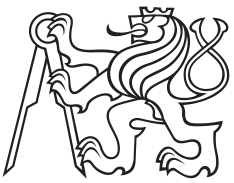
Bakalářská práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra počítačů

Podpora aspektově orientovaného přístupu v algebraických specifikacích

Denis Baručić
Softwarové technologie a management
Softwarové inženýrství

Květen 2017
Vedoucí práce: Ing. Jiří Šebek

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra počítačů

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: Baručić Denis

Studijní program: Softwarové technologie a management
Obor: Softwarové inženýrství

Název tématu: Podpora aspektově orientovaného přístupu v algebraických specifikacích

Pokyny pro vypracování:

Prostudujte možnosti implementace aspektově orientovaného programování (AOP).
Navrhněte metodiku, která využije AOP přístup pro přepisovací pravidla v jazyce Maude.
Vytvořte framework pro použití AOP v maude včetně aspect weaveru, který otestujte.
Demonstrujte použití na modelových příkladech, empiricky otestujte a porovnejte.
Výsledná implementace musí být snadno rozšiřitelná. Zhodnoťte výhody a možná omezení řešení.

Seznam odborné literatury:

1. ŠEBEK, J. and K. RICHTA. Aspect-oriented User Interface Design for Android Applications [online]. In: DATESO 2015. Databases, Texts, Specifications, and Objects 2015, Nepřívěc u Sobotky, Jičín, 2015-04-14/2015-04-16. Praha: MATFYZPRESS, vydavatelství Matematicko-fyzikální fakulty UK, 2015, pp. 121-130. CEUR Workshop Proceedings. vol. 1343. ISSN 1613-0073. ISBN 9788073782856. Available from: http://www.cs.vsb.cz/dateso/2015/
2. Kiczales; Gregor J.; Lamping; John O.; Lopes; Cristina V.; Hugunin; James J.; Hilsdale; Erik A.; Boyapati; Chandrasekhar, Aspect Oriented Programming, United States Patent 6,467,086, October 15, 2002
3. ŠEBEK, J., M. TRNKA, and T. ČERNÝ. On Aspect-Oriented Programming in Adaptive User Interfaces. In: Proceedings of the 2nd International Conference on Information Science and Security. The 2nd International Conference on Information Science and Security, Seoul, 2015-12-14/2015-12-16. Piscataway: IEEE, 2015, pp. 147-151. ISBN 978-1-4673-8611-1

Vedoucí: Ing. Jiří Šebek
Platnost zadání do konce letního semestru 2017/2018

L.S.

prof. Dr. Michal Pěchouček, MSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 23.2.2017

Poděkování / Prohlášení

Chtěl bych poděkovat vedoucímu bakalářské práce Ing. Jiřímu Šebkovi za ochotu a cenné rady.

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 24. 5. 2017

.....

Abstrakt / Abstract

Tato práce se zabývá přidáním podpory aspektově orientovaného programování pro systém algebraických specifikací Maude. Byl vytvořen jazyk pro definování aspektů. Dále byl naimplementovaný aspect weaver podporující daný jazyk. S využitím vytvořeného systému lze aplikovat aspekty pomocí přepisování hodnot proměnných jednotlivých rovnic. Výsledný systém umožňuje snížit počet duplicit v kódu.

Klíčová slova: aspektově orientované programování, Maude, aspect weaver

The purpose of this thesis is to add AOP support to Maude system. Within the thesis, a language for aspect definition was created. An aspect weaver, supporting the language, was implemented, too. Using the created system, it is possible to apply aspects by rewriting variable values. The resulting system allows programmer to reduce the number of duplicate code.

Keywords: aspect oriented programming, Maude, aspect weaver

Title translation: Support of aspect-oriented approach in algebraic specification

Obsah /

1 Úvod 1
1.1 Cíle práce 1
2 Rešerše 2
2.1 Aspektově orientované programování 2
2.1.1 Oddělení zodpovědností ... 2
2.1.2 Průřezové problémy 2
2.1.3 Paradigma 3
2.1.4 Join point model 3
2.1.5 Aspect Weaving 4
2.2 Maude 4
2.2.1 Syntaxe 4
2.3 Haskell 5
2.3.1 Čisté funkce a funkce s vedlejším efektem 5
2.3.2 Líné vyhodnocování 5
2.3.3 Statické typování 6
2.3.4 Příklad kódu 6
3 Související práce 7
3.1 AspectJ 7
3.1.1 JPM 7
3.1.2 Využití 7
3.2 AspectFaces 8
3.2.1 Koncept 8
3.2.2 Využití 8
3.3 AOP v Prologu 9
3.3.1 Koncept 9
3.3.2 Využití 9
4 Analýza 10
4.1 Požadavky 10
4.1.1 Povinné požadavky 10
4.1.2 Přirozené požadavky 10
4.2 Syntaxe Maude 11
4.2.1 Závěr 11
5 Design 12
5.1 JPM 12
5.1.1 Join points 12
5.1.2 Pointcut 12
5.1.3 Advice 13
5.2 Jazyk 13
5.2.1 Jazyk pro definici pointcut 13
5.2.2 Ukázka jazyka 14
5.3 Aspect weaver 15
5.3.1 Obecný návrh programu . 15
5.3.2 Načtení kódu 15
5.3.3 Weaving 15
5.3.4 Ukázka weavingu 16
5.3.5 Zapsání výsledku 17
5.3.6 Rozhraní programu 17
5.4 Omezení 17
5.4.1 Omezení jazyka 17
5.4.2 Vlastnosti aspect weaveru 18
6 Implementace 19
6.1 Výběr jazyka 19
6.2 Program stack 20
6.2.1 Struktura projektu 20
6.3 Moduly programu 20
6.4 Problém výjimek 21
6.4.1 Monády Maybe a Either . 21
6.4.2 Monáda Result 22
6.5 Program 23
6.6 Rozhraní příkazového řádku ... 24
6.7 Načítání zdrojového souboru .. 24
6.7.1 Preprocessing 25
6.8 Weaving 25
6.8.1 Modul Parser 25
6.8.2 Sestavení aspektového kontextu 26
6.8.3 Sestavení operačního kontextu 27
6.8.4 Transformace 27
6.8.5 Transformace rovnic 27
6.8.6 Aplikace advices 28
6.9 Zapsání výsledku 29
6.9.1 Postprocessing 29
6.9.2 Výstup do souboru 29
7 Testování 30
7.1 Framework HUnit 30
7.2 Jednotkové testy 30
7.2.1 Implementace testů 31
7.2.2 Pokrytí kódu 31
8 Modelový příklad 33
8.1 Úspora 34
9 Závěr 36
Literatura 37
A Seznam zkratek 39
B Obsah CD 40

Tabulky / Obrázky

6.1. Vlastnosti zvažovaných jazyků 19
6.2. Přehled modulů programu 21
7.1. Pokrytí kódu testy 31

2.1. Ukázka průřezového problému ... 3
3.1. Fáze komponent ve frameworku AspectFaces 8
4.1. Části frameworku 10
5.1. Fáze aspect weaveru 15
5.2. Aplikace advice 16
6.1. Činnost weaveru 24
8.1. Graf úspory pro n 35

Kapitola 1
Úvod

Při vývoji aplikací a v programování obecně je doporučováno držet se několika principů. Patří mezi ně i princip oddělení zodpovědnosti nebo princip DRY [1], tedy rozdělovat kód do soudržných modulů, resp. minimalizovat výskyty opakovaného kódu, neboť duplicity mají řadu nežádoucích důsledků. Jako nejzávažnější důsledky lze považovat například obtížné udržování kódu či složité testování. Každé programovací paradigma zpravidla disponuje možnostmi, jak výskyt duplicit minimalizovat. Přes všechny snahy ale existují problémy, které nelze efektivně separovat, a tak je vznik duplicitního či nečitelného kódu nevyhnutelný. Jako typické příklady takových problémů se uvádí logování, práce s transakcemi nebo také trasování (anglicky *tracing*) či ladění (anglicky *debugging*). Přestože by se mohlo zdát, že se téma duplicit týká pouze imperativního programování, není tomu tak. Problematika duplicit se vyskytuje i v programování deklarativním, kdy nevytváříme algoritmus jako takový, ale pouze popisujeme cíl. Řešením těchto problémů se zabývá aspektově orientované programování.

Bakalářská práce se zabývá přidáním podpory pro aspektově orientované přístupu v algebraických specifikacích. Jedním ze zástupců jazyků pro popis algebraických specifikací je jazyk Maude, který bude zároveň použit pro demonstraci aspektově orientovaného přístupu s cílem minimalizovat duplicity v definici algebraických struktur.

Nejprve se seznámíme s jazykem Maude a základní syntaxí, která je v rámci bakalářské práce využívána. Také si podrobněji představíme aspektově orientované padarigma a termíny, jenž se často vyskytují v souvislosti s AOP.

Dále si ukážeme některé projekty, které tématicky souvisí s touto prací.

V následující části definujeme formálněji požadavky na výsledný model, popíšeme si průběh řešení a zdůvodníme, proč bylo dané řešení zvoleno. Poté si předvedeme modelové příklady použití.

Na závěr zhodnotíme výsledky práce.

1.1 Cíle práce

Cílem této bakalářské práce je prostudovat možnosti implementace a následně vytvořit model, který přidá podporu aspektově orientovaného programování do Maude. V rámci modelu bude definována syntaxe aspektů, která bude v podobném stylu jako je syntaxe Maude. Dalším cílem je implementace a dostatečné otestování aspect weaveru.

Vytvořený framework bude následně použit pro demonstraci užití aspektově orientovaného programování v Maude v modelovém příkladu.

Součástí práce je také shrnutí výhod a definice omezení frameworku.

[1] Don't Repeat Yourself — neopakuj se

Kapitola 2
Rešerše

2.1 Aspektově orientované programování

K vysvětlení AOP je potřeba znát několik základních pojmů, které jsou klíčové a v souvislosti s tímto paradigmatem se často užívají. Proto se několik následujících podsekcích bude nejprve věnovat těmto pojmům.

2.1.1 Oddělení zodpovědností

Oddělení zodpovědností (anglicky *separation of concerns*) je jedním ze základních principů softwarového inženýrství pro rozdělení kódu programu do modulů tak, aby různé zodpovědnosti byly rozděleny do různých modulů [1].

Zodpovědností se rozumí funkcionalita daného programu. Tyto funkcionality můžou být různé úrovně od vysokoúrovňových (např. autorizace) až po nízkoúrovňové (např. zaznamenávání stavu aplikace) [2]. Implementace modulů se liší v závislosti na programovacím paradigmatu, respektive na výběru jazyka.

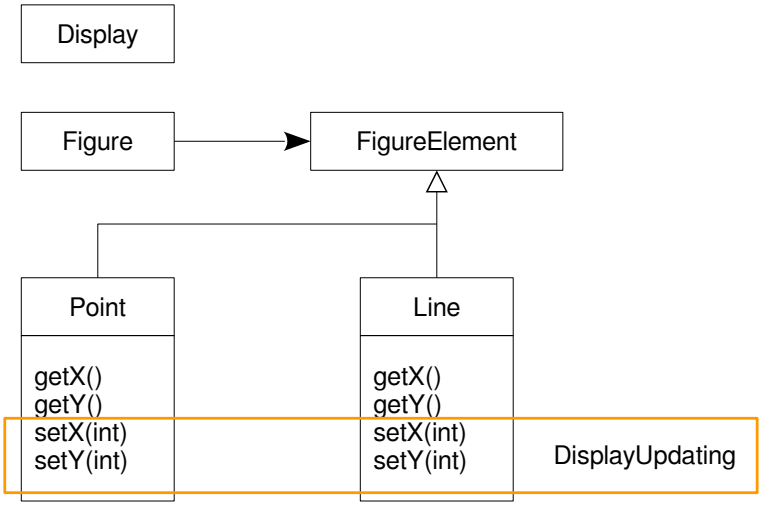
Oddělením zodpovědností se zvyšuje udržovatelnost programu, protože správné oddělení zodpovědností umožňuje znovupoužitelnost jednotlivých modulů. Znovupoužitelnost modulů snižuje výskyt duplicit.

2.1.2 Průřezové problémy

Jak již bylo zmíněno v úvodu, existují problémy (resp. zodpovědnosti), které nelze efektivně oddělit. Takové problémy nazýváme průřezové (anglicky *cross-cutting concerns*) a vyskytují se například v těchto oblastech [2]:

- zabezpečení (autorizace),
- logování,
- synchronizace,
- řízení transakcí a dalších.

Průřezové problémy způsobují rozložení implementace jedné funkcionality do několika modulů, což vede k rozptýlenému kódu (anglicky *code scattering*). Dále zapříčiňují jev zvaný *code tangling* – ten můžeme znát pod českým pojmem *špagetový kód*, který se vyskytuje, když má modul více zodpovědností [2].



Obrázek 2.1. Ukázka průřezového problému (převzato z [2])

Obrázek 2.1 demonstruje pomocí UML diagramu jednoduchý příklad průřezového problému. V ukázce lze vidět hierarchii tříd jednoduchého editoru obrazců. Při pohybu objektů typu `FigureElement` je potřeba notifikovat screen manager o změně pozice daného objektu. `FigureElement` je abstraktní třída, která ale má dvě konkrétní třídy `Point` a `Line`. Pokud chceme notifikovat screen manager o pohybu instancí těchto tříd, je nutné, aby každá metoda, která hýbe danou instancí, tyto notifikace vytvářela. Rámeček `DisplayUpdating` vyznačuje metody, jenž ovlivňují pozici objektů, a tudíž musejí vytvářet notifikace. Protože `DisplayUpdating` nepatří do žádné z uvedených tříd, ani je neobaluje, ale pouze jimi prořezává, lze tento problém označit jako průřezový [2].

2.1.3 Paradigma

Se znalostí pojmů zavedených v předchozích odstavcích lze přejít k vysvětlení aspektově orientovaného paradigmatu jako takového. V následujících dvou odstavcích jsou použity informace dostupné z [2].

Protože problematika oddělení zodpovědností není nová, existují možnosti, jak průřezové problémy řešit. Tyto možnosti ale většinou vyžadují podřízení jádra programu řešení průřezových problémů. AOP naopak přináší nástroj, kterým lze průřezové problémy zachytit a vyčlenit je do samostatných modulů bez podřízení jádra.

Moduly v AOP se nazývají *aspekty*. Aspekty jsou pak podle definice moduly, které prořezávají struktury ostatních modulů. To znamená, že aspekty obsahují chování, které ovlivňuje ostatní moduly (v OOP například třídy).

2.1.4 Join point model

Join point model popisuje jazyk pro popis aspektů, díky tomu je join point model jeden z nejdůležitějších prvků aspektově orientovaného jazyka [3]. Odlišné join point modely poskytují různé přístupy a možnosti při implementaci průřezových problémů [4].

JPM definuje kdy, kam a jak se mají aspekty aplikovat [4]. V souvislosti s tímto existují tři termíny (join point, pointcut a advice), které se běžně užívají a každý aspektově orientovaný jazyk je využívá. Tyto pojmy přehledně vysvětlil Tomáš Turek ve své diplomové práci [5]:

- *„Join point je místo v programu, kam je možné aplikovat průřezovou zodpovědnost pomocí AOP. Může sem patřit volání metod, inicializace tříd, struktura tříd (jméno třídy, názvy a datové typy atributů a jejich anotace) apod.“*

- *„Advice je rozšiřující kód, který chceme použít k rozšíření stávajícího modelu.“* Advice lze aplikovat v join pointech.
- Pointcut je množina join pointů, pro které je aplikována advice. *„Jedná se o místo v aplikace, kde je potřeba aplikovat průřezovou zodpovědnost.“*

Skutečnost, že různé jazyky mají různé možnosti (což je ovlivněno například i programovacím paradigmatem), způsobuje rozdíly v JPM. Proto se implementace AOP například v Prologu [6] (nejen) syntakticky liší od implementace například v Javě [7]. Podoba JPM tedy závisí na aspektově orientovaném jazyku, ke kterému náleží.

2.1.5 Aspect Weaving

Aspect Weaving *„je proces integrace aspektů do specifických míst aplikace“* [5]. Nástroj realizující integraci se pak nazývá *Aspect Weaver*. Existují dva způsoby, jakými lze weaving provádět: statický a dynamický.

Statický weaving upravuje zdrojový kód přidáním advice do vybraných join pointů a probíhá v době kompilace. Jeho nevýhodou je obtížné rozpoznání přidaného chování (advice) ve výsledném kódu [8] [9].

Dynamický weaving probíhá při zavádění aplikace či přímo při běhu. Tento přístup umožňuje změnu programu za běhu [9]. V tomto případě není integrace aspektů realizována modifikací kódu, ale často se používá například návrhový vzor proxy [10] nebo dědičnost [8].

2.2 Maude

Maude je programovací jazyk a systém podporující rovnicovou (*equational*) a přepisovací (*rewriting*) logiku vyvinutý v SRI International [1]. Je to nástroj vhodný pro dokazování různých teorémů nebo pro simulaci výpočetních modelů a jiné. Proto má Maude využití převážně v akademickém prostředí [11].

Hlavní důraz je v návrhu Maude kladen na tyto tři aspekty:

- *Jednoduchost:* programy by měly být jednoduché a měly by mít jasný význam.
- *Vyjadřovací schopnost:* jazyk by měl být použitelný pro širokou škálu aplikací (od jednoduchých až po složité abstraktní systémy).
- *Výkon:* implementace v Maude by měly být srovnatelně výkonné jako implementace v jiných efektivních programovacích jazycích [12].

2.2.1 Syntaxe

Tato podsekce vysvětluje pouze základní konstrukty dostupné v Maude, které jsou pro potřeby této bakalářské práce většinou dostačující. Pokud bude v dalších kapitolách použita vlastnost Maude, jenž není na následujících řádkách zmíněna, bude vysvětlena v místě použití [2].

Kód je v Maude členěný do modulů. Modul se skládá ze dvou částí. První část je deklarační (jinak se nazývá *signatura*) a určuje množiny, operátory a podobně, které daný modul používá. Druhá část obsahuje věty, které definují vlastnosti nebo chování operátorů definovaných v signatuře.

[1] Americká výzkumná instituce, sídlící v Kalifornii
[2] Pro kompletní seznam vlastností doporučuji [12] nebo [11]

V Maude rozlišujeme dva druhy modulů: *funkcionální* a *systémové*. Jejich signatury se neliší, ale liší se ve větách, které mají k dispozici. Funkcionální moduly podporují rovnice. Systémové moduly podporují ještě navíc přepisovací pravidla [12]. V následujícím výpisu je uveden příklad funkcionálního modulu, který byl převzat z [12]:

```
fmod SIMPLE-NAT is

  sort Nat .
  op zero : -> Nat .
  op s _ : Nat -> Nat .
  op _ + _ : Nat Nat -> Nat .

  vars N M : Nat .
  eq zero + N = N .
  eq s N + M = s (N + M) .

endfm
```

Výpis 2.1. Ukázka funkcionálního modulu v Maude

Modul představený v ukázce implementuje operace (konkrétně pouze sčítání) nad přirozenými čísly za použití Peano-notace. První řádek definuje nový modul s názvem `SIMPLE-NAT`, následuje řádek deklarující, že modul používá množinu `Nat`. Dále je deklarována konstanta `zero` a dvě operace `s_` a `_+_`. Podtržítka určují, kde se v dané operaci budou vyskytovat operandy; v ukázce tak můžeme pozorovat, že Maude dovoluje použít libovolnou notaci operátorů (prefixovou, infixovou či postfixovou). Řádek s `vars` deklaruje používané proměnné a jejich typy. Následující řádky definují chování výše deklarovaných operací.

2.3 Haskell

Haskell se od většiny populárních programovacích jazyků liší neobvyklou syntaxí a úzkým vztahem k matematice. Jeho vlastnosti jsou přehledně popsány a vysvětleny ve [13], odkud jsou také čerpány informace použité v této kapitole.

Haskell je čistě funkcionální programovací jazyk. To znamená, že v Haskellu nemají funkce žádné vedlejší efekty a zároveň je zaručeno, že funkce, podobně jako v matematice, vrací pro stejné parametry stejné výsledky. Takové funkce f se v Haskellu označují jako čisté (anglicky *pure*) a pro vstupní hodnoty X_1 a X_2 platí

$$X_1 = X_2 \Longrightarrow f(X_1) = f(X_2).$$

2.3.1 Čisté funkce a funkce s vedlejším efektem

Přestože je Haskell čistě funkcionální, existuje způsob, jakým lze interagovat s uživatelem či zapisovat nebo číst ze souboru. Taková interakce je realizována funkcemi označovanými jako nečisté (anglicky *impure*), protože mají vedlejší efekt (např. ve formě změny stavu souboru). Nečisté funkce jsou dobře navrženým systémem odděleny od čistě funkcionální zbytku programu, a tudíž neovlivní pozitivní vlastnosti čistých funkcí.

2.3.2 Líné vyhodnocování

Složené výrazy jsou v Haskellu vyhodnocované *líně*. Při líném vyhodnocování nejsou výrazy (funkce) vyhodnoceny, dokud není jejich výsledek potřeba. To má několik důsledků. Jedním z nich je například možnost použití nekonečných datových struktur

(např. nekonečný seznam čísel). Díky línému vyhodnocování a velké abstrakci, se kterou Haskell pracuje, lze o programech uvažovat jako o sérii transformací dat.

2.3.3 Statické typování

Další z vlastností Haskellu je statické typování. Protože v případě statického typování je nutné, aby v době kompilace byly jasně určeny datové typy všech entit programu, může kompilátor odhalit velkou část potencionálních chyb.

2.3.4 Příklad kódu

```
module Main where

import System.Environment

main = do
  putStrLn "Enter an integer: "
  n <- getLine
  putStrLn (show (pow2 n))

pow2 :: String -> Int
pow2 = (pow 2) . read

pow :: Int -> Int -> Int
pow e base = base^e
```

Výpis 2.2. Ukázka kódu v Haskellu

Výše uvedený jednoduchý příklad načte z konzole číslo a vypíše jeho druhou mocninu. V příkladu lze pozorovat několik dalších vlastností Haskellu, na které stručně upozorním:

- kód se dělí do modulů, které lze importovat,
- vstupním bodem programu je funkce `main` v modulu `Main`,
- funkce s vedlejším efektem mohou spolupracovat s čistými funkcemi bez vedlejšího efektu (např. `putStrLn` a `pow2`),
- jednotlivé funkce lze skládat pomocí operátoru `(.)`,
- využívá se curryfikace a částečně aplikované funkce (tj. využití funkce `pow` ve funkci `pow2`).

Kapitola 3
Související práce

Aspektově orientované rozšíření jazyka Maude ještě vytvořeno nebylo. Obvyklé jsou implementace v imperativních jazycích jako je Java nebo C#. Implementace ale například existuje i v jazyku Prolog, což je jazyk logický, deklarativní.

3.1 AspectJ

AspectJ je aspektově orientované rozšíření jazyka Java a jde o jednu z nejpopulárnějších (ne-li nejpopulárnější) implementací AOP vůbec. Autoři při vytváření dbali, aby výsledné rozšíření bylo kompatibilní s Javou. Kompatibilitu vnímali ve čtyřech rovinách:

- validní programy v Javě musí být i validními AspectJ programy,
- AspectJ programy musí fungovat na standardní JVM,
- AspectJ musí být kompatibilní se současnými nástroji (IDE, apod.),
- pro programátora se musí AspectJ jevit jako přirozené rozšíření Javy [14].

3.1.1 JPM

JPM obsahuje join points pro:

- volání a provádění metod,
- vracení a nastavování hodnot vlastností objektů nebo
- zachytávání výjimek [14].

3.1.2 Využití

AspectJ je integrovaný například do frameworku Spring [15], což je široce rozšířený enterprise framework v Javě. Využití pro AspectJ nacházíme zejména pro použití ve vrstvách backendu.

```
pointcut moves():
  receptions(void Line.setP1(Point)) ||
  receptions(void Line.setP2(Point));

static after(): moves() {
  flag = true;
}
```

Výpis 3.1. Ukázka kódu v AspectJ

Výše uvedený příklad byl převzat z [14] a následně upraven. Ukazuje způsob, jakým lze v AspectJ definovat pointcut a advice. První blok definuje, které join points má obsáhnout pointcut `moves`. Druhý blok říká, že advice se má přidat za místo, které definoval pointcut `moves`. Tento kus kódu tedy zajistí, že po zavolání setteru proměnné `P1` nebo `P2` bude proměnná `flag` nastavena na `true`.

Příklad také ilustruje, že autorům AspectJ se podařilo navrhnout jazyk, který svou syntaxí skutečně připomíná čistou Javu, což byl jeden z původních cílů projektu AspectJ.

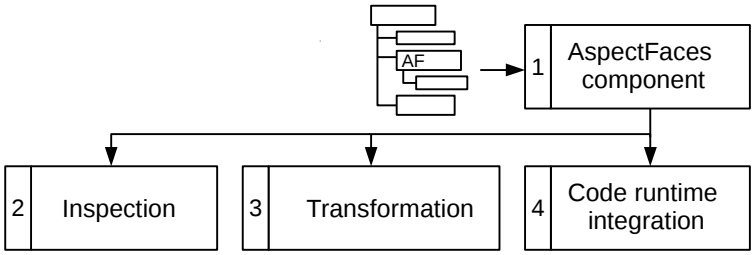
3.2 AspectFaces

AspectFaces je framework, který na základě vytvořeného meta-modelu integruje několik aspektů do jednoho výstupu. Cílem AspectFaces je snížení počtu řádků kódu, které musí vývojář při tvorbě frontendu napsat [16]. Framework umožňuje efektivně oddělit například tyto zodpovědnosti:

- uživatelská autorizace,
- omezení vstupů (délka řetězce apod.),
- validace vstupů nebo
- rozvržení komponent na stránce.

3.2.1 Koncept

Obrázek 3.1 ilustruje, že AF komponenty mají tyto tři fáze: *Inspection*, *Transformation* a *Runtime code integration*.



Obrázek 3.1. Fáze AF komponent (převzato z [16])

Ve fázi *Inspection* dochází k načtení informací ze zdrojového kódu a následně k vytvoření meta-modelu reprezentujícího data.

Další fází je *Transformation*, kde dochází k aplikaci zobrazovacích pravidel. Zobrazovací pravidlo určuje, jak se mají mapovat datové pole na uživatelská rozhraní. Poté dojde k vybrání správné šablony a rozvržení použitých komponent.

Runtime code integration je poslední fáze, ve které se z výsledku předchozích fází sestaví strom komponent.

3.2.2 Využití

AspectFaces má ze své podstaty využití pouze při tvorbě uživatelského rozhraní. Následující příklad AF komponenty (převzato z [16]) ukazuje, že AspectFaces je také, podobně jako AspectJ, syntakticky kompatibilní s původním tradičním jazykem, kterým je zde JavaServer Faces.

```
<af:ui instance="#{controller.personInstance}"/>
```

Výpis 3.2. Komponenta v AspectFaces

Zde je pro porovnání příklad kódu napsaného v JSF:

```
<h:inputText id="username"
             required="true"
             value="#{bean.username}"
             rendered="empty #{bean.username}"/>
```

Výpis 3.3. Komponenta v JSF

3.3 AOP v Prologu

Tato implementace AOP je ze všech uvedených nejblíže implementaci v Maude, které se snaží tato práce docílit. Podobnost lze pozorovat zejména ve způsobu, jakým se v Prologu nebo v Maude programuje. Ani v jednom z uvedených programů nepopisujeme algoritmus, ale popisujeme fakta a pravidla, na základě kterých lze posléze vyhodnotit či dokázat nějaký výraz.

3.3.1 Koncept

Systém poskytuje protokol pro definování aspektů a kompilátor, který funguje jako aspect weaver.

Zmíněný protokol definuje dva konstrukty `aspect_declaration` a `aspect`, jejichž syntaxe a způsob použití v kódu připomíná prologovské predikáty:

```
:- use_module(library(aspects), [aspect/3, aspect_declaration/1]).

:- use_module(library(ring_utils)).
:- aspect_declaration(use_module(library(ring_utils))).

:- aspect(around,
          get_active_ring(A, Rings, Ring),
          get_active_ring_proxy(A, Rings, Ring)).
```

Výpis 3.4. Ukázka použití AOP v prologu (převzato z [6])

Predikát `aspect_declaration` má jeden parametr a označuje deklaraci, kterou je potřeba vložit do souboru s aspekty, aby bylo možné daný soubor zkompilovat.

Druhý predikát, `aspect`, má tři parametry: typ, cíl a přidané chování. Parametr typ nabývá hodnoty z množiny {`around`, `before`, `after`} a určuje typ aspektu (resp. moment, kdy se má přidané chování volat). Druhý parametr specifikuje cíl, ke kterému se přidává chování, které se definuje v posledním, třetím parametru.

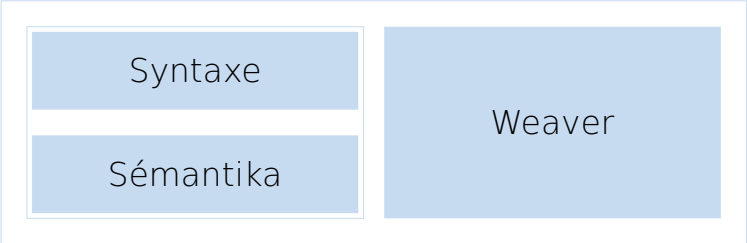
Tento systém nepodporuje žádný jazyk pro specifikaci pointcuts, a tak není možné kvantifikovat cíle jednotlivých aspektů.

3.3.2 Využití

Díky tomu, že je framework – stejně jako Prolog – všestranně použitelný, nabízí se široké možnosti využití. Jako modelový příklad užití je v [6] uvedeno trasování toku programu pomocí aspektů.

Kapitola 4
Analýza

Práce má za cíl vytvořit framework pro přidání podpory AOP do Maude. Protože součástí frameworku je i aspect weaver, lze vnímat práci ve dvou částech. První částí je navržení jazyka a celého systému, druhou pak je implementace aspect weaveru.



Obrázek 4.1. Schéma celků obsažených ve vytvářeném frameworku

4.1 Požadavky

Požadavky na framework vychází částečně z konceptů souvisejích frameworků zmíněných dříve v kapitole 3 a částečně ze zadání bakalářské práce.

Požadavky vycházející ze zadání bakalářské práce nazvěme povinnými; zbylé požadavky může označit například jako přirozené, neboť jsou výsledkem prostého úsudku či vyplývají přirozeně z povahy výchozího jazyka – Maude.

4.1.1 Povinné požadavky

Povinné požadavky jsou poměrně obecné a byly zmíněné již v úvodu práce v kapitole 1.1, pro ucelenost je ale uvedu i zde:

- **P1:** *Framework musí být rozšiřitelný* — tento požadavek platí pro obě zmiňované části frameworku. V případě systému znamená tento požadavek hlavně rozšiřitelnost jazyka pro specifikaci pointcuts. To znamená, že i weaver musí být rozšiřitelný, aby byl schopný reagovat na přidané prvky.
- **P2:** *Framework musí být otestovaný* — testováním bude zajištěna korektnost implementace, což se týká převážně weaveru.

4.1.2 Přirozené požadavky

Přirozené požadavky jsem identifikoval následující:

- **P3:** *AOP jazyk by měl mít dostatečnou vyjadřovací schopnost* — jazyk musí mít dostatečně velkou přidanou hodnotu v podobě vyjadřovací schopnosti. Aspekty musí programátorovi přinést možnost ovlivnit libovolný operand v operaci.
- **P4:** *Syntaxe AOP jazyka musí působit v kombinaci s Maude přirozeně* — taková vlastnost je velice důležitá zejména pro programátora užívajícího daný framework. O důležitosti tohoto požadavku vypovídá i skutečnost, že kompatibility s původním jazykem se snažily docílit všechny frameworky zmíněné v kapitole 3.

4.2 Syntaxe Maude

Pro splnění požadavku na vyjadřovací schopnost (P3), požadavku na syntaktickou kompatibilitu (P4) a částečně i požadavku na rozšiřitelnost (P1) je potřeba analyzovat syntaxi jazyka Maude. V rámci analýzy budou použity termíny a fakta popsané v kapitole 2.2.1.

První prostý poznatek je, že většina vět v Maude (viz např. ukázka kódu ve zmíněné kapitole) je zakončena tečkou. Každá věta také začíná klíčovým slovem, které určuje, co daná věta znamená, např. klíčové slovo `op` znamená deklaraci operace, `eq` znamená definici operace pomocí rovnice apod. Klíčové slovo navíc bývá krátké.

Jak bylo zmíněno v rešerši, moduly v Maude mají deklarační a definiční část. Deklarace operací v sobě typicky obsahují podtržítka, což značí místo pro vstup. Pokud operace nemá žádné vstupy, pak představuje konstantu. Každá deklarace operace obsahuje dvojtečku, která odděluje část s názvem od části představující množiny, se kterými operace pracuje. Dvojtečka se objevuje i v definiční části, kde určuje množiny, do kterých patří jednotlivé proměnné.

Operátory mohou mít také tzv. atributy, které se uvádí do hranatých závorek a které poskytují dodatečné informace o dané operaci [12]:

```
op zero : -> Zero [ctor] .
op _ + _ : Nat Nat -> Nat [comm] .
```

Výpis 4.1. Použití atributů operátorů v Maude

Pro účely této práce není podstatné vědět, co tyto atributy znamenají, pro představu ale uvedu, že například `comm` označuje komutativní operace.

V definicích rovnice se používá symbol `=` pro oddělení hlavičky a těla rovnice.

4.2.1 Závěr

Z výše uvedeného rozboru vyplývá, že pro Maude je přirozené uvádět klíčové slovo, které zpravidla bývá dlouhé dva znaky, určující typ věty jako první slovo ve větě. Na konci vět se pak běžně vyskytuje tečka, která větu ukončuje, a mezi často vyskytující se symboly patří `=`, `:` nebo `_`.

Kapitola 5
Design

S ohledem na definované požadavky a výstupy analýzy (viz kapitola 4) bude v této kapitole navržený join point model a jazyk vytvářeného frameworku. Dále zde bude popsán algoritmus, který bude použit pro aspect weaver, a budou zde uvedena také jeho omezení.

5.1 JPM

Návrh JPM předchází návrhu syntaxe, protože podoba JPM přímo ovlivňuje syntaxi jazyka. Pro připomenutí uveďme, že join point model určuje:

- kam lze přidat chování (tj. join point),
- jak jednoznačně definovat místo, kam se chování skutečně přidá – tedy kvantifikace join pointů (tj. pointcut),
- jak definovat přidané chování (tj. advice).

5.1.1 Join points

Velké možnosti, co se do počtu uznávaných join points týče, poskytuje framework AspectJ [14], kde existuje join point skutečně pro každé z několika míst, jenž Java nabízí a join point pro ně dává smysl. Mezi taková místa patří volání či provádění metod, nastavování hodnot třídním proměnným apod. V tomto ohledu je Maude poněkud chudší jazyk.

Uvážíme-li způsob, jakým jsou v Maude vytvářeny programy, můžeme vyvodit závěr, že jediná místa, kam má smysl přidávat chování, jsou operace (a jejich definice), a to z toho důvodu, že operace jsou jediný prvek Maude, který představuje chování. Systém tedy bude mít pouze jeden join point – *provádění operace*.

5.1.2 Pointcut

Protože byl identifikován pouze jeden potencionální join point, budou nároky na specifikaci pointcuts menší než ve složitějších systémech (např. zmíněný AspectJ). V rámci našeho AOP systému tedy pouze požadujeme, abychom pomocí pointcut mohli vybrat určitou množinu operací.

Operace se mezi sebou liší v několika aspektech, na základě kterých je můžeme vybírat. Mezi takové aspekty patří:

- název operace,
- počet vstupů,
- uspořádání proměnných v názvu operace.

Poslední bod seznamu zachycuje situace, kdy dvě operace mají stejný název i počet vstupů, ale přesto jsou různé. Například obě tyto operace `_ add _` nebo `add _ _` jsou validní, jmenují se stejně, mají stejný počet vstupů, ale i přesto jsou (z důvodu syntaktického uspořádání) různé.

Pointcut proto umožní adresovat operace na základě třech výše uvedených aspektů (název, počet vstupů, uspořádání), což nám dovolí vybrat libovolnou operaci. Zároveň bude možná kvantifikace ve smyslu výběru operací s podobným názvem, a tak bude možné vybírat více operací splňujících specifikovaná kritéria.

5.1.3 Advice

Chování bude možné přidat pomocí změny obsahu vstupních proměnných operace. Systém umožní přepsat hodnotu konkrétní proměnné na libovolný výraz, ve kterém lze použít kteroukoli proměnnou či operaci dostupnou z vybrané operace, na níž se aspekt aplikuje.

5.2 Jazyk

Jazyk vytvářeného systému bude rozšiřovat původní jazyk Maude, tzn. validní kód v Maude bude validním kódem v AOP systému. Pro definování aspektů je potřeba vytvořit syntaktické konstrukty tak, aby respektovaly kladené požadavky a definovaný JPM, proto jsou navrženy dva druhy vět – pro deklaraci nových aspektů, resp. pro definování advice:

⟨*AspectDeclaration*⟩ `::= as` ⟨*Label*⟩ ⟨*Variable*⟩`*` `:` ⟨*Pointcut*⟩

⟨*AdviceDefinition*⟩ `::= ad` ⟨*Label*⟩ ⟨*VariableId*⟩`*` `:` ⟨*RewriteExpr*⟩

⟨*RewriteExpr*⟩ `::=` ⟨*VariableId*⟩ `=` ⟨*Expression*⟩

⟨*Pointcut*⟩ `::= (`⟨*Variable*⟩ `|` ⟨*Label*⟩ `|` ⟨*SpecialChar*⟩`)*`

⟨*Label*⟩ `%%% identifikátor`

⟨*Variable*⟩ `%%% podtržítko`

⟨*VariableId*⟩ `%%% identifikátor`

⟨*Expression*⟩ `%%% výraz`

⟨*SpecialChar*⟩ `%%% výraz mající speciální význam`

Výpis 5.1. Pravidla pro syntaxi navrženého jazyka

Výše uvedená pravidla jsou zapsána za použití následující rozvité BNF notace:

- symboly `(` a `)` vyznačují celek (na který lze například aplikovat kvantifikátor),
- symbol `|` od sebe odděluje alternativy – lze číst jako „nebo“,
- symbol `*` znamená opakování předcházející jednotky (minimálně 0krát, maximálně neomezeno),
- sekvence `%%%` představuje začátek řádkového komentáře.

5.2.1 Jazyk pro definici pointcut

Pomocí jazyka pro definování pointcuts musí být možné specifikovat tato kritéria:

- název operace,

- počet vstupů operace,
- uspořádání proměnných v názvu operace.

Tato kritéria byla diskutována a zdůvodněna v kapitole 5.1.2, kde byla také zmíněna potřeba kvantifikace.

Navržený jazyk pro definici pointcuts je velice podobný standardní definici operací až na dva rozdíly. Prvním rozdílem je absence klíčového slova, které by bylo zcela zbytečné, neboť pointcut vybírá pouze operace (viz kapitola 5.1.2). A druhý rozdíl je možnost nahradit část názvu symbolem `*`, což představuje libovolnou sekvenci znaků (až do konce slova), a tudíž je tak realizovaná kvantifikace.

Takový pointcut může vypadat například takto `foo* _`. Mezi operace vybrané tímto pointcutem by pak patřily například `foo _` nebo `foobar _`. Naopak operace jako `_ foo`, `barfoo _` nebo `foo _ bar` nebudou pointcutem vybrány – první z důvodu uspořádání vstupů, druhá z důvodu jiného názvu a třetí také z důvodu jiného názvu (`bar` je navíc oproti pointcut).

Tímto jazykem jsme schopni vyjádřit všechna specifikovaná kritéria. Název navíc můžeme zobecnit pomocí symbolu `*`, čímž dojde ke kvantifikaci.

5.2.2 Ukázka jazyka

Pro lepší ilustraci představené syntaxe je v následující ukázce uveden jednoduchý aspekt, na kterém bude popsána jeho sémantika.

```
1  fmod TEST is
2
3    ...
4
5    op _ plus _ : Nat Nat -> Nat .
6    eq a plus b = a + b .
7
8    as aspectPlus _ _ : _ pl* _ .
9
10   ad aspectPlus x y : y = 0 .
11   ad aspectPlus x y : x = x + y .
12
13 endfm
```

Výpis 5.2. Použití vytvářeného jazyka v standardním modulu

V ukázce je představený jednoduchý funkcionální modul `TEST`. Kromě jedné obyčejné operace `plus`, která má dva vstupy a je naimplementována pomocí rovnice, si ale můžeme povšimnout ještě dalších vět na řádcích 8, 10 a 11.

Klíčové slovo `as` naznačuje, že věta na osmém řádku deklaruje nový aspekt. Každý aspekt má nadefinované své jméno, v této ukázce se bavíme o aspektu s názvem `aspectPlus`. Za názvem následuje deklarace vstupů pomocí podtržítek, kdy počet podtržítek je roven počtu vstupů. Počet vstupů aspektu se musí rovnat počtu vstupů použitých v pointcut (důvod viz v kapitole 5.3.3). Dvojtečkou je od deklarace oddělena specifikace pointcut – specifikaci pointcuts se bude věnovat následující podkapitola. Na osmém řádku tedy byl pojmenovaný nový aspekt, bylo naznačeno, kolik bude mít vstupů a díky pointcut byly vybrány operace, které daný aspekt ovlivní.

Na desátém a jedenáctém řádku jsou věty začínající klíčovým slovem `ad`, což značí, že jsou to definice advice. Podle druhého slova ve větě lze poznat, ke kterému aspektu daná advice patří. To je důležité znát zejména proto, aby došlo k weavingu dané advice do správných míst, která určuje pointcut. Za názvem aspektu následuje výčet vstupních

proměnných, jejichž počet se musí shodovat s počtem podtržítek v deklaraci aspektu. Za dvojtečkou je uvedený výraz, který může využívat všechny proměnné deklarované v první části věty.

5.3 Aspect weaver

Protože veškeré přidávané chování je známé při kompilaci a za běhu se měnit nebude, bude implementovaný statický weaving.

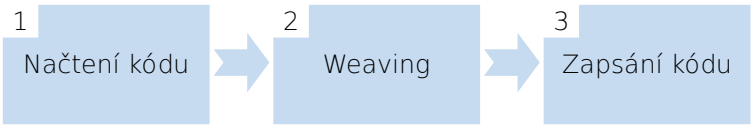
Činnost statického weaveru lze formalizovat. Nechť množina $\mathbb{M}$ obsahuje všechny věty, které lze v Maude zapsat, a množina $\mathbb{A}$ obsahuje všechny věty, které lze zapsat ve vytvářeném AOP systému, přičemž platí $\mathbb{M} \subseteq \mathbb{A}$. Pak můžeme definovat dva programy P a P' jako uspořádané množiny $P = \{v_{\mathbb{A}} \mid v_{\mathbb{A}} \in \mathbb{A}\}$ a $P' = \{v_{\mathbb{M}} \mid v_{\mathbb{M}} \in \mathbb{M}\}$, což jsou sekvence vět dostupných v příslušných množinách. Na základě toho lze weaving definovat jako zobrazení w z prostoru A všech programů P do prostoru M všech programů P'

$$w : A \to M.$$

5.3.1 Obecný návrh programu

V návrhu programu můžeme využít výše uvedeného formálního popisu činnosti, na základě kterého si lze weaver představit jako transformaci vět z jednoho prostoru do jiného. V nejvyšší úrovni můžeme činnost programu rozdělit do těchto třech fází:

1. Načtení souboru s kódem programu.
2. Weaving.
3. Zapsání výsledku do souboru.



Obrázek 5.1. Schéma fází aspect weaveru

Načtení kódu a pozdější zápis v první resp. třetí fázi jsou triviální. Druhá fáze obsahuje celou logiku weaveru a realizuje zmíněnou transformaci, při které dochází k weavingu aspektů.

5.3.2 Načtení kódu

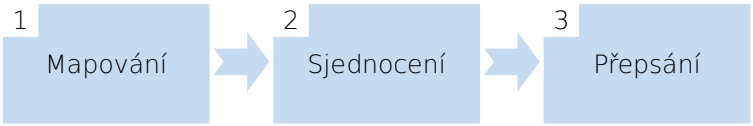
Program bude očekávat jako parametr cestu k souboru se zdrojovým kódem (viz kapitola 5.3.6) a tento soubor následně načte. Po načtení souboru dojde i k předzpracování kódu, kdy se odstraní prázdné řádky a smaže se odsazení jednotlivých řádků.

5.3.3 Weaving

Do fáze weavingu vstupuje Maude kód s aspekty, které se v průběhu fáze integrují do kódu, a výsledkem fáze pak je validní Maude kód obsahující funkcionalitu rozšířenou o (v aspektech definované) advices.

Způsob, jakým bude weaving proveden, závisí na jazyce definovaném výše. Pro náš jazyk bude dostačující lineární průchod skrz definované advices a jejich postupné aplikování v pointcutech. Algoritmus bude procházet vstupní zdrojový kód řádek po řádku

a na každou větu se pokusí aplikovat advices každého aspektu[1] v pořadí, v jakém byly advices zapsány. Postup pro aplikaci advice ilustruje následující schéma.



Obrázek 5.2. Schéma postupu pro aplikování advice

Jednotlivé bloky schématu lze popsat slovy takto:

1. Namapuj vstupní proměnné advice na vstupní proměnné rovnice.
2. V těle advice přepiš vstupní proměnné na jejich namapovanou proměnnou.
3. V těle rovnice přepiš proměnnou, kterou daná rada upravuje, na upravenou hodnotu.

Přestože k aplikaci advice jako takové dochází až ve třetím kroku, je mapování a sjednocení nedílnou součástí procesu, a to z toho důvodu, že systém umožňuje programátorovi pojmenovat proměnné v radě a v původní rovnici různými názvy. Proto musí nejprve dojít k synchronizaci názvů.

Aby došlo ke korektnímu namapování proměnných aspektu a dané konkrétní rovnice, je nutné, aby byl počet vstupů aspektu (resp. všech jeho rad) roven počtu vstupů rovnice. Z toho důvodu, existuje požadavek, aby v deklaraci aspektu byl počet vstupů roven počtu vstupů v pointcut. To přirozeně zajistí, že rovnice vybraná takovým pointcutem bude mít shodný počet vstupů jako daný aspekt.

5.3.4 Ukázka weavingu

Mějme jednu operaci vyjádřenou rovnicí a aspekt se dvěma radami:

```
1  op f _ : ... .
2  eq f x = k x .
3
4  as asp1 _ : f _ .
5
6  ad asp1 t : t = i t .
7  ad asp1 t : t = j t .
```

Výpis 5.3. Kód pro ilustraci weavingu

Díky pointcut `f _` se bude aspekt `asp1` aplikovat na operaci `f _`, která je realizovaná rovnicí. Pro tento aspekt byly nadefinovany dvě jednoduché advices. První přepisuje proměnnou tak, že před ni přidá symbol `i`. Druhá funguje velmi podobně, ale před proměnnou přidává symbol `j`.

Weaver bude aplikovat advices v pořadí, v jakém jsou zapsány, proto se nejprve aplikuje řádek č. 6, poté řádek č. 7 a následně, protože již nebyly nadefinovány další advices, weaving skončí. Po aplikaci řádku č. 6 bude mít rovnice takovou podobu

```
eq f x = k i x .
```

Následná aplikace advice ze sedmého řádku upraví rovnici do finální podoby

```
eq f x = k i j x .
```

Pokud by weaver advices aplikoval v opačném pořadí (tzn. nejprve sedmý řádek, pak šestý), získali bychom odlišný výsledek

[1] Zda má weaver daný aspekt použít, pozná podle pointcut.

```
eq f x = k j i x .
```

Z tohoto důvodu záleží na tom, v jakém pořadí jsou advices zadefinované. Pro programátora, užívajícího tento systém, to může být nevýhoda, protože tak přebírá přímou zodpovědnost za korektní výsledek a bude muset dbát na pořadí v jakém jednotlivé advices definuje. Na druhou stranu tak získává určitou volnost.

5.3.5 Zapsání výsledku

V této fázi dojde k finální úpravě výstupu, poté se pouze zapíše výsledek po trasformaci (resp. weavingu) do souboru. Protože fáze Weaving bude produkovat již validní Maude kód, bude možné tento soubor načíst standardním Maude interpreterem.

5.3.6 Rozhraní programu

Aspect Weaver bude konzolová aplikace s jednoduchým rozhraním, jejíž manuálová stránka by vypadala takto:

```
MAUDEAOP(1)

NÁZEV
       maudeaop - aspect weaver pro AOP systém Maude

SOUHRN
       maudeaop [-o VÝSTUP] SOUBOR
       maudeaop [-h]

POPIS
       Realizuje weaving aspektů Maude na základě vstupního
       souboru SOUBOR.

       -o VÝSTUP
              Definuje název výstupního souboru. Pokud nebude
              nastaveno, použije se generický název "a.maude".
       -h
              Vypíše nápovědu a skončí.
```

Výpis 5.4. Manuálová stránka výsledné aplikace

Z manuálu lze vypozorovat, že aplikace má jeden povinný argument, kterým je zdrojový kód, jenž může (ale nemusí) obsahovat aspekty. Dále je možné specifikovat název výstupního souboru, což ale není povinné. V případě, že žádný výstupní soubor není uveden, použije se název `a.maude`.

5.4 Omezení

5.4.1 Omezení jazyka

- Syntaxe umožňuje přidat chování pouze skrze úpravy hodnoty vstupních proměnných operací.
- Jediný join point, který ve vytvářeném systému existuje, je provádění operace.
- Jazyk pro výběr pointcuts neumožňuje generalizovat operace ve smyslu počtu vstupních proměnných.

- Jazyk pro výběr pointcuts nabízí pouze jednoduchý nástroj (symbol ∗) pro kvantifikaci.

První omezení vychází z podstaty vytvářeného AOP systému a druhé je důsledkem prvního. Proto lze tato omezení považovat za nezvratná a nelze je odstranit rozšířením systému.

Třetí a čtvrté omezení se týká čistě jazyka pro výběr pointcuts. Třetí omezení znamená, že programátor může pomocí pointcut vybrat pouze ty operace, které mají stejný počet vstupů. Jinými slovy – nelze zobecnit výběr na libovolný počet vstupních proměnných. Obě tato omezení lze posunout rozšířením jazyka pro specifikaci pointcuts (samozřejmě následně i aspect weaveru), které je možné později udělat.

5.4.2 Vlastnosti aspect weaveru

- podporuje weaving pouze do rovnic,
- nepodporuje *pattern matching* v rovnicích [1],
- neprovádí sémantickou kontrolu zdrojového kódu Maude ani kódu v těle advice,
- neupozorní programátora na advice, pro kterou neexistuje aspekt.

Omezení weaveru plyne ze skutečnosti, že se v případě této bakalářské práce jedná pouze o model, a při implementaci bude kladen důraz na možnost snadného rozšíření.

[1] Standardní Maude toto umožňuje.

Kapitola 6
Implementace

Po kapitole zabývající se návrhem lze považovat za hotovou část práce, která měla za cíl vytvořit jazyk pro definici aspektů. Tato kapitola se proto bude týkat pouze implementace aspect weaveru.

6.1 Výběr jazyka

První důležité rozhodnutí, které musíme učinit, je, v jakém jazyce bude weaver implementovaný. Jazyk bude zvolený na základě jeho schopnosti naplnit potřeby, které v průběhu implementace vzniknou. Díky předcházejícímu návrhu lze tyto potřeby identifikovat ještě před započetím samotné implementace, protože již z návrhu jsou evidentní některé úkony, které bude muset program nevyhnutelně provádět. Mezi tyto úkony patří:

- čtení ze souboru (resp. zápis do souboru) při načítání zdrojového kódu (resp. zápisu výsledného kódu po weavingu),
- komunikace přes rozhraní příkazové řádky (načtení parametrů při spouštění),
- zpracování textu.

Mezi další vhodné vlastnosti, které by měl jazyk mít, lze zařadit možnost kompilace do nativního kódu, což je výhodou zejména pro budoucího uživatele frameworku, protože nebude potřeba žádného interpeteru. Navíc lze díky překladu do nativního kódu uvažovat o vyšším výkonu programu. Vítaná je zároveň i vyšší úroveň abstrakce, a to hlavně z důvodu přímočarosti vytvořeného kódu, v důsledku čehož bude snažší argumentace pro potřeby bakalářské práce.

Tři jazyky – C, Haskell, Java – jsou brané v úvahu. Tabulka 6.1 uvádí zastoupení požadovaných vlastností v jednotlivých jazycích.

vlastnost	C	Haskell	Java
práce se soubory	ano	ano	ano
podpora CLI	ano	ano	ano
snadné zpracování textu	ne	ano	ne
kompilovaný jazyk	ano	ano	ne
vysoká abstrakce	ne	ano	ano

Tabulka 6.1. Přehled požadovaných vlastností v uvážených jazycích

Přestože se na základě hodnot z tabulky 6.1 může zdát výsledek jednoznačný, není tomu tak. Například pracovat se soubory lze ve všech třech jazycích, avšak standardní Java API je, co se práce se soubory týče, mnohem bohatší než standardní API Haskellu. Z pohledu výkonu bude bezpochyby nejlepší adept jazyk C a faktor výkonu hraje v aplikacích typu weaveru jistě důležitou roli. Naopak zpracování textu v Haskellu je díky

línému vyhodnocování a skládání funkcí mnohem elegantnější a jednodušší než v Javě či C.

Nicméně implementace bude probíhat v jazyce Haskell, protože v aspektech důležitých pro tuto bakalářskou práci (zejména zpracování textu) vyniká a v aspektech, ve kterých nabízí Java či C více možností (například vstupně-výstupní operace), je pro naše účely dostačující.

6.2 Program stack

Při vývoji byl pro správu projektu použit program *stack* ve verzi 1.3.2, který poskytuje nástroje pro kompletní vývojový cyklus projektů v Haskellu. Nabízí snadnou instalaci kompilátoru GHC, instalaci dodatečných balíčků (knihoven), sestavování a testování projektu [17].

Stack využívá ke kompilaci GHC – pro tuto bakalářskou práci byla použita verze 8.0.2. GHC je kompilátor a interaktivní prostředí pro jazyk Haskell podporující všechny obvyklé platformy (Windows, Mac, Linux) [18]. V interaktivním režimu nabízí GHC i debugger umožňující vytvářet break-points, krokovat exekuci kódu, vypisovat hodnoty užívaných proměnných apod.

Pro implementaci jednotkových testů byl použit framework HUnit, který je integrován v programu stack, díky čemuž lze spouštět testy nebo sledovat pokrytí kódu nástroji, jenž stack obsahuje. HUnit nabízí podobné možnosti jako často používaný framework JUnit pro jednotkové testy v jazyce Java. Jednotkovým testům se bude dále věnovat kapitola 7.

6.2.1 Struktura projektu

Struktura projektu byla vygenerována výše uvedeným programem `stack`, ve výpisu 6.1 jsou vypsány nejdůležitější položky adresářové struktury.

Adresář `app` obsahuje hlavní modul `Main` – vstupní bod programu. V adresáři `src` se nachází všechny zdrojové soubory s moduly programu. Moduly definující jednotkové testy jsou v adresáři `test`.

Soubor `maudeaop.cabal` obsahuje metadata o projektu jako je název a verze, krátký popis projektu či jméno autora. Dále jsou zde v sekcích popsány jednotlivé knihovny, spustitelné soubory, testovací sady. Pro každou sekci lze definovat závislosti na jiných balíčcích anebo určit, které z naprogramovaných modulů mají být její součástí.

```
maudeaop/
|-- app/
|-- src/
|-- test/
|-- maudeaop.cabal
```

Výpis 6.1. Struktura projektu

6.3 Moduly programu

Jak již bylo zmíněno v kapitole 2.3.4, v Haskellu je kód členěn do modulů. Tabulka 6.2 obsahuje přehled vytvořených modulů se stručným popisem jejich účelu.

Podrobnějšímu vysvětlení účelu a činnosti některých modulů se budou věnovat následující kapitoly.

název modulu	popis
`AdviceApplicator`	aplikuje advices na rovnice
`AspectContext`	sestavuje kontext aspektů
`CommandLineInterface`	analyzuje vstupní argumenty programu
`Input`	zpracovává vstup ze zdrojových souborů
`Main`	vstupní bod programu, řeší vstupně/výstupní operace
`OperationContext`	sestavuje kontext operací
`Output`	zpracovává výstup programu
`Parser`	převádí syntaktické konstrukty do datových struktur
`PointcutMatcher`	rozhoduje, zda daný pointcut vybírá konkrétní operaci
`Result`	poskytuje datovou strukturu pro výsledky funkcí
`Utils`	obsahuje užitečné funkce bez specifického užití
`Weaver`	realizuje weaving

Tabulka 6.2. Přehled vytvořených modulů

6.4 Problém výjimek

Weaver se musí, stejně jako každý software, vypořádat s chybami či výjimkami, které mohou při běhu nastat. Typický příklad, ve kterém může dojít k výjimce, lze vidět ve výpisu 6.2, kde je definována funkce `addSquareDivide`, která volá druhou funkci `divide`. Obě funkce očekávají celočíselný parametr a vrací celé číslo. Problém nastává, když je funkce `addSquareDivide` zavolána s nulovým parametrem, protože po zavolání dojde k vyhodnocení výrazu `divide 0` a z důvodu dělení nulou, skončí celé volání chybou.

```
1  addSquareDivide :: Int -> Int
2  addSquareDivide x = x^2 + (divide x)
3    where
4      divide :: Int -> Int
5      divide y = 20 `div` y
```

Výpis 6.2. Místo s možným vznikem výjimky

Imperativní jazyky, jako je Java nebo C/C++, řeší nastalé chyby vyhazováním výjimek, což Haskell nepodporuje (resp. podporuje, ale jiným způsobem, který je méně přehledný), nebo případně vrácením chybového kódu (např. „vrať `-1` pokud funkce selže, jinak vrať `0`“).

Řešit chyby vracením chybového kódu lze i v Haskellu, ale jsme tak nuceni ke každému volání funkce připojit kontrolu její návratové hodnoty. Tento přístup je ale pro rozsáhlejší programy nevhodný, protože důsledkem této kontroly jsou složitější těla funkcí, méně přehledný kód a opakované kontroly výsledku volání.

Elegantnějším řešením, které Haskell nabízí, je použití monád a funkce `>>=`, která se často nazývá *bind*. Díky této funkci lze chybu, vzniklou hluboko ve stromu volání funkcí, propagovat nahoru.

6.4.1 Monády Maybe a Either

Ve standardní knihovně Haskellu je dostupná monáda `Maybe` (pro její definici viz výpis 6.3), která by problém s chybami pomohla vyřešit. Funkce, ve kterých by mohla vzniknout chyba, by vracely typ `Maybe`; v případě chyby by došlo k navrácení `Nothing`, jinak `Just` *hodnota*.

```
data Maybe a = Just a | Nothing
instance Monad Maybe where
  return x = Just x
  Nothing >>= f = Nothing
  Just x >>= f  = f x
  fail _ = Nothing
```

Výpis 6.3. `Maybe` monáda

Nevýhodou monády `Maybe` je, že nelze popsat, jaká chyba nastala, protože konstruktor `Nothing` nemá žádný parametr. Monáda `Maybe` je proto vhodná pro jednoduché funkce, u kterých nás nezajímá původ chyby (viz výpis 6.4), pro komplexnější správu chyb je ale nedostačující.

```
1 addSquareDivide :: Int -> Maybe Int
2 addSquareDivide x = do
3   divided <- divide x
4   return (x^2 + divided)
5
6 divide :: Int -> Maybe Int
7 divide 0 = Nothing
8 divide y = Just (20 `div` y)
```

Výpis 6.4. Použití monády `Maybe`

Další datovou strukturou, kterou bychom mohli použít, je `Either`, která dovoluje chybu reprezentovat libovolným datovým typem. Pro naše potřeby by postačoval typ `Either String a`, pak by bylo možné chybu vracet pomocí konstruktoru `Left`, a pomocí *bind* by se daná chyba opět zpropagovala do nejvyššího místa volání.

```
1 data Either a b = Left a | Right b
2
3 instance Monad (Either e) where
4   return = Right
5   Right m >>= k = k m
6   Left e  >>= _ = Left e
```

Výpis 6.5. Monáda `Either a`

6.4.2 Monáda Result

Typ `Either` je velmi obecný. Výrazy, použité jako název datového typu i jednotlivých konstruktorů, jsou poněkud zavádějící a zároveň instance některých typových tříd jsou implementované genericky, což by v některých případech bylo nevyhovující. Proto byl vytvořen modul `Result` obsahující algebraický datový typ (ADT) se shodným jménem, jehož definice je ve výpisu 6.6.

```
1 Result a = Value a | Error String
2
3 instance Monad Result where
4   return = Value
5   Error x >>= _ = Error x
6   Value x >>= f = f x
7   fail x = Error x
```

Výpis 6.6. Definice ADT `Result` (včetně monády)

Takto nadefinovaná funkce *bind* umožňuje řetězit jednotlivá volání například pomocí `do` notace (což je přehlednější syntaktický zápis, viz 6.7) tak, že chyba vzniklá v libovolné části řetězu bude výsledkem celého řetězu.

Zmíněné zřetězení je ilustrované v příkladu z výpisu 6.7, kde předpokládáme, že funkce `f`, `g` a `h` vrací typ `Result a`. Pokud bude chyba `Error` výsledkem funkce `a`, `b` nebo `h`, bude výsledkem i celé funkce `compute`, což vyplývá z definice funkce `(>>=)`. V případě, že volání všech vnitřních funkcí postupně vrátí hodnotu `Value`, bude výsledek funkce `compute` také hodnota `Value`.

```
1  compute :: Int -> Result Int
2  compute x = do
3    a <- f x
4    b <- g a
5    h b
```

Výpis 6.7. Ukázka použití monády Result a `do` notace

Pro typ `Result` bylo vytvořeno několik podpůrných funkcí jako je `isError` nebo `isValue`, které očekávají vstup typu `Result a` a vrací `True`, pokud je zadaný vstup chyba, resp. hodnota, jinak `False`. Další definovanou funkcí je `fromValue`, která dokáže z konstruktoru `Value` v získat a vrátit hodnotu v.

Nejvyužívanější z funkcí definovaných v modulu `Result` je funkce `foldResult`, kterou lze vidět ve výpisu 6.8. Tato funkce přijímá na vstupu funkci $f : A \rightarrow Result\ B$ a seznam prvků $a \in A$, který označme S. Poté postupně aplikuje funkci f na každý prvek a seznamu S v pořadí, v jakém jsou prvky v seznamu uvedeny, a vytváří tak nový seznam S'. V případě, že funkce f skončila bez chyby (tzn. nevrátila `Error`) pro všechny prvky a seznamu S, bude výsledkem `Value` S'. Pokud funkce f vrátí chybu pro alespoň jeden prvek seznamu, bude tato chyba výsledkem celého volání `foldResult`. K implementaci této vlastnosti byla využita funkce *bind*.

```
1  foldResult :: (a -> Result b) -> [a] -> Result [b]
2  foldResult _ []     = Value []
3  foldResult f (l:ls) = do
4    v  <- f l
5    v' <- foldResult f ls
6    return (v:v')
```

Výpis 6.8. Funkce `foldResult`

6.5 Program

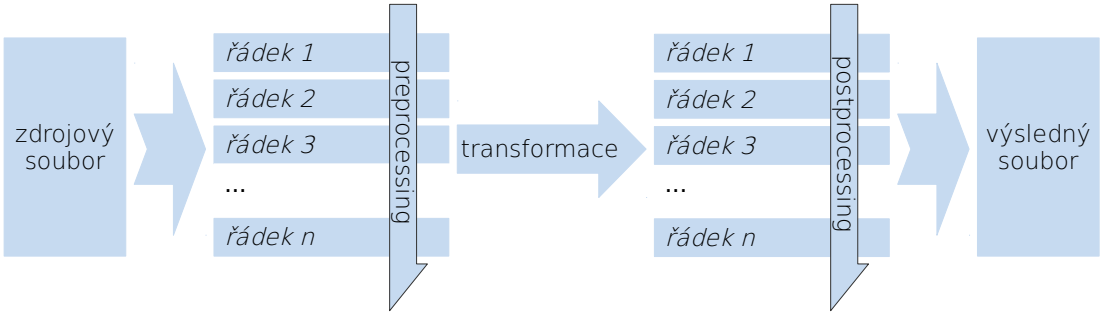
Implementace se bude řídit postupem navrženým v kapitole 5.3.1. Program bude interagovat s uživatelem přes jednoduché rozhraní příkazové řádky, konkrétně pomocí argumentů programu (viz 5.3.6).

Po načtení argumentů lze jeho další činnost rozdělit do tří fází či celků, které byly zmíněné v návrhu. Schéma činnosti lze pozorovat na obrázku 6.1 a jednotlivé fáze budou vysvětleny v dalším textu.

V první fázi dojde k načtení zdrojového kódu, který následně projde preprocessingem, což připraví zdrojový kód pro další zpracování.

Dále bude na vstupní zdrojový kód aplikována transformace, jenž realizuje weaving (způsobu, jak zaplétat aspekty, se věnovala kapitola 5.3.3). Výsledkem transformace je zdrojový kód zbavený vět souvisejících s aspekty (tj. typu `as` a `ad`) a s upravenými rovnicemi podle definovaných aspektů.

Výsledek po transformaci projde postprocessingem a zapíše se do souboru.



Obrázek 6.1. Činnost weaveru schématicky

6.6 Rozhraní příkazového řádku

Program při spuštění očekává povinný argument, který určuje umístění zdrojového souboru. Uživatel může také pomocí volby `-o` nepovinně specifikovat, kam chce uložit výsledek po weavingu. Případně lze vypsat jednoduchou nápovědu pomocí přepínače `-h`.

O načtení argumentů se stará modul `Main`, což je v rámci této práce jediný modul obsahující funkce s vedlejším efektem. Ve funkcích s vedlejším efektem lze využívat funkce interagující s okolím jako je např. funkce `getArgs :: IO [String]` z balíčku `System.Environment`, která je použita pro načtení argumentů. Pokud zavoláme program s argumenty `./maudeaop -o output.maude input.maude`, vrátí funkce `getArgs` hodnotu `IO ["-o", "output.maude", "input.maude"]`.

Po načtení argumentů lze použít modul `CommandLineInterface`, aby program porozuměl vstupním argumentům. Tento modul definuje typ `Settings` (viz výpis 6.9), který v sobě uchovává informace, které lze získat ze vstupních argumentů.

```
data Settings = Settings { inputFile  :: String,
                           outputFile :: String,
                           showHelp   :: Bool }
                           deriving (Eq, Show)
```

Výpis 6.9. Typ pro uchování nastavení

Protože některé možnosti nastavení nejsou povinné (výstupní soubor a zobrazení nápovědy), musí pro ně existovat implicitní hodnota. Z toho důvodu byla vytvořena funkce `defaultSettings`, jenž vrací implicitní nastavení, ve kterém je vstupní soubor `inputFile` nastavený na prázdný řetězec `""`, výstupní soubor `outputFile` na `a.maude` (viz kapitola 5.3.6) a zobrazení nápovědy `showHelp` na `False`.

Modul `CommandLineInterface` obsahuje, kromě typu `Settings` a dalších funkcí, ještě funkci `cli`. Tato funkce přijímá seznam argumentů (který byl dříve získán funkcí `getArgs`) a vrací výsledné nastavení `Settings`, které vzniká úpravami výchozího nastavení (vytvořené funkcí `defaultSettings`) na základě hodnot argumentů.

Podle výsledného nastavení se program následně zachová.

6.7 Načítání zdrojového souboru

Cesta ke zdrojovému souboru je uložena ve struktuře `Settings`, jejíž hodnoty byly nastaveny ze vstupních argumentů. Pro načtení obsahu souboru je použita funkce `readFile`, což je opět funkce s vedlejším efektem, která jako parametr očekává cestu k souboru a vrací jeho obsah.

6.7.1 Preprocessing

Po načtení obsahu dochází k předzpracování zdrojového kódu. Funkce pro předzpracování, která je aplikována na všechny zdrojové řádky, je součástí modulu `Input` a její podobu lze vidět ve výpisu 6.10. Během předzpracování jsou nejprve všechny řádky zbaveny mezer zleva a zprava, což zajistí funkce `strip` definovaná v modulu `Utils`, a následně jsou odfiltrovány řádky splňující predikát `(not . null)`, který vrátí `True` pro prázdný řetězec (tzn. odstranění prázdných řádek).

```
1  preprocess :: [String] -> [String]
2  preprocess = filter (not . null) . map strip
```

Výpis 6.10. Funkce pro předzpracování textu

Důvodem k předzpracování je komfort v další činnosti weaveru, neboť není nutné kontrolovat, zda aktuálně zpracovávaný řádek (v rámci transformace) není prázdný apod. Výsledkem načtení a předzpracování je proto zdrojový kód připravený k nadcházející transformaci.

6.8 Weaving

O weaving se stará funkce `weave` (definice ve výpisu 6.11) z modulu `Weaver`. Před transformací, realizující weaving, se v rámci zmíněné funkce vytváří dva typy kontextů: *aspektový*, který v sobě uchovává všechny aspekty (včetně příslušných advices) definované ve zdrojovém souboru, a *operační*, který obsahuje deklarace všech operací, pro něž existuje alespoň jeden aspekt (tzn. pointcut daného aspektu zahrnuje i tuto konkrétní operaci). Kontexty jsou vytvářeny z důvodu opakovaného využívání informací v nich uchovaných.

```
1  weave :: [String] -> Result [String]
2  weave ls = do
3    aspects          <- parseAspects aspects'
4    advices          <- parseAdvices advices'
5    aspectContext    <- buildAspectContext aspects advices
6    operationContext <- buildOperationContext aspects operations
7    foldResult (transform aspectContext operationContext) ls
8    where
9      aspects'   = filter isAspect ls
10     advices'   = filter isAdvice ls
11     operations = filter isOperation ls
12     equations  = filter isEquation ls
```

Výpis 6.11. Funkce realizující weaving

K sestavení operačního kontextu je nutné mít již sestavený aspektový kontext – a před sestavením aspektového kontextu je nezbytné zjistit, jaké aspekty byly ve zdrojovém souboru nadefinované. K tomu slouží modul `Parser`.

6.8.1 Modul Parser

Kromě jiného definuje modul `Parser` dva algebraické datové typy

- `AspectSen` (jako *aspect sentence*),
- `AdviceSen` (jako *advice sentence*).

Tyto typy představují věty definující aspekt (mají klíčové slovo `as`) a věty definující advice (s klíčovým slovem `ad`). Oba typy (viz výpis 6.12) mají pouze jeden konstruktor pojmenovaný shodně s názvem příslušného typu.

Konstruktor `AspectSen` má 3 parametry:

1. název daného aspektu,
2. počet vstupů,
3. pointcut, na který má být aspekt aplikován.

Konstruktor `AdviceSen` má také 3 parametry:

1. název aspektu, ke kterému advice přísluší,
2. názvy proměnných dané advice,
3. tělo advice.

Všechny položky použité v konstruktorech vycházejí přímo z vět zdrojového kódu (viz 5.2 pro definici syntaxe).

```
data AspectSen = AspectSen String Int String
                 deriving (Eq, Show)
data AdviceSen = AdviceSen String [String] String
                 deriving (Eq, Show)
```

Výpis 6.12. Datové typy pro reprezentaci `as` a `ad` vět

Modul `Parser` také definuje funkce `parseAspects` a `parseAdvices`, které se používají ve funkci `weave` z ukázky 6.11 (na řádcích 3 a 4) a jejichž typy jsou ve výpisu 6.13.

Funkce `parseAspects` očekává na vstupu seznam vět deklarujících aspekt a následně se pokusí pro každou větu vytvořit novou instanci typu `AspectSen`. Během tohoto procesu dochází k syntaktické kontrole každé věty. V případě chybné syntaxe alespoň jedné věty dochází k vrácení hodnoty `Error` e, kde e je stručný text popisující nastalou chybu. Pokud k žádné chybě nedojde, vrátí funkce `Value` v, kde v je seznam instancí `AspectSen` uspořádaný tak, jak byly uspořádané korespondující věty ve vstupním seznamu.

Funkce `parseAdvices` funguje obdobně jako funkce `parseAspects`, popsaná v předchozím odstavci, avšak pro věty definující nové advices. Proto je typ výstupu `Result [AdviceSen]`.

Zachovávání uspořádání vět je velice důležitá vlastnost pro pozdější aplikaci aspektů. Pokud by se uspořádání změnilo, docházelo by k neočekávaným výsledkům weavingu (viz 5.3.4).

```
parseAspects :: [String] -> Result [AspectSen]
parseAdvices :: [String] -> Result [AdviceSen]
```

Výpis 6.13. Typy funkcí pro parsování vět

6.8.2 Sestavení aspektového kontextu

Po načtení a připravení aspektů a advices lze nechat sestavit kontexty (k tomu dochází na řádcích 5 a 6 výpisu 6.11). Protože k sestavení operačního kontextu je potřeba aspektový kontext, musí být nejprve sestaven ten.

Pro sestavení aspektového kontextu slouží modul `AspectContext` a jeho funkce `buildAspectContext`, která očekává na vstupu seznam instancí `AspectSen` a seznam instancí `AdviceSen`. Poté se pokusí ke každému aspektu najít všechny advices k němu příslušící. Může se stát, že pro aspekt nebude nalezena žádná advice a z toho důvodu vrací funkce typ `Result [Aspect]` – v případě, že alespoň pro jeden aspekt nebude nalezena žádná advice, vrátí funkce `Error` e, kde e popisuje, který aspekt nemá žádnou

advice. Jinak je vráceno `Value` v, kde v je seznam typů `Aspect`, což je dosud nezmíněný typ. Typ `Aspect` označuje dvojici typu `AspectSen` a seznamu prvků `AdviceSen` (viz výpis 6.14), tzn. spojuje dohromady deklaraci aspektů a všechny definice advices patřící k dané deklaraci aspektu.

```
type Aspect = (AspectSen, [AdviceSen])
```

Výpis 6.14. Typ spojující deklaraci aspektu a seznam příslušných advices

6.8.3 Sestavení operačního kontextu

K sestavení operačního kontextu slouží funkce `buildOperationContext` z modulu `OperationContext`. Funkce má dva parametry: seznam deklarací aspektů a seznam operací. Funkce vrátí seznam deklarací (deklarací je myšlen název včetně vstupů, bez datového typu a klíčového slova `op`) těch operací, pro které existuje alespoň jeden aspekt, jehož pointcut zahrnuje danou operaci.

6.8.4 Transformace

Po sestavení kontextů lze spustit transformaci každého řádku (viz řádek 7 z výpisu 6.11). Transformace je realizována funkcí `transform` z výpisu 6.15. Funkce očekává na vstupu aspektový a operační kontext a větu k transformaci.

Pro implementaci byly použity *guards* (stráže), jenž fungují podobně jako série podmínek `if-then-else`. Oproti standardním podmínkám jsou ale guards přehlednější. Jednotlivé podmínky se zapisují za svislou čáru; kód, který se má s danou podmínkou provést, je pak umístěn za symbol `=`. Provede se vždy pouze první kód, jehož podmínka je vyhodnocena jako `True`, a podmínka `otherwise` je vždy pravdivá. Z toho vyplývá způsob, jakým funkce `transform` pracuje.

```
1 transform :: [Aspect] -> [String] -> String -> Result String
2 transform aspCxt opCxt s
3   | isAspect s   = Value ""
4   | isAdvice s   = Value ""
5   | isEquation s = transformEquation aspCxt opCxt s
6   | otherwise    = Value s
```

Výpis 6.15. Funkce pro transformaci řádků

Pokud transformovaná věta deklaruje aspekt nebo definuje advice, je transformovaná na prázdný řetězec (tzn. je odstraněna z výstupu). Pokud je věta cokoliv jiného, co zároveň není ani rovnice `eq`, transformace ji nezmění. V případě rovnice se volá funkce pro transformaci rovnic `transformEquation`, která se pokouší na danou rovnici aplikovat advices.

6.8.5 Transformace rovnic

```
1 transformEquation :: [Aspect] -> [String] -> String -> Result String
2 transformEquation aspCxt opCxt s =
3   case searchOperationContext opCxt s of
4     Nothing -> Value s
5     Just op -> applyAdvices op s (advices op)
6   where
7     aspects = searchAspectContext aspCxt
8     advices = concatMap snd . aspects
```

Výpis 6.16. Funkce pro transformaci rovnic

Aplikace advices na rovnici vychází z funkce `transformEquation`, jejíž definice je ve výpisu 6.16. Funkce má stejný typ, jako funkce `transform` z výpisu 6.15 – očekává na vstupu aspektový kontext, operační kontext a transformovanou rovnici, vrací výsledek po transformaci.

Ještě před aplikací advices se funkce `transformEquation` pokusí najít deklaraci operace dané rovnice v operačním kontextu. Hledání je realizováno pomocí funkce `searchOperationContext` z modulu `OperationContext`, která vrací `Nothing`, pokud operace nebyla nalezena, jinak `Just` *op*, kde *op* je nalezená deklarace z operačního kontextu.

V případě, že nebyla operace v operačním kontextu nalezena, znamená to, že daná rovnice (resp. operace) nemá žádný aspekt – vyplývá to z definice operačního kontextu, viz 6.8.3, a proto se rovnice nezmění.

Naopak pokud byla deklarace operace nalezena, znamená to, že existuje alespoň jeden aspekt, jehož pointcut zahrnuje danou operaci (resp. rovnici). Pro takovou rovnici jsou nalezeny všechny aspekty s pointcutem zahrnujícím danou operaci. Toto hledání aspektů je implementováno skládáním funkcí a částečným vyhodnocováním, jak lze vidět na řádcích 7 a 8 výpisu 6.16. Nejprve je na 7. řádku použita funkce `searchAspectContext`, která očekává dva parametry, aspektový kontext a deklaraci operace, a vrací seznam aspektů, které jsou pro danou operaci relevantní. Funkci je ale předán pouze jeden parametr (aspektový kontext) a takto částečně vyhodnocená funkce je pak použita ve skládání na 8. řádku. Druhým členem zmíněného skládání je funkce `(concatMap snd)`, která na každý prvek vstupního pole použije funkci `snd` (ta očekává na vstup dvojici prvků a vrátí druhý z nich) a výsledky spojí. Skládáním tudíž vznikla funkce nazvaná `advices`, která na vstupu očekává deklaraci operace a vrací seznam všech advices (ze všech aspektů).

6.8.6 Aplikace advices

Za předpokladu, že byla nalezena deklarace operace a byly pomocí funkce `advices` vyhledány všechny advices, volá se funkce `applyAdvices` (viz 5. řádek ve výpisu 6.16) z modulu `AdviceApplicator`. Tento modul zodpovídá za aplikování advices do rovnice.

Funkce `applyAdvices` očekává na vstupu deklaraci operace, rovnici a seznam advices, které mají být aplikovány. Funkce poté postupně pomocí funkce `applyAdvice` aplikuje jednotlivé advices ve stejném pořadí, v jakém jsou v seznamu.

Funkce `applyAdvice` je naimplementována přesně podle postupu navrženého v kapitole 5.3.3, jak lze vidět na definici ve výpisu 6.17.

```
1  applyAdvice :: String -> String -> AdviceSen -> Result String
2  applyAdvice op eq ad = do
3    mapping  <- mapVariables op eqDecl ad
4    mappedAd <- applyMapping mapping adBody
5    return (rewriteEquation eq mappedAd)
6    where
7      adBody = adviceBody ad
8      eqDecl = eqDeclaration eq
```

Výpis 6.17. Funkce pro aplikaci advice na rovnici

Funkce `mapVariables` realizuje mapování proměnných, proto vrací seznam dvojic, kde první prvek z dvojice je původní proměnná a druhý je proměnná cílová. Tento seznam získá na základě vstupních proměnných, kterými jsou deklarace operace, rovnice a advice. Funkce musí pracovat s deklarací operace, aby správně nalezla názvy proměnných v dané rovnici.

S formátem, který je výsledkem volání funkce `mapVariables`, dokáže pracovat funkce `applyMapping`, která přepíše výskyty všech prvních prvků z dvojic na korespondující druhé prvky, čímž dojde k sjednocení názvů proměnných rovnice a advice.

Funkce `rewriteEquation` konečně nahradí výskyt přepisované proměnné za cíl specifikovaný v těle advice. Weaver se tudíž nezabývá sémantikou těl advices, ale pracuje pouze na úrovni syntaxe.

6.9 Zapsání výsledku

Transformací došlo k zapletení aspektů do rovnic, a tak je weaving u konce. V poslední fázi běhu programu projde výsledný kód postprocessingem a zapíše se do výstupního souboru.

6.9.1 Postprocessing

Postprocessing je realizovaný v modulu `Output` funkcí `postprocess`, jenž odstraní prázdné řádky. Ty mohou vzniknout při transformaci vět deklarující aspekty nebo definující advice (viz kapitola 6.8.4 a výpis 6.15).

6.9.2 Výstup do souboru

Zapsání do souboru je implementované v modulu `Main` pomocí funkce s vedlejším efektem `writeFile` z modulu `System.IO`. Jako název souboru je použita hodnota z vytvořeného nastavení `Settings` (viz kapitola 6.6)

Kapitola 7
Testování

Testování bylo realizováno formou jednotkových testů, které byly implementované pomocí frameworku HUnit.

7.1 Framework HUnit

HUnit je framework pro jednotkové testování inspirovaný známým, široce používaným Java frameworkem JUnit. HUnit dovoluje vytvářet testy, seskupovat je do sad a vyhodnocovat je [19].

7.2 Jednotkové testy

Pro každý modul byl vytvořený samostatný testovací modul. Testovací moduly byly pojmenované přidáním přípony `Test` k názvu příslušného testovaného modulu, např. pro modul `Parser` existuje testovací modul `ParserTest`. Všechny testovací moduly jsou umístěné v adresáři `test/` projektu.

Každý testovací modul obsahuje funkci `tests`, která vrací seznam testů definovaných v daném modulu. Příklad takové funkce je ve výpisu 7.1.

```
tests = TestList [
  TestLabel "MatchAspectAdviceTest" testMatchAspectAdvice,
  TestLabel "PairAspectAdvicesTest" testPairAspectAdvices,
  TestLabel "BuildAspectContextTest" testBuildAspectContext,
  TestLabel "SearchAspectContextTest" testSearchAspectContext]
```

Výpis 7.1. Definice seznamu testů

Mezi testovací moduly patří také testovací modul `Main` (jiný modul než modul `Main` v programu), který obsahuje funkci `main`, což je funkce, která je volána při spuštění testů. Z funkce `main` jsou spouštěné všechny testy, jak lze pozorovat ve výpisu 7.2.

```
1  import qualified AdviceApplicatorTest
2  import qualified AspectContextTest
3  import qualified CommandLineInterfaceTest
4
5  main = do
6    runTestTT AdviceApplicatorTest.tests
7    runTestTT AspectContextTest.tests
8    runTestTT CommandLineInterfaceTest.tests
```

Výpis 7.2. Ukázka kódu v testovacím modulu Main

7.2.1 Implementace testů

V jednotkových testech byly použity funkce `assertBool` nebo `assertEqual` podle toho, jaký typ vrací testovaná funkce – `assertBool` pro funkce, které vrací `Boolean`, a `assertEqual` pro všechny ostatní.

Funkce `assertBool` má 2 vstupní parametry: popisek případné chyby a výraz, který testujeme. Funkce po zavolání zkontroluje, zda se zadaný výraz vyhodnotí na `True`. V případě, že se výraz vyhodnotí na `False`, vznikne upozornění na chybný výsledek.

Funkce `assertEqual` očekává tři vstupy: popisek případné chyby, očekávaný výsledek a testovaný výraz. Funkce upozorní na chybu, pokud se výraz nerovná očekávanému výsledku.

Každý jednotkový test kontroluje alespoň jeden pozitivní výraz (tj. takový, u kterého je daná odpověď očekávaná) a alespoň jeden negativní výraz (tj. takový, který by neměl být vyhodnocen jako True, nebo by měl například vrátit chybu `Error`). Tento přístup je demonstrován na příkladu z výpisu 7.3.

```
1  testMatchAspectAdvice = TestCase(do
2    assertBool "testing aspect-advice matching" $
3      matchAspectAdvice (AspectSen "foo" 2 "")
4                        (AdviceSen "foo" ["x", "y"] "")
5    assertBool "testing aspect-advice matching"
6      $ not $ matchAspectAdvice (AspectSen "foo" 1 "")
7                                (AdviceSen "foo" ["x", "y"] "")
8    assertBool "testing aspect-advice matching"
9      $ not $ matchAspectAdvice (AspectSen "foo" 2 "")
10                               (AdviceSen "foobar" ["x", "y"] ""))
```

Výpis 7.3. Ukázka jednotkového testu

Ve výpisu 7.3 je testována funkce `matchAspectAdvice`, která rozhoduje, zda dvojice `AspectSen` a `AdviceSen` (tzn. deklarace aspektu a definice advice) patří k sobě. První `assertBool` testuje pozitivní případ, kdy testovaná dvojice skutečně patří k sobě. Další dvě funkce `assertBool` testují, zda testovaná funkce rozezná případy, kdy k sobě aspekt a advice nepatří, tzn. mají různý počet vstupů nebo se jinak jmenují.

7.2.2 Pokrytí kódu

Pokrytí kódu testy uvádí tabulka 7.1.

název modulu	definice nejvyšší úrovně	alternativy	výrazy
`AdviceApplicator`	100%	100%	90%
`AspectContext`	100%	75%	85%
`CommandLineInterface`	78%	85%	75%
`Input`	100%	-	100%
`OperationContext`	100%	100%	100%
`Output`	100%	-	100%
`Parser`	76%	100%	80%
`PointcutMatcher`	100%	100%	100%
`Result`	100%	94%	89%
`Utils`	100%	100%	100%
`Weaver`	100%	100%	95%
celkem	87%	94%	86%

Tabulka 7.1. Pokrytí kódu testy

Data použitá v tabulce vychází z reportu, který byl vygenerovaný pomocí programu stack. Sloupec *definice nejvyšší úrovně* uvádí, jaká část definovaných funkcí nejvyšší úrovně byla otestována.

Alternativy vznikají při výskytu podmínek, to znamená, že sloupec *alternativy* popisuje, na kolik procent byly otestovány všechny podmínky.

Sloupec *výrazy* říká, kolik procent použitých výrazů bylo při testování vyhodnoceno.

Údaje o pokrytí mohou být zavádějící, protože například definice nejvyšší úrovně zahrnují i odvozené funkce pro porovnávání (ze třídy `Eq`) nebo pro výpis (ze třídy `Show`), které v rámci jednotkových testů nemělo smysl testovat.

Kapitola 8
Modelový příklad

Pro ukázku činnosti systému byl vytvořen jednoduchý program ve standardním Maude, který lze vidět ve výpisu 8.1. V programu jsou definované 4 operace `f`, `g`, `h` a `i`. Každá z operací má různou definici, avšak ve všech se vyskytuje společný člen `(a * a + b)`.

```
1  fmod EXAMPLE is
2    protecting NAT .
3    op f _ _ : Nat Nat -> Nat .
4    op g _ _ : Nat Nat -> Nat .
5    op h _ _ : Nat Nat -> Nat .
6    op i _ _ : Nat Nat -> Nat .
7    op j _ _ : Nat Nat -> Nat .
8    op k _ _ : Nat Nat -> Nat .
9    vars a b  : Nat .
10   eq f a b = 2 * (a * a + b) + b .
11   eq g a b = (a * a + b) + 3 * b .
12   eq h a b = (g (a * a + b) b) + b .
13   eq i a b = b + (a * a + b) * 12 .
14   eq j a b = (a * a + b) .
15   eq k a b = 7 + b * (a * a + b) .
16 endfm
```

Výpis 8.1. Příklad kódu ve standardním Maude

Společný člen lze díky vytvořenému systému vytknout ze všech operací do jednoho aspektu a zredukovat tak duplicitní kód, což je demonstrováno v ukázce z výpisu 8.2.

```
1  fmod EXAMPLE is
2    protecting NAT .
3    op f _ _ : Nat Nat -> Nat .
4    op g _ _ : Nat Nat -> Nat .
5    op h _ _ : Nat Nat -> Nat .
6    op i _ _ : Nat Nat -> Nat .
7    op j _ _ : Nat Nat -> Nat .
8    op k _ _ : Nat Nat -> Nat .
9    vars a b  : Nat .
10   eq f a b = 2 * a + b .
11   eq g a b = a + 3 * b .
12   eq h a b = (g a b) + b .
13   eq i a b = b + a * 12 .
14   eq j a b = a .
15   eq k a b = 7 + b * a .
16   as aspect _ _  : * _ _ .
17   ad aspect x y : x = (x * x + y) .
18 endfm
```

Výpis 8.2. Shodný kód zapsaný za užití aspektů

Byl vytvořený aspekt s názvem `aspect` a jako pointcut bylo nastaveno `* _ _`, což zajistí, že do výběru budou zahrnuty všechny 4 definované operace. Pro aspekt byla nadefinovaná jedna advice, která do operací přidává opakovaně se vyskytující část.

Kód s aspekty byl následně transformovaný pomocí vytvořeného aspect weaveru. Výsledek po weavingu lze pozorovat ve výpisu 8.3. Výstup je v porovnání s původním kódem z výpisu 8.1 téměř totožný. Liší se pouze v mezerách kolem závorek, které vznikají během přepisování, což ale neovlivňuje funkcionalitu, a v odstraněném odsazení, což je důsledek postprocessingu.

```
1  fmod EXAMPLE is
2  protecting NAT .
3  op f _ _ : Nat Nat -> Nat .
4  op g _ _ : Nat Nat -> Nat .
5  op h _ _ : Nat Nat -> Nat .
6  op i _ _ : Nat Nat -> Nat .
7  op j _ _ : Nat Nat -> Nat .
8  op k _ _ : Nat Nat -> Nat .
9  vars a b : Nat .
10 eq f a b = 2 * ( a * a + b ) + b .
11 eq g a b = ( a * a + b ) + 3 * b .
12 eq h a b = (g ( a * a + b ) b) + b .
13 eq i a b = b + ( a * a + b ) * 12 .
14 eq j a b = ( a * a + b ) .
15 eq k a b = 7 + b * ( a * a + b ) .
16 endfm
```

Výpis 8.3. Výsledný kód po zpracování aspektů systémem

V uvedeném příkladu není výhoda frameworku, tj. úspora napsaných znaků, tolik zřejmá, protože se projevila jen mírně. K viditelnější úspoře dojde při větším počtu operací s opakujícím se kódem či při větším počtu znaků v opakujícím se kódu. Například při 10 operacích, obsahujících duplicitní kód o délce 25 znaků, může být úspora až 20 %. Tuto závislost vysvětluje následující kapitola 8.1.

8.1 Úspora

Jedním z faktorů, které můžeme u aspektově orientovaného systému sledovat, je úspora napsaného kódu.

K úspoře napsaného kódu nedochází, pokud by délka kódu byla chápána jako počet řádků, protože pro počet řádků bude vždy platit vztah

$$R_a \geq R_m + 2,$$

kde R_a je počet řádků při použití aspektů a R_m je počet řádků kódu standardního Maude bez použití aspektů. Důvodem tohoto vztahu je nutnost deklarace aspektu a definice alespoň jedné advice pro zmíněný aspekt.

Úsporu napsaného kódu lze ale pozorovat v počtu napsaných znaků, protože systém umožňuje ve specifických případech vytknout duplicitní kód do aspektu. V tomto případě platí pro úsporu U vztah

$$U = 1 - \frac{Z_a}{Z_m},$$

kde Z_a je počet znaků v kódu s aspekty a Z_m počet znaků v kódu standardního Maude. Vztah uvádí podíl dosažené úspory. Složky Z_a a Z_m se budou lišit v definicích rovnic (aspektový přístup vytkne duplicity) a v aspektovém přístupu budou samozřejmě navíc definice aspektů. Na základě této skutečnosti lze jednotlivé složky pro představu vyjádřit pro jeden aspekt:

$$Z_a = S_a + S_m + M + (N + 1) \cdot P,$$

$$Z_m = S_m + N \cdot M.$$

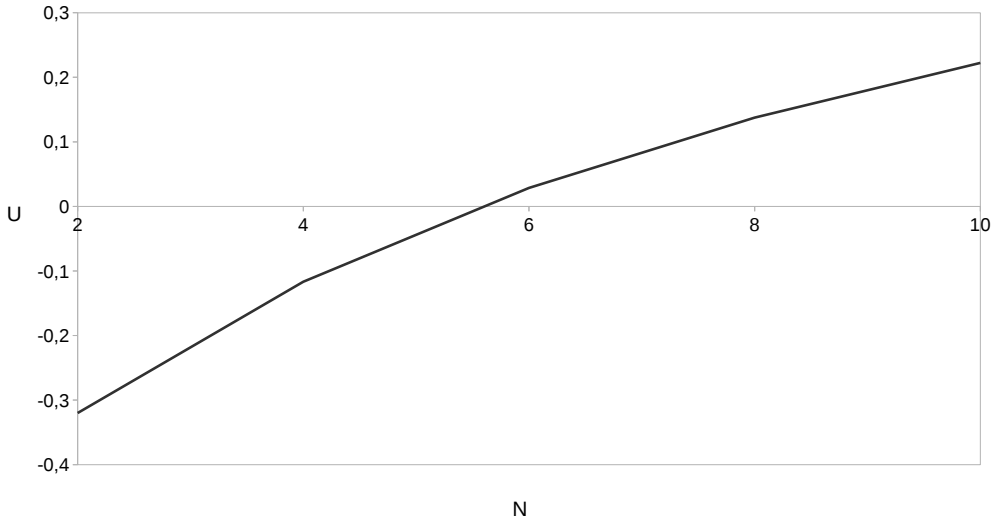
Proměnná S_a vyjadřuje počet znaků nutných pro zadefinování aspektu a S_m počet znaků v syntaktických konstrukcích, které mají oba přístupy společné (definovaní modulů, rovnice bez duplicitního kódu apod.), proměnná M vyjadřuje počet znaků opakujícího se kódu a proměnná N znamená počet rovnic, ve kterých se opakující se kód vyskytuje. Proměnná P znamená počet znaků proměnné, která se použije pro weaving.

Ze vztahu lze pozorovat, že úspora velmi závisí na počtu znaků opakujícího se kódu a na počtu rovnic, které tento kód obsahují. Dále výrazně závisí na počtu znaků proměnné, která je použita pro weaving; proměnné ale obvykle mívají pouze několik znaků.

Na obrázku 8.1 lze pozorovat graf závislosti úspory na počtu operací obsahujících duplicitní kód, který lze vytknout do aspektu. Při vytváření grafu byl uvažován konstantní počet znaků

- nutných pro definování aspektu $S_a = 39$,
- společných konstrukcí pro oba přístupy $S_m = 80$,
- opakujícího se kódu $M = 10$,
- proměnné použité pro weaving $P = 1$.

Zvolené počty znaků byly vybrány tak, aby daly představu reálného průběhu funkce a reflektují hodnoty reálného, jednoduchého příkladu.



Obrázek 8.1. Graf závislosti úspory na počtu operací, obsahujících opakující se kód

Z grafu je patrné, že k úspoře dochází pouze ve specifických případech, kdy počet znaků duplicitního kódu převáží počet znaků syntaxe nutné pro definici aspektu.

Kapitola 9
Závěr

V rámci bakalářské práce byly prostudovány existující implementace aspektově orientovaných jazyků. Na základě těchto implementací byl vytvořen vlastní jazyk, který umožňuje definovat aspekty a splňuje definované požadavky.

Pro vytvořený jazyk byl v jazyce Haskell naimplementovaný aspect weaver. Kód weaveru byl rozdělený do modulů a funkcí tak, aby byla možná implementace dalších rozšíření. Pro aspect weaver byly vytvořeny jednotkové testy pokrývající všechny definované funkce.

Způsob použití vytvořeného systému byl demonstrovaný na modelovém příkladu, přičemž bylo ukázáno, že weaver vytváří korektní výstup.

Mezi výhody výsledného frameworku patří přehledná syntaxe aspektů, která připomíná kód standardního Maude. Framework dále umožňuje pomocí aspektů snížit počet duplicit v kódu.

Omezení frameworku spočívá ve způsobu, jakým je prováděn weaving. Protože jsou aspekty vplétané pomocí přepisování hodnot proměnných, jsou možnosti weavingu striktně limitované.

Literatura

[1] Bart De Win, Frank Piessens, Wouter Joosen a Tine Verhanneman. *On the importance of the separation-of-concerns principle in secure software engineering.* In: *Workshop on the Application of Engineering Principles to System Security Design.* 2002. 1–10.

[2] Niklas Pahlsson. Aspect-oriented programming. *Topic Report for Software Engineering.* 2002, 11–03.

[3] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm a William Griswold. Getting started with AspectJ. *Communications of the ACM.* 2001, 44 (10), 59–65.

[4] Noorazean Mohd Ali a Awais Rashid. *A state-based join point model for aop.* In: *Proceedings of the 1st ECOOP Workshop on Views, Aspects and Role (VAR'05), in 19th European Conference on Object-Oriented Programming (ECOOP'05), Glasgow, Scotland.* 2005.

[5] Tomáš Turek. *Využití aspektově-orientovaného přístupu pro tvorbu adaptivních uživatelskỳch rozhraní*. Diplomová práce, 2015.

[6] *Aspect-Oriented Programming in Prolog.* `https://bigzaphod.github.io/Whirl/dma/docs/aspects/aspects-man.html`. Navštíveno: 12. 3. 2017.

[7] *The AspectJ Project.* `https://eclipse.org/aspectj/`. Navštíveno: 12. 3. 2017.

[8] Kai Böllert. *On Weaving Aspects.* 1999.

[9] Michal Forgáč a Ján Kollár. *Static and dynamic approaches to weaving.* In: *Proceedings of 5th Slovakian-Hungarian Joint Symposium on Applied Machine Intelligence and Informatics Poprad, Slovakia.* 2007.

[10] Faisal Akkai, Atef Bader a Tzilla Elrad. *Dynamic weaving for building reconfigurable software systems.* In: *Proceedings of OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems.* 2001. 152–184.

[11] *The Maude System.* `http://maude.cs.illinois.edu`. Navštíveno: 17. 3. 2017.

[12] [Manuel Clavel ...]. *All about Maude - a high-performance logical framework*. Berlin u.a.: Springer, 2007. ISBN 9783540719403.

[13] Miran. Lipovača. *Learn you a Haskell for great good!* San Francisco, CA: No Starch Press, 2012. ISBN 978-1-59327-283-8.

[14] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm a William G Griswold. *An overview of AspectJ.* In: *European Conference on Object-Oriented Programming.* 2001. 327–354.

[15] *Aspect Oriented Programming with Spring.* `http://maude.cs.illinois.edu`. Navštíveno: 22. 3. 2017.

[16] *AspectFaces documentation*.
`http://wiki.codingcrayons.com/display/af`. Navštíveno: 22. 3. 2017.

[17] *The Haskell Tool Stack*.
`https://docs.haskellstack.org/`. Navštíveno: 10. 5. 2017.

[18] *The Glasgow Haskell Compiler*.
`https://www.haskell.org/ghc/`. Navštíveno: 10. 5. 2017.

[19] *HUnit - A unit testing framework for Haskell*.
`https://github.com/hspec/HUnit`. Navštíveno: 16. 5. 2017.

Příloha A
Seznam zkratek

ADT	algebraický datový typ
AF	AspectFaces
AOP	aspektově orientované programování
API	Application programming interface
CLI	Command-line interface
GHC	The Glasgow Haskell Compiler
JPM	join point model
JSF	JavaServer Faces
JVM	Java Virtual Machine

Příloha B
Obsah CD

- `thesis.pdf` — elektronická verze textu bakalářské práce
- `maudeaop-text.zip` — zdrojové soubory textu
- `maudeaop-source.zip` — zdrojové soubory systému
- `example.maude` — příklad použití aspektového přístupu v Maude