny okrajové případy může být velmi obtížné. Pokud je navíc program generován v závislosti na vstupech od uživatele, vzniká velké nebezpečí zneužití. Nejznámějšími příkladem je SQL, kde je tento problém znám pod jménem SQL injection.

Kapitola 2

Vnořené doménově specifické jazyky

V úvodní části diplomové práce jsme se zabývali doménově specifickými jazyky obecně a shrnuli jsem jejich výhody a nevýhody. Nadále se budeme věnovat už jen DSL jazykům vytvořených uvnitř jiného obecného programovacího jazyka.

Hudak [**?**] zavedl pro tyto jazyky název vnořené doménově specifické jazyky – DSEL (z angl. Domain Specific Embedded Languages). Terminologie ovšem není ustálená – používá se také zkratka EDSL [**?**] (Embedded Domain Specific Languages).

2.1 Proč DSEL?

Navrhnout a implementovat programovací jazyk od úplného začátku není triviální záležitost. Navíc je velká pravděpodobnost, že první verze nebude dostačující, bude se dále vyvíjet a my budeme muset implementaci rozšiřovat a upravovat.

Jinými slovy, počáteční náklady na vývoj jsou dosti velké a vývoj nového jazyka představuje rizikovou část projektu, což omezuje praktické nasazení DSL na situace, kdy lze snadno odvodit, jak by měl jazyk vypadat, a zároveň kdy přínos z použití DSL bude dostateč velký – natolik, aby pokryl náklady na vývoj.

Vnořené doménově specifické jazyky představují vynikající alternativu ke klasickému přístupu, protože odbourávají potřebu vytvářet vše od začátku. DSEL dědí infrastrukturu hostitelského jazyka a přetváří ji v patřičných oblastech na míru dané doméně. Díky tomuto „dědictví“ se autor jazyka může zaměřit na

sémantiku nového jazyka (syntax je do značné míry předurčena).

2.2 Co je DSEL?

Jazyk lze považovat za kombinaci dvou doplňujících se aspektů [**?**]. Jeden je obecný a zahrnuje základní syntaktické a sémantické pojmy jako definice, použití jmen pro hodnoty a typy, vytváření a volání funkcí, řízení toku programu, ošetření výjimek, typovací pravidla. Druhý aspekt jazyka je doménově specifický slovník popisující například matematické operace, manipulace s řetězci, seznamy, stromy.

Snaha DSEL je realizovat tento slovník takovým způsobem, aby vznikl dojem nového jazyka, který je navrhnut na míru dané doméně. Abychom mohli provést vnoření, musí být hostitelský jazyk dostatečně pružný, s minimem klíčových slov a jiných omezení na syntaxi. Mainstreamové programovací jazyky jako C, Java, C#, VB a podobně nejsou vhodnými kandidáty.

Naopak ve funkcionálních jazycích typu Scheme nebo Haskellu je až překvapivě snadné navrhnout DSEL pro širokou škálu aplikací. Proces vkládání je ovšem velmi silně vázán na hostitelský jazyk. Lze vysledovat určité způsoby, jak postupovat, ty se ovšem pro různé jazyky liší, protože staví na jiných vlastnostech a nejsou tak přenosné. Volba hostitelského jazyka je tedy zásadní.

Přesto nejde o extrémně složitou záležitost. Návrh DSEL v obou jazycích je vcelku přirozený a lze ho přirovnat k návrhu knihovny v mainstreamových jazycích.

Například v Haskellu bývá DSEL prostá knihovna, která zveřejňuje datové typy a několik funkcí, které s nimi pracují. Bývá také označována jako *knihovna kombinátorů*, protože definuje malou sadu primitiv a potom množinu „kombinátorů“, které je dokáží spojit (zkombinovat) do větších a složitějších struktur.

2.3 Výhody DSEL

Hlavní výhoda integrace externího DSL je to, že lze vytvořit jazyk, který perfektně slouží svému účelu – jak po stránce sémantické, tak po stránce syntaktické. Vnoření na druhou stranu vyžaduje určitou toleranci ke kompromisům, které musí být učiněny při přizpůsobování obecného programovacího jazyka specifickým požadavkům. Odměnou za tyto kompromisy je možnost využití již existující infrastruktury.

Praktické DSL, které není jen hračkou nebo výzkumným projektem (i když mohlo jako takové začít), nutně potřebuje několik částí, které musí být dobře navrženy a implementovány:

- **Definice jazyka**. I přes největší snahu svých autorů, mají úspěšné DSL (používané) tendenci růst a absorbovat do sebe čím dál víc vlastností obecného programovacího jazyka. Výsledek takových rozšíření, pokud nejsou anticipovány, bývá nepřehledný a nekonzistentní.

- **Implementace jazyka**. V závislosti na doméně může stačit jednoduchý interpret, ale v některých případech (například real-time animace) je kompilace důležitá. Vytvořit dobrý kompilátor jako je GCC nebo GHC (kompilátor Haskellu) trvá roky.

- **Nástroje**. Programátoři potřebují debuggery a profilery, aby mohli psát programy, které pracují správně a efektivně.

- **Interoperabilita**. Ze specializace DSL také vychází potřeba zabalit kusy funkcionality tak, aby mohly pracovat s jinými komponentami implementovanými v jiných jazycích, ať už doménově specifických či nikoliv.

Ve všech těchto ohledech nám může být DSEL nápomocno. Snazší implementace jazyka je hlavní motivací pro použití DSEL. Nejde ovšem jen o úsporu práce. Hostitelský jazyk nám může poskytnout vlastnosti, které bychom pro jejich náročnost buďto vypustili, nebo je zavedli v horší kvalitě. Například napsat kvalitní garbage collector pro jeden omezený jazyk není reálné – řešením je buďto ho z jazyka vypustit (a tím výrazně zasáhnout do jeho návrhu), nebo se spokojit s jednoduchou variantou typu počítání referencí.

Nástroje v případě DSEL jsou zděděny z hostitelského jazyka. To sice není optimální, protože se ztrácí doménově specifický charakter jazyka. Uživatel je nucen ladit a profilovat implementaci DSEL, místo toho, aby ladil svůj program. Na druhou stranu jen málokteré externí DSL má vůbec takové nástroje k dispozici.

Interoperabilita je další silnou stránkou DSEL. Programy v DSEL bývají typicky podprogramy (funkce), které jsou napsány pomocí dané knihovny. Tyto podprogramy lze pak snadno volat z hlavního kódu. I uvnitř programu v DSEL lze přejít k hostitelskému jazyku a využít jeho schopnosti nebo třeba začít používat jiné DSEL.

2.4 Vhodné jazyky

Nyní se můžeme pokusit určit, které vlastnosti jazyka dovolují snadné vytváření DSEL.

- **Výrazy**. Výrazy se snadno skládají, lze je přirozeně zanořovat do sebe a vedou k manipulaci „hodnot“ místo efektů. Dovolují tedy deklarativnější přístup než příkazy.

- **Definice**. Abychom mohli jednotlivé konstrukty použít vícekrát v různých kontextech a oddělili definici od použití, je důležité mít k dispozici obecný mechanismus, který umožní pojmenovávat hodnoty a výrazy pokud možno s omezitelnou viditelností, aby šlo zavést jak globálně platná jména, tak i pomocné lokální definice.

- **Programování vyššího řádu** (first-class funkce). Funkce vyššího řádu umožňují vyjádření a zapouzdření doménově specifických „programových vzorů“. Typickým příkladem funkcí vyššího řádu je `map`, který aplikuje danou funkci na všechny prvky datové struktury přičemž uživatel takové funkce je odstíněn od způsobu, jakým je struktura procházena.

- **Silné statické typování**. Doménově specifický jazyk typicky obsahuje několik různých druhů hodnot a operací, které s těmito hodnotami pracují. Nelze ovšem aplikovat všechny operace na všechny typy hodnot. Navíc může typový systém sloužit jako nápověda, která vede uživatele k smysluplným definicím. Další efekt statického typování je zvýšení výkonu díky eliminaci runtimových kontrol. Má-li mít DSL jednoduchou a přehlednou syntaxi, je záhodno, aby byly typy automaticky odvoditelné a nemusely se do kódu ručně dopisovat.

- **Flexibilní notace**. Schopnost dát starým jménům nový, doménově specifický význam je velmi žádoucí.

  Je krajně nepříjemné, pokud jsou některé identifikátory nelegální, protože jsou zabrané v hostujícím jazyce (např. klíčová slova), i když v DSL se nikde nevyskytují. Typickým příkladem jsou aritmetické operátory.

  Sice jde jen o „syntaktické pozlátko“, nicméně právě tato vlastnost stojí z větší částí za „look'n'feelem“ doménově specifických jazyků a dělá výsledné programy čitelnější a srozumitelnější, než kdybychom museli zadefinovat úplně novou kolekci jmen.

- **Garbage collection**. Správa paměti je nízkoúrovňový detail. Správná alokace a dealokace je nepříjemná rutina, která nabízí spoustu příležitostí k chybám. Automatická správa paměti dává DSL deklarativnější ráz a osvobozuje uživatele od otrocké práce. Co je ale důležitější, dává daleko větší bezpečnostní záruky.

  Bezpečnost DSL je velmi důležitá, protože program v DSL je ve skutečnosti programem v hostujícím jazyce. Pokud je ve vstupním programu nějaká chyba, projeví se tato později na jiné úrovni abstrakce. Pokud například na některém místě předpokládáme, že uživatel v DSL zadá výraz, který vrací hodnoty jen z určitého oboru, ale tuto vlastnost explicitně nekontrolujeme a přímo používáme hodnotu výrazu například k indexaci pole, dočkáme se nepříjemného pádu programu. Uživatel uvidí, že pád je způsobený použitím indexu mimo rozsah pole, která používá implementace, ale bude vyžadovat velké detektivní úsilí vystopovat skutečnou příčinu problému.

- **Líné vyhodnocování**. Princip lenosti spočívá v odkládání výpočtu až do doby, kdy je skutečně potřeba. Některé části programu ani nemusejí být vyhodnoceny.

  To, co je běžně bráno jako primitivní příkazy pro kontrolu řízení (podmínky, cykly), lze často v líném jazyce definovat. Lze tedy definovat i doménově specifické řídící struktury.

  Lenost také hraje komplementární roli ke garbage collectoru – paměť není konzumovaná, dokud není nezbytně třeba, zatímco garbage collector ji uvolní po tom, co už není potřeba.

- **Moduly**. I programy psané v DSL mohou být docela komplikované a proto je důležité, aby šlo rozumně vytvářet části psané různými autory, různé znovupoužitelné knihovny.

Haskell je experimentální, čistě funkcionální[1] jazyk, který má většinu výše uvedených vlastností. Proto jsme si ho také vybrali jako hostitelský jazyk v rámci této diplomové práce.

Možnosti DSEL v Haskellu jsou opravdu široké – tímto způsobem byly implementovány například tyto domény: parsery [**?**], grafika [**?**], skládání hudby [**?**], návrh hardwaru [**?**, **?**, **?**], pretty printing [**?**], skriptování COM komponent [**?**].

[1] funkce nemohou mít vedlejší efekty

2.5 PIC

Nyní si navrhneme a naimplementujeme jednoduché DSEL, abychom získali konkrétnější představu, jak takový vnořený doménově specifický jazyk vlastně vypadá.

Náš jazyk se bude jmenovat PIC[2] a bude sloužit k tvorbě obrázků. Původní návrh tohoto jazyka pochází z Hendersonova článku Functional Geometry [**?**]. Další varianty se postupně objevili například v [**?**] nebo [**?**] a v revizi původního článku [**?**].

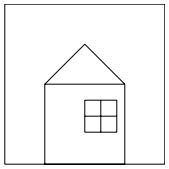
PIC je navržen tak, aby šlo snadno experimentovat s obrázky, které jsou do určité míry pravidelné, založené na opakujících se vzorech. Právě takovými díly je známý nizozemský umělec M.C. Escher. Snaha o „naprogramování“ jeho dřevořezu Čtvercová limita byla původní motivací Hendersonova článku.

2.5.1 Základní přehled

Z pohledu uživatele obsahuje jazyk jediný typ – `Picture`. Tento typ reprezentuje obrázek ať už dodaný v rámci knihovny nebo vytvořený uživatelem. Obrázky se navenek chovají jako černé krabičky – nemají žádné explicitní rozměry ani se neskládají z jiných primitiv (pixelů, čar, křivek).

Jediný způsob jak s nimi lze manipulovat je pomocí množství jednoduchých transformací, které jsou součástí jazyka. Ty uživateli umožňují vytváření nových, komplikovanějších obrázků z těch stávajících. Výsledný obraz tak vznikne jako skládačka z menších stavebních bloků slepených dohromady pomocí sady geometrických operací.

Celý jazyk je deklarativní, uživatel se nemusí starat o souřadnicové systémy, jednotky, ani o způsob, jakým se spočítá výsledek. Pouze při finálním vykreslení zadá uživatel velikost boxu a výsledek je přepočítán tak, aby tento box zaplnil.

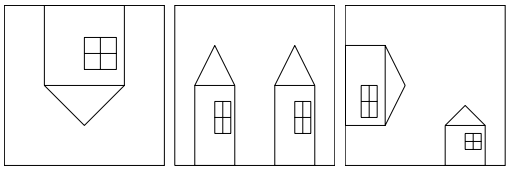


Obrázek 2.1: house

Na obrázku **??** vidíme základní obrázek – *house*. Nyní si představíme primitivní operace:

[2] neplést s existujícím jazykem pro popis diagramů navržený Brianem Kernighanem

- Operátor *beside* vezme dva obrázky a vytvoří výsledek tím, že je položí vedle sebe.
- Operátor *above* položí dva obrázky nad sebe.
- Operátor *rotR* otočí obrázek o 90° doprava (po směru hodinových ručiček).
- Operátor *flipV* otočí obrázek okolo vodorovné osy procházející prostředkem obrázku – otočí tedy obrázek "vzhůru nohama".
- Operátor *flipH* zase obrázek zrcadlově přetočí zleva doprava.
- Konstruktor *nil* vytvoří speciální obrázek, který nic neobsahuje. Lze ho považovat za nulární operátor, který použijeme, když budeme potřebovat nějakému operátoru předat „nic" na místo skutečného obrázku.
- Konstruktor *house* vytvoří obrázek s domečkem, který používáme v ukázkách.



Obrázek 2.2: Ukázka základních operátorů

Na obrázku **??** můžeme vidět, jak dopadnou aplikace těchto operátorů na základní obrázek s domečkem. V pořadí zleva doprava jsou obrázky vytvořeny takto:

- `flipV house`,
- `house ‘beside‘ house`,
- `(rotR house) ‘beside‘ (blank ‘above‘ house)`.

V prvním případě je obrázek jednoduše převrácen, V druhém je výsledek rozdělen na dvě poloviny a do každé je vykreslen stejný obrázek. Ten je automaticky přizpůsoben oblasti, do které je vykreslován, proto jsou oba domky roztažené do výšky. Druhý případ je podobný, jen obrázek vlevo je otočen a nad obrázek vpravo jsme umístili `blank`, abychom předešli jeho deformaci.

Přeskočme nyní otázku vygenerování grafického výstupu a také otázku jak získat počáteční obrázek (takový, jako je `house`), abychom měli od čeho začít s konstrukcí. K oběma se vrátíme při analýze implementace PICu v sekci **??**.

Tím jsme představili celý jazyk PIC – 5 operátorů. Kupodivu i s takto minimalistickými prostředky lze dosáhnout zajímavých výsledků.

2.5.2 Možnosti jazyka

Důležitou součástí každého jazyka je schopnost abstrakce – možnost definovat složené objekty a zacházet s nimi jako s jednoduchými.

Kolik úsilí bude potřeba vynaložit na implementaci takovýchto prostředků? Jaké bude mít uživatel možnosti definovat nové obrázky nebo kombinátory pomocí těch stávajících?

Zde je práce návrháře DSEL velmi jednoduchá, protože nemusí dělat nic – vše zdědí z hostitelského jazyka. Výše uvedené operátory nejsou nic jiného než funkce a uživatel s nimi může zacházet stejně jako se všemi ostatními funkcemi. V tom je právě síla vnořených DSL.

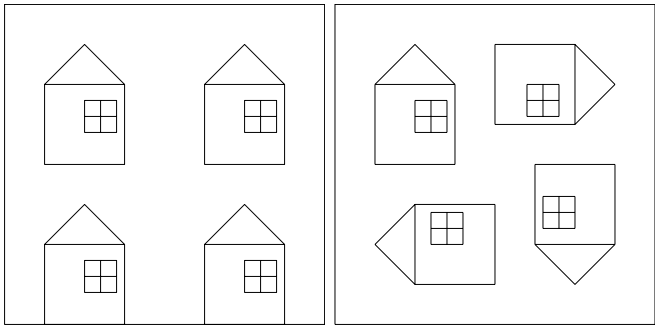
Pozornému čtenáři jistě neušlo, že ve výčtu operátorů chybí otočení o 180° a o 90° doleva. Není nic snazšího, než si takové operátory pořídit sám (tečka je v Haskellu operátor složení dvou funkcí):

```
rot180 = rotR . rotR
rotL   = rot180 . rotR
```

Než se pustíme do větších experimentů, bude vhodné zadefinovat si pomocné operátory, které zachytí různé složitější vzory. Operátor `quartet` bere čtyři obrázky a vydláždí s nimi čtverec, operátor `cycle` ho použije k tomu, aby zobrazil všechna čtyři otočení jednoho obrázku.

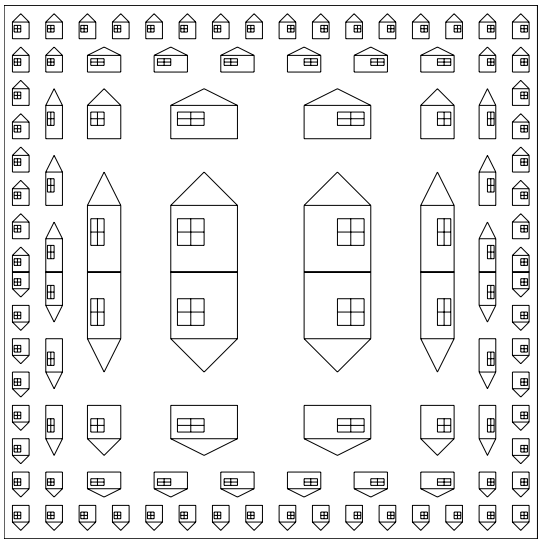
```
quartet p1 p2 p3 p4 = upper `above` lower
  where
    upper = p1 `beside` p2
    lower = p3 `beside` p4

cycle p = quartet p (rotR p) (rotL p) (rot180 p)
```

Všimněme si, jak je definování nových kombinátorů přirozené. Zde se právě projevuje velmi volná syntaxe Haskellu, která z něho činí ideální hostitelský jazyk.



Obrázek 2.3: `quartet house house house house` a `cycle house`

Na závěr ukážeme o něco složitější příklad, kdy vydláždíme plochu obrázky, které se postupně od středu zmenšují. Tento příklad využije rekurzi, aritmetiku i pattern matching, přestože jsme nic takového do našeho jazyka explicitně nedodali.



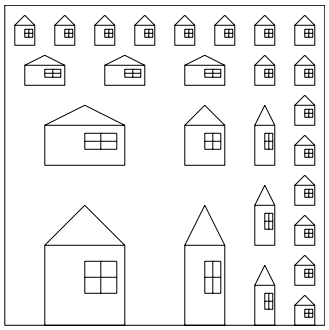
Obrázek 2.4: Dlaždičky, `tile 4 house`

```
up_split 1 p = p
up_split n p = (u~‘beside‘ u)  ‘above‘ p
  where
    u~= up_split (n-1) p
```

```
right_split 1 p = p
right_split n p = p `beside` (l `above` l)
  where
    l = right_split (n-1) p
```

Operace `up_split n` vykreslí zadaný obrázek, prostor nad ním rozdělí vertikálně napůl a do každého takto vzniklého sloupce vykreslí rekurzivně sebe sama (s n o 1 menším). Operace `right_split` se chová podobně jenom rozděluje prostor napravo místo nad sebou. S pomocí těchto operátorů už můžeme vytvořit kus výsledného obrazu (obr. **??**).

```
corner_split 1 p = p
corner_split n p = quartet top_left top_right bot_left bot_right
  where
    top_right = corner_split (n-1) p
    top_left  = up_split     (n-1) p
    bot_left  = p
    bot_right = right_split  (n-1) p
```



Obrázek 2.5: corner_split 4 house

Vidíme, že jsme získali horní pravý roh našeho obrázku – zbytek dostaneme snadno pomocí operátorů `flipV` a `flipH`.

```
tile n p = left `beside` right
  where
    left = flipH right
    right= corner `above` flipV corner
    corner = corner_split n p
```

2.5.3 Implementace

V předchozí části jsme představili doménově specifický jazyk PIC a ukázali část jeho možností. Víme dost na to, abychom mohli v tomto jazyce zapsat různé obrázky, ale prozatím nám chybí implementace! Bez dalšího nemáme jak tyto elegantní definice proměnit v grafický výstup.

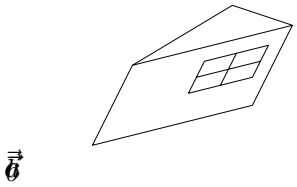
V případě klasického DSL implementovaného například v jazyce C bychom začali definicí gramatiky, napsáním parseru, určením vnitřní reprezentace načteného programu (výstup parseru) a poté napsáním interpretu nad touto reprezentací.

V případě DSEL ale syntaktické otázky odpadají a my se můžeme plně soustředit na sémantickou stránku věci. Základním stavebním kamenem je `Picture`, typ který reprezentuje obrázky.

Pro jednoduchost budeme vycházet z toho, že obrázek je černobílý a skládá se výhradně z jednoduchých čar, tj. neobsahuje křivky ani žádný druh oblastí vyplněných černou barvou.

Přirozenou myšlenkou je reprezentovat `Picture` jako seznam úseček. Musíme ovšem vzít v potaz operace, které s obrázky chceme provádět – potřebujeme je různě zmenšovat, otáčet a posouvat.

Všechny tyto transformace lze snadno vyjádřit pomocí trojice vektorů – jeden určí posunutí vůči počátku, zbylé dva potom rovnoběžnostěn, do něhož bude výsledek vykreslen. Konkrétní příklad můžeme vidět na obrázku **??**.



Obrázek 2.6: Vykreslení domku podle vektorů $\vec{a}, \vec{b}, \vec{c}$

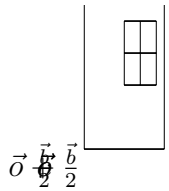
Využijeme toho, že Haskell je funkcionální jazyk a definuje `Picture` prostě

jako funkci, která bere na vstupu tři vektory, a vrací seznam úseček, jejichž souřadnice už jsou odpovídajícím způsobem přepočítány.

```
type Point = (Float,Float)
type Vector = (Float, Float)
data Primitive = Line Point Point deriving (Eq,Show)
type Picture = Vector->Vector->Vector->[Primitive]
```

Jakmile máme zvolenu takovouto reprezentaci je implementace operátorů velmi snadná:

```
-- pomocne funkce
half (a,b) = (a/2, b/2)
(a,b) `plus` (c,d) = (a+c, b+d)

beside::Picture->Picture->Picture
beside p1 p2 o~b c = p1 o~nb c ++ p2 (o~`plus` nb) nb c
    where nb = half b
```



Obrázek 2.7: beside

Jak funguje operátor *beside* můžeme vidět schématicky na obrázku **??**. Vektory $\vec{o}, \vec{b}, \vec{c}$ určují kam se má vykreslit výsledek. Operátor *beside* nejdříve přepočte

tyto vektory a dostane trojici $\vec{o}, \frac{\vec{b}}{2}, \vec{c}$, která odpovídá levé polovině celkové plochy a trojici $\vec{o} + \frac{\vec{b}}{2}, \frac{\vec{b}}{2}, \vec{c}$, která odpovídá pravé polovině. První obrázek aplikuje na první trojici, druhý obrázek na druhou trojici. Dostane tak dva seznamy úseček, které spojí pomocí standardního operátoru `++`. Ostatní operátory jsou implementovány obdobně.

```
above p1 p2 o~b c = p1 (o~‘plus‘ nc) b nc ++ p2 o~b nc
    where nc = half c

flipH p (ox,oy) (x1,y1) (x2,y2) = p (ox+x1,oy+x2) (-x1,-y1) (x2,y2)
flipV p (ox,oy) (x1,y1) (x2,y2) = p (ox+y1,oy+y2) (x1,y1) (-x2,-y2)
rotR  p (ox,oy) (x1,y1) (x2,y2) = p (ox+x2,oy+y2) (-x2,-y2) (x1,y1)
```

K tvorbě prvotních obrázků zadefinujeme dva konstruktory: `blank`, který vrátí prázdný obrázek (lze použít jako výplň u binárních operátorů) a `grid`, který bere seznam úseček a rozměry obrázku (předpokládá se, že všechny body jsou v 1.kvadrantu uvnitř vymezeného prostoru).

```
blank _ _ _ = []

grid::Integer->Integer->[Primitive]->Picture
grid width height lines (ox,oy) (x1,y1) (x2,y2) = map fun lines
    where
      w = fromIntegral width
      h = fromIntegral height
      fun (Line beg end) = Line (rp beg) (rp end)
      rp (x,y) = (ox + x*x1/w + y*x2/h, oy + y*y2/h + x*y1/w)
```

Posledním zbývajícím kamínkem do celé skládačky je funkce, která by výsledný obrázek vykreslila. Zde se uchýlíme k menšímu triku a použijeme další DSL, tentokrát externí – konkrétně *PostScript*. Díky němu je vykreslovací funkce velmi jednoduchá: nechá vykreslit obrázek do jednotkového čtverce a za každou úsečku výsledku přidá příkaz `moveto` a `lineto` do souboru:

```
renderPS::FilePath->Float->Float->Picture->IO ()
renderPS filename width height pic =
    writeFile filename $ toPs $ pic (0,0) (1,0) (0,1)
    where
      toPs lines = prelude ++
```

```
              (concatMap formatLine lines) ++
              closure
prelude    = "%!\n" ++ show width ++
              " " ++ show height ++ " scale\n" ++
              ".1 .1 translate\n" ++
              "0 setlinewidth\n" ++
              "0 0 moveto 1 0 lineto 1 1 lineto " ++
              "0 1 lineto 0 0 lineto\n"
closure    = "stroke\nshowpage\n"
formatLine (Line (x1,y1) (x2,y2)) =
    show x1 ++ " " ++ show y1 ++ " moveto " ++
    show x2 ++ " " ++ show y2 ++ " lineto\n"
```

2.5.4 Zhodnocení

V předchozí části jsme předvedli doménově specifický jazyk PIC. A to jak z hlediska uživatele tohoto jazyka tak i z hlediska programátora, který je zodpovědný za jeho implementaci.

Uživatel dostal k dispozici jazyk, který slouží svému účelu. Může pohodlně vyjadřovat různě strukturované obrázky pomocí primitivních operátorů. Nemusí se zatěžovat technickými detaily – výpočty jsou implicitní a schované.

PIC má na doménově specifický jazyk poměrně bohatou syntaxi. Lze v něm používat lokální definice pro pomocné hodnoty i globální definice pro pomocné funkce. Dokonce lze využít i systém modulů. Uživatelé si sami mohou psát rozšíření, která mohou umístit do samostatných modulů a potom se na ně při psaní programů odkazovat.

PIC je silně staticky typovaný. Špatně zformované programy budou odmítnuty už při překladu. Jazyk navíc dokáže sám odvodit typy výrazů, takže uživatel se nemusí zatěžovat psaním typových anotací.

Z hlediska implementátora PICu je nejspíš nejdůležitější, jak snadné je takovou implementaci vyvinout. Celá knihovna, tak jak jsme ji předvedli v předešlých částech, se vejde do jediného souboru o 56 řádkách! Neobsahuje žádný parser, žádné kontroly, zda je program syntakticky správně, jenom pár pomocných definic a definice operátorů, většinou 1 řádek na 1 operátor.

Vzhledem k tomu, jak je implementace PICu kompaktní, je také snadno udržovatelná. Je velmi snadné s ní experimentovat a zkoušet nové věci.

Důležitým aspektem této implementace je také její modulárnost. Za prvé mohou PIC rozšiřovat sami uživatelé tím, že si pojmenují některé kombinace

operátorů z knihovny jako jsme to viděli v případě otáčení o 180°. Za druhé je snadné rozšířit jazyk i o operace, které nelze vyjádřit pomocí těch stávajících (to už musí provést programátor jazyka). Operátor je charakterizován tím, jakým způsobem přepočítává vektory, které dostanou jeho argumenty jako parametr. V podstatě nezávisí na tom, jak jsou obrázky konkrétně reprezentovány. Tím se dostáváme k tomu, že i primitiva lze snadno měnit. V současnosti umí PIC pouze úsečky, ale nebyl by problém rozšířit ho o mnohoúhelníky, kružnice, elipsy, různé druhy křivek. Stejně tak by šlo primitiva parametrizovat. Zavést například tloušťku čar, nebo jejich styly (tečkované, čárkované, čerchované, ...). Jediné co by bylo potřeba upravit, je implementace funkce `grid` a `renderPS`. Operátory, ale i programy napsané uživateli, by zůstaly nedotčeny a fungovaly by stejně dál. Funkce `renderPS` je další směr, ve kterém lze PIC snadno rozšířit. Jedná se v podstatě o jakýsi „ovladač“, který se stará o vykreslení vnitřní reprezentace. Pokud by nevyhovoval výstup do PostScriptu, lze snadno napsat obdobné funkce s výstupem do jiných grafických formátu. Nebyl by ani problém využít nějakou knihovnu pro GUI a implementovat „ovladač“ tak, aby otevřel okno a obrázek do něj vykreslil pomocí volání grafických rutin této knihovny.

DSEL mají také nevýhody, které se v tomto případě zdají spíš marginální, ale i tak je zmíníme. Především není možné ovlivnit syntaktické nuance nového jazyka. V PICu lze použít komentáře, ale způsob jejich uvození je stejný jako v Haskellu – pomocí `--` nebo `{- ... -}`. Pokud bychom chtěli komentáře jako má C/C++, máme smůlu. Stejně tak pravidla týkající se klíčových slov, priority vestavěných operátorů, omezení kladených na jména identifikátorů a podobně jsou předem daná.

Typový systém PICu je také zděděný z Haskellu. Je sice dostatečně silný, ale pokud uživatel udělá chybu, může být chybová hláška matoucí vzhledem k tomu, že kompilátor Haskellu nemá žádnou znalost o PICu.

Pokud například uživatel zapomene, že funkce s názvem složeným z písmen lze v Haskellu používat jako infixové operátory pouze, pokud jsou uzavřeny do zpětných apostrofů, a napíše `blank beside blank` místo `blank ‘beside‘ blank`. Dostane od kompilátoru mírně matoucí chybové hlášení:

```
*FG> blank beside blank

Couldn’t match expected type ‘Vector’
  against inferred type ‘((t1, t2) -> (t1, t2) -> t -> [a])
                         -> ((t1, t2) -> (t1, t2) -> t -> [a])
                         -> (t1, t2)
                         -> (t1, t2)
```

```
                          -> t
                          -> [a]’
In the first argument of ‘blank’, namely ‘beside’
In the expression: blank beside blank
In the definition of ‘it’: it = blank beside blank
```

Kapitola 3

Pokročilé techniky

Haskell patří do rodiny staticky typovaných funkcionálních jazyků. Jako většina takových jazyků je založen na Hindley-Milnerově typovém systému [**?**].

Haskell byl od počátku koncipován jako platforma pro výzkum funkcionálních jazyků. Je to patrné jednak ze snahy o čistě funkcionální řešení (například v oblasti vstupu/výstupu), jednak z neustálého vývoje – pro Haskell bylo vyvinuto velké množství rozšíření. Někdy bývá také označován jako laboratoř typových systémů [**?**]. Nejvýznačnější vlastnost typového systému Haskellu – typové třídy – jsou jeho součástí už od úplného začátku, ale i ony se dočkali postupně několika zobecnění.

Typový systém Haskellu je díky svým rozšířením velmi pružný a umožňuje vyjádřit koncepty, které v jiných jazycích, jako je například Standard ML, vyjádřit nelze.

V této kapitole se budeme snažit zmapovat největší přínosy typového systému Haskellu pro implementaci doménově specifických jazyků.

3.1 Monády

DSL vnořená do funkcionálních jazyků jsou typicky vykonávána pomocí jednoduchých interpretů. Monády a monádové transformátory představují důležité ingredience k jejich efektivní (z hlediska náročnosti) implementaci. Usnadňují také další rozšiřování a změny interpretu v důsledku změny jazyka.

Na monády můžeme pohlížet jako na nástroj pro strukturování výpočtů. Základním stavebním kamenem programů zapsaných v monadickém stylu jsou hodnoty a posloupnosti výpočtů používajících tyto hodnoty. Monáda umožňuje programátorovi definovat nový výpočet jako sekvenci jednodušších výpočtů.

Přesný způsob jakým jsou tyto výpočty zkombinovány, záleží výhradně na konkrétní použité monádě. Programátor jej tedy nemusí pokaždé mezi dva výpočty ručně dopisovat. Ve skutečnosti ho ani nemusí detailně znát a může od něj abstrahovat.

Monády jsou v Haskellu realizovány jako instance typové třídy `Monad`:

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

Vidíme, že monáda je takzvaný typový konstruktor – typ parametrizovaný typem. Pro konkrétní typ `a` představuje `m a` reprezentaci výpočtu, který vrací hodnotu typu `a`.

Třída `Monad` sice obsahuje čtyři funkce, ale (`>>`) a `fail` mají implicitní definici. Vlastnosti každé monády jsou tedy určeny prvními dvěma operacemi – `return` a `>>=`. Význam jednotlivých operací je tento:

- `return` – konverzní funkce, převede hodnotu libovolného typu `a` na výpočet vracející tuto hodnotu.
- `>>=` (`bind`) – zřetězí dva výpočty za sebe, druhý výpočet dostane výsledek prvního na svůj vstup.
- `>>` – zkratka za `m1 >>= (\_ -> m2)`, použije se v případě, že druhý výpočet nepotřebuje výsledek prvního.
- `fail` – zkratka za `error`, vyvolá běhovou výjimku.

Funkce `fail` slouží k zabalení chybové zprávy do monády. Tím, že je součástí interfacu třídy `Monad`, umožňuje každé monádě implementovat vlastní pojem chyby – obecně jde o vlastnost, která se příliš nepoužívá a některými je považována spíše za chybu v návrhu jazyka.

Můžeme si všimnout určité nesymetričnosti v signatuře operace `bind`: na vstupu bere reprezentaci výpočtu a funkci, která bere jako vstup hodnotu, nikoliv výpočet. Nemůže tedy přímo aplikovat funkci na svůj první argument.

Pokud na monádu budeme pohlížet jako na druh kontejneru, tak operace `bind` funguje takto:

1. Vybalí hodnotu z kontejneru, který dostala jako první parametr.
2. Aplikuje na ní funkci, kterou dostala jako druhý parametr.

Zde je potřeba učinit několik důležitých pozorování:

- Operace `bind` vrací monadickou hodnotu. Přesnější by tedy bylo mluvit o tom, že vrací reprezentaci výpočtu, který provede kroky jedna a dva.
- Obecně neexistuje opak funkce `return`. Nelze tedy monadickou hodnotu `m a` zkonvertovat zpět na `a`. Jediné co lze s abstraktní monádou dělat, je konvertovat hodnoty na výpočty a vyrábět nové výpočty ze starých. Nelze tak „nekontrolovaně“ získat hodnoty ukryté v monádě.
- Konkrétní monády typicky mají funkci (často pojmenovanou `run`), která spouští výpočet a vrací jeho výsledek. Tato funkce je ovšem specifická pro danou monádu.

3.1.1 Maybe monáda

Abychom dostali jasnější představu, jak se s monádami pracuje, ukážeme si jednoduchý příklad. Řekněme, že máme skupinu funkcí, které mohou selhat, a potřebuje s jejich pomocí sestavit nějaký výpočet.

Tradiční řešení takové situace je rozšířit návratové hodnoty funkcí o chybový kód, který indikuje selhání, a poté toto selhání explicitně ošetřit. V případě, že chyba nastane v pomocné funkci, nelze ani určit, jak na chybu reagovat, a nezbude než chybovou hodnotu propagovat výš.

Tento způsob ovšem vyžaduje od programátora otrockou opakující se práci a je náchylný k chybám. Stačí někde zapomenout zkontrolovat návratovou hodnotu a chyba bude tiše ignorována. Navíc jsou kontroly rozprostřeny všude po celém programu, i když mají stále stejné schéma: ověřit návratovou hodnotu a v případě chyby přerušit výpočet a rovnou vrátit chybu.

Řekněme, že máme obchodní systém, ve kterém sledujeme produkty. Produkt má k sobě přiřazenou nabídku, ta je nabízena obchodníkem a ten má zase jméno. Každá část je ovšem nepovinná - obdoba NULL v databázích.

Chtěly bychom napsat funkci, která k danému produktu zjistí příjmení obchodníka, který ho nabízí (pokud existuje). Přímočaré, ale nepěkné řešení je:

```
data Maybe =  Nothing | Just a
getOffering :: Product  -> Maybe Offering
```

```
getMerchant :: Offering -> Maybe Merchant
getName     :: Merchant -> Maybe Name
getSurname  :: Name     -> Surname

getMerchantNameFromProduct product =
    case (getOffering product) of
      Nothing    -> Nothing
      Just offer ->
          case (getMerchant offer) of
            Nothing -> Nothing
            Just merch ->
                case (getName merch) of
                  Nothing -> Nothing
                  Just name -> Just (getSurname name)
```

Monadický přístup nám umožní abstrahovat opakující se vzor na jedno místo - datový typ `Maybe` definujeme jako monádu. Operace `bind` při složení s nedefinovanou hodnotou ignoruje druhý argument a rovnou vrací `Nothing`. V opačném případě vybalí vrácenou hodnotu (odstraní `Just`) a použije funkci předanou v druhém argumentu:

```
instance Monad Maybe where
    return a = Just a

    (>>=) Nothing  _ = Nothing
    (>>=) (Just a) f = f a

    fail  _          = Nothing
```

Nyní můžeme přepsat funkci pro získání příjmení:

```
getMerchantNameFromProduct p =
    getOffering p >>=
    getMerchant >>=
    getName     >>=
    (\name -> return (getSurname name))
```

Vidíme, že testy na `Nothing` zmizely – jsou skryty v operaci `bind`. Ne vždy máme ovšem k dispozici funkce v takovém tvaru, že je stačí jen pospojovat

pomocí `bind`. V takovém případě přijdou ke slovu anonymní funkce, stejně jako v posledním kroku našeho příkladu. Pokud by takových funkcí bylo zapotřebí více, což je běžný případ, byl by výsledný výraz velmi nepřehledný. Monády jsou natolik důležité, že Haskell pro ně obsahuje speciální do-syntaxi. Náš příklad lze pomocí ní přepsat do této podoby:

```
getMerchantNameFromProduct p = do
    offer <- getOffering p
    merch <- getMerchant offer
    name  <- getName merch
    return (getSurname name)
```

Výše uvedený kód je ekvivalentní předchozímu, jen předávané hodnoty jsou v tomto stylu explicitně pojmenované. Na první pohled tak vznikl v Haskellu specializovaný jazyk, který umožňuje za sebe skládat příkazy (i když Haskell žádné příkazy neobsahuje, jen výrazy). Na rozhraní těchto „příkazů“, tam, kde je v jazycích vycházejících z C středník, pak překladač vloží kód, který ošetřuje nedefinované hodnoty. Klíčovým faktem je, že tento kód je uživatelsky definovatelný, daný implementací operací `bind` a `return`. Různé instance třídy `Monad`, budou vykonávat různý kód. Proto se pro monády občas používá neformální označení „programovatelný středník“.

3.1.2 Vlastnosti monád

K jednotlivým operacím třídy `Monad a` se ještě vážou určitá pravidla, která zpřesňují jejich sémantiku a která se považují za obecně platná. Ověření těchto pravidel je mimo možnosti typového systému a tudíž je na programátorovi, aby si ohlídal, že jsou splněna. Jedná se celkem o tři tzv. *monadické zákony*:

- `(return x) >>= f == f x`
- `m >>= return == m`
- `(m >>= f) >>= g == m >>= (\x -> f x >>= g)`

V řeči algebry: `return` je identita zleva i zprava vůči `bind` a `bind` je asociativní operace.

Operaci `fail` nemá smysl definovat pro každou instanci, má však jednu důležitou vlastnost, kvůli které je zahrnuta do rozhraní třídy: pokud v do-syntaxi

dojde k chybě při pattern matchingu, je volána právě tato funkce (v normálním kódu dojde k výjimce, která ukončí běh programu).

Monáda může mít ještě další algebraickou strukturu. Standardní knihovna proto obsahuje rozšíření třídy `Monad`. Třída `MonadPlus` reprezentuje monády, pro které má smysl spojení dvou nezávislých výpočtů do jednoho. Tato třída má dvě funkce – `mzero` a binární operaci `mplus` (názvy jsou inspirovány sčítáním na přirozenými čísly).

```
class Monad m => MonadPlus m where
    mzero :: m a
    mplus :: m a -> m a -> m a
```

Funkce `mzero` vrací „nulový“ (prázdný, selhávající) výpočet a `mplus` sloučí dva výpočty do jednoho, přičemž v případě selhání se vrátí k druhé variantě – mělo by tedy platit:

```
    mzero `mplus` f = f
    f `mplus` mzero = f
```

Monáda `Maybe`, kterou jsme si představili v předešlé sekci, je instancí `MonadPlus`:

```
instance MonadPlus Maybe where
    mzero = Nothing

    Nothing `mplus` y = y
    x       `mplus` _ = x
```

3.1.3 Ostatní monády

Maybe monáda není zdaleka jediná. Standardní knihovna obsahuje několik velice užitečných instancí této třídy:

- Error – umožňuje vyhodit a zachytit výjimku.
- State – umožňuje udržovat si stav, který lze měnit.
- Reader – poskytuje sdílené prostředí, ze kterého lze číst.
- Writer – poskytuje log/výstupní proud, do kterého lze zapisovat.
- Cont – umožňuje přerušit výpočet a znovu ho obnovit.

- IO – umožňuje provádět I/O operace.

Jak jsem již uvedli v **??**, monáda je abstraktní datový typ, který slouží ke strukturování výpočtu. Konkrétní monády se liší v tom, jaké služby poskytují svým klientům. Kromě nutných funkcí z třídy `Monad a` exportuje monáda typicky ještě další funkce, pomocí kterých lze využít její speciální vlastnosti.

Například stavová monáda `State s` poskytuje funkce `get :: State s s` a `put :: s -> State s ()`, kterými lze měnit stav.

Monáda `Reader e` zase poskytuje funkci `ask :: Reader e e`, která poskytne prostředí ke čtení (obdoba `get` u monády `State s`) a `local :: (e->e) -> Reader e a -> Reader e a`. První argument je funkce, která transformuje prostředí, druhý je výpočet reprezentovaný reader monádou. Výsledkem je výpočet, který proběhne s modifikovaným (lokálním) prostředím.

3.1.4 Shrnutí

Monády jsou užitečný nástroj strukturování funkcionálních programů díky těmto svým vlastnostem:

1. **Modulárnost**. Umožňují sestavit výpočty z jednodušších výpočtů a oddělit strategii, s jakou jsou skládány, od výpočtů samotných.

2. **Pružnost**. Umožňují programům snazší adaptaci, protože monády koncentrují řízení výpočtu do jednoho místa, místo toho, aby bylo rozprostřeno po celém programu.

3. **Isolace**. Umožňují vytvoření výpočtů v imperativního stylu (sekvence příkazů). Ty jsou bezpečně isolovány od zbytku programu – lze tak jazyk rozšířit o různé side-efekty.

Haskell byl v počátku 90. let průkopníkem používání monád v programování. Monády se stali jeho pevnou součástí, protože dokáží vyjádřit I/O operace v čistě funkcionálním stylu (side-efekty jsou schované v monádě `IO`).

Postupem času začaly pronikat i do jiných (funkcionálních) jazyků jako je F# nebo rozšíření ML. Typový systém těchto jazyků ovšem nedokáže vyjádřit koncept monády přímo (nemají typové třídy) a ty jsou proto napevno zachycené v definici jazyka.

3.2 Monádové transformátory

Monády představují důležitý prostředek abstrakce a mohou ušetřit množství práce tím, že soustředí obecný mechanismus (zpracování výjimek, chybových hodnot, vnitřního stavu) do jednoho místa, místo toho, aby byl rozprostřen po celém zdrojovém kódu.

Standardní monády skýtají velké možnosti pro znovupoužití kódu. Zachycují ty nejdůležitější vlastnosti, které lze pomocí monád vyjádřit. Málokdy je potřeba chování, které by neodpovídalo některé standardní monádě. Z praktického hlediska však mají jednu velkou slabinu – většinou potřebujeme vlastnosti více standardních monád najednou.

Přímočarý způsob řešení je pro každou aplikaci navrhnout její vlastní monádu. Vzít monády, jejichž vlastnosti chceme, pouze jako inspiraci a pokaždé implementovat všechny operace znovu. Tím sice získáme absolutní kontrolu nad vlastnostmi monády, ale budeme zbytečně znovu vynalézat kolo.

Monádové transformátory [**?**] představují způsob, jak vyřešit tento problém systematicky. Tyto transformátory umožňují vytvořit monádu s požadovanou sadou vlastností prostým složením bloků, z nichž každý představuje jednu konkrétní vlastnost (transformátory víceméně odpovídají jedna k jedné monádám, které jsme představili v části **??**).

Otevírá se tak cesta k implementaci modulárních interpretů, jejichž jednotlivé části jsou na sobě nezávislé a přidání vlastností se obejde bez zásahu do stávajícího kódu.

To je zvláště důležité při vytváření DSEL, kde je potřeba počítat s tím, že minimálně zpočátku nebude jisté, jak přesně daný jazyk bude vypadat a tudíž ani co všechno bude umět jeho interpret.

V Haskellu nejsou transformátory součástí standardní knihovny, ale jsou umístěny v knihovně `mtl`. Stejně jako monáda je i transformátor typová třída. Definice je velmi jednoduchá:

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

Vidíme, že každý transformátor je typový konstruktor, který bere o jeden parametr víc než monády, prvním parametrem je právě monáda, na kterou se má aplikovat. Operace `lift` umožňuje konvertovat hodnoty v původní monádě na hodnoty v nové (transformované) monádě.

Řekněme, že chceme vytvořit monádu pro interpret, která si bude pamatovat vnitřní stav interpretu, bude poskytovat prostředí s proměnnými, které jsou v daném místě dostupné a bude schopná pracovat s výjimkami.

Pomocí monádových transformátorů to není těžké zařídit:

```
data Inter = ... -- vnitrni stav interpretu
data Env = ... -- informace o~promennych
data Err = ... -- typ pro vyjimky

data InterMonad a = ErrorT Err (StateT Inter
                                (ReaderT Env (Identity a))
```

Monáda `InterMonad` má všechny vlastnosti chybové, stavové i čtecí monády. Transformátory sami nejsou monádami, musíme je nejdřív na nějakou aplikovat. Zde jsme jako základ použili identitu, která nemá žádné zvláštní vlastnosti.

Pomocí transformátorů na sebe můžeme navrstvit jednotlivé funkcionality. Kýžený výsledek samozřejmě je, že vznikne monáda, která má sjednocení všech těchto vlastností. Bohužel to není až tak snadné. Není obecně možné propagovat jednotlivé vlastnosti monád nezávisle na ostatních.

V podstatě je nutné při implementaci transformátoru určit, jak bude interagovat s ostatními monádami. Operace, které jsou na sobě nezávislé, půjdou snadno, u závislých operací bude potřeba se zamyslet nad sémantikou složení. Z teoretického hlediska má skládání monád řadu zádrhelů:

- Počet možných kombinací roste kvadraticky s počtem transformátorů.
- Nelze vytvořit transformátory pro všechny monády.
- Skládání není komutativní.

Prakticky ovšem nejde o závažné problémy. Běžně si vystačíme se standardní sadou monád. Z nich je seznam jediná monáda, kterou nelze skládat, a tudíž ji stačí vždy použít jako výchozí monádu, na kterou se aplikují další transformátory. Knihovna `mtl` pak obsahuje transformátory pro ostatní monády a vyřeší programování interakcí za nás. Rozhodnutí o správném pořadí transformátorů nám naopak dává možnost přesněji určit sémantiku. Chceme-li například jak sdílený stav, tak možnost signalizovat chybu, použijeme `StateT` na `Error`, nebo `ErrorT` na `State`?

V prvním případě bude sdílený stav vždy k dispozici a hodnota výpočtu bude buďto chyba nebo běžná hodnota. V druhém případě je výsledek buďto v pořádku a máme k dispozici stav, nebo nastala chyba (a stav je už nepřístupný).

3.3 Chytré konstruktory

Chytré konstruktory jsou jednoduchou technikou mající široké použití. Modulový systém Haskellu umožňuje exportovat datový typ bez jeho konstruktorů. Vnitřní struktura datového typu je v takovém případě uživateli skryta – nemůže instance tohoto typu ani vytvořit ani na ně použít pattern matching. Tvůrce modulu tak má plnou kontrolu nad vnitřní reprezentací daného typu.

Chytré konstruktory jsou pak obyčejné funkce, které modul exportuje, aby uživateli umožnil vytvářet instance daného typu. Takové konstruktory mají oproti těm běžným několik výhod:

- Skrývají skutečnou implementaci. Lze měnit reprezentaci daného typu, aniž by se muselo zasahovat do kódu, který ho používá.
- Nemusejí odpovídat datovým konstruktorům jedna k jedné. Kompaktní reprezentace je výhodná, proto je snaha mít co nejméně datových konstruktorů – zmenší se tak počet případů, které je potřeba rozebírat při pattern matchingu. Na druhou stranu nic nebrání tomu, aby existoval velký počet chytrých konstruktorů – typicky budou navíc pokrývat některé časté případy použití.
- Konstruktory mohou kromě zkonstruování daného datového typu, provádět ještě další operace (odtud přívlastek „chytré“), například udržovat nějaký implementační invariant.

Při implementaci DSL se často tato technika používá, aby se uživatel DSL odstínil od některých detailů implementace.

3.4 Fantomové typy

Předpokládejme, že součástí DSL, které chceme do Haskellu vnořit, jsou i jednoduché aritmetické výrazy spolu s hodnotami typu boolean a podmíněnými příkazy. Budeme tedy mít operace jako `Plus`, `IsZero` a `If`. Všechny tyto operace se chovají jako kombinátory: berou na vstupu reprezentaci podvýrazů a podmínek a vracejí reprezentaci nových výrazů. Přímočará implementace by použila jednoduchý datový typ na reprezentaci operací:

```
data Expr = Bool Bool | Int Int |              -- Literals
            Add Expr Expr | Minus Expr Expr | -- Arithmetic Ops
            If Expr Expr Expr | IsZero Expr   -- Conditionals
```

Zde ovšem narazíme na drobný problém – některé výrazy při vyhodnocení vracejí číslo, jiné boolean a v daném kontextu nelze vždy použít libovolný typ: například výrazy `IsZero (Bool False)` a `Add (Bool True) (Int 10)` nemají dobře definovaný význam.

Nic ovšem nebrání uživateli DSL takové výrazy vytvářet. Interpret DSL potom musí za běhu kontrolovat, zda je výraz dobře zformovaný a případně vyvolat výjimku. To je ovšem nežádoucí, zvlášť když jde o chybu, kterou v běžném jazyce zachytí kompilátor.

Vytvoření zvláštního datového typu pro aritmetické výrazy a pro podmínky není řešení. Potřebujeme, aby byly spolu výrazy kompatibilní a bylo možné napsat jedinou funkci eval (s typem `Expr -> Result`), která vyhodnotí výraz zapsaný v DSL.

Stěžejní myšlenkou fantomových typů je rozšířit definici `Expr` na polymorfní datový typ – přidat typovou proměnnou, která označí jakého „typu“ výraz je:

```
data Expr a = Bool Bool | Int Int         |
              Add (Expr Int) (Expr Int)   |
              Minus (Expr Int) (Expr Int) |
              If (Expr Bool) (Expr a) (Expr a) |
              IsZero (Expr Int)
```

Výraz `Bool True` by měl mít typ `Expr Bool`, zatímco výraz `Add (Int 10) (Int 2)` typ `Expr Int` a `Add (Bool True) (Int 10)` by měl být typecheckerem odmítnut jako nesprávný.

Typový systém Haskellu neumožňuje, aby datové konstruktory vracely cokoli jiného než typ na levé straně rovná se, tedy `Expr a`. Proto se uchýlíme k technice „chytrých konstruktorů“ (viz **??**). V našem případě ale nebudou konstruktory příliš chytré:

```
bool :: Bool -> Expr Bool
bool = Bool

int :: Int -> Expr Int
int = Int

add :: Expr Int -> Expr Int -> Expr Int
add = Add

isZero :: Expr Int -> Expr Bool
```

```
isZero = IsZero

if_ :: Expr Bool -> Expr a -> Expr a -> Expr a
if_ = If
```

Vidíme, že jsou prakticky totožné s datovými konstruktory. Jediný rozdíl je v návratovém typu konstruktorů: například `IsZero` je typu `Expr Int -> Expr a`, kdežto `isZero` je typu `Expr Int -> Expr Bool`.

Díky tomu jsou výrazy zapsané pomocí těchto konstruktorů typově bezpečné – pokus o vytvoření nesmyslného výrazu nyní selže:

```
> add (bool True) (int 10)

<interactive>:1:5:
Couldn't match expected type 'Int' against inferred type 'Bool'
  Expected type: Expr Int
  Inferred type: Expr Bool
In the first argument of 'add', namely '(bool True)'
In the expression: add (bool True) (int 10)
```

Typ `Expr a` je v určitém ohledu dost neobvyklý:

- `Expr a` není kontejner – hodnota typu `Expr Int` je výraz, který se vyhodnotí na celé číslo, ale ne datová struktura, která obsahuje celé číslo.
- Nemůžeme zadefinovat obecnou mapovací funkci typu `(a -> b) -> (Expr a -> Expr b)` jako v případě mnoha jiných datových typů.
- Typ `Expr b` ani nemusí být obydlen – například neexistuje výraz, který by měl typ `Expr String`.

Právě proto, že typová proměnná v `Expr a` není spojená s žádnou jeho komponentou (neobjevuje se na pravé straně, leda jako parametr jiných typů), nazývá se `Expr a` *Fantomový typ*.

Technika fantomových typů je základem mnoha zajímavých použití typových systémů. Byla použita k odvození počátečních implementací rozšiřitelných záznamů [**?**, **?**, **?**], ke zprostředkování komunikace s COM komponentami [**?**], k otypování výrazů vloženého kompilátoru [**?**, **?**], k zaručení, že SML scriptlety produkují validní XHTML dokumenty [**?**].

Tato technika také není nijak specifická pro Haskell. Lze ji použít v mnoha jiných jazycích – všude, kde je podporován parametrický polymorfismus. Do této skupiny patří, mimo jiné, běžně používané staticky typované funkcionální jazyky jako OCaml, Standard ML, F#.

3.5 GADT

Fantomové typy umožňují vnoření typovaného jazyka do hostitelského jazyka. Takové vnoření sice zaručuje, že mohou být zkonstruovaná jen správně zformovaná data, nedovoluje už však typově bezpečnou dekonstrukci bez dalších běhových kontrol.

Z hlediska uživatele našeho jednoduchého DSL vypadá vše pěkně – místo datových konstruktorů se používají exportované funkce a typový systém nedovolí vznik nesmyslných hodnot. Interpret však už tak hezky nevypadá.

Výraz `Add (Int 10) (Int 10)` má stále typ `Expr a` nikoliv `Expr Int` a lze ho tedy předat i do funkce, která očekává `Expr Bool`. Není tedy naděje, že by náš interpret mohl mít typ, který bychom asi na první pohled očekávali – `(Expr a) -> a`.

Implementace interpretu pro jazyk z minulé sekce by vypadala takto:

```
data Result = B Bool | I~Int

eval::Expr a -> Result
eval (Add a b)  = I~(x+y) where
    I~x = (eval a)
    I~y = (eval b)

eval (IsZero a) = B (x == 0) where
    I~x = (eval a)

eval (If a b c) = if cond then (eval b) else (eval c) where
     B cond = (eval a)

eval (Bool b) = B b
eval (Int  i) = I~i
```

V podstatě tedy interpret vypadá stejně jako kdybychom fantomové typy nepoužili: vyhodnocení výrazu vrací buďto pravdivostní hodnotu nebo celé číslo.

Zavedeme tedy nový algebraický typ, který nám umožní rozlišit tyto dvě hodnoty a za běhu musíme vždy kontrolovat, která možnost nastala.

Díky fantomovým typům si můžeme být jistí, že pattern matching ve výše uvedeném kódu nikdy neselže (pokud byl výraz zkonstruován pomocí chytrých konstruktorů), ale přesto musíme tyto kontroly za běhu provádět.

Možným řešení tohoto problému jsou tzv. „Zobecněné algebraické datové typy“ (angl. Generalised Algebraic Data Types, zkráceně GADTs). Jde o rozšíření jazyka, které umožní u datových konstruktorů uvádět typové signatury. S jejich pomocí je jak definice datového typu, tak i interpretu pozoruhodně přímočará:

```
data Expr a where
    Bool   :: Bool -> Expr Bool
    Int    :: Int  -> Expr Int
    Add    :: Expr Int -> Expr Int -> Expr Int
    IsZero :: Expr Int -> Expr Bool
    If     :: Expr Bool -> Expr a -> Expr a -> Expr a

eval::Expr a -> a
eval (Bool b)   = b
eval (Int  i)   = i
eval (Add a b)  = eval a + eval b
eval (IsZero z) = 0 == eval
z~eval (If a b c) = if (eval a) then (eval b) else (eval c)
```

Na rozdíl od fantomových typů je ovšem potřeba, aby pro ně jazyk obsahoval podporu, přičemž se jedná o relativní novinku – do Haskellu se dostaly jako rozšíření překladače GHC teprve v roce 2006.

Kapitola 4

Optimalizace

Vnořená DSL jsou velmi efektivní technika, co se týče vynaloženého úsilí, protože velkou část své infrastruktury dědí DSL z hostitelského jazyka.

Bohužel, takto vytvořená implementace může být příliš neefektivní. Důvodem je to, že doménové datové typy jsou často reprezentovány pomocí algebraických datových typů a jsou interpretovány rekurzivní funkcí, která prochází celou reprezentaci programu. Režie interpretu se vyskytuje i v generovaném kódu, protože kompilátor hostujícího jazyka nemá žádnou znalost DSL a nemůže provést specifické optimalizace.

Ve většině případů není tato režie problematická, určitá neefektivita je cena, kterou si můžeme dovolit zaplatit výměnou za zjednodušenou implementaci, možnost rychlého prototypování a snadných změn jazyka. Dokladem toho je vzestup skriptovacích jazyků, které jsou oproti kompilovaným jazykům řádově pomalejší, ale poskytují za to prostředky pro snadnější vývoj aplikací.

Jsou ovšem oblasti, kde je výkon důležitý – například počítačová grafika. Znamená to však, že v případě takových domén bychom měli předem upustit od myšlenky implementovat DSEL z důvodu výkonu? Domnívám se, že nikoliv, protože existuje několik technik, které dokáží provádění programů zapsaných v DSEL podstatně zefektivnit. Jejich představení je obsahem této kapitoly.

Společným jmenovatelem těchto technik je metaprogramování, tj. psaní programů, které vytvářejí jiné programy nebo s nimi manipulují. Těmto programům se také říká generátory programů. Jazyk, kterým je napsán generátor, se označuje jako *metajazyk*, jazyk vygenerovaného kódu je pak *objektový jazyk*. Oba dva jazyky mohou být totožné (ale nemusí).

4.1 Pan

Jak už jsme zmínili v úvodu kapitoly, počítačová grafika je oblast vyžadující efektivní programy. DSEL navržené pro tuto doménu se tedy musí s tímto faktem nějak vypořádat. Možná řešení si předvedeme na jednom konrétním příkladu. Nejdříve si představíme samotné DSEL, v dalších částech (**??**, **??**) pak rozebereme dvě jeho efektivní implementace.

Pan [**?**] je podobně jako PIC (viz **??**) jazyk pro popis 2D obrázků. Oproti PICu je ale založen na trochu jiné myšlence. Ústředním typem Panu (stejně jako u PICu) je obrázek. Zde je to ovšem spojitá funkce, která bodům přiřazuje barvu. Použití takového modelu má řadu výhod: lze zadefinovat nekonečné obrázky (funkce jsou definované na celém $\mathbb{R}^2$), snadné vyjádření různých geometrických transformací (skládání funkcí), jednodušší práce s obrázky definovanými spojitými funkcemi jako sinus nebo cosinus (odpadá práce se zaokrouhlováním – to se provede až při finálním vykreslování uvnitř Panu).

```
type Image c   = Point -> c
type Point     = (Float, Float)
type Color     = (Float, Float, Float, Float)
type Transform = Point -> Point
```

Typ `Point` určuje bod v dvourozměrném prostoru, `Color` barvu (RGBA), a `Image` je parametrizovaný výsledkem, což umožňuje zadefinovat několik synonym:

```
type ImageC = Image Color
type Region = Image Bool
```

`ImageC` je barevný obrázek, `Region` lze použít pro vyjádření libovolně složitých masek. Část funkcí v Panu je univerzálních a dokáže pracovat s obecným typem `Image a`, jiné jsou specifické pro jeden datový typ. V Panu existuje konvence přidávat na konec funkce písmeno určující typ, nad kterým funkce pracuje (C pro obrázky, R pro regiony).

Pan obsahuje řadu užitečných metod, které se hodí při konstrukci obrázků:

- Různé 2D transformace: zvětšování, rotace, posuny ...
- Okolo typu `Region` je postavena skupina funkcí, které umožňují konstrukci masek pomocí množinových operací.

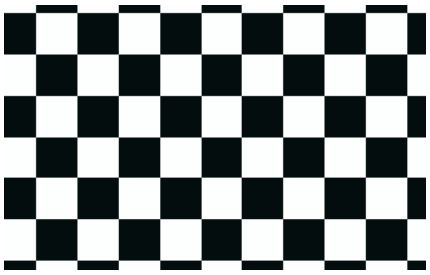
- Kromě kartézských souřadnic lze používat i polární souřadnice – existují konverzní funkce pro oba systémy – lze tak snadno vyjádřit různé deformace.

Ukazuje se, že tento přístup lze ještě o něco zobecnit. Je-li obrázek funkce prostoru, pak animace je funkce času. Pan tedy není omezen jen na statické obrázky, ale může vytvářet i animace a to s libovolnou frekvencí snímků (čas je opět spojitá veličina, stejně jako prostor).

```
type Time   = Float
type Anim a = Time -> Image a
```

Jeden obrázek řekne víc než tisíc slov, a proto tuto podkapitolu ukončíme jednoduchou ukázkou schopností Panu (převzatou z původního článku [**?**]).

```
checker (x,y)  = even (fx + fy) where
    fx = floor x
    fy = floor y

checkerBoard sqSize c1 c2 =
    ustretch sqSize (cond checker (const c1) (const c2))

ustretch s~im = im . scale (1/s, 1/s)
```



Obrázek 4.1: checkerBoard 30 blackH whiteH

Funkce `checker` je typu region. Pomocí sudosti součtu souřadnic (zaokrouhlených dolů) vytváří mřížkovanou masku s roztečí jeden pixel. Funkce `cond` je vlastně `if` „zvednutý“ do typu `Image`: `cond` *pod obr1 obr2* je obrázek definovaný takto, pokud je pro bod *p* splněna podmínka *pod p*, pak je barva tohoto bodu *obr1 p*, jinak *obr2 p*. Obrázek `checkerBoard` tedy nejdříve pomocí `cond` zkonvertuje masku na dvoubarevný obrázek a následně použije zvětšení, aby byla políčka daný počet pixelů veliká.

4.2 Vložený kompilátor

Literatura [**?**] popisuje, že parser pro LL(1) gramatiku lze jednoduše implementovat pomocí postupu zeshora dolů, kde každému neterminálu odpovídá funkce, která rekurzivně volá funkce odpovídající neterminálům na pravé straně pravidla. Volání skončí ve chvíli, kdy je zkonzumován vstup odpovídající celému pravidlu. Přechod od gramatiky k takové implementaci je jednoduchý, ale únavný, náchylný k chybám a v podstatě mechanický.

Kamin [**?**] přišel s myšlenkou využít moderní funkcionální jazyky jako programové generátory. Ve svém článku předvedl, jak pomocí vhodných kombinátorů lze v jazyce ML takové gramatiky zapsat. Každý kombinátor produkoval fragment kódu v jazyce C (z pohledu ML jde o obyčejný textový řetězec).

Gramatika popsaná v tomto DSEL je jeden výraz vytvořený pomocí těchto kombinátorů. Jeho vyhodnocení pak má za následek vypočtení velkého řetězce, který obsahuje program pro parsování jazyka daného touto gramatikou. V podstatě tak do ML vložil jednoduchý generátor parserů podobný programům jako Yacc nebo Bison (až na to, že tyto zpracovávají jiný typ gramatik).

Elliot, Finne a de Moor [**?**] se rozhodli tento postup použít při implementaci Panu. V případě Panu autoři tento přístup ještě vylepšili o to, že pro reprezentaci nepoužili netypované řetězce, ale zakódovali fragmenty kódu pomocí algebraických datových typů, díky čemuž je mohly snáze analyzovat a provádět optimalizace (Kamin se o žádnou analýzu nepokoušel, jednotlivé fragmenty byly prostě slepeny za sebe).

Místo interpretu tedy vložili do hostujícího jazyka *optimalizující kompilátor*. Implementace takového kompilátoru je v mnoha ohledech odlišná od interpretu, přesto se v případě Panu podařilo zachovat původní jednoduché rozhraní jen minimálně ovlivněné tím, že je DSEL kompilováno.

4.2.1 Architektura Panu

Pan je DSEL vnořené do Haskellu. Uživatel v něm tedy napíše svůj program a může k tomu využít všechny prostředky, které Haskell nabízí. Následně uživatel tento program spustí.

Na rozdíl od obvyklé implementace pomocí interpretu má takovéto spuštění za následek vytvoření optimalizovaného programu zapsaného v mezijazyku, který je oproti plnohodnotnému Haskellu podstatně omezenější. Tento mezikód je následně předán do jednoduchého kompilátoru, který vygeneruje kód v jazyce C. Haskell funguje jako programový generátor, ale v době běhu výsledného pro-

gramu už je úplně eliminován. Funguje tedy podobně jako makro systém, na rozdíl od většiny makro jazyků je však staticky typovaný a má větší vyjadřovací schopnost než cílový jazyk.

Pan sestává z několika málo typů: barvy, body jsou n-tice čísel; obraz je funkce, která bodu přiřadí barvu, a transformace je funkce, která převede bod na druhý bod. K tomu obsahuje sadu kombinátorů a pomocných funkcí. Haskell, na druhou stranu, obsahuje spoustu dalších typů a velmi užitečných funkcí, které může uživatel Panu libovolně použít. Díky tomu má také tento jazyk takovou vyjadřovací sílu.

Autoři zvolili při implementaci takovouto cestu:

1. Aritmetické výrazy jsou reprezentovány explicitně.
2. Panovské typy jsou upraveny tak, aby pracovaly s touto reprezentací.
3. Používají se funkce a n-tice hostujícího jazyka.

Funkce nejsou tedy nijak přímo reprezentovány a analyzovány. Při překladu budou jednoduše vyhodnoceny (jako obyčejné funkce v Haskellu) a teprve jejich výsledek se použije v reprezentaci. Výsledek programu využívající knihovnu Pan je tedy jeden velký výraz popisující strukturu výpočtu, který je dále zpracováván.

Tento velký výraz už je v „mezijazyku“, který je podstatně chudší než Haskell: neobsahuje seznamy, rekurzi, funkce vyšších řádů. Jsou v něm jen aritmetické operace, vybrané matematické funkce ($\sin, \cos \ldots$), operace s poli a podobné primitivní operace. Jedním z důsledků použití funkcí hostujícího jazyka je také *inlinování zadarmo* – ve výsledku se žádné funkce neobjeví.

V určitém smyslu se základní struktura Panu nezměnila: obraz je stále funkce, která bodům přiřazuje barvu. Tentokrát jsou to však *syntaktické body* (dvojice výrazů, které popisují souřadnice) a výsledkem je taky výraz (popis výsledné barvy).

V tomto bodě velmi pomohly typové třídy. Díky nim bylo možné většinu operací přetížit a funkce jako $\sin, \cos$ fungují i nad výrazy, aniž by došlo ke konfliktu jmen.

Pro „syntaktické“ varianty původních typů se v Panu používá koncovka E (z angl. expression). Například výrazy v pohyblivé desetinné čárce jsou reprezentovány typem `FloatE`, kde každá primitivní operace má svůj konstruktor.

4.2.2 Typy

Pan obsahuje datové typy pro výrazy různých typů - booleany, floaty, inty. Z hlediska implementátora jsou si velmi podobné a sdílí množství vlastností. Jejich implementace pomocí různých datových typů by vedla k velké redundanci. Z pohledu uživatele je to však výhodné, protože typová kontrola může odhalit více chyb.

Autoři v tomto případě využili techniku fantomových typů (viz **??**). Uvnitř kompilátorů je použit typ `DExp` (*dynamically-typed expression*), který reprezentuje výrazy všech typů.

Nad tímto typem pak zavedli „bezpečný“ typ `Exp a` a synonyma pro jeho použití:

```
data Exp a = E DExp
type BoolE = Exp Bool
type IntE = Exp Int
type FloatE = Exp Float
```

Takto mohli jednoduše zadefinovat typově bezpečné operace pomocí tříd: `instance (Num IntE)`, `instance (Num FloatE)`, ale už žádná `instance (Num BoolE)`. Operace $+, -$ a podobně jsou tak typově bezpečné a lze je aplikovat jen na výrazy správného typu.

Menší nepříjemnost představuje typ `Bool`, ten je totiž do Haskellu pevně zabudován a operátory jako `||` a `&&` nejsou součástí žádné třídy a mají napevno typ `Bool->Bool->Bool`. Pan tedy nemůže tato jména přetížit a poskytuje svoje operátory s mírně odlišenými jmény jako `.&&.` a `notE`, které fungují nad typem `BoolE` místo nad `Bool`.

4.2.3 Chytré konstruktory

Ruku v ruce s fantomovými typy kráčí i chytré konstruktory (viz **??**). Kromě poskytnutí staticky typovaného rozhraní ke konstrukci výrazů se v Panu starají i optimalizace:

- nahrazování konstantních výrazů konstantami (constant-folding),
- přeskupení těl do sebe zanořených if příkazů (if-floating),
- algebraické optimalizace – specifická přepisovací pravidla (použití identit a krácení).

Tyto optimalizace probíhají ve chvíli, kdy jsou hodnoty konstruovány. Celý proces tedy probíhá odspoda – nejdřív jsou optimalizované zanořené výrazy a konstruktory vytvářejí optimalizované výrazy z už optimalizovaných podvýrazů.

Tento postup má výhodu v tom, že nejsou opakovaně přepisovány sdílené podvýrazy, a nevýhodu v tom, že je z principu omezen na bezkontextové optimalizace.

4.2.4 Další zpracování

Chytré konstruktory produkují acyklický orientovaný graf reprezentující poměrně veliký (už částečně optimalizovaný) strom výrazu.

Mnoho výrazů může mít více než jednoho otce, ale toto sdílení je pouze implicitní. Aby bylo možné provést složitější globální optimalizace, je strom výrazu převeden na explicitní graf a odtud zpět na strom výrazu, tentokrát jsou však sdílené výrazy explicitně vyznačeny pomocí datového konstruktoru `Let`.

Jakmile je program v takovémto tvaru, lze provést další optimalizace: eliminaci společných podvýrazů, optimalizaci přístupu k polím a optimalizaci smyček.

Posledním krokem je potom konvertování zoptimalizované reprezentace do jazyka C, což už je celkem jednoduché.

4.3 Template Haskell

Template Haskell (TH) je rozšíření Haskellu 98, které umožňuje metaprogramování v době kompilace [**?**]. Je dostupný jako součást překladače GHC od roku 2002.

Haskell je v tomto systému jak objektový jazyk tak i metajazyk. Je tedy možné psát programy (v Haskellu), které generují jiné programy nebo jejich fragmenty (ty jsou také v Haskellu).

Template Haskell umožňuje snadnou integraci ručně psaného a generovaného kódu. Poskytuje prostředky pro usnadnění metaprogramování a také dodává určité záruky ohledně bezpečnosti.

Základní motivací Template Haskellu bylo dát uživatelům prostředek, kterým by si mohli rozšířit svůj kompilátor a naučit ho nové „triky“ [**?**] – například zavést typově bezpečnou funkci `printf`.

V pozadí celého TH je *uvozovací monáda* `Q` (z angl. quotation). Hodnoty tohoto typu představují fragmenty Haskellovského kódu. Má-li například výraz `e` typ `Q Exp`, potom `e` obsahuje *reprezentaci* Haskellovského výrazu (pod reprezentací si víceméně můžeme představit abstraktní syntaktický strom). Název

monády `Q` se odvozuje od *kvazi-uvození*, což je společný název pro dva konstrukty, kterými Template Haskell rozšiřuje syntax běžného Haskellu.

Prvním je *vlepení* (angl. splice): pokud překladač narazí na výraz `$(...)`, znamená to pro něj, že musí vnitřek vyhodnotit a výsledek „vlepit“ na místo původního výrazu. Kód uvnitř vlepení musí být typu `Q a` pro vhodné `a`.

Druhou konstrukcí jsou takzvané *Oxfordské závorky*, které dovolují zkonvertovat konkrétní syntaxi Haskellu (ručně psaný kód) na její reprezentaci. Tedy:

- `[| e |]`, kde `e` je výraz má typ `Q Exp`.
- `[d| e |]`, kde `e` je deklarace má typ `Q Dec`.
- `[t| e |]`, kde `e` je typ má typ `Q Typ`.

Vidíme tedy, že pomocí Template Haskellu lze vytvářet nejen výrazy, ale rovnou i typy a deklarace. Vztah mezi `$(...)` a `[| e |]` je takový, jaký bychom asi očekávali – platí: `$( [| e |] )` $\equiv$ `e`

Template Haskell může mít mnoho využití. To, které nás zajímá, je optimalizace. Pomocí Template Haskellu totiž můžeme část výpočtu přesunout do doby překladu. Vše si ilustrujeme na jednoduchém příkladu s mocněním čísel.

```
pow x n = if (n==0) then 1 else x * pow x (n-1)
cube = \x -> pow x 3
```

V kódu výše vidíme přirozenou definice funkce pro obecné mocnění (`pow`). Pod ní je pak funkce `cube`, definované pomocí `pow`, která provádí mocnění na třetí.

Z hlediska sémantiky je vše v pořádku. Funkce `cube` je však neefektivní. Pokud ji aplikujeme na nějaký výraz, zavolá se funkce `pow` s n rovno třem, následně se třikrát provede její zavolání (pokaždé s nižším n). Výsledek výrazu `cube x` je tedy vždy `x*x*x*1`, ale jeho vyhodnocení zahrnuje tři rekurzivní volání `pow`.

Template Haskell může pomoci právě v takových případech:

```
{-# LANGUAGE TemplateHaskell #-}
module Pow where

power :: Int -> Q Exp -> Q Exp
power n x = if (n==0) then [|1|] else [|$x * $(power (n-1) x)|]
```

```
mk_power :: Int -> Q Exp
mk_power n = [| \x -> $(power n [| x |]) |]
...
{-# LANGUAGE TemplateHaskell #-}
module Main where
import Pow
cube = $(mk_power 3)
main = do putStrLn $ show $ cube 3
...
*Main> main
27
```

V kódu výše můžeme vidět efektivní implementaci funkce `cube`. Program je rozdělen na dva moduly, protože nelze v době kompilace volat funkce ze stejného modulu (omezení Template Haskellu). Na začátku každého modulu si také můžeme všimnout speciální anotace, která zapíná podporu pro Template Haskell.

Jádrem programu je funkce `power`. Obě větve ifu obsahují Oxfordské závorky, takže vrací `Q Exp`. Pokud jí překladač zavolá s n rovno nule, vrátí reprezentaci jedničky. Pokud je ovšem n větší než jedna, tak vrátí reprezentaci kódu, který překladač bude muset teprve spočítat, protože obsahuje operátor vlepení. Vidíme tedy, že vyhodnocování výrazů v době kompilace se může libovolně zanořovat.

Není těžké dovodit, že výsledkem volání `power 3 [| x |]` je výraz `[| x*(x*(x*1)) |]`.

Funkce `cube` je tedy v tomto programu naprogramována bez zbytečné režie rekurzivních volání. Jako malý bonus můžeme označit, že kompilátor uvidí výraz `x*(x*(x*1))` jako celek a tudíž se s největší pravděpodobností zbaví přebytečné jedničky.

V Template Haskellu musíme rozlišovat mezi hodnotou a její reprezentací. Pro programátora ovšem může být obtížné vyznat se ve změti `[| |]` a `$()`. V tomto ohledu stojí za pozornost funkce `mk_power`. Na první pohled nemusí být vidět, proč je do `power` předáváno `[| x |]` místo `x`. Samozřejmě je to proto, že hodnota `x` bude známá až za běhu, a volání `mk_power` probíhá v době kompilace. Typový systém nás na takový omyl upozorní.

Explicitní syntaktické stromy

Obě anotace poskytují pohodlné rozhraní k metaprogramování, bohužel jejich vyjadřovací síla je omezená [**?**]: pokud například budeme chtít napsat funkci,

která vybere i-tý prvek z n-tice, tak se nám to nepodaří.

Template Haskell proto umožňuje explicitně konstruovat syntaktické stromy. Funkci, která vybírá i-tý prvek, potom můžeme zapsat takto [**?**]:

```
sel i n = lam [pvar "x"] (caseE (var "x") [simpleM pat rhs])
  where
    pat = ptup (map pvar as)
    rhs = var (as !! (i-1))
    as  = ["a" ++ show i | i <- [1..n]]
```

Funkce `lam`, `pvar`, `caseE` a další jsou zástupci speciální třídy funkcí – „syntaktických konstruktorů“. Tímto způsobem lze programově sestavit libovolný Haskellovský kód. Je to však způsob podstatně méně přehledný, hlavně proto, že každý element syntaxe Haskellu je potřeba explicitně konstruovat. Na oplátku takto můžeme v době kompilace například vytvořit n-tici pro libovolné n pomocí `ptup`, což jinak nelze.

Kvazi-uvození má oproti explicitně konstruovaným syntaktickým stromů kromě syntaxe ještě jednu výhodu - rozumí rozsahu platnosti proměnných. Napíšeme-li funkci:

```
f y = [| \x -> x + $y |]
```

Tak volání `f` bude vracet funkci, která bude přičítat hodnotu `y` ke svému argumentu, ať už se bude jmenovat jakkoliv – tedy i `x`. Výraz `$(f x)` bude ekvivalentní λ`x1 -> x1 + x`, nikoliv λ`x -> x + x`.

V případě explicitní konstrukce si musíme dávat pozor na nechtěné kolize jmen (aby nenastal druhý případ). Řešením je využít monádu `Q`, která obsahuje metodu `newName` typu `String -> Q Name`, která generuje „čerstvá“ jména.

Typování

Template Haskell je stejně jako Haskell silně typovaný. Otypovat výraz `$(f x) y` ovšem představuje výzvu, protože typ `$(f x)` závisí na tom, co funkce `f` dělá.

Typování je proto v Template Haskellu prokládané s kompilací. K syntaktickým konstruktům jako `[| |], $(...)` se vážou pravidla (např. nelze zanořit dvě `[| |]`), která lze snadno staticky ověřit již při parsování. Kód uvnitř Oxfordských závorek je obyčejný kód v Haskellu a je ověřen běžnými metodami, stejně tak kód uvozený `$(...)`. Ten druhý je však následně zkompilován, spuštěn a nahrazen výsledkem svého spuštění. Program je pak zkontrolován stejně, jako by vygenerovaný kód byl na dané místo napsán ručně.

Reifikace

Reifikace je způsob, jakým si programátor v Template Haskellu může zpřístupnit část interního stavu kompilátoru. Pomocí funkce `reify` se může programátor dotázat na typ dané funkce, získat reprezentaci deklarace nějakého typu (tj. získat syntaktický strom popisující jak daný typ vypadá, jaké má konstruktory apod.), zjistit místo (číslo řádky, jméno modulu) v programu, kde se nachází.

`Q` monáda také obsahuje konverzní funkci `qIO`, která převede výpočet typu `IO a` na typ `Q a`. Programy tak za času kompilace mohou provádět různé operace – načítat soubory nebo do nich naopak zapisovat.

4.4 PanTHeon

PanTHeon je název knihovny pro manipulaci s obrázky založenou na Template Haskellu, kterou vyvinul Sean Seefried [**?**].

Jde o reimplementaci Panu, která nepoužívá C jako svůj cílový jazyk, ale místo toho využívá metaprogramovací techniky, kterými se snaží převést vstupní Haskellovský kód (zapsaný v DSEL) na efektivní Haskellovský kód. Pro svůj přístup si Sean Seefried vybral název *rozšiřitelné metaprogramování* [**?**].

Základní architektura PanTHeonu je tato: přímočará implementace kombinátorů Panu pomocí interpretu, optimalizační modul napsaný v Template Haskellu a klient, který vykresluje výslednou grafiku.

Optimalizační modul je založen na třech optimalizacích:

1. unboxing aritmetických výrazů,
2. inlining,
3. algebraické optimalizace.

Unboxing

Standardně jsou všechny typy v Haskellu boxované, tj. nepoužívá se jejich primitivní reprezentace daná procesorem, na kterém program běží, ale jiná, která obsahuje další informace. Aritmetická operace tedy znamená konverzi (unboxing) argumentů na nativní reprezentaci, provedení příslušné instrukce a opětovnou konverzi tentokrát opačným směrem (boxing).

GHC obsahuje speciální rozšíření, které umožňuje pracovat s neboxovanými primitivními datovými typy. Díky němu se lze zbavit režie způsobené konverzemi. Navíc neboxované typy lze umístit přímo do registrů procesoru.

Optimalizační modul transformuje původní kód na kód využívající toto rozšíření.

Inlinování

Inlinování je v PanTHeonu implementováno vcelku jednoduše – existuje seznam inlinovaných funkcí (obsahuje všechny funkce PanTHeonu, ale už neobsahuje knihovní funkce samotného Haskellu) a pokud je v analyzovaném kódu taková funkce použita, je nahrazena ekvivalentní anonymní funkcí. Zde se spoléhá na kompilátor hostitelského jazyka, že sám provede β-redukci (dosazení argumentů do těla funkce) a následně kód zoptimalizuje.

Problém možného zacyklení při rekurzivně definovaných funkcí je vyřešen prostě – k inlinování dochází jen do určité předem stanovené hloubky.

Algebraické optimalizace

Algebraické transformace jsou v Template Haskellu jednoduše vyjádřitelné. Platí-li například pravidlo `empty ‘over‘ a = a`, lze ho zapsat takto [**?**]:

```
trans (AppE (AppE (VarE 'over) (VarE 'empty)) image) = trans image
```

Určitou komplikací je snad jedině to, že stejný výraz může vyjádřen mnoha způsoby, jednotlivá pravidla může být potřeba aplikovat opakovaně či rekurzivně. Explozi počtu různých případů, které je potřeba uvažovat, lze omezit převedením na kanonický tvar [**?**].

4.5 Zhodnocení

V této části se pokusíme zhodnotit jednotlivé přístupy k efektivní implementaci DSEL, které jsem rozebrali v této kapitole.

Vložený kompilátor je podstatně pracnější metoda. Zde můžeme původní interpretující kód zahodit (lépe je ponechat si ho jako referenční implementaci - může se hodit při ladění), protože z původního jazyka zbude jen torzo – exportované jména a signatury funkcí, implementace se musí udělat kompletně nová. Seefriedův přistup je flexibilnější. Původní kód zůstává zachován, jenom se připíše kód v Template Haskellu, který rozšíří kompilátor o doménově specifické optimalizace.

Z hlediska efektivity nejspíš vede vložený kompilátor, i když je to těžké přesně stanovit, vždy bude záležet na konkrétní aplikaci. Nabízí se přímé srovnání

přístupů Elliota a Seefrieda vzhledem k tomu, že oba implementovali tutéž knihovnu. Dle měření [**?**] ve většině případech vedl vložený kompilátor (mnohdy dosti výrazně). Ovšem je potřeba vzít v potaz i velikost obou programů – Elliotova implementace Panu je více než třikrát větší [**?**] systematicky se snaží o maximální optimalizaci, kdežto PanTHeon se spokojí s přímočarými metodami:

- Neprovádí všechny optimalizace.
- Používá heuristiky – inlinuje jen do úrovně k, místo aby zkonstruoval graf volání funkcí.
- Spoléhá se na kompilátor hostitelského jazyka - inlinuje prostým vlepením definice funkce uvnitř anonymní funkce na místo volání; dosazení do těla této funkce, řešení duplicit vzniklých inlinováním pomocí common subexpression elimination je ponecháno na kompilátoru.

Přesto má Template Haskell řadu omezení, která tento přístup znevýhodňují. Není možné v době kompilace analyzovat nebo používat právě zpracovávaný (a to i nepřímo) modul. Právě proto musel PanTHeon použít explicitní seznam funkcí [**?**]. Další velké omezení je nedostupnost typových informací. Díky způsobu jakým je typování prolínáno s kompilací, nemá kód v Template Haskellu vždy k dispozici všechny typové informace. Není například možné zjistit, jakého typu jsou literály [**?**].

Dalším důležitým rozdílem je, že vložený kompilátor analyzuje jen mezikód – tedy zrestringovanou množinu konstrukcí, tak jak si je zadefinoval programátor implementace. Rozšiřitelné metaprogramování se ovšem musí vyrovnat s plnohodnotným Haskellovským kódem, což znamená nutnost větší opatrnosti při transformacích a zároveň méně specifické informace o transformovaném kódu.

To ovšem neznamená, že by bylo použití Template Haskellu zbytečné. Měření, která provedl Seefried [**?**], ukázala dvojnásobné zrychlení oproti neoptimalizované verzi. Přičemž algebraické transformace (třída optimalizací, na kterou je Template Haskell asi nejvhodnější) nebyly do měření zahrnuty, protože jejich přínos byl na daných datech minimální.

Kapitola 5

Závěr

5.1 Shrnutí

Smyslem doménově specifických jazyků je zvýšit úroveň abstrakce, na které jsou programy popisovány, umožnit uživateli přirozené vyjádření problému nezastřené technickými detaily. Programy zapsané v DSL jsou typicky přehlednější a kratší, než jejich protějšky napsané v GPL.

Klasický způsob implementace DSL spočívá v napodobení implementace obecných programovacích jazyků. Na začátků se vytvoří syntaxe jazyka, stanoví se jeho vlastnosti, naprogramuje se parser, který převede textový vstup do interní reprezentace a ta se pak dále zpracovává.

Obtížnost takové implementace představuje hlavní překážku nasazení DSL. Je třeba vyvážit přínos DSL a náročnost vývoje a údržby nástrojů na jeho zpracovaní. Častou implementační strategií je alespoň část práce na novém DSL převést na již hotové komponenty: použít generátor parserů, přizpůsobit si překladač jiného jazyka, přímočaře překládat z DSL do existujícího programovacího jazyka a podobně.

V této práci jsme zabývali hlavně vnořenými doménově specifickými jazyky [**?**] (DSEL). Ty využívají přímo jiný obecný programovací jazyk, aniž by do něho zasahovaly (měnily kompilátor, přidávaly preprocessing). Celý jazyk je realizován pomocí sady operátorů, funkcí a typů, které v sobě zachycují doménovou znalost. Program zapsaný v DSEL je tedy zároveň i programem zapsaným v hostitelském jazyce.

Pro účely práce jsme použili funkcionální jazyk Haskell, protože má vlastnosti, které z něj činí vhodný hostitelský jazyk.

Ukázali jsme, že vytvořit funkční DSEL je velmi jednoduché a nevyžaduje

žádné speciální znalosti z oblasti překladačů. Výsledná implementace je velmi kompaktní a modulární. To předurčuje nasazení DSEL jako prototypovacího nástroje.

Důležitou technikou implementace DSEL je využití monád. Ty jsou schopny abstrahovat obecné vlastnosti kódu (způsob ošetřování výjimek, sdílení globálního stavu apod.) do jednoho místa.

Kód psaný v monadickém stylu má několik výhod. Jednak jsou abstrahované vlastnosti schované v monádě a tak je kód čistší – technické detaily nejsou řešeny na úrovni kódu, který monádu používá. Jednak lze pro nejčastější vlastnosti použít už předpřipravené monády a tak podstatně zredukovat úsilí potřebné k implementaci. V práci jsme ukázali, jak lze tento přístup ještě zjednodušit pomocí monádových transformátorů, které dokáží vytvořit monádu s potřebnými vlastnostmi jako kombinaci knihovních monád.

Při vkládání do staticky typovaného jazyka dědí DSEL typový systém svého hostitele. Ukázali jsme dvě techniky, které toho dokáží využít a přizpůsobit si typový systém pro své účely. Zobecněné algebraické datové typy (GADT) jsou elegantnější, ale vyžadují podporu překladače, přičemž nejde o běžnou vlastnost programovacích jazyků. Fantomové typy jsou naopak technikou univerzální, použitelnou v mnoha jazycích.

Pokud vyžadujeme od DSEL i efektivitu, musíme tomu přizpůsobit jeho implementaci. V práci jsem prozkoumali dva přístupy k efektivní implementaci: při prvním kombinátory daného DSEL nevyjadřují přímo výsledný program, místo toho jen staví syntaktický strom, ten je pak dále zpracováván a přetvářen až do podoby finálního programu. Druhý způsob je založen na rozšíření běžné implementace o modul, který provádí transformace nad zdrojovým kódem.

První způsob je náročný, vyžaduje znalosti z oblasti návrhu překladačů, ale ukázal se být praktický a přináší výsledky. Druhý způsob je podstatně omezenější. Hlavně proto, že musí pracovat s obecným kódem, který je bohatší než DSEL. Lze ho tak použít jen v omezené míře na určité jednoduché dobře formulovatelné optimalizace.

5.2 Kdy použít DSEL?

V práci jsme shrnuli některé přístupy používané při návrhu DSEL. Kdy je tedy jejich nasazení vhodné a kdy raději použít externí DSL?

V podstatě můžeme DSEL přistup doporučit vždy, pokud nepotřebujeme stoprocentní kontrolu nad DSL, včetně detailů syntaxe a chybových hlášek. Zvláště

výhodné jsou, pokud si nejsme jisti návrhem jazyka a budeme potřebovat nějaký prostor pro vyhodnocení více přístupů.

Klíčovým bodem při návrhu DSEL je výběr hostitelského jazyka. Samotné DSEL bývá typicky modulární a snadno změnitelné, ale hostitelský jazyk předurčuje naše možnosti při jeho implementaci. Případný přechod k jinému jazyku by znamenal nejen přepis celé implementace, ale narazily bychom i při pokusu přenést některé techniky (například monádové transformátory podstatně využívají bohatý typový systém Haskellu a těžko by se přenášely do jiných jazyků).

Z praktického hlediska je potřeba vzít v úvahu také nároky hostitelského jazyka. Pokud bychom se mu vyhnuli při implementaci běžného programu z dané domény, ať už kvůli chybějícím nástrojům, špatné kvalitě překladačů, nebo třeba příliš velkým paměťovým nárokům, pak nemá smysl v něm vyvíjet DSEL.

Jazyky vhodné pro implementaci DSEL nejsou mezi běžně používanými. Rozhodnutí vytvořit DSEL tedy s sebou přináší nejen problém s návrhem jazyka, ale i se zavedením nového jazyka do projektu. Podobně uživatelé mohou mít větší problémy naučit se nový jazyk, protože dědí vlastnosti hostitelského jazyka a je tedy bohatší.

5.3 Jiné přístupy

Způsob vnoření doménově specifického jazyka uvedený v této práci není jediný možný. Vytváření takových jazyků má delší tradici v rodině LISPových jazyků [**?**]. Slabinou tohoto přístupu je jednak velká syntaktická odlišnost LISPu od ostatních jazyků, kterou vnořené DSL zdědí. Dále pak LISP není staticky typovaný.

Vnořené DSL vlastně vytváří jakousi vrstvu nad obecným programovacím jazykem. Má vlastní abstrakce, vlastní invarianty. V dynamicky typovaných jazycích je třeba být obzvláště opatrný, protože případná chyba na úrovni DSL se projeví až za běhu v obecném kódu, který o DSL nic neví.

Další přístup navrhl Sheard [**?**], který vyvinul metodiku pro vytváření DSL s ohledem na jejich efektivitu. Tato metodika se příliš neodlišuje od postupů uvedených v této práci. Rozdílem je využití podpory pro vícestupňové programování (angl. multistage programming) v MetaML. Toto rozšíření jazyka ML umožňuje pomocí explicitních anotací rozdělit výpočet do více fází. Výsledkem každé fáze je kód, který má být spuštěn v další fázi. Kód je tedy generován za běhu a může být optimalizován pro dané hodnoty parametrů. Sheard [**?**] ukázal, jak lze tento přístup použít k eliminaci režie interpretu.

Celý postup můžeme zařadit někam mezi vložený kompilátor a rozšiřitelné metaprogramování, kterými jsme se zabývali v kapitolách **??** a **??**. Výchozím bodem je jednoduchý interpret, který je ale postupně upravován, a pomocí anotací zefektivňován. Oproti rozšiřitelnému metaprogramování, tedy není původní kód nedotčen – optimalizace se dotýkají kódu interpretu; na druhou stranu je základní kostra zachována narozdíl od vloženého kompilátoru.

Tento přístup je plně závislý na podpoře vícestupňové programování – jazyky, které mají taková rozšíření, jsou ML (MetaML) a Ocaml (MetaOcaml).

Příloha A

Stručný přehled jazyka Haskell

V této příloze probereme nejdůležitější součásti programovacího jazyka Haskell. Naším záměrem je pouze pokrýt syntaktické konstrukty, které se vyskytují v této práci, aby bylo možné porozumět kódu bez předchozích znalostí.

A.1 Definice

Program v Haskellu je posloupnost definic, například:

```
msg = "Hello, world!"
msg2 = upper msg                        -- "HELLO, WORLD!"
upper string = map toUpper string
```

První řádka definuje `msg` jako konkrétní řetězec, druhá definuje `msg2` jako výsledek aplikace funkce upper na `msg` (výsledek je v komentáři), třetí definuje funkci, která převede řetězec na velká písmena (`map` a `toUpper` jsou standardní funkce jazyka).

Můžeme si všimnout, že argumenty funkce se zadávájí prostým zapsáním za funkci, nepoužívají se žádné závorky nebo čárky jako oddělovače. Pořadí definic nehraje roli. Narozdíl od jazyka C nepotřebujeme dopředné definice, můžeme se odkazovat i na funkce, které budou definovány někde dále.

Haskell je čistě funkcionální jazyk: není zde žádný operátor přiřazení, který by měnil hodnoty proměnných, žádné příkazy. Jednotlivé definice se skládají výhradně z výrazů, přičemž funkce jsou na stejné úrovni jako ostatní datové typy. Lze je tedy předávat do funkcí jako parametry a mohou být i výsledky funkcí.

Haskell podporuje pattern matching – na levé straně funkce nemusí být jen jména proměnných, mohou se tam vyskytovat i částečně instanciované datové typy. V tom případě se definice napravo použije jen v případě, že skutečný argument odpovídá tomuto vzoru. Každá funkce může být definovaná více rovnicemi – potom se postupně zkouší jedna podruhé (v pořadí v jakém jsou v zdrojovém kódu) dokud se matching nepodaří, případně se vyvolá běhová výjimka, pokud se nenajde žádná použitelná definice (což je u staticky typovaného jazyka nepříjemné – proto kompilátor varuje v případě, že nejsou pokryty všechny možnosti).

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

Takto lze definovat funkce, která vrací délku seznamu (`:` je datový konstruktor seznamu).

A.2 Infixní operátory

Haskell dovoluje definovat i nové infixní operátory – binární funkce, které se zapisují mezi své argumenty. Funkce, jejíž název sestává ze speciálních znaků jako `&!+-/` je automaticky infixní operátor. Haskell má navíc unikátní vlastnost: běžnou binární funkci lze použít jako infixní operátor, pokud se její název zavře do zpětných apostrofů – tyto dva výrazy jsou tedy ekvivalentní:

```
elem element list
element `elem` list
```

Lze dokonce každému operátoru určit asociativitu a prioritu.

A.3 Typy

Haskell je staticky typovaný jazyk, jehož typový systém je založený na Hindley-Milnerově typovacím systému. Právě typový systém je nejčastěji rozšiřovanou částí standardního Haskellu. Ve většině případů dokáže překladač odvodit typy výrazů sám, ale programátor může (ve výjimečných případech i musí) oanotovat výraz jeho typem:

```
msg :: String
upper:: String->String
map :: (a -> b) -> [a] -> [b]
```

Symbol "`->`" značí funkci a je asociativní doprava, "`[]`" je označení pro datový typ seznam. Konkrétní datové typy začínají velkým písmenem, typové proměnné malým.

A.4 Curryfikace

Funkce v Haskellu berou pouze jeden parametr. Víceparametrické funkce jsou realizované pomocí *curryfikace*: funkce která bere n parametrů je ve skutečnosti funkce s jedním parametrem, která vrací funkci, která „má" $n-1$ parametrů.

Volání `map toUpper msg` je totéž, co `((map toUpper) msg)`. Víceparametrické fukce jsou tedy jen syntaktický cukr, který zjednodušuje zápis. Důležitým důsledkem tohoto přístupu je možnost *částečně aplikovat funkci*. Funkci `upper` jsme mohli definovat také takto (i když `map` bere dva argumenty):

```
upper = map toUpper
```

A.5 Lokální definice

Výrazy na pravé straně definice mohou být dost komplikované a některé části se mohou opakovat. Proto lze pojmenovat podvýrazy pomocí klíčového slova `where`, přičemž definice v těle funkce a ve `where` klauzuli mohou být vzájemně rekurzivní.

```
max3 a b c = max2 c d
 where
   max2 a b = if a < b then a else b
   d        = max2 a b
```

Jednotlivé definice do sebe můžou být libovolně zanořeny. Pro stanovení oboru platnosti jednotlivých identifikátorů se používá odsazení. Detaily jsou trošku komplikované, ale v podstatě blok začíná s nedokončeným řádkem a končí řádkem, který má stejné nebo menší odsazení. V praxi tak odpadá používání středníků a složených závorek k určování bloků, i když jsou součástí syntaxe jazyka.

A.6 Polymorfismus

Polymorfismus označuje schopnost nějakého kódu pracovat s více typy hodnot. Nejčastěji je tento pojem skloňován v souvislosti s objektově-orientovaným programováním, kde většinou odkazuje na schopnost objektu zareagovat na zaslanou zprávu podle svého. Lze tedy napsat obecný kód, který vždy zasílá dané zprávy, ale s různými typy objektů se bude program chovat různě. Polymorfismus je důležitá vlastnost jazyka, protože zpřehleďnuje kód a umožňuje vyhnout se zbytečné redundanci psaním více verzí stejného kódu. Existuje ovšem více druhů polymorfismů [**?**], mezi kterými je potřeba rozlišovat.

A.6.1 Parametrický polymorfismus

Je typem polymorfismu, který je Haskellu vlastní. Typ, se kterým funkce pracuje, může obsahovat typovou proměnnou. Potom funkce zpracovává libovolný typ, který vznikne dosazením za tuto proměnnou. Kupříkladu funkce `length :: [a] -> Int` je polymorfní: spočte délku seznamu, ať už jde o seznam čísel, znaků nebo třeba funkcí. Polymorfní funkce níjak nevyužívá vlastnosti parametrizovaného typu (všechny instance jsou pro ní stejné).

V mainstreamových jazycích se tento polymorfismus objevuje pod názvem generika.

A.6.2 Ad-hoc polymorfismus

Umožňuje hodnotám projevovat různé chování při různém otypování. Formou ad-hoc polymorfismu je přetěžování – funkce mají různé implementace pro různé typy. Haskell podporuje tento druh polymorfismu pomocí konstruktu *typových tříd*. Typová třída definuje signatury funkcí, které musejí její *instance* implementovat.

Polymorfní kód tyto „abstraktní“ funkce používá a v závislosti na tom, jakého konkrétního typu jsou jejich argumenty, se určí příslušné definice. Použití dané třídy se projeví v kontextu typu.

```
noteq :: (Eq a) => a -> a -> Bool
noteq a b = not (a == b)
```

Operátor (`==`) je součástí třídy `Eq`, která reprezentuje hodnoty, které lze mezi sebou porovnávat na rovnost. Funkce `noteq` dokáže porovnávat na nerovnost

libovolné dvě instance (stejného typu) této třídy. Konkrétní kód použitý k porovnávní závisí na typu parametrů.

Objektově orientované jazyky jsou známé tím, že dokáží přetěžovat metody podle příjemce zprávy (tzv. virtuální metody), přetěžování pomocí ostatních parametrů (multimetody) je už méně časté.

Jednotlivé implementace nemusí mít spolu moc společného, jde jen o kód sdružený pod společným jménem. Přetěžování je v podstatě jen doručovácí mechanismus – odtud název ad-hoc.