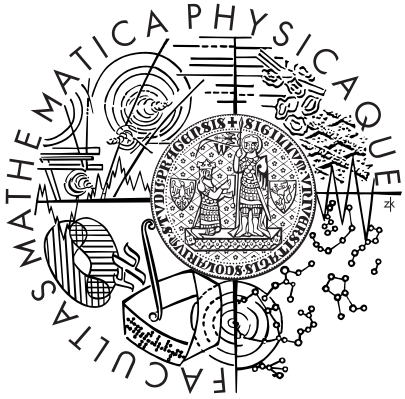
Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Vít Šefl

Komonády (nejen) pro programátory

Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: RNDr. Jan Hric

Studijní program: Informatika

Studijní obor: Programování

Praha 2014

Na tomto místě bych rád poděkoval všem lidem, kteří se podíleli na této práci. Především děkuji RNDr. Janu Hricovi za poskytnutou ochotu, připomínky ohledně struktury a obsahu a za pomoc při psaní této práce. Dále RNDr. Petru Pudlákovi a Prof. RNDr. Aleši Pultrovi DrSc. za další nápady a připomínky. Děkuji také své rodině a přátelům za podporu při psaní této práce.

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

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

V Praze dne 17.7.2014

Název práce: Komonády (nejen) pro programátory

Autor: Vít Šefl

Katedra / Ústav: Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: RNDr. Jan Hric

Abstrakt: Monády (a jejich kategorický duál – komonády) jsou důležitým konceptem teorie kategorií a zatímco monády jsou velmi oblíbenými nástroji ve funkcionálních jazycích (hlavně díky jazyku Haskell), komonády se příliš nepoužívají. V této práci prezentujeme definici komonád vhodnou pro potřeby funkcionálního programování a dále uvádíme příklady jejich praktického použití. Jedním z důležitějších příkladů je zipper – struktura, která reprezentuje určitou pozici. Ukážeme, že zipper lze automaticky odvodit pro každý regulární typ a také že tato operace připomíná derivaci z matematické analýzy. Kromě toho také uvádíme několik řešených příkladů v jazyce Haskell, které ilustrují, jak lze komonády použít pro řešení různých problémů. Všechny důkazy v teoretické části jsou provedeny v jazyce Agda a jsou zkontrolovány počítačem.

Klíčová slova: Haskell Agda komonáda zipper derivace

Title: Comonads (not only) for programmers

Author: Vít Šefl

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: RNDr. Jan Hric

Abstract: Monads (and their categorical dual – comonads) are important concepts in category theory and while monads enjoy their popularity in functional languages (mainly due to the programming language Haskell), comonads are often forgotten. In this work we present a definition of comonads suitable for programming and give examples of their use. One of the more important examples is zipper – a structure used to represent position. We show that zipper can be automatically derived for any regular type and show that this operation is very reminiscent of derivative from mathematical analysis. We also show worked examples of various problems that comonads can help solve in the language Haskell. All relevant proofs for the theoretical part of this work are machine checked in the language Agda.

Keywords: Haskell Agda comonad zipper derivative

Obsah

Úvod

Čistě funkcionální jazyky na rozdíl od dnešních procedurálních jazyků neumožňují neomezené vedlejší efekty (jako je např. I/O) uvnitř kódu a zároveň také omezují či naprosto vylučují mutaci datový struktur. Ačkoliv jsou kvůli těmto omezením vedlejší efekty náročnější na použití – speciálně musejí být explicitně označeny (obvykle prostřednictvím typů) – ve výsledku máme mnohem lepší kontrolu nad tím, kde a v jaké míře se tyto efekty mohou vyskytovat.

Právě snaha o toto rozlišení vedla ke vzniku monád, tak jak je známe z funkcionálního programování. Tento koncept byl převzat v upravené podobě z teorie kategorií. Monády nám umožňují popsat sekvenční výpočet a všechny jeho možné vedlejší efekty. K monádám existuje duální koncept – tzv. komonády. Ačkoliv jsou komonády používané v teorii kategorií, v praktickém programování se objevují pouze ojediněle. V této práci se mimo jiné podíváme právě na definici a praktickou aplikaci komonád.

Datové struktury ve funkcionálním programování taktéž mají různá omezení oproti svým protějškům v procedurálních jazycích. Stejně jako u monád nám tato omezení poskytují jisté výhody, ale na druhou stranu znemožňují mnoho postupů známých právě z procedurálních jazyků. Pěkným příkladem tohoto problému je možnost vést si ukazatel dovnitř struktury a libovolně jej měnit; tím snadno dosáhneme efektivního pohybu po celé struktuře. Jelikož jsou datové struktury ve funkcionálním programováním neměnné (*immutable*), může se zdát, že tento postup nemá svůj ekvivalent ve funkcionálním světě.

Tento problém vyřešil koncept známý pod jménem *zipper*, který nám umožňuje reprezentovat původní struktury vhledem k jednomu vybranému prvku a jeho okolí. Složitost konstrukce zipperu odpovídá přibližně složitosti struktury, pro kterou tento zipper konstruujeme. Nabízí se otázka, jestli není možné tento postup automatizovat – díky čemuž bychom snadno mohli vytvářet zippery i pro ty nejsložitější struktury. Také se podíváme, jaká je souvislost mezi zippery a komonádami.

Struktura této práce je následující:

V první kapitole se věnujeme úvodu do programovacího jazyka *Agda*, ve kterém jsou formalizována a dokázána tvrzení o vlastnostech funkcí, které definujeme a používáme v této práci. Nejprve zmíníme souvislost s matematickou logikou (speciálně s konstruktivní variantou). Dále ukážeme, co jsou to závislé typy a jak je lze použít. Nakonec dokážeme jednoduché tvrzení a zmíníme se o obecném použití Agdy – ať jako nástroj pro dokazování nebo jako praktický programovací jazyk.

Druhá kapitola popisuje regulární datové typy. Podíváme se na jejich použití v jazyce *Haskell* a jak se toto použití vztahuje k jejich teoretické definici. Dále se podíváme, jak nám tato definice může pomoci v každodenním programování, a

ukážeme ukážeme další zajímavé teoretické vlastnosti.

Třetí kapitola definujeme pojem komonády jakožto duálu monády. Kromě definice vhodné pro funkcionální jazyky zde nalezneme alternativní prezentaci založenou na teorii kategorií. Tyto dvě definice porovnáme a ukážeme, že to jsou skutečně jen dva pohledy na stejnou strukturu. Tuto kapitolu zakončíme přehledem běžně používaných datových typů, na kterých lze definovat komonadické operace.

Čtvrtá kapitola popisuje zipper. Nejprve probereme, proč bychom zipper vůbec měli používat. Poté následuje detailní popis této struktury spolu požadovanými operacemi. Na konci kapitoly se věnujeme souvislosti s komonádami.

V páté kapitole definujeme analogii derivace z matematické analýzy pro datové typy a ukážeme, že výsledek lze interpretovat jako kontext, který určuje pozici jednoho prvku uvnitř dané struktury. Dále ukážeme, jak pomocí tohoto kontextu můžeme zkonstruovat zipper pro libovolný regulární typ a definovat vybrané operace.

Poslední šestá kapitola je soubor příkladů použití konceptů probraných v předchozích kapitolách. Tato kapitole je tvořena čtyřmi řešenými příklady, ve kterých využijeme jak zipper, tak i komonády: simulátor Turingova stroje, rozostření obrazu jakožto obrazový filtr, simulátor dvourozměrného celulárního automatu a interpret jednoduchého dotazovacího jazyka pro pohyb n-árním stromem.

1 Agda

Agda [1][2] je čistě funkcionální programovací jazyk se závislými typy (*dependent types*), který lze díky *Curry-Howardovo isomorfismu* [3] také použít pro dokazování tvrzení v konstruktivní logice. Proto se také Agda označuje jako tzv. *proof assitant*, což je obecně software, ve kterém lze vytvářet formální důkazy za spolupráce počítače (jiný takovýto systém je např. *Coq*). Základem je intuicionistická teorie typů (*intuitionistic type theory*), která nabízí alternativní základ matematiky. Samotný jazyk svou syntaxí připomíná *Haskell* [4]. Důkazy tvrzení, která jsou uvedena v této práci, jsou provedeny právě v jazyce Agda.

1.1 Závislé typy

V programovacích jazycích obvykle nacházíme hodnoty, které závisejí na jiných hodnotách (např. výsledek aplikace funkce); hodnoty, které závisejí na typech (polymorfismus ve funkcionálních jazycích, *generics*). Občas také máme typy, které zavisejí na jiných typech (typové funkce, např. pragma *TypeFamilies* v Haskellu). Závislé typy odkazují na poslední druh takovéto závislosti – typy, které závisejí na hodnotách.

Příkladem této závislosti jsou například seznamy, jejichž délka je součástí typu. Nové typy se v Agdě deklarují podobně jako *GADT* v Haskellu (tj. vyjmenujeme všechny konstruktory a každému explicitně definujeme typ).

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ

data Vec (A : Set) : ℕ → Set where
  nil  : Vec A zero
  cons : {n : ℕ} → A → Vec A n → Vec A (suc n)
```

Všimněme si, že `nil` vytváří pouze seznamy, jejichž délka je nula (`zero`) a pokud použijeme `cons` na seznam, jehož délka je n, výsledný seznam má délku $n + 1$ (`suc n`). Pokud bychom chtěli napsat funkci, která pro daný prvek vytvoří seznam obsahující n kopií tohoto prvku, je nutné vyjádřit, že délka výsledného seznamu závisí právě na n.

```
replicate : (n : ℕ) {A : Set} → A → Vec A n
replicate zero    x = nil
replicate (suc n) x = cons x (replicate n x)
```

Seznam `replicate 4 0` tedy bude mít typ `Vec ℕ 4`.

1.2 Důkazy

Curry-Howardův isomorfismus říká, že typy v čistě funkcionálním jazyce odpovídají tvrzením v určitém logickém systému. Hodnoty daného typu pak odpovídají důkazu tvrzení daného tímto typem. Typ funkcí například odpovídá implikaci, aplikace funkce pak pravidlu modus ponens.

Na tomto principu je založeno dokazování v Agdě. Intuicionistická teorie typů odpovídá konstruktivní logice vyššího řádu – pokud se nám tedy podaří vyjádřit dané tvrzení prostřednictvím typů a nalézt hodnotu daného typu, díky isomorfismu jsme dokázali i původní tvrzení; samotná hodnota je pak důkazem tohoto tvrzení.

Podívejme se, jak bychom dokázali asociativitu sčítání přirozených čísel. Nejprve potřebujeme přirozená čísla, stejně jako v předchozí podkapitole použijeme unární reprezentaci:

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

Nyní definujeme sčítání. Agda umožňuje definovat libovolné operátory – podtržítko označuje, kde se daný argument nachází; prioritu a asociativitu můžeme obdobně jako v Haskellu definovat klíčovými slovy `infix`, `infixr` a `infixl`.

```
infixl 6 _+_

_+_ : ℕ → ℕ → ℕ
zero  + n = n
suc m + n = suc (m + n)
```

Zbývá definovat, kdy jsou si dvě hodnoty rovny.

```
infix 4 _≡_

data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

Takto definovaná rovnost splňuje všechny důležité vlastnosti – reflexivita, symetrie, tranzitivita, atp. Pro naše účely budeme potřebovat kongruenci, tj. pokud $x = y$ pak také $f(x) = f(y)$.

```
cong : {A B : Set}
       (f : A → B) {x y : A} → x ≡ y → f x ≡ f y
cong f refl = refl
```

Nyní již máme vše potřebné, abychom formulovali a dokázali tvrzení, že takto definované sčítání je asociativní.

```
+-assoc : (m n o : ℕ) → (m + n) + o ≡ m + (n + o)
+-assoc zero    _ _ = refl
+-assoc (suc k) n o = cong suc (+-assoc k n o)
```

Pokud $m = 0$, dostáváme z definice sčítání $n + o = n + o$, což je triviálně splněno. Naopak, pokud je m následníkem čísla k, přistoupíme k indukci. Typ indukčního předpokladu `+-assoc k n o` je $(k + n) + o = k + (n + o)$. Přičteme-li 1 k obou stranám této rovnice (tj. aplikujeme `suc` pomocí `cong`), dostaneme požadovaný typ.

Všimněme si, že funkce `+-assoc` je rekurzivní. Neomezená rekurze vede k nekonzistentní logice, příkladem je funkce `fix`, pomocí které můžeme dokázat libovolné tvrzení.

```
fix : {A : Set} → (A → A) → A
fix f = f (fix f)

bad : zero ≡ suc zero
bad = fix λ x → x
```

Kvůli tomu je součástí kontroly správnosti programu také tzv. *termination checker*, který kontroluje, jestli rekurzivní funkce vždy vrátí svůj výsledek v konečném čase. *Halting problem* nám ale říká, že toto není možné provést v obecnosti pro každou funkci – termination checker je tedy konzervativní: existují funkce, jejichž výpočet je konečný pro libovolné argumenty, ale přesto se přes tento test nedostanou.

1.3 Použití

Chceme-li k řešení daného problému využít externí knihovnu, obvykle je veškerá interakce s touto knihovnou definovaná pomocí *API*, tj. sady funkcí (metod, tříd, atp. – v závislosti na programovacím jazyce), která zpřístupňuje veškerou funkčnost dané knihovny. Od API očekáváme platnost určitých vlastností, které ale obvykle nemohou být vynucovány vlastním jazykem a proto jsou uvedeny pouze v dokumentaci dané knihovny.

Ideálním příkladem takovéhoto API, jehož vlastnosti nejsou na úrovni jazyka vynucovány, jsou typové třídy standardní knihovny Haskellu. U spousty typových tříd se předpokládá platnost určitých vlastností (např. typová třída `Eq` by měla definovat ekvivalenci, typová třída `Ord` by měla definovat lineární uspořádání, atp.), programátor ale vždy může napsat instanci této třídy, která tyto vlastnosti nesplňuje.

Agda nám umožňuje jít o krok dále a vynutit platnost těchto vlastností přímo na úrovni jazyka. V této práci toho využijeme při definici komonád (kapitola 3), kde kód v Agdě nejen poskytuje definici samotných operací, ale také důkazy, že dané vlastnosti skutečně platí. V textu této práce jsou slovně či v pseudokódu uvedeny dané vlastnosti a pak následuje odkaz na přílohu, ve které jsou tyto vlastnosti formálně popsány a dokázány.

2 Regulární datové typy

2.1 Regulární datové typy v Haskellu

Pokud se podíváme na definici datových typů v Haskellu, všimneme si, že v zásadě umožňují tři způsoby konstrukce: výběr mezi alternativami, agregaci více položek a rekurzi.

Výběr mezi alternativami (tzv. koprodukt) je reprezentován svislou čarou (`|`), jako například u datového typu `Bool`:

```
data Bool = False | True
```

Agregace (tzv. produkt), tedy v určitém smyslu možnost vytvářet n-tice dalších datových typů, je jednoduchá juxtapozice:

```
data BoolPair = Pair Bool Bool
```

Po koproduktu i produktu je rozumné vyžadovat existenci neutrálního prvku. Pro koprodukt by to měl být výběr mezi nula alternativami, tedy žádná alternativa. Standardy *Haskell 98* [5] ani *Haskell 2010* [4] však tuto možnost nepřipouštějí, ale nejrozšířenější implementace, GHC, podporuje skutečně prázdné datové typy přes pragma *EmptyDataDecls*. Můžeme tedy definovat:

```
{-# LANGUAGE EmptyDataDecls #-}

data Empty
```

Pro produkt by naopak neutrálním prvkem měl být datový typ, který má právě jednu alternativu bez jediné položky (má tedy právě jednu hodnotu). Takovýto datový typ je již součástí modulu `Prelude` pod jménem `()`, pokud by bylo možné tento typ definovat v Haskellu, vypadala by definice následovně:

```
data () = ()
```

Typy vytvořené těmito způsoby jsou ale relativně nezajímavé. Vždy se jedná o m alternativ, každá s n_i položkami. To, co nám umožňuje vytvářet zajímavé a velice užitečné datové typy, je rekurze. Základem je rekurzivní výskyt vlastního datového typu v jedné z alternativ, díky tomu odpovídá struktura části datového typu struktuře celku, což na rozdíl od dříve uvedených způsobů umožňuje vytvářet libovolně velké hodnoty (podle vlastního způsobu utváření vždy konečné nebo potenciálně nekonečné).

```
data IntList = End | More Int IntList
```

Kromě toho máme také k dispozici typové proměnné, které umožňují vytvářet polymorfní typy. Např. místo seznamů znaků, čísel, řetězců, atp. budeme mít jeden seznam hodnot typu *a* a pokud potřebujeme seznam prvků konkrétního typu, stačí za *a* vhodně dosadit.

```
data List a = End | More a (List a)

type IntList  = List Int
type CharList = List Char
```

Než se přesuneme k samotné definici regulárního datového typu, podívejme se na příklady neregulárních datových typů. Obecně je datový typ neregulární, pokud definice obsahuje něco jiného než koprodukt, produkt nebo rekurzi (se stejným typovým argumentem). Použití funkčních typů je obvyklé pro neregulární datové typy:

```
data T a = T (a -> T a)
```

Dalším příkladem je netriviální rekurze:

```
data Perfect a = Stop a | Go (Perfect (a, a))
```

Všimněme si, že rekurzivní výskyt má jiný typ parametru. S tímto typem rekurze se obvykle nesetkáváme, ale může být velice užitečný. Příkladem je dříve uvedený typ `Perfect a`, který reprezentuje perfektní binární stromy (tj. binární stromy, jejichž všechny listy jsou na stejné úrovni), nebo také *Finger Tree* (což je datová struktura, která stojí za kolekcí `Data.Sequence.Seq` z knihovny *containers*).

Na závěr poznamenejme, že pokud je rekurzivní výskyt součástí jiného regulárního typu, je výsledný typ taktéž regulární. Obvyklým případem jsou stromy s libovolným počtem podstromů; regularitu tohoto typu využijeme v dalších kapitolách.

```
data Tree a = Node a [Tree a]
```

2.2 Definice

Pokud bychom neuvažovali rekurzi ani typové proměnné, pak lze regulární datové typy definovat velice snadno induktivně:

- 0 je regulární typ.
- 1 je regulární typ.
- Pokud jsou T a S regulární typy, pak také $T + S$ je regulární.
- Pokud jsou T a S regulární typy, pak také $T \times S$ je regulární.

0 je dříve uvedený datový typ `Empty`, 1 je `()` (za chvíli bude jasné, proč jsme zvolili takováto jména), + je koprodukt dvou typů a × je produkt dvou typů. Na × se také můžeme dívat jako na kartézský součin. Pokud potřebujeme koprodukt více než

dvou typů, stačí použít + vícekrát, obdobně pak pro produkt a ×. Odpovídající typy v Haskellu jsou `Either` pro + a `(,)` pro × (typový konstruktor dvojic).

Kromě toho, že pomocí této definice nemůžeme reprezentovat typové proměnné, nemůžeme také reprezentovat rekurzi. Pokud budeme chtít vytvořit rekurzivní typ, bude nutné označit, ve kterých místech se rekurze nachází. Překvapivě se pro to také hodí typové proměnné: jednu z proměnných použijeme na označení rekurzivního výskytu.

Pro samotnou definici proměnných se odkážeme na matematickou logiku. Předpokládáme tedy spočetnou množinu jmen; každá proměnná je pak jedním prvkem z této množiny (jednoduše si tuto množinu můžeme představit jako datový typ `String`). Stejně tak máme volné a vázané proměnné – vázané proměnné ale na rozdíl od logických formulí vznikají použitím μ nebo ν (které slouží právě k zavedení rekurzivních typů).

Také si explicitně vedeme seznam volných proměnných (dále pouze kontext), který je obvykle reprezentován symbolem Γ. Pro úplnost dodejme, že Γ, x znamená kontext Γ spolu s novou proměnnou x. Matematická logika proměnné obvykle takto neřeší, ale pro naše účely je důležité znát, přes které proměnné je datový typ parametrizován.

Poznamenejme, že následující definice je uvedena ve stylu přirozené dedukce – vše nad vodorovnou čarou jsou předpoklady, vše pod čarou jsou důsledky. Vzhledem k tomu, že také používáme volné proměnné, jsou všechna pravidla opatřena kontextem Γ, který zaručuje, že výsledný typ neobsahuje volnou proměnnou, která se v kontextu nevyskytuje.

$$\frac{}{\Gamma \vdash 0 \text{ reg}}\ (0\text{-reg}) \qquad \frac{}{\Gamma \vdash 1 \text{ reg}}\ (1\text{-reg})$$

$$\frac{\Gamma \vdash T \text{ reg} \quad \Gamma \vdash S \text{ reg}}{\Gamma \vdash T + S \text{ reg}}\ (+\text{-reg}) \qquad \frac{\Gamma \vdash T \text{ reg} \quad \Gamma \vdash S \text{ reg}}{\Gamma \vdash T \times S \text{ reg}}\ (\times\text{-reg})$$

$$\frac{\Gamma, x \vdash T \text{ reg}}{\Gamma \vdash \mu x.T \text{ reg}}\ (\mu\text{-reg}) \qquad \frac{\Gamma, x \vdash T \text{ reg}}{\Gamma \vdash \nu x.T \text{ reg}}\ (\nu\text{-reg})$$

$$\frac{}{\Gamma, x \vdash x \text{ reg}}\ (\text{var-reg})$$

Pokud se podíváme na kontexty jednotlivých pravidel, můžeme si všimnout, že μ a ν se skutečně chovají jako kvantifikátory. Pokud je původní kontext Γ, x a aplikujeme μ nebo ν, pak se ve výsledném kontextu již volná proměnná x nevyskytuje.

Pravidla 0-reg, 1-reg, +-reg a ×-reg jsou pouze přepsané induktivní kroky z definice na začátku této podkapitoly. Pravidlo var-reg je nám umožňuje do regulárního typu

přidávat proměnné: pokud je v kontextu volná proměnná x, tak tuto proměnnou můžeme využít na vytvoření regulárního typu. Zbylá dvě pravidla, μ-reg a ν-reg, ošetřují rekurzi. Pokud máme regulární typ v nějakém kontextu Γ, který navíc obsahuje proměnnou x, tak můžeme tuto proměnnou využít pro označení rekurzivních výskytů. Dříve uvedený typ `List a` by například vypadal takto:

$$\text{List } a = \mu x.\ 1 + a \times x \ (= 1 + a \times \text{List } a)$$

Jedná se skutečně o regulární datový typ, o čemž svědčí toto odvození:

$$\dfrac{\dfrac{}{a, x \vdash 1 \text{ reg}} \quad \dfrac{\dfrac{}{a, x \vdash a \text{ reg}} \quad \dfrac{}{a, x \vdash x \text{ reg}}}{a, x \vdash a \times x \text{ reg}}}{\dfrac{a, x \vdash 1 + a \times x \text{ reg}}{a \vdash \mu x.1 + a \times x \text{ reg}}}$$

Zbývá vysvětlit, proč máme dva způsoby zavedení rekurze – μ a ν. Základní idea je taková, že μ vytváří rekurzivní typy, jejichž hodnoty mohou být sice neomezeně velké, ale vždy konečné. Tedy například typ `List a` definovaný pomocí μ obsahuje všechny konečné seznamy prvků typu `a`, ale nekonečný seznam jím reprezentovat nelze. Na druhou stranu ν vytváří rekurzivní typy, které mohou být nekonečné. Nejlépe je tento rozdíl vidět na následujícím datovém typu:

```
data Silly = S Silly
```

Pokud použijeme μ, tedy `Silly` = $\mu x.\ x$, pak díky tomu, že pomocí μ lze vytvořit pouze konečně velké hodnoty, je tento typ efektivně ekvivalentní typu `Empty`. Jediná možná hodnota vypadá takto:

```
silly = S silly
```

Ale tato hodnota je jistě nekonečně velká. Na druhou stranu ν, tedy `Silly` = $\nu x.\ x$, vytvoří datový typ, jehož hodnoty jsou potenciálně nekonečně velké. Díky tomu je takto definovaný `Silly` ekvivalentní typu `()`, jediná hodnota je totiž dříve uvedená `silly`.

V Haskellu toto rozlišení neexistuje, rekurzivní typy mohou být vždy nekonečné a navíc díky existenci neomezené rekurze existují hodnoty, které neodpovídají žádnému konstruktoru daného typu (např. mimo hodnotu `silly` obsahuje typ `Silly` také hodnotu `let x = x in x`).

Pokud se omezíme na programy, které tuto neomezenou rekurzi nevyužívají, rozlišení mezi ν a μ může být velice užitečné; rozumná disciplína umožňuje předcházet zacyklení programu vzniklé v důsledku použití rekurze na potenciálně nekonečné struktury (typickým příkladem je použití funkce `foldl` na nekonečně velké seznamy). V jiných programovacích jazycích a obzvláště v dokazovacích

asistentech (*proof assistant*, *theorem prover*) je toto rozlišení klíčové – umožňuje nám pracovat s nekonečnými strukturami, aniž bychom přišli o logickou konzistenci.

Závěrem poznamenejme, že od proměnných a kontextů očekáváme obvyklé vlastnosti:

- Dva regulární datové typy jsou shodné pokud jsou shodné až na přejmenování proměnných.
- Do kontextu vždy můžeme přidávat nové volné proměnné (viz pravidlo weaken níže).
- Proměnné v kontextu můžeme libovolně permutovat.
- Za proměnnou můžeme substituovat jiný regulární typ (ale pouze pokud by se z původně volné proměnné nestala vázaná – což se vždy dá snadno ošetřit vhodným přejmenováním).

Substituce a přidávání nových proměnných jsou stručně shrnuty těmito pravidly:

$$\frac{\Gamma \vdash T \text{ reg}}{\Gamma, x \vdash T \text{ reg}} \text{ (weaken)} \qquad \frac{\Gamma \vdash T \text{ reg} \quad \Gamma, x \vdash S \text{ reg}}{\Gamma \vdash [T/x]S \text{ reg}} \text{ (subst)}$$

Pro substituci uvádíme jednoduchý příklad:

$$\begin{aligned} T &= a + \mu a.\, 1 + b \times a \\ [\text{List } x/a]T &= \text{List } x + \mu a.\, 1 + b \times a \\ [\text{List } x/b]T &= a + \mu a.\, 1 + \text{List } x \times a \end{aligned}$$

Pro regulární typy také můžeme definovat konstruktory.

$$\begin{aligned} \langle\rangle &: 1 \\ \langle \_,\_ \rangle &: T \to S \to T \times S \\ \text{inl} &: T \to T + S \\ \text{inr} &: S \to T + S \\ \text{rec} &: [\mu x.\, T/x]T \to \mu x.\, T \\ \text{crec} &: [\nu x.\, T/x]T \to \nu x.\, T \end{aligned}$$

Pokud máme např. hodnoty x_1, x_2 typu a, tak z nich můžeme vytvořit seznam o dvou prvcích typu List a:

$$\begin{aligned} \text{nil} &= \text{rec}\,(\text{inl}\,\langle\rangle) \\ \text{cons}\,x\,xs &= \text{rec}\,(\text{inr}\,\langle x, xs\rangle) \\ \text{list} &= \text{cons}\,x_1\,(\text{cons}\,x_2\,\text{nil}) \end{aligned}$$

2.3 Komutativní polookruh typů

Některé regulární typy obsahují části, které se zdají být zbytečné. Produkt s typem 1 nepřidává nic nového, dvojice $1 \times T$ bude vždy obsahovat triviální hodnotu typu 1 a pak nějakou hodnotu typu T. Stejnou informaci ale lze reprezentovat samotným typem T. Stejně tak $0 + T$ nepřidává nic nového. Nabízí se otázka, jestli neexistují pravidla, která by řešila, kdy můžeme nahradit typ T za jiný typ S, aniž bychom ztratili informaci obsaženou v původním typu.

Část těchto pravidel se dá shrnout do následujícího tvrzení: množina všech typů spolu s konstantami 0, 1 a operacemi +, × tvoří komutativní polookruh.

Problém nastává s rovností – konkrétně není úplně jasné, kdy jsou si dva typy rovny. Například typy `Bool` a `Either () ()` reprezentují stejnou informaci (oba typy mají právě dvě hodnoty), ale přesto se jedná o různé typy. Místo toho řekneme, že dva typy jsou si rovny, pokud jsou *isomorfní*. Dva typy A a B jsou isomorfní, pokud existují zobrazení splňující:

$$\begin{aligned} \text{from} &: A \to B \\ \text{to} &: B \to A \\ \\ \text{id} &= \text{to} \circ \text{from} \\ \text{id} &= \text{from} \circ \text{to} \end{aligned}$$

Pokud je naší rovností isomorfismus, pak jsou si typy `Bool` a `Either () ()` skutečně rovny:

```
to :: Bool -> Either () ()
to True  = Left  ()
to False = Right ()

from :: Either () () -> Bool
from (Left  ()) = True
from (Right ()) = False
```

Matematické struktury, jako je například grupa, jsou obvykle definovány nad rovností tak jak ji známe z teorie množin. Není tedy nutné od této rovnosti explicitně vyžadovat určité vlastnosti (např. že se jedná o ekvivalenci slučitelnou s operacemi dané struktury), protože jsou automaticky splněny. V našem případě ale používáme jinou rovnost – isomorfismus. Musíme tedy mírně upravit obvyklé definice těchto struktur, abychom tyto vlastnosti pokryli.

Než uvedeme samotné definice, tak jen poznamenejme, že vlastnostmi jako je např. asociativita se implicitně míní asociativita vzhledem k námi definované rovnosti.

Pologrupa S je trojice $(A, =, \times)$, kde A je množina, $=$ je ekvivalence nad A (tj. relace, která je reflexivní, tranzitivní a symetrická) a $\times$ je asociativní binární operace nad A, která respektuje ekvivalenci $=$ a je asociativní.

Pomocí pologrupy můžeme definovat komutativní monoid. Komutativní monoid M je čtveřice $(A, =, \times, 1)$, kde opět A je množina, $=$ je binární relace nad A, $\times$ je binární operace nad A a 1 je konstanta z množiny A. Zároveň musí platit, že $(A, =, \times)$ je pologrupa, 1 je (levá) identita vzhledem k $\times$ (tj. $1 \times x = x$, pro každé x) a $\times$ je navíc komutativní.

Nakonec můžeme definovat komutativní polookruh R jako šestici $(A, =, \times, +, 1, 0)$, kde A je množina, $=$ je binární relace, $\times$ a $+$ jsou binární operace a 1 a 0 jsou konstanty. Navíc platí, že $(A, =, \times, 1)$ a $(A, =, +, 0)$ jsou komutativní monoidy, distributivita násobení zprava (tj. $(y + z) \times x = (y \times x) + (z \times x)$) a nulování zleva (tj. $0 \times x = 0$). Díky komutativitě na pořadí násobení, resp. sčítání, nezáleží, bylo by tedy možné vzít ekvivalentně distributivitu zleva a nulování zprava.

Když už tedy víme, jak je nutné pozměnit tyto struktury, abychom si byli jisti, že operace korektně respektují ekvivalenci, můžeme uvést naše tvrzení. Množina všech typů, spolu s isomorfismem, operacemi $+$ a $\times$ a konstantami 0 a 1, tvoří komutativní polookruh.

Důkaz tohoto tvrzení je proveden v programovacím jazyce Agda a je k nalezení v příloze B.1.

Pro využití tohoto tvrzení nemusíme chodit daleko. V následujících kapitolách zavedeme pojem derivace regulárního datového typu. Po aplikaci derivace ale často dostáváme sice jednoduché, ale značně rozsáhlé typy, které obsahují mnoho zbytečných částí. Derivujme např. typ $a \times b$ podle proměnné a:

$$\begin{aligned} \partial_a(a \times b) &= a \times \partial_a b + \partial_a a \times b \\ &= a \times 0 + 1 \times b \\ &= 0 + b \\ &= b \end{aligned}$$

Díky tomu, že typy tvoří komutativní polookruh, můžeme aplikovat jednoduché algebraické transformace a výsledný typ značně zjednodušit.

3 Komonády

3.1 Definice

Komonáda je struktura z teorie kategorií, ale pokud bychom zde chtěli uvést plnou definici, museli bychom zavést většinu základů teorie kategorií. Soustřeďme se tedy na konkrétní definici pro praktické programování (a to konkrétně ve stylu vhodném pro funkcionální programování). Pro samotnou definici použijeme typové třídy; další vlastnosti, které nelze pomocí typových deklarací vynutit, pak uvádíme separátně.

Abychom mohli o nějakém (unárním) typovém konstruktoru prohlásit, že se jedná o komonádu, musíme nejprve ověřit, že se jedná o `Functor`. Pro připomenutí, zde je definice funktoru spolu s očekávanými vlastnostmi:

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


-- fmap musí splňovat:

1) fmap id      == id
2) fmap (f . g) == fmap f . fmap g
```

Typovou třídu `Comonad` [6] lze definovat více způsoby; zde použijeme definici, která vychází z teorie kategorií, a v další podkapitole uvedeme alternativní prezentaci.

```
class (Functor f) => Comonad f where
    extract   :: f a ->      a
    duplicate :: f a -> f (f a)

-- fmap, extract a duplicate musejí splňovat

1) extract      . fmap f    ==            f   . extract
2) duplicate    . fmap f    == fmap (fmap f)  . duplicate
3)      extract . duplicate == id
4) fmap extract . duplicate == id
5) duplicate    . duplicate == fmap duplicate . duplicate
```

Při srovnání s definicí monády není vidět žádná přímá souvislost. To je způsobeno tím, že obvyklá definice typové třídy `Monad` se mírně odklání od definice z teorie kategorií. Pokud bychom monády zavedli podobně, dostali bychom:

```
class (Functor f) => Monad f where
    return ::      a  -> f a
    join   :: f (f a) -> f a
```

Nyní je souvislost zřejmá: pokud vezmeme operace definující monádu a zaměníme jejich domény za kodomény (obory hodnot), dostáváme právě komonádu. To je také

důvod, proč jsou operace komonády občas nazývány jako `coreturn` a `cojoin` místo `extract` a `duplicate`.

Z typů lze vidět, co nám monády a komonády poskytují a v čem se liší.

Monády se obvykle používají na popis výpočtů s jakýmsi vedlejším efektem (ať je to měnitelný stav, nedeterminismus nebo I/O). Z konkrétní hodnoty je vždy možné vytvořit výpočet bez vedlejšího efektu, jehož výsledkem je právě tato hodnota. Na druhou stranu nemůžeme (obecně) z výpočtu vyextrahovat výsledek, tj. nelze definovat funkci typu `f a -> a`, která by vyextrahovala výsledek a vedlejší efekt by jednoduše zahodila. Pro konkrétní monády tyto funkce existovat mohou, ale funkce, která v obecnosti funguje pro všechny monády, neexistuje.

Podíváme-li se na komonády, všimneme si, že dříve zmíněná funkce je naopak součástí definice. Na druhou stranu nemáme možnost přenést obyčejnou hodnotu do kontextu komonády, nelze tedy napsat funkci typu `a -> f a`. Intuice je taková, že zatímco monády reprezentují jakési vedlejší efekty a vždy máme možnost vytvořit prázdný efekt, tak komonády reprezentují hodnoty v kontextu, který můžeme kdykoliv zahodit. Pokud se například podíváme na prvek spolu s jeho pozicí v seznamu (což je náš kontext), tak neexistuje žádný prázdný kontext. I pozice odpovídající prvnímu prvku v seznamu je netriviální kontext.

`join` a `duplicate` slouží ke spojování výpočtů. Pokud máme funkci, která generuje vedlejší efekty (tj. `a -> f b`), můžeme ji aplikovat uvnitř monády (pomocí `fmap`) a pak pomocí `join` sloučit původní a nové vedlejší efekty (tj. z `f (f a)` na `f a`). Na straně komonád máme naopak funkce, které na základě kontextu spočítají jednu hodnotu (tj. `f a -> b`). Původní kontext zkopírujeme pomocí `duplicate` a pak na něj aplikujeme tuto funkci (opět pomocí `fmap`).

3.2 Alternativní prezentace

V předchozí podkapitole jsme zmínili alternativní definici monády pomocí operací `return` a `join` (a také `fmap` z typové třídy `Functor`). Na první pohled je tato definice velice odlišná od standardní definice:

```
class Monad f where
    return :: a -> f a
    (>>=)  :: f a -> (a -> f b) -> f b
```

Nicméně operace `(>>=)` lze definovat pomocí operací `join` a `fmap`:

```
(>>=) :: (Monad f) => f a -> (a -> f b) -> f b
m >>= f = join (fmap f m)
```

Stejně tak můžeme definovat `join` a `fmap` pomocí `(>>=)` a `return`:

```
fmap :: (Monad f) => (a -> b) -> f a -> f b
fmap f m = m >>= return . f

join :: (Monad f) => f (f a) -> f a
join m = m >>= id
```

U komonád je situace obdobná. Podívejme se tedy na alternativní definici:

```
class Comonad f where
    extract ::  f a -> a
    extend  :: (f a -> b) -> f a -> f b

-- extract a extend musejí splňovat

1) extend extract      == id
2) extract  . extend f == f
3) extend f . extend g == extend (f . extend g)
```

`extend` lze definovat pomocí operací `fmap` a `duplicate` následovně:

```
extend :: (Comonad f) => (f a -> b) -> f a -> f b
extend f w = fmap f (duplicate w)
```

A na druhou stranu, `fmap` a `duplicate` definujeme pomocí `extend` a `extract`:

```
fmap :: (Comonad f) => (a -> b) -> f a -> f b
fmap f w = extend (f . extract) w

duplicate :: (Comonad f) => f a -> f (f a)
duplicate = extend id
```

Zbývá ověřit, že tyto operace jsou definované korektně, tedy že splňují požadované vlastnosti uvedené u obou definic. Tento důkaz je proveden v programovacím jazyce Agda a je k nalezení v příloze B.2 (definice) a B.3 (důkaz).

Nyní můžeme uvést finální definici, která umožňuje programátorovi definovat komonádu oběma způsoby.

```
instance (Functor f) => Comonad f where
    extract   ::  f a -> a
    extend    :: (f a -> b) -> f a -> f b
    duplicate ::  f (f a) -> f a

    extend f  = fmap f . duplicate
    duplicate = extend id
```

Pokud se rozhodneme implementovat všechny operace (např. kvůli efektivitě), tak bychom se měli ujistit, že si tyto operace odpovídají, tedy:

```
1) fmap f    == extend (f . extract)
2) extend f  == fmap f . duplicate
3) duplicate == extend id
```

Stejně jako pro monády lze i pro komonády definovat mnoho pomocných funkcí, pro nás budou ale důležité hlavně protějšky operátoru `<=<` (resp. `>=>`) a typu `Kleisli`. Funkce typu `Kleisli m a b` (= `a -> m b` pro monádu `m`) jsou instancí typové třídy `Category` (která je založená na definici kategorie z teorie kategorií), `return` je identita a `<=<` je skládání. Pro komonády lze definovat obdobný operátor `=<=` (resp. `=>=`). Funkce typu `Cokleisli w a b` (= `w a -> b` pro komonádu `w`) jsou

taktéž instancí typové třídy `Category`, `extract` je identita a `=<=` je skládání:

```
(=<=) :: (Comonad w) => (w b -> c) -> (w a -> b) -> (w a -> c)
f =<= g = f . extend g

newtype Cokleisli w a b = C (w a -> b)

instance (Comonad w) => Category (Cokleisli w) where
    id        = C extract
    C f . C g = C (f =<= g)
```

Typová třída `Category` také klade na tyto operace určité požadavky.

```
1)     id . f   ==  f
2)      f . id  ==  f
3) f . (g . h)  == (f . g) . h
```

Tyto požadavky jsou ověřeny v příloze B.4.

3.3 Konkrétní komonády

V několika dalších podkapitolách uvedeme příklady konkrétních komonád. Všechny definice a důkazy vlastností jsou k nalezení v příloze B.5.

3.3.1 Identita

Identita je příkladem triviální komonády. Jedná se pouze o hodnotu daného typu v prázdném kontextu. Stejně jako její protějšek, monáda `Identity`, má především význam jako neutrální prvek pro komonadické transformátory. Definice je skutečně triviální:

```
newtype Identity a = Identity a

instance Functor Identity where
    fmap f (Identity a) = Identity (f a)

instance Comonad Identity where
    extract   (Identity a) = a
    duplicate (Identity a) = Identity (Identity a)
```

Komonadické transformátory v této práci sice nepopisujeme, ale použití `Identity` je prakticky stejné jako u monadických transformátorů: pokud máme transformátor `SomeT w a`, tak komonádu `Some` (bez možnosti transformace) můžeme získat tak, že za `w` zvolíme právě `Identity`:

```
type Some a = SomeT Identity a
```

3.3.2 Reader

`Reader` je dobře známý jako monáda, která umožňuje distribuovat přístup k jedné globální hodnotě. Toho se dosahuje pomocí funkcí: hodnoty typu `Reader e a` jsou ve skutečnosti funkce, jejichž prvním argumentem je právě globální hodnota typu `e`.

Na straně komonád je situace odlišná. Abychom byli schopni extrahovat hodnotu z takovéto funkce, musíme mít k dispozici jeden vybraný prvek typu `e`, a pro duplikaci naopak potřebujeme binární operaci (tj. funkci typu `e -> e -> e`) – výsledkem `duplicate` je funkce typu `e -> e -> a`, ale k dispozici máme pouze funkci typu `e -> a`, tato operace nám tedy určí, jakým způsobem máme zkombinovat dva argumenty typu `e` výsledné funkce. Pokusíme-li se dokázat, že komonáda definovaná níže uvedenými operacemi splňuje všechny vlastnosti, zjistíme také, že je zapotřebí asociativita binární operace a neutrálnost vybraného prvku vůči této operaci. Od typu `e` tedy vyžadujeme strukturu monoidu (tj. `<>` a `mempty`, kde `<>` je asociativní binární operace a `mempty` její neutrální prvek):

```
type Reader e a = e -> a

instance (Monoid e) => Comonad ((->) e) where
    extract   f = f mempty
    extend  g f = \e1 -> g \e2 -> f (e1 <> e2)
```

Instanci typové třídy `Functor` není nutné definovat, je již součástí modulu `Prelude`. Pro úplnost ji ale uvedeme:

```
instance Functor ((->) e) where
    fmap = (.)
```

3.3.3 Env

Dříve uvedená komonáda `Reader` má sice stejnou strukturu jako `Reader` monáda, ale na sdílení globální hodnoty se nedá použít. Komonáda, která do určité míry umožňuje právě toto sdílení, je `Env` (známá také pod názvem `Coreader`).

Na rozdíl od `Reader` monády, která globální hodnotu předává prvním argumentem funkce, funguje `Env e a` jednoduše tak, že sdruží globální hodnotu typu `e` s hodnotou typu `a` ve dvojici.

```
type Env e a = (e, a)

instance Functor ((,) e) where
    fmap f (e, a) = (e, f a)

instance Comonad ((,) e) where
    extract   (_, a) = a
    duplicate (e, a) = (e, (e , a))
```

Stejně jako `Reader` monáda má i `Env` komonáda operaci `ask`, která vrací globální hodnotu.

```
ask :: Env e a -> e
ask (e, _) = e
```

3.3.4 Store

`Store` je první komonáda, která reprezentuje kontext v obvyklejší podobě struktura

spolu s vybraným prvkem. `Store s a` je dvojice tvořená hodnotou typu `s`, která reprezentuje pozici ve struktuře, a hodnotou typu `s -> a`, což je funkce, která pro danou pozici vrátí odpovídající prvek ve struktuře.

Zajímavé je, že struktura je určena právě typem `s`. Například pro přirozená čísla je odpovídající struktura nekonečný seznam, pro seznamy hodnot typu `Bool` to může být nekonečný binární strom (jednotlivé hodnoty `True` a `False` určují, zda-li máme sestoupit do levého či pravého podstromu).

Příklad na použití této komonády je k nalezení v podkapitole 6.3. Dále uvádíme pouze definici instancí typových tříd `Functor` a `Comonad`.

```
data Store s a = Store (s -> a) s

instance Functor (Store s) where
    fmap f (Store g s) = Store (f . g) s

instance Comonad (Store s) where
    extract   (Store f s) = f s
    duplicate (Store f s) = Store (Store f) s
```

Klíčové operace jsou `pos` (která vrací vybranou pozici) a `peek` (která vrací hodnotu na zvolené pozici):

```
pos :: Store s a -> s
pos (Store _ s) = s

peek :: s -> Store s a -> a
peek s (Store f _) = f s
```

3.3.5 Koprodukt

Pokud se podíváme na možnosti kombinování dvou monád či komonád, dostaneme v zásadě dvě možnosti: produkt a koprodukt. Nabízí se také kompozice, ale monády a komonády nejsou obecně uzavřeny na tuto operaci.

```
data Product   f g a = P (f a) (g a)

data Coproduct f g a = L (f a) | R (g a)
```

Dvě monády lze zkombinovat pomocí produktu (tj. dvojice, která obsahuje obě monády), ale naopak je nelze zkombinovat koproduktem (tj. hodnota, která obsahuje jednu ze dvou monád, ale ne obě). Příklad kombinování dvou monád je k nalezení například v knihovně *monad-products* [7].

U komonád je situace opačná. Dvě komonády lze sloučit koproduktem, ale ne produktem. Můžeme tedy snadno reprezentovat výběr mezi dvěma komonádami.

```
instance (Functor f, Functor g) => Functor (Coproduct f g) where
    fmap f (L fa) = L (fmap f fa)
    fmap f (R ga) = R (fmap f ga)

instance (Comonad f, Comonad g) => Comonad (Coproduct f g) where
    extract (L fa) = extract fa
    extract (R ga) = extract ga

    duplicate (L fa) = L (fmap L (duplicate fa))
    duplicate (R ga) = R (fmap R (duplicate ga))
```

3.3.6 Neprázdné seznamy

Samotné seznamy nemohou tvořit komonádu, protože není možné implementovat operaci `extract` pro prázdný seznam. Co se ale stane, pokud se omezíme na seznamy s alespoň jedním prvkem? Neprázdné seznamy se dají reprezentovat mnoha různými způsoby:

```
data NonEmpty a = NonEmpty a [a]

data NonEmpty a = NonEmpty a (Maybe (NonEmpty a))

data NonEmpty a = End a | More a (NonEmpty a)
```

V příloze B.5 jsme použili poslední reprezentaci, protože se pak lépe provádí důkazy. Nicméně pro obyčejné použití je příjemnější reprezentace první.

```
toList :: NonEmpty a -> [a]
toList (NonEmpty x xs) = x:xs

instance Functor NonEmpty where
    fmap f (NonEmpty x xs) = NonEmpty (f x) (map f xs)

instance Comonad NonEmpty where
    extract (NonEmpty x _) = x

    duplicate n@(NonEmpty x [])
        = NonEmpty n []
    duplicate n@(NonEmpty x (y:ys))
        = NonEmpty n . toList . duplicate $ NonEmpty y ys
```

4 Zipper

4.1 Motivace a popis

Ve funkcionálním programování se často používají persistentní datové struktury, které neumožňují přímou (*in-place*) modifikaci, pokud tedy chceme tuto strukturu upravit, musíme vyrobit novou verzi. To může být výhoda i nevýhoda – persistentní datové struktury jsou z principu *thread-safe* (tj. lze je bezpečně použít i v přítomnosti vláken), části struktury lze snadno sdílet; nemožnost přímé modifikace na druhou stranu může vést k vyšší paměťové náročnosti (změna struktury vyžaduje výrobu nové verze i když původní strukturu už dále nepotřebujeme) nebo například k nemožnosti implementovat hašovací tabulky, které jsou na přímé modifikaci založeny.

Nutnost kopírovat celou strukturu se dá do jisté míry omezit sdílením částí datové struktury. Pokud do binárního vyhledávacího stromu vložíme novou hodnotu, stačí zkopírovat uzly na cestě k novému vrcholu; zbytek stromu může být sdílen.

Kromě výše zmíněných nevýhod také na první pohled persistentní struktury neumožňují pohyb uvnitř struktury a modifikaci vybraných prvků. Nejen že nepersistentní struktury mohou mít zpětné ukazatele a současně umožňovat modifikaci (což není možné u persistentních struktur; zpětné ukazatele vytvářejí cyklické struktury, které se nedají efektivně kopírovat), ale také si mohou vést ukazatel na vybraný prvek a ten velice jednoduše měnit.

Tento problém lze vyřešit pro stromovité (obecně rekurzivně definované) datové struktury pomocí *zipperu* [8], což je datová struktura, která v určitém smyslu reprezentuje jeden ukazatel – jednu pozici uvnitř struktury. Díky tomu lze zipper použít nejen pro pohyb uvnitř struktury, ale také k efektivní modifikaci právě vybraného prvku.

Základem je rozdělení původní struktury na tři části: ještě nenavštívená část (která umožňuje pohyb směrem dolů), vybraný prvek a již navštívená část, která je ale uložena v obráceném pořadí – část struktury nejblíže vybranému prvku je uložena na začátku (tato část naopak umožňuje pohyb směrem nahoru). Vždy máme přístup nejen k vybranému prvku, ale také k celé struktuře – z těchto částí můžeme totiž opět složit celou strukturu.

Pěkným příkladem je zipper pro binární stromy. První část tvoří dva podstromy vybraného vrcholu. Druhou část tvoří hodnota vybraného vrcholu a třetí část je popis cesty k němu, tedy seznam hodnot „vlevo“ a „vpravo“. Abychom byli schopni rekonstruovat původní strukturu, tak si vedle směru, kterým jsme se vydali, ukládáme také navštívený vrchol a podstrom, do kterého jsme nesestoupili.

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

```
data Direction = Left | Right

data Zipper a
    = Zipper (Tree a, Tree a) a [(Direction, a, Tree a)]

up :: Zipper a -> Maybe (Zipper a)
up (Zipper _      _ []) = Nothing
up (Zipper (l, r) x ((d, x', t):rest)) = Just $ case d of
    Left  -> Zipper (Node x l r, t) x' rest
    Right -> Zipper (t, Node x l r) x' rest

left :: Zipper a -> Maybe (Zipper a)
left (Zipper (Nil, _)          _ _) = Nothing
left (Zipper (Node x' l r, t) x rest) = Just $
    Zipper (l, r) x' ((Left, x, t):rest)

right :: Zipper a -> Maybe (Zipper a)
right (Zipper (_, Nil)         _ _) = Nothing
right (Zipper (t, Node x' l r) x rest) = Just $
    Zipper (l, r) x' ((Right, x, t):rest)
```

Vybraný prvek lze velice snadno modifikovat – stačí upravit druhou část zipperu.

```
modify :: (a -> a) -> Zipper a -> Zipper a
modify f (Zipper l x u) = Zipper l (f x) u
```

Zbývá definovat dvě operace – první operace vytvoří zipper z původní struktury a druhá operace naopak ze zipperu původní strukturu rekonstruuje.

```
start :: Tree a -> Maybe (Zipper a)
start Nil          = Nothing
start (Node x l r) = Just (Zipper (l, r) x [])

plug :: Zipper a -> Tree a
plug = finish . last . takeWhile isJust . iterate (>>= up) . Just
  where
    isJust (Just _) = True
    isJust _        = False

    finish (Just (Zipper (l, r) x [])) = Node x l r
```

Operace `start` je specifická pro každý zipper: některé struktury mohou mít vždy k dispozici startovní pozici (třeba neprázdné seznamy) – pak není zapotřebí typ `Maybe`; pokud jsou k dispozici dvě startovní pozice (např. neprázdný strom, jehož kořen má vždy dva prvky), může `start` vracet seznam nebo můžeme mít dvě operace, atp.

Na druhou stranu operace `plug` je vždy přítomná v této podobě. Z popisu plyne, že každý zipper má možnost rekonstruovat původní strukturu. V této implementaci jsme jednoduše využili dříve definovanou funkci `up`, ale `plug` můžeme implementovat přímo bez použití parciální funkce `finish`.

4.2 Zipper a komonády

Pokud se na zipper díváme jako na hodnotu (vybraný prvek původní struktury) a kontext, který určuje jeho pozici, tak se nabízí otázka, jestli se nejedná o komonádu. Pro příklady využijeme zipper pro seznamy.

```
data Zipper a = Zipper [a] a [a]

left :: Zipper a -> Maybe (Zipper a)
left (Zipper []     _ _) = Nothing
left (Zipper (l:ls) x r) = Just (Zipper ls l (x:r))

right :: Zipper a -> Maybe (Zipper a)
right (Zipper _ _ [])     = Nothing
right (Zipper l x (r:rs)) = Just (Zipper (x:l) r rs)

start :: [a] -> Maybe (Zipper a)
start []     = Nothing
start (x:xs) = Just (Zipper [] x xs)

plug :: Zipper a -> [a]
plug (Zipper l x r) = reverse l ++ [x] ++ r
```

První operací komonády je `extract`, která by měla zahodit kontext (tj. část struktury, která určuje pozici vybraného prvku). Jelikož zipper vždy obsahuje právě vybraný prvek, je implementace velice jednoduchá.

```
extract :: Zipper a -> a
extract (Zipper _ x _) = x
```

Operace `duplicate` je ale náročnější. Z obyčejného zipperu musíme vytvořit zipper, jehož prvky jsou další zippery. Nabízí se nahradit prvky původního zipperu za zippery, které určují pozici těchto prvků.

```
zip :: Zipper Int
zip = Zipper [2,1] 3 [4,5]

duplicatedZip :: Zipper (Zipper Int)
duplicatedZip = Zipper
    [Zipper [1]       2 [3,4,5]   ,
     Zipper []        1 [2,3,4,5] ]
    (Zipper [2,1]     3 [4,5]     )
    [Zipper [3,2,1]   4 [5]       ,
     Zipper [4,3,2,1] 5 []        ]
```

Všimněme si, že první seznam `duplicatedZip` je také v obráceném pořadí – prvním prvkem je zipper ukazující na prvek 2. Samotná implementace může vypadat následovně.

```
duplicate :: Zipper a -> Zipper (Zipper a)
duplicate z = Zipper (go left z) z (go right z)
  where
    go f = map fromJust . tail
         . takeWhile isJust . iterate (>>= f) . Just
```

```
isJust (Just _) = True
isJust _        = False

fromJust (Just x) = x
```

Také musíme ověřit, že takto definované operace skutečně splňují požadované vlastnosti. Pro případ tohoto konkrétního zipperu je důkaz proveden v příloze B.5. Definice `duplicate` je mírně odlišná pro potřeby důkazu (explicitně rekurzivní definice umožňuje provést jednoduchý důkaz indukcí), ale dokazatelně ekvivalentní s výše uvedenou definicí.

Obecný zipper s takto definovanými operacemi splňuje požadavky komonády, pokud lze každou pozici původní struktury reprezentovat právě jednou hodnotou zipperu. Jakmile můžeme reprezentovat jednu pozici dvěma možnými způsoby, nastává problém s vlastností `extract . duplicate == id`. Z výše uvedeného popisu plyne, že pokud původní hodnota označovala prvek x, tak i nová hodnota označuje prvek x – ale nemáme již garanci, že se jedná o tu samou hodnotu. Důkaz pro obecný zipper se samozřejmě bude lišit v závislosti na konkrétním zipperu.

5 Derivace a kontext

5.1 Definice

Nejprve definujeme, jak derivovat regulární datový typ podle proměnné x (značeno jako ∂_x). Induktivní strukturu regulárních typů můžeme efektivně využít i pro definici derivace, stačí uvést, jak se derivace zachová pro každý případ.

$$
\begin{aligned}
\partial_x(x) &= 1 \\
\partial_x(y) &= 0 \\
\partial_x(1) &= 0 \\
\partial_x(0) &= 0 \\
\partial_x(T + S) &= \partial_x(T) + \partial_x(S) \\
\partial_x(T \times S) &= \partial_x(T) \times S + T \times \partial_x(S) \\
\partial_x(\mu y.\ T) &= \mu z.\ [\mu y.\ T/y]\partial_x(T) + [\mu y.\ T/y]\partial_y(T) \times z \\
\partial_x(\nu y.\ T) &= \mu z.\ [\nu y.\ T/y]\partial_x(T) + [\nu y.\ T/y]\partial_y(T) \times z
\end{aligned}
$$

Obecně předpokládáme, že jsou všechny uvedené proměnné různé, tj. $x \neq y \neq z$. Pokud tomu tak není, pomůžeme si vhodným přejmenováním. Proměnná z se navíc uvažuje jako nová volná proměnná, která se v popisu derivovaného typu nevyskytuje. Kromě posledních dvou případů se jedná o známá pravidla derivace z matematické analýzy. Tato souvislost je o to zvláštnější, že význam derivace typů je značně odlišný od významu derivace funkcí v matematické analýze. Interpretace této definice je uvedena v následující podkapitole.

Derivaci μ a ν lze definovat dalším způsobem. Pokud se pozorně podíváme na strukturu derivace $\partial_x(\mu y.\ T)$, všimneme si, že odpovídající výraz na pravé straně je ve skutečnosti určitý druh seznamu. Porovnejme tento typ s definicí seznamu:

$$\text{List}\ x = \mu y.\ 1 + x \times y$$

Rozdíly jsou v tom, že typ prvků seznamu je $[\mu y.\ T/y]\partial_y(T)$ a místo triviální hodnoty typu 1 na konci seznamu máme hodnotu typu $[\mu y.\ T/y]\partial_x(T)$. Mohli bychom tedy použít definici:

$$\partial_x(\mu y.\ T) = [\mu y.\ T/y]\partial_x(T) \times \text{List}\ ([\mu y.\ T/y]\partial_y(T)) \qquad (1)$$

Tyto typy jsou ale isomorfní:

```
data D a b = End b | More a (D a b)

from :: D a b -> (b, [a])
from (End  b)   = (b, [])
from (More a r) = (b, a:r')
  where
    (b, r') = from r

to :: (b, [a]) -> D a b
to (b, [])   = End  b
to (b, a:as) = More a (to (b, as))
```

Z čehož plyne, že obě definice jsou až na isomorfismus ekvivalentní. Stejným způsobem lze upravit definici pro případ $\partial_x(\nu y.\ T)$.

Nakonec se podívejme na konkrétní příklad. Derivujme dříve uvedený typ List x podle proměnné x.

$$\begin{aligned}
\partial_x(\text{List } x) &= \partial_x(\mu y.\ 1 + x \times y) \\
&= \mu z.\ [\text{List } x/y]\partial_x(1 + x \times y) + \\
&\qquad [\text{List } x/y]\partial_y(1 + x \times y) \times z \\
&= \mu z.\ [\text{List } x/y](0 + \partial_x(x \times y)) + \\
&\qquad [\text{List } x/y](0 + \partial_y(x \times y)) \times z \\
&= \mu z.\ [\text{List } x/y](0 + \partial_x(x) \times y + x \times \partial_x(y)) + \\
&\qquad [\text{List } x/y](0 + \partial_y(x) \times y + x \times \partial_y(y)) \times z \\
&= \mu z.\ [\text{List } x/y](0 + 1 \times y + x \times 0) + \\
&\qquad [\text{List } x/y](0 + 0 \times y + x \times 1) \times z \\
&= \mu z.\ [\text{List } x/y]y + \\
&\qquad [\text{List } x/y]x \times z \\
&= \mu z.\ \text{List } x + x \times z \\
&= \text{List } x \times \text{List } x
\end{aligned}$$

Pro připomenutí, $[S/x]T$ znamená nahrazení (substituci) všech volných výskytů proměnné x v T za typ S. Všimněme si, že se zbavujeme nadbytečných částí typu 0 a 1. To si můžeme dovolit právě díky tomu, že typy tvoří komutativní polookruh, jak je uvedeno v podkapitole 2.3.

5.2 Derivace je kontext

Uvažme regulární datový typ T parametrizovaný proměnnou a, jehož hodnoty mohou obsahovat prvky typu a. Zvolme si jeden takový prvek a odeberme ho z původní struktury. Výsledkem je struktura s jednou dírou po odebraném prvku, tzv. *kontext* [9] (specificky *one-hole context*, protože kontext určuje pouze jednu díru). Tato struktura určuje nejen okolí odebraného prvku, ale také jeho pozici v původní struktuře. Otázkou je, jestli lze pro typ T definovat odpovídající typ reprezentující

právě tyto kontexty, dále Ctx(T). Podívejme se, co by tento typ musel splňovat.

Pro typ 0 dostáváme prázdný kontext, protože 0 neobsahuje žádné hodnoty typu a. Ze stejného důvodu je kontextem typu 1 taktéž 0. Pokud narazíme na hodnotu typu a, tak je kontext triviální – existuje pouze jedna pozice, na které se hodnota může vyskytovat. Typ kontextu je tedy 1. Pro typ odlišný od typu a, třeba b, dostáváme opět prázdný kontext – b neobsahuje žádné hodnoty typu a.

Pokud máme výběr $T + S$, tak se prvek typu a může nacházet vlevo (tj. v hodnotě typu T) nebo vpravo (tj. v hodnotě typu S). Pokud se nachází vlevo, tak je jeho pozice určená kontextem levé hodnoty; pokud je vpravo, tak naopak kontextem pravé hodnoty. Typ výsledného kontextu je tedy Ctx(T) + Ctx(S).

Pro dvojici $T \times S$ je situace mírně odlišná. Pokud se vybraný prvek nachází v levé části, tak potřebujeme kontext levé části, ale také celou pravou část (jinak „ztratíme“ polovinu původní struktury). Analogicky vypadá kontext pro případ, kdy je prvek v pravé části. Výsledný typ je Ctx(T) $\times S + T \times$ Ctx(S).

Výsledné typy připomínají derivaci z minulé podkapitoly. A skutečně, typ kontextů odpovídá derivaci.

$$\mathrm{Ctx}(T) = \partial_a(T)$$

Zbývá vyřešit regulární typy formy $\mu x.\ T$ a $\nu x.\ T$, které nemají ekvivalent na straně derivace funkcí z matematické analýzy. Na typ $\mu x.\ T$ se můžeme dívat jako na určitý druh stromu, ve kterém je vzhled jedné úrovně dán právě typem T (volné výskyty proměnné x pak určují, ve kterých místech se nacházejí podstromy). Podívejme se na cestu k vybranému prvku – jak procházíme stromem, postupně navštěvujeme jednotlivé úrovně.

Pokud se prvek nachází na dané úrovni, je jeho pozice určena kontextem jedné úrovně, tedy hodnotou typu $\partial_a(T)$. Samozřejmě, v původní struktuře všechny volné výskyty proměnné x v T odkazovaly na podstromy, které nemůžeme zahodit. Stačí tedy vzít tento výsledný typ kontextu a nahradit volné výskyty x právě za typ podstromů, pro celou úroveň tedy dostaneme $[\mu x.\ T/x]\partial_a(T)$.

Je-li prvek na nižší úrovni, musíme si na dané úrovni zapamatovat, do kterého podstromu jsme sestoupili. To se dá vyřešit snadno tak, že se podíváme na typ kontextu vzhledem k proměnné x. Hodnota typu $\partial_x(T)$ určuje pozici vybraného podstromu. Narážíme na stejný problém jako v předchozím případě: je nutné zapamatovat si ostatní podstromy, kterými cesta nevede. Řešení je identické, stačí nahradit x za typ podstromů: $[\mu x.\ T/x]\partial_x(T)$.

Důležité je si také uvědomit, že jak $[\mu x.\ T/x]\partial_a(T)$ tak i $[\mu x.\ T/x]\partial_x(T)$ korektně obsahují ostatní hodnoty typu a na dané úrovni. Hodnoty typu $\partial_a(T)$ oproti hodnotám typu T obsahují o jeden prvek typu a méně – což je námi vybraná pozice. Hodnoty typu $\partial_x(T)$ naopak obsahují všechny prvky typu a díky tomu, že derivujeme podle jiné proměnné.

Na nižších úrovních se situace opakuje. Pokud je prvek např. na třetí úrovni, je jeho pozice určená hodnotou typu $[\mu x.\ T/x]\partial_x(T) \times [\mu x.\ T/x]\partial_x(T) \times [\mu x.\ T/x]\partial_a(T)$. Všimněme si, že typ kontextu je stejně jako původní typ také rekurzivní: kontext je buď hodnotou typu $[\mu x.\ T/x]\partial_a(T)$, nebo je to hodnota typu $[\mu x.\ T/x]\partial_x(T)$ spolu s kontextem pro nižší úroveň. Tímto dostáváme přesně definici uvedenou v předchozí podkapitole:

$$\partial_a(\mu x.\ T) = \mu y.\ [\mu x.\ T/x]\partial_a(T) + [\mu x.\ T/x]\partial_x(T) \times y$$

Definice pro ν je obdobná, jedná se také o určitý druh stromu, jehož struktura je daná typem T, ale na rozdíl od $\mu x.\ T$ může být tento strom nekonečný. Nabízí se otázka, proč nemůže být výsledný kontext také nekonečný. Problém tkví v tom, že nekonečné kontexty efektivně neurčují žádnou pozici. Pokud bychom měli nekonečný binární strom, tak nekonečný kontext „neustále jdi vlevo“ neurčuje žádnou pozici stejně jako „největší přirozené číslo“ neurčuje žádné číslo.

Podívejme se na derivaci typu List x z minulé podkapitoly a srovnejme ji s intuitivní představou pozice v seznamu.

$$\partial_x(\text{List } x) = \text{List } x \times \text{List } x$$

Vybereme-li si určitý prvek seznamu, můžeme podle tohoto prvku rozdělit seznam na dvě části: seznam prvků, které se nacházejí před vybraným prvkem, a seznam prvků, které jsou za vybraným prvkem. Tato dvojice seznamů ale jednoznačně určuje pozici vybraného prvku.

```
type Ctx a = ([a], [a])

value :: [Int]
value = [1, 2, 3]

-- kontext prvku 1
ctx1 :: Ctx Int
ctx1 = ([], [2, 3])

-- kontext prvku 2
ctx2 :: Ctx Int
ctx2 = ([1], [3])

-- kontext prvku 3
ctx3 :: Ctx Int
ctx3 = ([1, 2], [])
```

Mnohdy je výhodnější udržovat si první seznam v obráceném pořadí (obecně seznam z alternativní definice derivace pro μ a ν). Tato reprezentace umožňuje efektivnější pohyb po struktuře, protože okolí, které je vybranému prvku nejblíže, je uloženo na začátku seznamu. Toto zlepšení je prozkoumáno v podkapitole 6.4.

5.3 Kontext jako součást zipperu

Z předchozí podkapitoly víme, že kontext prvku x ve struktuře s není nic jiného, než struktura s, ze které byl odstraněn právě prvek x. Tímto nám kontext určuje pozici prvku x v původní struktuře s. Rozdíl mezi kontextem a zipperem je ten, že v zipperu je kromě pozice uložen i samotný prvek; díky tomu jsme schopni určit, jaký prvek se na vybrané pozici nachází a případně ho také můžeme měnit. Pokud tedy vezmeme kontext a přidáme k němu odebraný prvek, dostaneme zipper. Jenže pro určení typu kontextu stačí derivace – pro každý regulární typ je tedy možné definovat zipper a to velice jednoduchým způsobem:

$$\mathrm{Zipper}(T, a) = \partial_a(T) \times a$$

Jedna z klíčových operací zipperu je *plug*, tedy operace, která zipper složí zpět do původní struktury. Pro definici pomocí derivace potřebujeme operaci typu:

$$\mathrm{plug}[T, a] \ : \ \partial_a(T) \to a \to T$$

Tuto operaci definujeme podle struktury regulárního typu.

$$\begin{aligned}
\mathrm{plug}[a, a]\ \langle\rangle\ e &= e \\
\mathrm{plug}[T + S, a]\ (\mathrm{inl}\ c)\ e &= \mathrm{inl}\ (\mathrm{plug}[T, a]\ c\ e) \\
\mathrm{plug}[T + S, a]\ (\mathrm{inr}\ c)\ e &= \mathrm{inr}\ (\mathrm{plug}[S, a]\ c\ e) \\
\mathrm{plug}[T \times S, a]\ (\mathrm{inl}\ \langle c, s\rangle)\ e &= \langle \mathrm{plug}[T, a]\ c\ e, s\rangle \\
\mathrm{plug}[T \times S, a]\ (\mathrm{inr}\ \langle t, c\rangle)\ e &= \langle t, \mathrm{plug}[S, a]\ c\ e\rangle \\
\mathrm{plug}[\mu x.\ T, a]\ (\mathrm{rec}\ (\mathrm{inl}\ c))\ e &= \mathrm{rec}\ (\mathrm{plug}[T, a]\ c\ e) \\
\mathrm{plug}[\mu x.\ T, a]\ (\mathrm{rec}\ (\mathrm{inr}\ \langle c, cs\rangle)\ e &= \mathrm{rec}\ (\mathrm{plug}[T, x]\ c\ (\mathrm{plug}[\mu x.\ T, a]\ cs\ e)) \\
\mathrm{plug}[\nu x.\ T, a]\ (\mathrm{rec}\ (\mathrm{inl}\ c))\ e &= \mathrm{crec}\ (\mathrm{plug}[T, a]\ c\ e) \\
\mathrm{plug}[\nu x.\ T, a]\ (\mathrm{rec}\ (\mathrm{inr}\ \langle c, cs\rangle)\ e &= \mathrm{crec}\ (\mathrm{plug}[T, x]\ c\ (\mathrm{plug}[\nu x.\ T, a]\ cs\ e))
\end{aligned}$$

Zajímavé je, že tato definice je korektní, ačkoliv nejsou pokryty všechny případy – chybí například případ, kdy T je 1. Nicméně, $\partial_a(1) = 0$ a tedy máme hodnotu e typu 0. Jenže neexistuje žádná hodnota typu 0; tento případ tedy nastat nemůže.

Pro alternativní definici se seznamem (5.1.1) lze použít akumulátor. Nejprve spojíme pomocí plug hodnotu typu a a $[\mu x.\ T/x]\partial_a(T)$, což je naše počáteční hodnota akumulátoru. Poté stačí projít seznam a do akumulátoru postupně připojovat další úrovně:

$$\begin{aligned}\text{plug-list } (\text{rec } (\text{inl } \langle\rangle))\ \mathit{acc} &= \mathit{acc} \\ \text{plug-list } (\text{rec } (\text{inr } \langle \mathit{c}, \mathit{cs}\rangle))\ \mathit{acc} &= \text{plug-list } \mathit{cs}\ (\text{plug}[\mathit{T}, \mathit{x}]\ \mathit{c}\ \mathit{acc})\end{aligned}$$

Samozřejmě záleží na orientaci seznamu, výše uvedená metoda funguje pro seznamy, kde nejvyšší úroveň stromu $\mu x.\ T$ je posledním prvkem seznamu.

6 Konkrétní použití

6.1 Turingův stroj

Simulátor Turingova stroje je vhodným příkladem použití zipperu: v každém kroku výpočtu je nutné znát pozici „čtecí hlavy“ nad páskou, dále pak musíme posunovat pásku vpravo nebo vlevo o jedno políčko, ideálně v konstantním čase. Nejdříve definujeme reprezentaci samotného automatu:

```
data TM state symbol
    = TM
    { blank :: symbol
    , start :: state
    , end   :: state -> Bool
    , trans :: state -> symbol -> (state, symbol, Direction)
    }
```

Jedná se tedy o čtveřici, parametrizovanou typy `state` (který určuje „množinu“ stavů) a `symbol` (který určuje abecedu, nad kterou stroj operuje), která obsahuje prázdný symbol (kterým se zpočátku vyplní páska), počáteční stav, množinu koncových stavů (zde reprezentována jako funkce `state -> Bool`, která vrací `True` pokud je daný argument koncovým stavem) a nakonec přechodovou funkci. Ještě zbývá definovat typ `Direction`:

```
data Direction = L | R
    deriving (Show, Read, Eq)
```

Nyní se můžeme přesunou k reprezentaci samotné pásky. Náš Turingův stroj se může hýbat libovolně v obou směrech, chtěli bychom tedy pásku nekonečnou v obou směrech. Toho lze docílit několika různými způsoby: můžeme vytvořit pásku, která je skutečně nekonečná v obou směrech, a díky línému vyhodnocování nebude s výpočtem problém; nebo budeme mít pásku konečnou a pokud bychom se měli posunout mimo pásku, přidáme na tomto konci další prázdný symbol. První reprezentace je sice elegantnější, ale druhá nám mnohem lépe umožňuje sledovat výpočet. Zde jsou obě reprezentace:

```
data Stream a = a :< Stream a

data WholeTape1 a
    = WT1
    { leftPart  :: Stream a
    , rightPart :: Stream a
    }


data WholeTape2 a
    = WT2
    { tape :: [a]
    }
```

Nyní můžeme tyto datové typy derivovat podle parametru `a` a tím dostaneme datový typ, který reprezentuje přesně jednu pozici uvnitř pásky (což bude pozice čtecí hlavy):

$$\begin{aligned}
\text{Stream } a &= \nu x.\ a \times x \\
\\
\partial_a(\nu x.\ a \times x) &= \mu y.\ (1 \times (\nu x.\ a \times x)) + (1 \times a) \times y \\
&= \mu y.\ (\nu x.\ a \times x) + a \times y \\
&= \text{List } a \times \text{Stream } a \\
\\
\partial_a(\text{WholeTape1 } a) &= 2 \times \text{Stream } a \times \text{List } a \times \text{Stream } a \\
\partial_a(\text{WholeTape2 } a) &= \text{List } a \times \text{List } a
\end{aligned}$$

Derivace typu `WholeTape2` není překvapivá, ale `WholeTape1` je poněkud zvláštní. Podívejme se na tuto čtveřici zleva doprava: typ 2 (= 1 + 1; efektivně tedy `Bool`) určuje, jestli se daný prvek vyskytuje v levé nebo v pravé části, Stream a označuje celou opačnou část (tj. pokud se prvek nacházel v levé části, reprezentuje Stream a celou pravou část) a nakonec List a × Stream a určuje (konečnou) část před vybraným prvkem a (nekonečnou) část za tímto prvkem. Stream a × List a × Stream a můžeme zjednodušit na Stream a × Stream a tak, že konečný prefix na jedné straně zahrneme do Stream a na straně druhé. Nakonec můžeme zapomenout, na které straně se daný prvek nachází (tj. vyhodíme typ 2); tyto úpravy jsou neekvivalentní (ztrácíme informaci o tom, kde byl předěl mezi levou a pravou částí), ale pro naše účely je dostačující uložit si prvky před a za vybraným místem. Dostáváme tedy dvě možné reprezentace pásky spolu s vybranou pozicí:

```
data Tape1
    = Tape1
    { before :: Stream a
    , here   :: a
    , after  :: Stream a
    }

data Tape2
    = Tape2
    { before :: [a]
    , here   ::  a
    , after  :: [a]
    }
```

Zbytek tohoto příkladu je založen na typu `Tape2` (dále pouze `Tape`).

Prvek pod čtecí hlavou můžeme extrahovat pomocí selektoru `here`, zbývá definovat posouvání a zapisování pásky:

```
left :: a -> Tape a -> Tape a
left  e (Tape []     x r) = Tape [] e (x:r)
left  _ (Tape (l:ls) x r) = Tape ls l (x:r)
```

```
right :: a -> Tape a -> Tape a
right e (Tape l x []    ) = Tape (x:l) e []
right _ (Tape l x (r:rs)) = Tape (x:l) r rs

write :: a -> Tape a -> Tape a
write x (Tape l _ r) = Tape l x r
```

První argument funkcí `left` a `right` v tomto případě označuje prázdný symbol, kterým vyplňujeme pásku, pokud narazíme na konec. V obou dvou funkcích se podíváme pouze na vrchol jednoho ze seznamů a rozšíříme druhý seznam (aplikací konstruktoru `(:)`). Díky tomuto jsou tyto operace skutečně $O(1)$.

Nyní můžeme přirozeným způsobem definovat simulaci Turingova stroje:

```
run :: TM state symbol -> Tape symbol -> [Tape symbol]
run (TM b s e t) = run' s
  where
    run' st tape
      | e st      = [tape]
      | otherwise = case t st (here tape) of
          (st', x, d) ->
              (tape:) . run' st' . move d . write x $ tape
      where
        move L = left  b
        move R = right b
```

Pomocná funkce `run'` provádí samotnou simulaci. Argument `st` označuje stav, ve kterém se Turingův stroj zrovna nachází, `tape` je pak páska. Pokud se dostaneme do koncového stavu (tj. `e st == True`), pak výpočet končí a vrátíme koncový stav pásky. V opačném případě aplikujeme přechodovou funkci `t` na momentální stav a symbol pod „čtecí hlavou“. Přechodová funkce vrátí trojici `(st', x, d)`, kde `st'` je nový stav, `x` je nový symbol, který máme zapsat, a `d` je směr, kterým se má páska posunout.

Na novou pásku a stav aplikujeme rekurzivně `run'` a k výsledku pak připojíme původní pásku. Tímto způsobem můžeme sledovat všechny kroky výpočtu: funkce totiž vrací tyto kroky jako neprázdný, potenciálně nekonečný seznam pásek. Pokud chceme pouze výsledek, stačí definovat:

```
runLast :: TM state symbol -> Tape symbol -> Tape symbol
runLast tm = last . run tm
```

Případně ještě můžeme vyžadovat funkci, která připraví prázdnou pásku:

```
empty :: symbol -> Tape symbol
empty s = Tape [] s []
```

Jako příklad uvádíme 5-stavový „busy beaver“:

```
main = do
    let start = empty 0
        end   = run bb start
    print (length end)
```

```
data S = A | B | C | D | E | Halt
    deriving (Show, Eq)

data A = 0 | P
    deriving (Show, Eq)

bb :: TM S A
bb = TM 0 A (== Halt) trans
  where
    trans A 0 = (B, P, R)
    trans A P = (C, P, L)
    trans B 0 = (C, P, R)
    trans B P = (B, P, R)
    trans C 0 = (D, P, R)
    trans C P = (E, 0, L)
    trans D 0 = (A, P, L)
    trans D P = (D, P, L)
    trans E 0 = (Halt, P, R)
    trans E P = (A, 0, L)
```

Ve skutečnosti lze predikát, který určuje, zda-li je stav koncový, velice jednoduše implementovat jako `(== stav)`, nebo v případě více stavů jako `(`elem` [stav_1, stav_2 .. stav_n])`. Přechodová funkce je pak definovaná jednoduše výčtem případů. Jedná se o parciální funkci, ale díky implementaci funkce `run` nám tento fakt nevadí.

Obvyklá definice Turingova stroje definuje přechodovou funkci pouze na nekoncových stavech a to tak, že doménou funkce je rozdíl množiny všech stavů a množiny koncových stavů. Z pohledu teorie množin je jednoduché mít funkci, jejíž doména je rozdílem dvou množin, v případě naší reprezentace by to mohlo být něco jako:

state - { x | x je koncový stav }

Ale v Haskellu nic takového nelze jednoduše reprezentovat. Na druhou stranu můžeme elegantně jít druhým směrem: začneme od částí a spojíme je v celek, například pomocí datového typu `Either`. Typ `state` tedy nebude reprezentovat všechny stavy, ale pouze stavy nekoncové. Pokud typ `finalstate` reprezentuje koncové stavy, dostaneme množinu všech stavů jako `Either state finalstate`.

V tomto případě následující problém nenastane, ale je dobré ho zmínit: pokud se snažíme takto manipulovat s typy jakožto s množinami, je nutné si uvědomit, že `Either` není „opravdové“ sjednocení, ale disjunktní sjednocení. Všechny prvky v průniku budou ve výsledném disjunktním sjednocení dvakrát.

Pokud nás koncový stav nezajímá, tak můžeme použít typ `Maybe`. Všechny takovéto stavy jsou pak reprezentované konstruktorem `Nothing`.

```
data TM state symbol
    = TM
    { blank :: symbol
    , start :: Maybe state
```

```
        , trans :: state -> symbol ->
                   (Maybe state, symbol, Direction)
        }
```

Průběžný stav Turingova stroje (tj. argument `st` v pomocné funkci `run'`) pak v sobě nese informaci o tom, zda-li je nebo není koncový. Díky tomuto můžeme vypustit funkci `end`. Plně funkční příklad je k nalezení v příloze A.1.

6.2 Obrazové filtry

Dalším možným použitím komonád je zpracování signálu a obrazu. Mnoho filtrů nad signálem či obrazem se dá popsat pomocí diskrétní konvoluce: chceme-li určit hodnotu na určité pozici výsledného signálu, stačí se podívat na okolí tohoto bodu v původním signálu a použít filtrovací funkci.

Takovéto okolí bodu bude mít jistě komonadickou strukturu: extrakce je pouze projekce vybraného bodu, duplikace je obohacení každého bodu jeho polohou v celém signálu.

Podívejme se tedy na konkrétní příklad: implementace Gaussovského rozostření. Pro načítání a ukládání obrazu je připraveno několik funkcí (definice jsou uvedeny v příloze A.2):

```
loadBitmap  :: FilePath -> IO (Maybe (Image PixelRGB8))
writeBitmap :: FilePath -> Image PixelRGB8 -> IO ()
imageToList :: Image PixelRGB8 -> [[PixelRGB8]]
listToImage :: [[PixelRGB8]] -> Image PixelRGB8
```

`loadBitmap` a `writeBitmap` slouží k načítání obrazů ve formátu `BMP`, `imageToList` a `listToImage` pak k převodu mezi (jednorozměrným) vektorem pixelů a dvourozměrným seznamem. Samotné typy `PixelRGB8` a `Image` jsou součástí knihovny *JuicyPixels* [10], která dále nabízí spoustu funkcí pro načítání, ukládání a manipulaci s obrázky.

Nyní potřebujeme reprezentaci bodu a jeho okolí. Pozice bodu je jednoznačně určena řádkem a sloupcem. Zde máme na výběr dvě možnosti – můžeme začít od jednoho řádku (tj. pozice je nejprve určená sloupcem – pozicí v jednom řádku) a pak přidáme ostatní řádky, nebo můžeme začít od sloupce. Tyto možnosti jsou víceméně ekvivalentní; v následujícím kódu je použita první implementace.

Pozice prvku na jednom řádku se dá popsat tím, co se nachází před a za vybraným prvkem:

```
data Line a
    = Line
    { before :: [a]
    , here   ::  a
    , after  :: [a]
    }
```

Například řádek `[1 .. 5]` s vybraným prvkem `3` bude reprezentován takto:

```
example :: Line Int
example = Line [2, 1] 3 [4, 5]
```

Všimněme si, že podobně jako u simulátoru Turingova stroje je seznam `before` obrácený a díky tomuto je posun $O(1)$ operace.

Teď můžeme dodat zbylé řádky. Opět budeme mít dva seznamy: seznam řádek nad a pod vybranou řádkou. Jejich struktura je shodná se strukturou vybrané řádky, tj. obsahuje opět polohu jedné vybrané řádky. Pro určení jednoho místa v obrázku to není nutné, můžeme mít pouze obyčejné seznamy, které reprezentují celou řádku, ale tato implementace nám umožní snadný pohyb nahoru i dolů.

```
data Plane a
    = Plane
    { above :: [Line a]
    , line  ::  Line a
    , below :: [Line a]
    }
```

Jako příklad vezměme matici (po řádcích) `[[1, 2, 3], [4, 5, 6], [7, 8, 9]]` s vybraným prvkem `5`:

```
example :: Plane Int
example = Plane
    [Line [1] 2 [3]]
    (Line [4] 5 [6])
    [Line [7] 8 [9]]
```

`Plane` a `Line` můžeme jednoduše sloučit zpět do seznamu pomocí funkcí `plugPlane` a `plugLine`:

```
plugPlane :: Plane a -> [Line a]
plugPlane (Plane u x d) = reverse u ++ [x] ++ d

plugLine :: Line a -> [a]
plugLine  (Line l x r)  = reverse l ++ [x] ++ r
```

Dále dodáme implementace typové třídy `Functor` pro `Line` i `Plane` a můžeme přestoupit k typové třídě `Comonad`. Připomeňme, že typová třída `Functor` definuje operaci `fmap`:

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

Implementace operace `duplicate` je relativně rozsáhlá, proto zde uvádíme pouze popis funkčnosti. `extract` je jednoduchá projekce vybraného prvku, jedná se tedy o pouhé zahození kontextu. `duplicate` naopak kontext zduplikuje: každému prvku v původní struktuře je přidán kontext, který určuje, kde se daný prvek nachází – tím dostaneme „kontext kontextů“. Pokud uvažujeme pouze obdélníkové seznamy (tj. vnitřní seznamy mají všechny stejnou délku), tak tyto operace skutečně korektně definují komonádu, vlastnosti operací lze ověřit obdobně jako u zipperu pro

seznamy. Pro ilustraci se podívejme, co se stane, aplikujeme-li `duplicate` na dříve uvedený `example`:

```
duplicate example == Plane
    [Line [tl] tm [tr]]
    (Line [ml] mm [mr])
    [Line [bl] bm [br]]

tl == Plane
    []
    (Line [] 1 [2,3])
    [Line [] 4 [5,6]
    ,Line [] 7 [8,9]]

mm == Plane
    [Line [1] 2 [3]]
    (Line [4] 5 [6])
    [Line [7] 8 [9]]

br == Plane
    [Line [5,4] 6 []
    ,Line [2,1] 3 []]
    (Line [8,7] 9 [])
    []
```

Ostatních 6 hodnot (`tm`, `tr`, `ml`, `mr`, `bl`, `bm`) vypadá obdobně. Podívejme se např. na hodnotu 1 v `example` a na odpovídající místo v `duplicate example`, tedy `tl`. Je vidět, že `tl` skutečně určuje pozici (kontext) hodnoty 1 v celé matici.

Pro daný prvek lze snadno určit, jaké prvky se nacházejí v jeho okolí:

```
takeClosest :: a -> Int -> Plane a -> [[a]]
takeClosest empty n = map (plugLine . takeLine) . plugPlane .
takePlane
  where
    e  = repeat empty
    eL = repeat (Line e empty e)
    takePlane (Plane u x d) =
        Plane (take n (u ++ eL)) x (take n (d ++ eL))
    takeLine  (Line l x r) =
        Line  (take n (l ++ e )) x (take n (r ++ e ))
```

`takeClosest a n p` vyextrahuje okolí velikosti n (v každém směru, tj. $(2n+1)^2$ bodů) bodu určeného `p`. Pokud se nacházíme u okraje obrázku, body, které by byly mimo obrázek, nahradíme `a`. Toto řešení je dostačující pro naše potřeby, ale pokud by bylo nutné rozeznávat okraje, tak můžeme vracet `[[Maybe a]]` místo `[[a]]` a místo nahrazení hodnotou `a` prostě použijeme konstruktor `Nothing`.

Tohle nám dává možnost jednoduše vyjádřit konvoluci: filtr definovaný čtvercovou maticí velikosti $2n+1$ a kus obrazu daný `takeClosest a n p` stačí po složkách vynásobit a pak sečíst a tím dostaneme hodnotu nového bodu na pozici určené `p`. Jednoduchá implementace této operace je např.:

```
conv :: (Num a) => (a -> b -> a) -> [[a]] -> [[b]] -> a
conv op a b = sum . map sum $ zipWith (zipWith op) a b
```

Všimněme si, že `conv` pracuje nad seznamy prvků různých typů. To se bude dále hodit při škálování RGB kanálů reálným číslem.

Nyní stačí definovat vhodnou matici: jednoduchá aproximace filtru pro Gaussovské rozostření může vypadat například takto:

```
simpleGauss :: [[Float]]
simpleGauss = map (map (/ s)) matrix
  where
    s      = sum (map sum matrix)
    matrix = [[1, 2, 1], [2, 4, 2], [1, 2, 1]]
```

Předtím, než napíšeme samotnou aplikaci filtru, musíme definovat startovací operaci, která z dvourozměrného seznamu vytvoří kontext, a ukončující operaci, která vezme vybraný prvek a jeho kontext a vytvoří z nich původní seznam.

Na pozici vybraného prvku víceméně nezáleží – zde popsaná komonáda je mnohem obecnější, ale této obecnosti zde nevyužíváme, následující implementace tedy vždy zvolí prvek v levém horním rohu (pokud existuje). Opět předpokládáme obdélníkový seznam. Pro lepší typovou bezpečnost by bylo vhodné definovat si nový datový typ, který bude reprezentovat takovéto seznamy, ale pro jednoduchost je tento příklad stavěn nad obyčejnými seznamy.

```
start :: [[a]] -> Maybe (Plane a)
start l
  | any null l = Nothing
  | otherwise  = Just (buildPlane l)
  where
    buildPlane ((l:ls):lss)
        = Plane [] (Line [] l ls) (buildLines lss)
    buildLines = map (\(l:ls) -> Line [] l ls)

end :: Plane a -> [[a]]
end = map plugLine . plugPlane
```

Teď již můžeme napsat aplikaci filtru:

```
applyGauss :: [[PixelRGB8]] -> [[PixelRGB8]]
applyGauss img = maybe img (end . extend gauss) (start img)
  where
    mult (PixelRGB8 a b c) x = PixelRGB8 (f a) (f b) (f c)
      where
        f a = floor (fromIntegral a * x)

    gauss ctx = conv mult (takeClosest 0 1 ctx) simpleGauss
```

Nejprve začneme tím, že z obrázku reprezentovaného dvourozměrným seznamem vytvoříme vhodný kontext (tj. `Plane`) pomocí funkce `start`. Takto získaný kontext duplikujeme: tím obohatíme každý prvek jeho pozicí v celém obrázku a můžeme tedy použít funkci `takeClosest`, která vyextrahuje nejbližší okolí daného prvku do dvourozměrného seznamu. Nyní již stačí (po prvcích) vynásobit takto získané matice s maticí Gaussovského rozostření a sečíst všechny hodnoty. Matice získaná aplikací `takeClosest` ale obsahuje RGB hodnoty, zatímco matice rozostření hodnoty

typu `Float`: potřebujeme tedy jakési násobení skalárem, k čemuž slouží pomocná funkce `mult`.

Duplikace a mapování jsou sloučené do jedné funkce `extend`. Pro připomenutí:

```
extend :: (Comonad w) => (w a -> b) -> (w a -> w b)
extend f = fmap f . duplicate
```

Všimněme si, že pokud chceme aplikovat více transformací, je vhodné zůstávat v komonádě, protože aplikace funkcí `end` a poté znovu `start` je značně náročná. Pokud bychom tedy chtěli např. aplikovat toto rozostření dvakrát, je mnohem lepší použít:

```
applyTwice :: [[PixelRGB8]] -> [[PixelRGB8]]
applyTwice img = maybe img (end . extend gauss . extend gauss)
    (start img)
```

Místo alternativního:

```
applyTwice = applyGauss . applyGauss
```

Na závěr poznamenejme, že reprezentace obrázku jakožto dvourozměrný (spojovaný) seznam není příliš vhodná pro obyčejné použití. Podobná reprezentace ale může být vysoce efektivní pro zpracování jednorozměrných signálů, kde seznamy nemají špatný vliv na efektivitu. Na druhou stranu tento příklad ukazuje použití dvourozměrné struktury jako komonády.

6.3 Celulární automaty

Další možnou aplikací komonád je simulace celulárních automatů. Tyto automaty se skládají z mřížky, množiny stavů a přechodového pravidla. Mřížka se skládá z buněk, které se nacházejí v jednom ze stavů; struktura mřížky může být různá, setkáváme se například s jednorozměrnými konečnými mřížkami, kde konec navazuje na počátek, nebo dvourozměrnými mřížkami, které jsou nekonečné ve všech směrech. Přechodové pravidlo určuje, jak se mění stav buňky v závislosti na jejím okolí. Zpočátku se buňky mřížky nacházejí ve startovním stavu, novou *generaci* vytvoříme tak, že aplikujeme pravidlo na všechny buňky současně. Připomeňme, že aplikace funkce pomocí `extend` je současná v každé komonádě.

Pro případ dvourozměrné nekonečné mřížky lze použít strukturu popisovanou v sekci 6.2. Místo seznamů `[a]` se použije `Stream a` a máme garanci, že mřížka bude skutečně nekonečná ve všech směrech. Pravidlo pak můžeme reprezentovat pomocí funkce typu `Plane a -> a`, které určuje stav dané buňky v nové generaci. Pomocí `extend` lze toto pravidlo rozšířit na funkci, která vytváří novou generaci mřížky.

Příkladem takovéhoto automatu je např. *Conway's Game of Life* [11]. Základem je dvourozměrná mřížka, jejíž buňky se nacházejí v jednom ze dvou stavů – *živá* nebo *mrtvá*. Přechodové pravidlo pak zkoumá 8 buněk v okolí (`takeClosest _ 1`): živá buňka, která má méně než dva nebo více než tři živé sousedy umírá (tj. její stav v

nové generaci je *mrtvá*), pokud má dva nebo tři živé sousedy, pak přežívá do další generace. Mrtvá buňka, která má právě tři živé sousedy, ožívá. Implementace tohoto pravidla pak může vypadat např. takto:

```
data State = Alive | Dead
    deriving (Show)

alive :: State -> Int
alive Alive = 1
alive _     = 0

countAlive :: [[State]] -> Int
countAlive xs = (sum . map alive . concat) xs - alive middle
  where
    middle = xs !! 1 !! 1

rule :: Plane State -> State
rule p = decide middle (countAlive area)
  where
    area   = takeClosest Dead 1 p
    middle = extract p

    decide Alive n
      | n >= 2 && n <= 3 = Alive
    decide Dead  n
      | n == 3           = Alive
    decide _     _       = Dead
```

V tomto příkladu se ale podíváme na jednorozměrné celulární automaty. Konkrétně nás zajímají ty, jejichž buňky se mohou nacházet ve dvou různých stavech a přechodové pravidlo zkoumá pouze dva nejbližší sousedy. Na vstupu pravidla se tedy nacházejí 3 buňky, máme tedy $2^3 = 8$ možných vstupů a každému vstupu může pravidlo přiřadit jeden ze dvou stavů. Celkový počet pravidel je tedy $2^8 = 256$.

Na identifikaci pravidla nám tedy stačí pouze 8 bitů, které můžeme reprezentovat primitivním typem `Word8` (z modulu `Data.Word`). Vezmeme všechny možné kombinace vstupních buněk – 111, 110, 101, 100, 011, 010, 001, 000 – a pak se podíváme na bity 8-bitového čísla, které určuje dané pravidlo. Nejvyšší bit říká, jaký je výsledný stav pokud je vstupem trojice buněk 111, druhý nejvyšší bit určuje výstup pro 110, atd. Toto kódování bylo navrženo a popsáno Stephen Wolframem v *A New Kind of Science* [12].

Než napíšeme vlastní implementaci převodu čísla na pravidlo, musíme si rozmyslet, jak bude vypadat mřížka. Je možné použít zipper ze sekce 6.1, ale v tomto příkladu použijeme odlišnou strukturu.

```
data Store s a
    = Store (s -> a) s

instance Functor (Store s) where
    fmap f (Store g s) = Store (f . g) s

instance Comonad (Store s) where
    extract   (Store f s) = f s
    duplicate (Store f s) = Store (Store f) s
```

`Store` obsahuje uložený stav typu `s` a funkci, která je schopná z daného stavu vyprodukovat hodnotu typu `a`. Lokální kontext je určen právě stavem `s`, který v určitém smyslu udává pozici v celé struktuře. Pěkným příkladem je `Stream a` – nekonečný seznam prvků typu `a` je možné reprezentovat jako funkci z přirozených čísel do `a`:

```
data Stream a = a :< Stream a

toStream :: (Nat -> a) -> Stream a
toStream f = map f (from 0)
  where
    from n = n :< from (n + 1)

fromStream :: Stream a -> Nat -> a
fromStream st n = st !! n

nthElem :: Stream a -> Nat -> Store Int a
nthElem st n = Store (fromStream st) n
```

Výše uvedenému kódu chybí vhodné definice typu `Nat` a operátoru `!!`, který operuje nad `Streamy`, ale je zde vidět, že lze snadno reprezentovat okolí *n*-tého prvku.

Pro naše účely se omezíme na funkce z `Int` do `Bool`. Okolí buňky tedy bude `Store Int Bool`, kde `Int` určuje pozici buňky a `Bool` je stav buněk. Jak využijeme hodnot uvnitř Store k přístupu k okolí dané buňky? Reprezentace `Stream a` jako `Nat -> a` dává nápovědu:

```
closest :: Store Int Bool -> [Bool]
closest (Store f s) = [f (s - 1), f s, f (s + 1)]
```

Nyní již máme vše potřebné pro implementaci převodu z `Word8` na pravidlo:

```
wolfram :: Word8 -> Store Int Bool -> Bool
wolfram rule (Store f s) = testBit rule area
  where
    fromBits = foldl'
        (\r b -> (if b then (+1) else id) (2 * r)) 0
    area     = fromBits [f (s - 1), f s, f (s + 1)]
```

Nejbližší okolí vybraného bodu převedeme na číslo tak, že interpretujeme `[Bool]` jako seznam bitů tohoto čísla – tento převod zajišťuje pomocná funkce `fromBits`. Pokud bylo okolí vybraného bodu např. `[False, True, False]`, dostaneme 2 a dle výše uvedeného popisu kódování se stačí podívat na druhý bit vstupního `Word8` (číslováno od nuly), k čemuž slouží funkce `testBit` (z modulu `Data.Bits`).

Pro vytvoření počátečního stavu mřížky můžeme použít `[Bool]` nebo dokonce `Stream Bool`. Zde se také nabízí několik variací: můžeme požadovat 2 hodnoty typu `Stream Bool`, abychom mohli postavit pásku nekonečnou v obou směrech nebo můžeme vyplnit zbytek pásky předem určenou hodnotou, atp. Implementace jsou celkem přímočaré: funkcí typu `Int -> Bool` bude varianta `!!` (s tím, že pro záporná čísla vrátíme pevně danou hodnotu nebo se podíváme do druhého seznamu).

Implementace použitá v příloze A.3 jednoduše vygeneruje náhodný seznam dané velikosti a zbytek vyplní hodnotami `False`.

```
start :: (RandomGen g) => g -> Int -> Store Int Bool
start gen size = Store look 0
  where
    list = take size (randoms gen)
    look n
      | n < 0     = False
      | n < size  = list !! n
      | otherwise = False
```

Obvykle se výstup těchto automatů zobrazuje do dvourozměrného pole, kde každý řádek odpovídá jedné generaci. Jednoduchý textový výstup např. přiřazuje hodnotám `True` znak `'#'` a hodnotám `False` pouze `' '`.

```
showLine :: Int -> Store Int Bool -> String
showLine size (Store f s) = map (toChar . f) [s .. s + size]
  where
    toChar b = if b then '#' else ' '
```

Je nemožné zobrazit celou pásku, proto se omezíme pouze na konečný úsek specifikovaný argumentem `size`. Vždy zobrazujeme pouze několik znaků vpravo od vybraného znaku (tj. ten, který je určený pozicí `s`).

Zbývá vyřešit přechod k nové generaci. Díky tomu, že `Store` je komonáda, máme k dispozici operaci `extend`, kterou můžeme aplikovat na pravidlo získané aplikací funkce `wolfram`. Výsledkem je funkce typu `Store Int Bool -> Store Int Bool`, kterou už lze jednoduše iterovat.

```
iterateRule :: (Store Int Bool -> Bool) -> Store Int Bool ->
               [Store Int Bool]
iterateRule rule = iterate (extend rule)

run :: Store Int Bool -> String
run = unlines . map (showLine 70) . iterateRule (wolfram 120)
```

Něco je ale v nepořádku. Prvních pár řádků se vygeneruje velice svižně, ale pak se výpočet znatelně zpomalí. Důvod je prostý: výpočet hodnoty jedné buňky na řádku n závisí na hodnotách 3 buněk na řádku n - 1, které závisí na 3^2 hodnotách na řádku n - 2, atp. Problém je ten, že pro `Store f s` se každá aplikace funkce `f` vyhodnocuje samostatně. Pokud tedy aplikujeme `f` dvakrát na tu samou hodnotu, výpočet se duplikuje. Nabízí se tedy tyto funkce memoizovat (tj. pro každé použití si uložíme výsledek do pomocné struktury a pokud by mělo dojít k opětovnému použití na dříve spočítaný výsledek, tak rovnou vrátíme hodnotu uloženou v pomocné struktuře). V tomto příkladu se ale memoizací nebudeme detailněji zabývat, použijeme tedy jednu z knihoven, která toto řeší za nás. V následujícím kódu je použita knihovna *data-memocombinators* [13].

```
iterateRule :: (Store Int Bool -> Bool) -> Store Int Bool ->
               [Store Int Bool]
iterateRule rule = iterate (extend rule . remember)
```

```
    where
      remember (Store f s) = Store (integral f) s
```

Předtím, než aplikujeme samotné pravidlo, tak provedeme memoizaci funkce `f` obsažené ve `Store f s` – toto opakujeme pro každý řádek a díky tomu se hodnota každé buňky bude počítat pouze jednou. Samotnou memoizaci obstarává funkce `integral`, která (jak jméno napovídá) zajišťuje memoizaci typů s instancí typové třídy `Integral`.

6.4 Cestování stromem

Další vhodnou aplikací zipperů je pohyb uvnitř *n*-árního stromu. Obvyklé imperativní řešení používá ukazatele pro určení pozice uvnitř stromu; pokud chceme tuto pozici změnit, je nutné provést destruktivní změnu hodnoty ukazatele, čemuž se chceme ve funkcionálním řešení vyhnout.

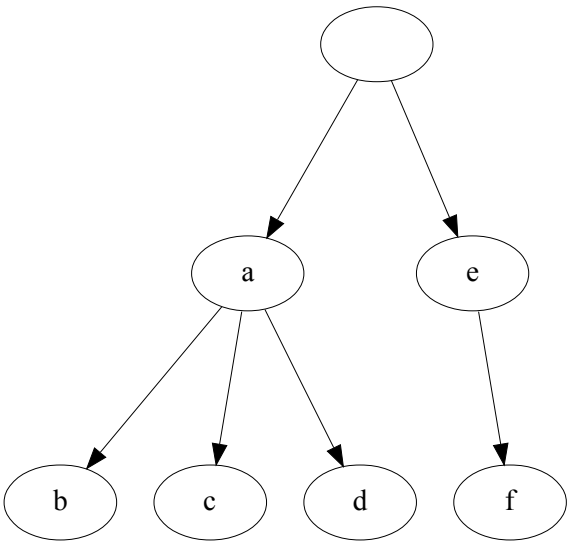
Použijeme strom, jehož uzly mohou mít libovolný počet potomků (takovýto typ stromu bývá znám pod názvem *Rose Tree*).

```
data Tree a = Node a [Tree a]
    deriving (Show, Eq)
```

Součástí vstupu je popis stromu: uzly jsou označeny posloupností písmen a jejich potomci jsou uvedeni v kulatých závorkách. Tato struktura tedy odpovídá *preorder* průchodu stromem. Kořen stromu má speciální hodnotu `""` (prázdný řetězec) a v popisu se neuvádí. Následující popis:

```
a (b c d) e (f)
```

odpovídá tomuto stromu:



Dále máme jednoduchý dotazovací jazyk pro pohyb uvnitř stromu. Základem jsou posloupnosti písmen (dále identifikátory) oddělené znakem `/`. Význam takového dotazu je následující: na začátku máme množinu uzlů, která obsahuje pouze kořen. Když narazíme na identifikátor, tak z množiny odstraníme všechny uzly a místo nich do množiny vložíme jejich potomky, jejichž hodnota odpovídá identifikátoru. Pokud je identifikátorem speciální hodnota `*` (která označuje, že chceme zahrnout všechny potomky), tak žádné filtrování neprobíhá; speciální hodnota `..` označuje, že místo potomků chceme rodiče (a množinu uzlů nenahrazujeme množinou jejich potomků, ale množinou jejich rodičů).

Každému identifikátoru lze přiřadit libovolný počet predikátů, které se zapisují do hranatých závorek. Predikáty nám umožňují dále měnit obsah množiny dvěma způsoby: pokud je predikátem číslo n, z množiny se vybere pouze n-tý prvek (pokud tato množina n-tý prvek nemá, výsledkem je prázdná množina), pokud je predikátem další dotaz, tak se tento dotaz provede pro každý prvek množiny a do výsledné množiny se zahrnou jen ty uzly, pro které dotaz vrátil jako výsledek neprázdnou množinu. Uveďme několik příkladů pro výše uvedený strom.

```
?> /*
[a,e]

?> /*/*
[b,c,d,f]

?> /*/*/..
[a,e]

?> /..
[]

?> /*[b]    -- všichni potomci kořene, kteří obsahují potomka „b“
[a]

?> /*[1]
[e]

?> /a/*[2]
[d]
```

Pravidla reprezentujeme pomocí tří datových typů:

```
data ActionType a
    = Parent
    | Children
    | Child a
    deriving (Show)

data Action a
    = Action (ActionType a) [Predicate a]
    deriving (Show)

data Predicate a
    = NthElement Int
    | Filter (Querry a)
    deriving (Show)
```

```
type Query a = [Action a]
```

Identifikátor spolu se všemi predikáty je reprezentován typem `Action`, typ identifikátoru určuje `ActionType` a vlastní predikát typ `Predicate`.

Samotné funkce pro zpracování vstupu zde neuvádíme; v příloze A.4 (resp. A.5) jsou implementovány pomocí knihovny *parsec* [14].

Podívejme se tedy na reprezentaci pozice uvnitř stromu. Nejdříve implementujeme operace přesně podle výsledné derivace a v další části pak navrhneme několik zlepšení.

$$\begin{aligned}
\text{List } a &= \mu x.\ 1 + a \times x \\
\text{Tree } a &= \mu x.\ a \times \text{List } x
\end{aligned}$$

$$\begin{aligned}
\partial_a(\mu x.\ 1 + a \times x) &= \mu y.\ \text{List } a + a \times y \\
&= \text{List } a \times \text{List } a
\end{aligned}$$

$$\begin{aligned}
\partial_a(\mu x.\ a \times \text{List } x) = \mu y.\ &\text{List (Tree } a) + a \times \text{List (Tree } a) \times \\
&\text{List (Tree } a) \times y
\end{aligned}$$

Z předchozího výpočtu vyplývá, že pozice prvku ve stromu lze určit pomocí tohoto datového typu:

```
data TreeContext a
    = End [Tree a]
    | Down a [Tree a] [Tree a] (TreeContext a)
    deriving (Show, Eq)
```

Pokud sestoupíme do určitého podstromu, můžeme si vybranou cestu poznamenat do konstruktoru `Down` tak, že uložíme hodnotu původního uzlu, seznam podstromů vlevo a vpravo od podstromu, kterým pokračujeme. Rekurzivní výskyt `TreeContext` nám pak umožňuje uložit cestu na nižší úrovni stromu. Jako příklad použijeme opět strom z úvodu a podíváme se, jak bychom uložili cestu k uzlu `c`:

```
path :: TreeContext String
path = Down ""  [] [Node "e" [Node "f" []]]  -- 1. úroveň
     ( Down "a" [Node "b" []] [Node "d" []]  -- 2. úroveň
     ( End      [] ))                        -- 3. úroveň
```

Jako obvykle je celý zipper tvořen kontextem a samotnou hodnotou:

```
data TreeZipper a = TreeZipper a (TreeContext a)
    deriving (Show, Eq)
```

Nyní se můžeme zaměřit na samotný průchod stromem. Pro implementaci dotazovacího jazyka potřebujeme pohyb nahoru a dolů, pro pohyb dolů také

potřebujeme filtrování výsledných uzlů. Díky struktuře stromu máme vždy k dispozici kořen, vždy tedy můžeme vytvořit zipper, který odpovídá pozici kořene:

```
start :: Tree a -> TreeZipper a
start (Node a l) = TreeZipper a (End l)
```

Cestování směrem dolů je snadné: z hodnoty typu `TreeZipper a` vyextrahujeme `TreeContext a`, který projdeme až na konec ke konstruktoru `End`, který obsahuje seznam všech podstromů. Stačí tedy rozdělit tento seznam všemi možnými způsoby na seznamy podstromů vlevo a vpravo od nově zvoleného podstromu. K tomuto rozdělení slouží funkce `picks`:

```
picks :: [a] -> [(a, [a], [a])]
picks []     = []
picks (x:xs) = (x,[],xs):[(a, x:l, r) | (a,l,r) <- picks xs]

?> picks [1,2,3]
[(1,[],[2,3]),(2,[1],[3]),(3,[1,2],[])]
```

Jediná věc, která poněkud znepříjemňuje implementaci, je nutnost znovu postavit `TreeZipper`. Při rekurzivním průchodu si musíme ukládat hodnoty uzlů a pomocí nich pak na konci postavit `TreeZipper`.

```
downWith :: (a -> Bool) -> TreeZipper a -> [TreeZipper a]
downWith p (TreeZipper x ctx) = map (uncurry TreeZipper) (helper
ctx)
  where
    helper (End xs) = [(a, Down x l r (End d))
        | (Node a d, l, r) <- picks xs, p a]
    helper (Down a l r rest) =
        map (second (Down a l r)) (helper rest)
```

Cestování nahoru je poněkud náročnější. Stejně jako v předchozím případě musíme rekurzivně projít hodnotou typu `TreeContext a`, ale jakmile narazíme na konstruktor `End`, musíme se posunout o úroveň výše – musíme tedy vyhodit jeden z konstruktorů `Down`. V zásadě máme dvě možnosti: při průchodu můžeme kontrolovat dva konstruktory (místo pouze jednoho konstruktoru) nebo si předchozí konstruktor můžeme předávat v dalším argumentu. V následujícím kódu je použit druhý přístup.

```
up :: TreeZipper a -> [TreeZipper a]
up (TreeZipper x ctx) = map (uncurry TreeZipper)
                            (helper ctx Nothing)
  where
    helper (End _)           Nothing
        = []
    helper (End xs)          (Just (y, l, r))
        = [(y, End (l ++ [Node x xs] ++ r))]
    helper (Down y l r rest) Nothing
        = helper rest (Just (y, l, r))
    helper (Down y l r rest) (Just (y', l', r'))
        = map (second (Down y' l' r'))
              (helper rest (Just (y, l, r)))
```

Pokud předchozí konstruktor neexistuje, je druhým argumentem pomocné funkce `helper` hodnota `Nothing`. V případě, že narazíme na konec cesty a předchozí vrchol neexistuje, tak vracíme prázdný list – tento případ nastane pouze pokud je vybraným uzlem kořen, který již nemá žádného předka.

Teď již máme vše potřebné pro interpretaci vstupního dotazu. Množinu uzlů reprezentujeme jednoduše jako seznam zipperů (což je také důvod, proč funkce `up` vrací seznam místo vhodnějšího `Maybe`). Identifikátor se převede na volání funkce `downWith`, speciální hodnoty `*` a `..` se převedou na `downWith (const True)` a `up`:

```
parents :: Eq a => [TreeZipper a] -> [TreeZipper a]
parents = nub . concatMap up

childrenWith :: (a -> Bool) -> [TreeZipper a] -> [TreeZipper a]
childrenWith = concatMap . downWith

children :: [TreeZipper a] -> [TreeZipper a]
children = childrenWith (const True)
```

Vyhodnocování bude tedy vypadat následovně:

```
eval :: Eq a => Query a -> Tree a -> [TreeZipper a]
eval query = evalAllA query . return . start
  where
```

Vyhodnocení jedné akce (tj. hodnoty typu `Action`) odpovídá výše uvedenému popisu:

```
evalA (Action a ps) = evalAllP ps . toA a
  where
    toA Parent    = parents
    toA Children  = children
    toA (Child c) = childrenWith (== c)
```

Vyhodnocení predikátu je náročnější. Pro případ, kdy je predikátem `NthElement n`, stačí vzít n-tý prvek seznamu (pokud existuje), pokud je ale predikátem `Filter q`, musíme rekurzivně vyhodnotit dotaz `q` pro každý prvek seznamu a zařadit pouze ty prvky, pro které je výsledek dotazu neprázdný:

```
evalP (NthElement n) = take 1 . drop n
evalP (Filter q)     = filter
                         (not . null . evalAllA q . return)
```

Všimněme si, že výsledkem pomocných funkcí `evalA` a `evalP` jsou další funkce. Pokud chceme vyhodnotit všechny akce a všechny predikáty, stačí aplikovat `evalA` na každou akci, `evalP` na každý predikát a všechny tyto funkce složit. Pro skládání všech funkcí v seznamu slouží pomocná funkce `ffold`:

```
ffold = foldr (>>>) id

-- ekvivalentně
ffold = foldr (flip (.)) id
```

```
evalAllA = ffold . map evalA
evalAllP = ffold . map evalP
```

Hodnoty uzlů se ve stromu mohou opakovat, proto pro výstup nestačí pouze vypsat tuto hodnotu. Uzly budeme tedy identifikovat plnou cestou z kořene stromu, uzel `c` v úvodním stromě by byl reprezentován takto:

```
?> printPath c
/a/c
```

V jedné úrovni se ale hodnoty uzlů mohou také opakovat, opatříme tedy uzly na každé úrovni číslem, které vyjadřuje pozici uzlu v úrovni (indexováno zleva od nuly):

```
?> printPath c
/a(0)/c(1)
```

Jelikož zipper přesně určuje pozici jednoho uzlu, lze očekávat, že budeme také schopni vyextrahovat cestu, která k němu vede. Potřebujeme tedy z hodnoty typu `TreeZipper a` vytvořit seznam `[(a, Int)]`, který označuje hodnoty a pozice uzlů v každé úrovni:

```
path :: TreeZipper a -> [(a, Int)]
path (TreeZipper x ctx) = helper ctx 0
  where
    helper (End _)           ix = [(x, ix)]
    helper (Down a l _ rest) ix = (a, ix):helper rest (length l)


?> path c
[("", 0), ("a", 0), ("c", 1)]
```

Převod do čitelnější a kompaktnější podoby obstarává funkce `printPath`, která zde není dále rozebrána.

Zaměříme se na problémy této reprezentace. Pokud bychom chtěli pouze převádět zipper na popis cesty, je tato reprezentace vhodná – první vrstva typu `TreeContext` odpovídá právě první úrovni stromu. Není tedy nutné hledat konstruktor `End` a pokračovat odsud zpět nahoru. Na druhou stranu nemáme efektivní pohyb stromem, obě operace `downWith` a `up` jsou $O(k)$, kde k je hloubka vybraného uzlu (`downWith` je navíc o něco pomalejší kvůli použití funkce `picks` a případnému filtrování).

Problém je ten, že údaje o částech, které jsou blízko vybranému uzlu, jsou uložené hluboko ve struktuře. Řešení je ale snadné: stačí tuto strukturu otočit tak, aby lokální informace byly snadno dostupné. Použijeme tedy alternativní definici derivace μ a dostáváme následující typ:

$$\partial_a(\text{Tree } a) = \text{List } (a \times \text{List } (\text{Tree } a) \times \text{List } (\text{Tree } a)) \times \text{List } (\text{Tree } a)$$

Nová reprezentace tedy vypadá takto:

```
data TreeStep a
    = Step a [Tree a] [Tree a]
    deriving (Show, Eq)

type TreeContext a = ([TreeStep a], [Tree a])

data TreeZipper a = TreeZipper a (TreeContext a)
    deriving (Show, Eq)
```

Jaká je ale výhoda? Díky tomu, že jsme byli schopni transformovat konstruktor `Down` na pouhý prvek seznamu, můžeme změnit pořadí těchto konstruktorů tak, že seznam budeme ukládat v obráceném pořadí – hodnoty, které byly původně hluboko uvnitř celé struktury, se dostanou na povrch. Jediné funkce, které je nutné změnit, jsou `start`, `downWith`, `up` a `path`. Samotné implementace jsou mnohem přehlednější a efektivnější:

```
start :: Tree a -> TreeZipper a
start (Node a l) = TreeZipper a ([], l)

downWith :: (a -> Bool) -> TreeZipper a -> [TreeZipper a]
downWith p (TreeZipper x' (prev, next)) =
    [TreeZipper x (Step x' l r:prev, lower)
        | (Node x lower, l, r) <- picks next, p x]

up :: TreeZipper a -> [TreeZipper a]
up (TreeZipper x' ([], _)) = []
up (TreeZipper x' (Step x l r:prev, lower)) =
    [TreeZipper x (prev, l ++ [Node x' lower] ++ r)]
```

Funkci `path` ale tato reprezentace nevyhovuje, musíme tedy kontext převést do původní podoby, kde prvním prvkem seznamu je hodnota odpovídající první úrovni stromu. Další možností je použít akumulátor.

```
path :: TreeZipper a -> [(a, Int)]
path (TreeZipper x (prev, _)) = helper (reverse prev) 0
  where
    helper []                ix = [(x, ix)]
    helper (Step a l r:rest) ix = (a, ix):helper rest (length l)
```

Závěr

Kontexty se ve funkcionálním programováním vyskytují v mnoha podobách – od jednoduchých jako je triviální kontext či jediná hodnota, ke složitým jako je celá datová struktura. Komonády se ukazují jako užitečný nástroj pro řešení mnoha úloh, které vyžadují informace o určitém kontextu – ať jsou to celulární automaty, zpracování signálu nebo jiné.

Taktéž jsme ukázali, že zipper lze automaticky určit pro každý regulární datový typ a to právě díky derivaci, (téměř) tak jak ji známe z matematické analýzy. Mimo jiné lze tento postup automatizovat i na úrovni jazyka – v případě Haskellu je možné automaticky vytvářet zipper pomocí `TemplateHaskell` nebo dokonce i jednodušší `TypeFamilies`.

Díky tomu, že se na zippery můžeme dívat jako na hodnotu spolu se zbytkem původní struktury, která určuje polohu této hodnoty, se zipper chová jako komonáda. Tento fakt nám umožňuje pohodlněji pracovat právě s kontextem vybrané hodnoty, mimo jiné komonadické operace umožňují pohyb po celé struktuře.

Zbývá však několik dalších otázek. Podařilo se nám sice napsat generickou definici pro operaci `plug`, ale jak (nebo dokonce jestli vůbec) lze napsat podobnou definici pro komonadickou operaci `duplicate` (jelikož definice `extract` je triviální) je zatím nevyřešené.

Stejně tak jako máme monadické transformátory pro obyčejné monády, tak jsou také komonadické transformátory pro komonády. Několik často používaných transformátorů je k nalezení v knihovně *comonad* [6]. Ale zatímco transformátory pro monády mají jasné použití a význam (tj. seskupování efektů), použití komonadických transformátorů bývá spíše *ad-hoc*.

Případná generalizace zipperu pro jednu vybranou hodnotu by mohl být zipper pro n hodnot. Ale námi popsaný zipper tuto možnost neposkytuje; je možné, že lze definovat jinou strukturu, která připomíná zipper a umožňuje ukládat více než jednu pozici uvnitř struktury a také s nimi efektivně manipulovat.

Derivaci jsme sice zavedli na regulárních datových typech, ale pro některé neregulární typy by se tato derivace dala zavést také:

$$\begin{aligned} \text{Bool} \to A &= A \times A \\ \partial_A(\text{Bool} \to A) &= 2 \times A \end{aligned}$$

Tohle nás zavádí k otázce, kdy takováto derivace vůbec dává smysl, případě jestli je možné interpretovat unární negaci nebo převrácenou hodnotu ve světě typů [15].

Jistě také existují jiné možnosti využití komonád kromě těchto námi popsaných. Příkladem může být použití komonád pro sledování koefektů [16] (tak jako jsou monády použity ke sledování efektů).

Případně jestli je možné pro komonády vytvořit syntaktický cukr, který by odpovídal „`do`“ notaci pro monády. Dnes již existuje několik nápadů pro tuto „`co-do`“ notaci, například [17].

Seznam použité literatury

[1] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

[2] Agda developers. *Agda* programming language wiki. `http://wiki.portal.chalmers.se/agda/pmwiki.php`

[3] William A. Howard. *The formulae-as-types notion of construction*. To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, Academic Press, pp. 479–490, ISBN 978-0-12-349050-6.

[4] Simon Marlow, ed. *Haskell 2010 Language Report*. 2010. `http://www.haskell.org/definition/haskell2010.pdf`

[5] Simon Peyton Jones, ed. *Haskell 98 Language and Libraries: The Revised Report*. 2003. `http://www.haskell.org/definition/haskell98-report.pdf`

[6] Edward A. Kmett. *comonad* package. `http://hackage.haskell.org/package/comonad`

[7] Edward A. Kmett. *monad-products* package. `http://hackage.haskell.org/package/monad-products`

[8] Gérard Huet. *The Zipper.* Journal of Functional Programming, 7 (5), Sep 1997, pp. 549–554.

[9] Conor McBride. *The Derivative of a Regular Type is its Type of One-Hole Contexts*. `http://strictlypositive.org/diff.pdf`

[10] Vincent Berthoux. *JuicyPixels* package. `http://hackage.haskell.org/package/JuicyPixels`

[11] Martin Gardner. *The fantastic combinations of John Conway's new solitaire game "life"*. Scientific American, 223, Oct 1970, pp. 120–123.

[12] Stephen Wolfram. *A New Kind of Science*. 2002. ISBN 1-57955-008-8.

[13] Luke Palmer. *data-memocombinators* package. `http://hackage.haskell.org/package/data-memocombinators`

[14] Daan Leijen, Paolo Martini. *parsec* package. `http://hackage.haskell.org/package/parsec`

[15] Roshan P. James, Amr Sabry. *The Two Dualities of Computation: Negative and Fractional Types*. 2012. `http://www.cs.indiana.edu/~sabry/papers/rational.pdf`

[16] Tomas Petricek, Dominic Orchard, Alan Mycroft. *Coeffects: Unified static analysis of context-dependence*. `http://www.cl.cam.ac.uk/~dao29/publ/coeffects-icalp13.pdf`

[17] Gabriel Gonzalez. *Comonads are objects*.
http://www.haskellforall.com/2013/02/you-could-have-invented-comonads.html

Přílohy

A Haskell

A.1 Modul `Turing`

Tento modul se nachází v souboru `appendix/haskell/src/Turing.hs` a obsahuje implementaci řešeného příkladu 6.1.

Hodnota `mainTuring` spočítá počet kroků 5-stavového automatu a vypíše ho na standardní výstup; kvůli výpočetní náročnosti v interpretovaném režimu doporučujeme tento modul zkompilovat (stačí zvolit `main = mainTuring`).

A.2 Modul `ImageFilter`

Tento modul se nachází v souboru `appendix/haskell/src/ImageFilter.hs` a obsahuje implementaci řešeného příkladu 6.2.

Hodnota `mainImageFilter` aplikuje Gaussovský filtr na obraz `../data/in.bmp` a výsledný obraz uloží do stejné složky pod jménem `out.bmp`.

A.3 Modul `Automata`

Tento modul se nachází v souboru `appendix/haskell/src/Automata.hs` a obsahuje implementaci řešeného příkladu 6.3.

Hodnota `mainAutomata` vykreslí na obrazovku prvních 40 řádek jednorozměrného celulárního automatu, který odpovídá číslu 122 (lze změnit taktéž v hodnotě `mainAutomata`). První řádka je určena náhodně.

A.4 Modul `SPath`

Tento modul se nachází v souboru `appendix/haskell/src/SPath.hs` a obsahuje implementaci řešeného příkladu 6.4.

Hodnota `mainSPath` načte strom uložený v souboru `../data/data.in`, dotaz uložený v souboru `../data/query.in` a výsledek tohoto dotazu vypíše na standardní výstup. Formát vstupních dat je popsán v kapitole 6.4.

A.5 Modul `SPathFast`

Tento modul se nachází v souboru `appendix/haskell/src/SPathFast.hs` a jedná se o vylepšení předchozího modulu `SPath` (A.5), metodou popsanou taktéž v kapitole 6.4.

B Agda

Níže uvedené moduly jsou také dostupné se zvýrazněným kódem ve složce `appendix/agda/html`.

B.1 Modul `TypeAlgebra`

Tento modul se nachází v souboru `appendix/agda/src/TypeAlgebra.agda` a obsahuje formulaci a důkaz tvrzení o polookruhu typů spolu s operacemi +, × a s konstantami 0, 1.

B.2 Modul `Comonad.Definition`

Tento modul se nachází v souboru `appendix/agda/src/Comonad/Definition.agda` a obsahuje definici operací komonády spolu s vlastnostmi, který by tyto operace měly splňovat. `Comonad.Definition` obsahuje definici pomocí dvojice `extract`, `extend` a také definici pomocí trojice `fmap`, `extract`, `duplicate`.

B.3 Modul `Comonad.Equivalence`

Tento modul se nachází v souboru `appendix/agda/src/Comonad/Equivalence.agda` a obsahuje formulaci a důkaz tvrzení o ekvivalenci dvou alternativních definic komonády (viz B.2).

B.4 Modul `Comonad.Cokleisli`

Tento modul se nachází v souboru `appendix/agda/src/Comonad/Cokleisli.agda` a obsahuje důkaz, že typ `Cokleisli` definovaný v kapitole 3.2 skutečně tvoří kategorii.

B.5 Modul `Comonad.Examples`

Tento modul se nachází v souboru `appendix/agda/src/Comonad/Examples.agda` a obsahuje příklady různých komonád z kapitoly 3.3 spolu s důkazy, že tyto operace splňují požadované vlastnosti.