GYMNASIUM JANA KEPLERA

Parléřova 2/118, 169 00 Praha 6

eru — Editor opetopických komplexů

Maturitní práce

Autor: Jan Růžička

Třída: 4.A

Školní rok: 2020/2021

Předmět: Informatika

Vedoucí práce: Šimon Schierreich

Praha, 2021

GYMNASIUM JANA KEPLERA
Kabinet informatiky

ZADÁNÍ MATURITNÍ PRÁCE

Student:	**Jan Růžička**
Třída:	**4.A**
Školní rok:	**2020/2021**
Platnost zadání:	**30. 9. 2021**
Vedoucí práce:	**Šimon Schierreich**
Název práce:	**Editor opetopických komplexů**

Pokyny pro vypracování:

Cílem práce je vyvinout nativní aplikaci pro Linux a macOS, která slouží jako grafický editor opetopických komplexů. Mezi hlavní funkce patří responzivní úpravy samotných komplexů, dekorování jejich komponentů o metadata a kontrola správnosti konstrukce.

Doporučená literatura:

[1] KOCK, Joachim, André JOYAL, Michael BATANIN a Jean-François MASCARI. Polynomial functors and opetopes. Advances in Mathematics. 2010, 224(6), 2690-2737. ISSN 0001-8708. Dostupné z: https://doi.org/10.1016/j.aim.2010.02.012

[2] BAEZ, John C. a James DOLAN. Higher-Dimensional Algebra III. n-Categories and the Algebra of Opetopes. Advances in Mathematics. 1998, 135(2), 145-206. ISSN 0001-8708. Dostupné z: https://doi.org/10.1006/aima.1997.1695

[3] CHENG, Eugenia. Weak n-categories: opetopic and multitopic foundations. Journal of Pure and Applied Algebra. 2004, 186(2), 109-137. ISSN 0022-4049. Dostupné z: https://doi.org/10.1016/S0022-4049(03)00139-7

URL repozitáře:

https://gitlab.com/opetopes/eru-app

vedoucí práce *student*

V Praze dne 26. 10. 2020

Prohlášení

Prohlašuji, že jsem svou práci vypracoval samostatně a použil jsem pouze prameny a literaturu uvedené v seznamu bibliografických záznamů. Nemám žádné námitky proti zpřístupňování této práce v souladu se zákonem č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších předpisů.

V Praze dne 7. dubna 2021 Jan Růžička

Poděkování

Chtěl bych poděkovat svému vedoucímu práce za poskytnuté rady a konzultace. Také jsem vděčný své rodině a přátelům za podporu v průběhu zpracovávání projektu. V neposlední řadě mé díky patří PhD. Ericu Finsterovi za jeho odborné rady ohledně konstrukčních vlastností opetopů.

Abstrakt

V práci definujeme pojem opetopického komplexu. Dále rozvedeme některé operace, které na ně lze uplatnit. Praktickou částí je editor umožňující opetopické komplexy konstruovat a manipulovat s nimi. Ten je v této práci zdokumentován.

Klíčová slova

opetop, opetopický komplex, editor, důkazový asistent

Abstract

The notion of opetopic complex is defined. Some operations applicable to them are further clarified. The practical part comprises an editor which enables one to construct and manipulate opetopic complexes. This editor is documented in this paper.

Keywords

opetope, opetopic complex, editor, proof assistant

Obsah

1. Teoretická část

1.1 Úvod

Opetopy jsou matematické objekty reprezentující n-ární poly- nebo monomorfické funkce.

Motivace k jejich vzniku vzešla z vyšší teorie kategorií. V teorii kategorií je zvykem reprezentovat soustavy rovnic / rovností mezi funkcemi jako tzv. *komutativní diagramy*, tj. grafy, jejichž hrany jsou funkce a vrcholy obory daných funkcí. Ovšem ve vyšší teorii kategorií jsou funkce zobecněny na *transformace*, přičemž obory transformací mohou být i transformace nižšího řádu — objekty ve vyšší kategorii jsou nazývány 0-transformace, funkce 1-transformace a mezi dvěma n-transformacemi může vést $n + 1$-transformace.

Pokud člověk pracuje s transformacemi na několika úrovních, komutativní diagramy se stávají nepřehlednými. Navíc je nelze snadno strojově zpracovat, a je tedy obtížné vytvořit pro vyšší teorii kategorií asistenční důkazový systém.

Opetopy jsou jedním z přístupů, které oba problémy řeší.

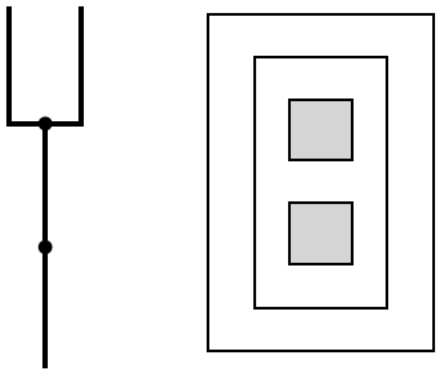
1.2 Definice

Pojem, který v definici opetopů hraje ústřední roli, je *strom*. Tím se zde myslí souvislý acyklický graf. Pojem strom ještě dále specializujeme na *buněčný strom* a *hranový strom*. Jde o dvě reprezentace stromů, a sice reprezentaci vnořenými množinami a reprezentaci grafem.

Další odlišností mezi nimi je, že zatímco u buněčného stromu nám záleží pouze na samotné stromové struktuře, u hranového záleží i na podkladovém grafu — zajímají nás hlavně hrany, z důvodu popsaného níže.

Pozn.: Podstromem hranového stromu se dále bude rozumět jeho souvislý podgraf.

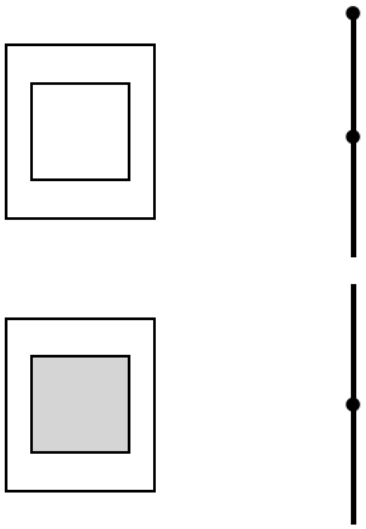
Příklad:



Obrázek 1.1: Dva stejné stromy; vlevo hranový, vpravo buněčný

V buněčných stromech značíme listy šedou barvou, zatímco větvení bílou; v hranových stromech jakákoliv hrana končící tečkou je větvení.

Tedy následující dvojice stromů jsou různé, ale buněčný strom vlevo je vždy stejný jako ten hranový napravo od něj:



Obrázek 1.2: Dvojice různých stromů

Nad stromy také můžeme zavést částečné uspořádání podle relace „bytí podstromem“. Například na předchozím obrázku je šedá buňka menší než bílá buňka kolem ní.

Nyní zaveďme tzv. *atomický diagram*. Atomický diagram je dvojice stromů (jeden nahlížíme jako buněčný a druhý jako hranový) spolu se zobrazením mezi množinami jejich podstromů, které je neklesající s ohledem na uspořádání dané relací „podstromovosti“.

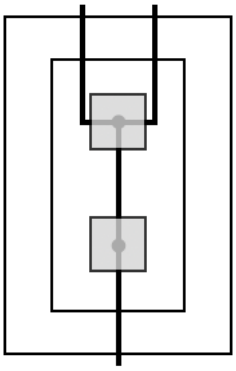
Tomuto zobrazení říkejme *kostra*.

Dále pro ně platí několik podmínek:

1. kostra listu (šedá buňka v popředí) je pouze vrchol hranového stromu,
2. mají-li dva buněčné podstromy A a B kostry, jejichž průnik obsahuje vrchol, potom platí buď $A \leqslant B$, nebo $B \leqslant A$.

Kostru si lze představit následovně: nakreslíme buněčný a hranový strom přes sebe tak, že listy buněčného stromu jsou v popředí a větve buněčného stromu jsou v pozadí a zároveň každá buňka má ve svém obsahu svoji kostru.

Příklad:

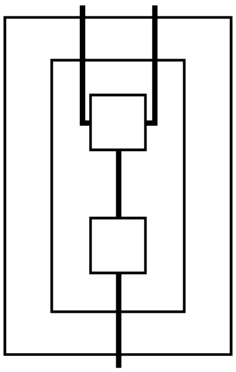


Obrázek 1.3: Atomický diagram

Zde má každý buněčný list za kostru jeden z vrcholů. Obě buněčné větve mají za kostru celý hranový strom.

Jelikož v atomickém diagramu lze buněčné listy odlišit od buněčných větví pomocí toho, že jsou v popředí před hranovým stromem (dále je budeme kreslit neprůhledně), nemusíme je již odlišovat barvou. Budeme je tedy kreslit bílé.

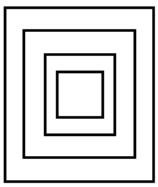
Po jmenovaných stylistických úpravách bude tentýž diagram vypadat takto:



Obrázek 1.4: Atomický diagram po grafické úpravě

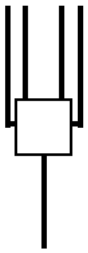
Nyní se dostáváme k poslední části definice, kterou je *opetop*. Opetop je posloupnost atomických diagramů, která splňuje následující:

1. Buněčný strom prvního diagramu je lineární. Vypadá tedy nějak takto:



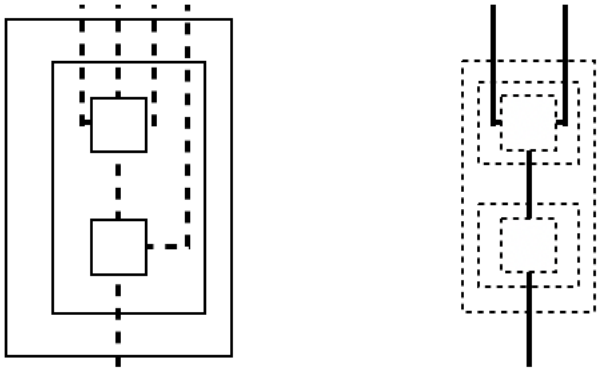
Obrázek 1.5: Lineární buněčný strom

2. Buněčný strom posledního diagramu sestává z jediného listu. Vypadá tedy nějak takto:



Obrázek 1.6: Svrchní vrstva opetopu

3. Pro každé dva po sobě následující atomické diagramy A a B platí, že buněčný strom A je izomorfní hranovému stromu B. Tedy například tato dvojice podmínku splňuje:



Obrázek 1.7: Dvě následující vrstvy opetopu

1.3 Vlastnosti

Jak již bylo řečeno, opetopy se dají interpretovat jako transformace v nějaké vyšší kategorii, což mimo jiné znamená, že samy určitou kategorii *tvoří*. Zde tedy rozebereme nějaké vlastnosti opetopů, které bychom jim rádi přisoudili (ač tyto vlastnosti nesouvisejí se strukturou opetopům vlastní, ale se strukturou vnější).

Kategorii $\mathcal{O}$ nazýváme *kategorie opetopů*. Její objekty jsou opetopy. Mezi opetopy A a B vede morfismus $A \to B$, právě když je A obsažen v B, tedy pokud se v B nachází buňka, jejíž *tvar* je totožný s A.

Tvar buňky lze získat následovně (C-like pseudokód):

```
Opetope tvar(Cell X) {
    if X in first atomic diagram {
        return X;

    } else {
        Cell X2 = X;

        X2.inputs = X.inputs.map(tvar);
        X2.output = tvar(X.output);

        return X2;
    }
}
```

Tedy tvar buňky je získán tak, že se zkopíruje buňka a její vstupy a výstupy (rekurzivně). Tato operace koresponduje k jakémusi "vynětí"dané buňky z mateřského opetopu.

Závěrem této sekce podáme poloformální důkaz, že výstupem této operace je vždy validní opetop. Důkaz povedeme indukcí přes vrstvy.

Pokud je buňka v 1. vrstvě, je důkaz triviální. Předpokládejme, že tvar každé buňky z n-té vrstvy je validní opetop. Vezmeme-li tvar buňky z $n+1$-ní vrstvy, musíme spočíst tvar jejích vstupů a výstupu. To jsou buňky v n-té vrstvě, tedy jejich tvary jsou validní opetopy.

Zároveň buňka odpovídající výstupu obsahuje všechny (a výhradně) buňky odpovídající vstupům. Z definice opetopu se tedy musí kostra výstupní buňky překrývat s kostrami vstupních buněk. Takže vstupní buňky musí tvořit souvislý (hranový, pomocí svých vstupů a výstupů) strom.

Tedy výsledkem je skutečně validní opetop.

Pozn. 1: Jelikož opetopy tvoří kategorii, ale jakožto struktury jsou schopny modelovat zobrazení a transformace (proto mluvíme o vstupech a výstupech), kategorii opetopů lze modelovat v ní samé (toto tvrzení platí dokonce pro obecnější struktury, nazývané operády [1]). Pomocí opetopů lze dokonce formalizovat i teorii typů [3], ač se tímto v této práci nebudeme zabývat.

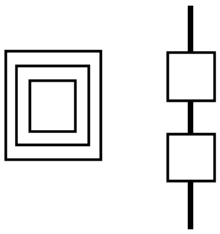
Pozn. 2: Na opetopech lze definovat ještě jednu operaci: *suspenzi*. Mějme opetop X. Jeho suspenze, značená ΣX, je opetop, jehož první diagram obsahuje lineární buněčný strom o dvou buňkách, druhý diagram má stejný buněčný strom jako první diagram v X a hranový strom s jediným uzlem a každý $n+2$-hý diagram v ΣX je shodný s $n+1$-ním diagramem v X.

Výstupem této operace je opět validní opetop, avšak díky ní můžeme definovat proces konstrukce opetopů s čím dál větším počtem vrstev, jehož limitou je množina tzv. *stabilních* opetopů, jejímiž prvky jsou jak opetopy konečné, tak nekonečné. Pro více informací a relevantní konstrukce viz [4].

1.4 Úpravy

V této sekci jsou popsány jednotlivé úpravy, kterými lze opetopy transformovat.

Popisované úpravy budou často pracovat s nevalidními opetopy: často se setkáme s opetopem, jehož svrchní vrstva neobsahuje jediný buněčný strom, ale les buněčných stromů. Například tento:



Obrázek 1.8: Opetop s neúplnou svrchní vrstvou

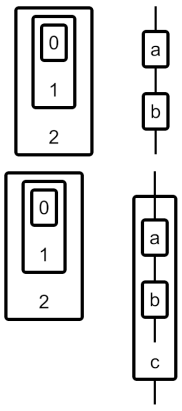
Těmto neúplným opetopům budeme říkat *skoro-opetopy*.

Úpravy opetopů úzce souvisí s logickým rámcem, v němž pracujeme. Opetopy jsou v základu vlastně datová struktura: každá buňka nese určitá data. Náš logický rámec pak obsahuje doménu možných symbolů či dat, kterými můžeme buňky osadit.

Pro účely této práce předpokládáme, že všechny použité symboly jsou v této doméně obsaženy. Pro účely důkazových asistentů a logických či matematických systémů se však hodí doménu symbolů omezit a místo toho přidat pravidla pro generování symbolů na základě aktuální struktury upravovaného opetopu (viz [2]).

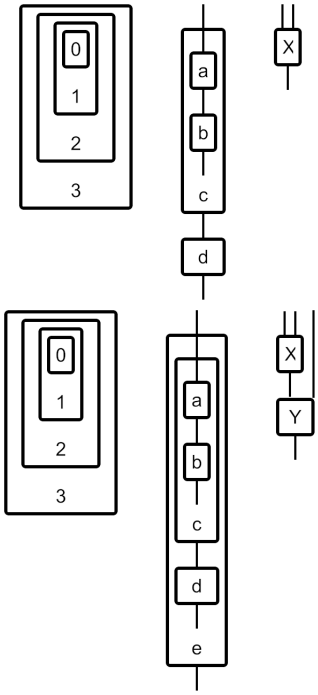
1.4.1 Seskupení

Máme-li v nějaké vrstvě množinu buněk, jejichž kostry dohromady tvoří souvislý hranový strom a zároveň mají společného rodiče v buněčném stromu, můžeme je vnořit do nové buňky (1.9). V následujícím diagramu se tato akce projeví tím, že se hrany, které odpovídají seskupeným buňkám, spojí novou buňkou (uzlem).



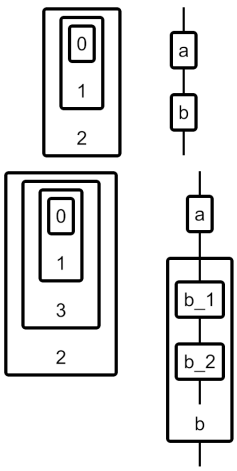
Obrázek 1.9: Buňky před seskupením a po seskupení

Pokud žádná ze seskupených buněk neměla rodiče (takže jsme jim touto operací poskytli prvního), potom stačí v následujícím diagramu připojit buněčný list "dospodu". Jeho vstupy jsou realizace seskupených buněk a jeho výstup je realizace vzniklé skupiny (1.10).



Obrázek 1.10: Buňka se vloží dospodu

V opačném případě vezmeme buňku, která má za vstupy realizace seskupených buněk, a vnoříme do ní dva nové buněčné listy: horní bude mít za výstup nově vzniklou skupinu a výstup spodního bude odpovídat starému společnému rodiči seskupených buněk (1.11).

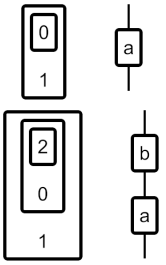


Obrázek 1.11: Buňka se "rozdělí"na dvě

1.4.2 Vetknutí

Tato operace dává smysl pouze pro buněčné listy. Vezmeme-li buněčný list L, pak do něj můžeme vnořit nějaký diagram buněk, který má stejnou signaturu. V následující vrstvě pak stačí přidat

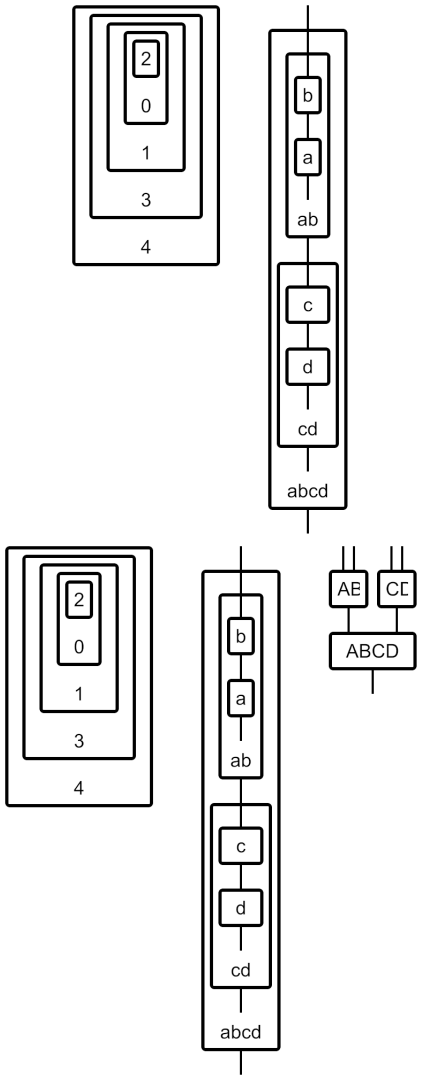
buňku, jejíž výstup je L a jejíž vstupy jsou diagram, který jsme vetknuli do L (1.12).



Obrázek 1.12: Buňka se vloží nahoru

1.4.3 Další vrstva

Máme-li nějaký skoro-opetop O, potom můžeme vytvořit další skoro-opetop O′ tak, že vezmeme buněčný strom O a převedeme jej na hranový strom O′, přičemž pro každý uzel tohoto hranového stromu musíme vytvořit buněčný list (1.13). Pokud má hranový strom O′ jediný uzel, dostali jsme takto validní opetop O′.



Obrázek 1.13: Buněčný strom se převede na hranový

2. Implementace

Implementace byla členěna do následujících fází:

1. datová reprezentace opetopů
2. výběr ekosystému na tvorbu aplikace
3. pomocné struktury
4. zobrazování opetopů

Následující sekce obsahují reflexi a informace o každé fázi.

2.1 Datová reprezentace opetopů

Vlastnosti opetopů umožňují překvapivě rozmanitou škálu možných reprezentací, přičemž každá má své výhody a nevýhody ve vztahu ke kompaktnosti reprezentace a ke složitosti implementace jednotlivých úpravných operací.

2.1.1 Zespoda nahoru

Vlastníkem celé struktury je buněčný list v 1. atomickém diagramu.

Motivací pro tuto reprezentaci je, že dobře zachycuje signaturu jednotlivých buněk.

Hlavní nevýhodou však je, že vyžaduje z velké části sdílet data mezi nezávislými objekty (třeba pomocí reference counting), upravování je mnohem složitější a celou strukturu je potřeba rekurzivně (s opakováným průchodem tím kterým uzlem) procházet.

2.1.2 Seshora dolů

Zde je vlastníkem naopak buněčný list v posledním atomickém diagramu.

Motivací pro tuto reprezentaci je snadná manipulace s opetopem a snadné ověřování kompatibility signatur. Nicméně i tato reprezantace vyžaduje extenzivní užití sdílených pointerů a ústí ve špagetokód.

2.1.3 Seshora dolů pomocí indexů

Toto je nejspíš nejpřírozenější reprezentace.

Každá vrstva je separátní objekt, který spravuje všechny buňky v sobě (skupiny i uzlové buňky) a vlastní vrstvu o úroveň níž (pokud existuje).

Tento přístup skýtá stejné výhody jako předchozí zmíněná reprezentace. Navíc díky použití indexů nevyžaduje aliasované pointery a dopomáhá k přehlednějšímu kódu. Proto je to reprezentace, která byla v editoru použita.

2.2 Výběr ekosystému na tvorbu aplikace

Zde bylo potřeba přihlédnout ke dvěma hlediskům: zobrazovacímu a programatickému.

Pro konvenční prostředí typu Java + JavaFX nebo webové technologie mluví fakt, že tyto systémy jsou velmi rozšířené a existuje v nich tedy mnoho dobrých GUI knihoven a návodů. Na druhou stranu, tyto jazyky nemají dobrou datovou sémantiku, a vývoj by byl tedy brzděn nutností častého ladění.

Touto nevýhodou netrpí programovací jazyk Rust, který umožňuje rozsáhlou analýzu datového toku, čímž zabraňuje mnohým chybám spojeným s nedokonalou správou paměti (zejména souběh či neočekávaný přístup).

Jelikož větší část zobrazování opetopů by stejně bylo potřeba napsat ručně (i v rozšířené knihovně jako je JavaFX), byl jako implementační jazyk zvolen Rust.

2.3 Pomocné struktury

Některé operace rozbrané v 1.4 mohou zneplatnit indexy již sestrojených buněk. To je problém, neboť buňky ve vyšších vrstvách se na ně potřebují odkazovat (musí si pamatovat své vstupy a výstupy). Zároveň kvůli serializaci není možné, aby držely prosté ukazatele (ty se při deserializaci téměř jistě změní). Proto bylo potřeba vyvinout struktury, které umožňují odkazování na validní data i zastaralými indexy.

Vyvinuty byly struktury dvě: verzovací seznam a trasovací seznam. Verzovací seznam obsahoval seznam verzí (seznam seznamů svých položek). Tento seznam řeší problém nevalidních indexů, ale vyvstává jiný problém. Za prvé, je potřeba data kopírovat při změně verzí (a také tedy plýtvá místem). Za druhé, i když lze data vyhledat na základě zastaralého indexu, tento index nelze aktualizovat (protože verzovací seznam nemůže sledovat pohyb svých položek).

Trasovací seznam toto řeší tak, že místo toho, aby jeho verze obsahovaly data položek, položky jsou uloženy v databance (jiném seznamu) a verze obsahují pouze jejich indexy v databance. Z této databanky nelze mazat (mazání odpovídá vytvoření nové verze, která pouze neobsahuje index "smazané"položky). Díky tomu lze zastaralý index aktualizovat (vyhledá se index položky v databance, vyhledá se poslední verze, která jej obsahuje, a vytvoří se index do této verze).

Ke konci vývoje bylo zjištěno, že by se hodilo implementovat ještě trasovací strom, ale bylo od toho z časových důvodů upuštěno.

2.4 Zobrazování opetopů

Tato část implementace byla velmi zajímavá, neboť se ukázalo, že vykreslování atomických diagramů není triviální záležitost.

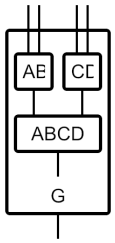
Vertikálně jde skutečně o jednoduchý problém: čáry, které nejsou zakončeny buňkou se protáhnou podle potřeby (to umí každé flex rozvržení). Horizontálně jde však o zapeklitější situaci.

Jelikož každý diagram sestává z buněk, které potenciálně obsahují další diagram, musí být vykreslování rekurzivní. Aby se vykreslil diagram, vykreslí se postupně každá jeho buňka (v pořadí pre-order traversal). Pokud má buňka nějaký obsah, vykreslí se jako diagram.

Každý diagram má ovšem jednu výstupní čáru a několik vstupních čar. Pokud je takový diagram v nějaké buňce vnořený, musí se nějakým způsobem přenést informace o rozestupech vstupních čar. Tento přenos navíc musí být obousměrný.

Dejme tomu, že čáry mají nějaký základní rozestup. Pokud ovšem nějaká čára vede do buňky, rozestupy kolem této čáry se rozšíří. Informace o tomto rozšíření se musí přenést jak vzhůru do naddiagramů (aby mohly upravit rozestupy svých výstupů), tak dolů do vnořených diagramů (aby mohly upravit rozestupy svých vstupů a buněk).

Řešení, které je v editoru uplatněno, využívá rekurzivní stromové struktury, která si pamatuje šířky jednotlivých buněk. Když se zobrazuje buňka, diagramy, které jsou situovány nad jejími vstupy, jsou předány oné stromové struktuře. Ta je seskupí do kontejnerů fixní šířky (viz 2.1), čímž se mezi nimi vytvoří stejné rozestupy, jaké mají buňky níž a výš po daných čarách.



Obrázek 2.1: Vstupy buňky G jsou seskupeny po šířkách buněk AB a CD

3. Technická dokumentace

Tato kapitola obsahuje informace o samotném editoru - o instalaci, vlastní kompilaci, spuštění a samotném používání.

3.1 Instalace

Nejdříve je potřeba stáhnout si zdrojový kód z adresy gitlab.com/opetopes/eru-app (nebo též github.com/ruza-net/eru). Nejjednodušší způsob je:

```
git clone --recursive https://gitlab.com/opetopes/eru-app
```

Pozn.: Je potřeba použít rekurzivní příkaz, protože repozitář obsahuje podmoduly.

Také je nutné nainstalovat si programovací jazyk Rust (viz Install Rust - Rust Programming Language).

Instalaci lze ověřit pomocí příkazu:

```
cargo -V
```

Jakmile je Rust nainstalován, stačí rozbalit zdrojový kód editoru, odnavigovat se do jeho adresáře (eru-app nebo eru) a spustit příkaz:

```
cargo run --release
```

První kompilace může trvat dlouho, poté lze ovšem použít spustitelný soubor, který se nachází v eru/target/release/eru.

Pozn.: Na Linuxu je problém se základním backendem, proto se musí použít alternativní. Ten nepodporuje zobrazování SVG obrázků, proto nebudou mít nástroje v panelu nástrojů ikony. Aby se projekt zkompiloval pro daný backend, místo předchozího příkazu použijte:

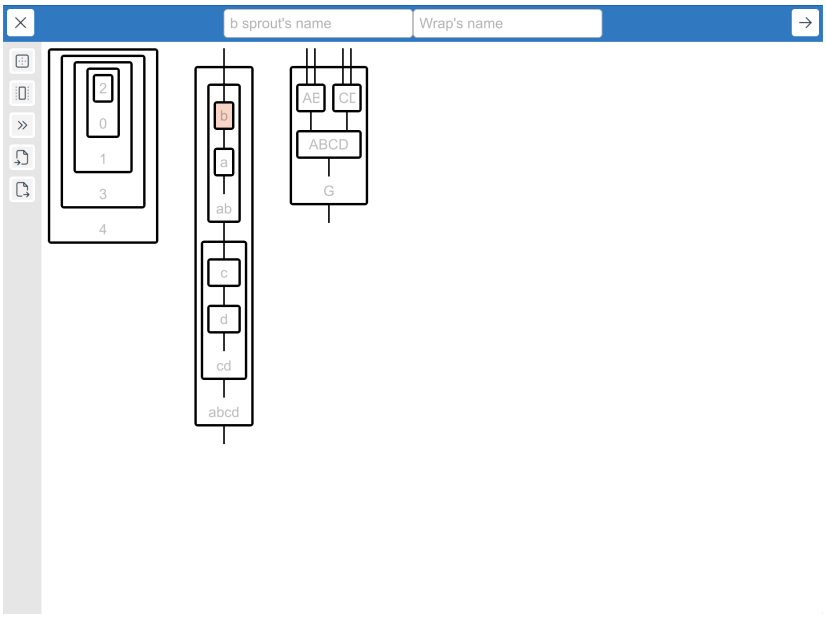
```
cargo run --release --features glow
```

3.2 Uživatelské rozhraní

Uživatelské rozhraní je velmi jednoduché. Sestává z panelu nástrojů, pracovní plochy a dialogového pole. V pracovní ploše probíhá zobrazování a interakce s opetopy, panel nástrojů obsahuje tlačítka a dialogové pole zobrazuje chybová hlášení a formuláře na vyplňování popisků buněk (3.1).

3.2.1 Pracovní plocha

Zde se zobrazuje aktuální stav upravovaného opetopu. Jedinou možnou interakcí zde je výběr buněk. Vybrané buňky jsou vysvíceny oranžově. Výběr buněk je omezen tak, aby byl validní pro



Obrázek 3.1: Uživatelské rozhraní

všechny operace. Například nelze současně vybrat nějakou buňku a jejího rodiče. Lze ovšem vybrat buňky, které netvoří souvislý strom. Například při vetknutí jde o validní výběr. Naproti tomu pokus o seskupení nesouvislých buněk vyústí v chybové hlášení.

3.2.2 Panel nástrojů

Panel nástrojů obsahuje nástroje na provádění úprav zmíněných v 1.4. Při najetí myší na jednotlivé ikony se zobrazí jejich popisek a v závorce klávesová zkratka. Popisky odpovídají operacím následovně:

1. enclose - seskupení
2. sprout - vetknutí
3. pass - další vrstva
4. cut - získání tvaru
5. rename - přejmenování
6. save - export rozpracovaného opetopu do souboru ~/opetope.json
7. load - import rozpracovaného opetopu ze souboru ~/opetope.json

Pozn.: Získání tvaru není v současné verzi plně implementováno. Proto tlačítko "cut"místo toho vyčistí pracovní plochu. Jelikož undo taktéž není plně implementováno, musí si uživatel dát pozor, aby neztratil svou práci.

3.2.3 Formulář na vyplnění popisků

Zobrazuje se při operacích, které generují nové buňky (seskupení, vetknutí, další vrstva) nebo upravují popisky existujících buněk (přejmenování). Z technických důvodů nelze, aby měla buňka prázdné jméno, takže místo bezejmenné buňky se vloží buňka jménem ""(mezera).

3.2.4 Chybový dialog

Zobrazuje poslední chybové hlášení, a to nejvýš po dobu 5 vteřin.

Závěr

Závěrem bych rád provedl reflexi celého projektu.

Tento projekt jsem si původně vybral pro své nadšení do jeho matematického uplatnění, brzy jsem však zjistil, že do této fáze se bohužel nedostanu. Značně jsem podcenil časovou náročnost zobrazování opetopů.

Tato fáze však byla také nejpřínosnější pro můj osobní růst. Potvrdil jsem si při ní programátorské maximy týkající se udržitelnosti kódu - nejen KISS, ale hlavně principu syntézy-analýzy. Vývoj by měl postupovat od nejvyšší úrovně abstrakce po tu nejnižší. Díky tomu jsou jednotlivé části kódu přehledné a snadno laditelné. Je to takový test-driven-development, ale jeho testy nejsou kontrolovány postuláty (assertion), ale zdravým rozumem (zejména proto, že GUI se automaticky ladí obtížně, a proto, že některé datové struktury jsou natolik složité, že hrozí, že automatický test bude napsán špatně - což by představovalo ještě horší problém).

Poučením tedy bylo: aspekty projektu nejdřív vyvíjej analyticky (od celkového zadání po detaily), a jaké podproblémy přitom vyřešíš, potom skládej synteticky. Pokud při syntetickém skládání narazíš na konflikt, pokus se přijít se zobecněním dané konfliktní části, jehož speciálními případy by byla konfliktní uplatnění.

Dalším přínosem, který vzešel z nelehké implementace, byla myšlenka verzovaných datových struktur. Verzované datové struktury by zasluhovaly samostatnou práci, ale konkrétním výstupem z mého projektu je knihovna pro jazyk Rust, která implementuje trasovací seznam. Je to seznam, který si pamatuje pohyb svých elementů, a je tedy schopen simulovat proměny svých verzí v čase. Původně jsem jej implementoval, abych nemusel při každé úpravě opetopu měnit indexy jeho jednotlivých komponentů. K tomu jsem byl nakonec z časových důvodů donucen (protože kromě trasovacího seznamu byl potřeba také trasovací strom), ale ona knihovna je stejně k dispozici v registru knihoven jazyka Rust (crates.io).

Posledním poučením, nebo spíše rozčarováním, bylo zjištění, jak neúplné a nedostatečné jsou grafické knihovny obecně. Samozřejmě grafické knihovny jazyka Rust jsou v rané fázi vývoje, takže nedosahují kvality webových technologií či JavaFX. Jenže ani ve zmíněných vyspělejších systémech (web, Java, Smalltalk, ...) by nebyla implementace zobrazování opetopů snazší. Paradoxně nějaký nástroj pro děti jako je Scratch by byl pro tento účel vhodnější (to samé platí o herních enginech jako Unity). Je zvláštní, že přestože jsou dnes grafické aplikace velmi rozšířené, stále nebylo vyvinuto skutečně dobré prostředí na vývoj grafických aplikací. Ty existující nejspíš vyhovují, dokud člověk dělá pouze "digitální papír"nebo jiná jednoduchá schémata. Ale jen o trochu složitější problém vyžaduje značnou ceremonii a výpomoc procedurálními prvky (viz potřeba přenášet informace o šířkách). Vývoj uživatelského rozhraní by přitom měl být výhradně deklarativní.

Poslední, co zbývá, je zhodnotit dokončenost projektu. V této fázi projekt splňuje zadání. Opetopy jsou skutečně obecné struktury, takže mohou nést jakýkoliv druh dat. Konstrukce jsou vždy garantované jako správné. Spoustu věcí, které jsem čekal, že stihnu implementovat, jsem nakonec nestihnul. První z nich je registr sestrojených buněk a univerzální konstrukce. V současné podobě editoru si uživatel může každou buňku pojmenovat, jak chce. Původně jsem si však představoval, že opetopy budou jakési "funkce". Některé buňky budou jeho *parametry* a jiné z nich půjdou sestrojit automaticky (viz [2]).

Také jsem chtěl umožnit uživateli upravovat více opetopů naráz. To by ovšem vyžadovalo scrolling, který není knihovnou, kterou používám, dostatečně podporován (a neměl jsem čas si ho napsat sám). Kompenzací tohoto je snadné a rychlé ukládání a načítání opetopů ze souboru (zde se mimochodem blýskla knihovna serde: (de)serializace je možná do široké škály formátů, základní používaný je JSON). Editor také nepodporuje mazání buněk, což lze zastoupit pomocí operace tvar (viz 1.3).

Ač byla práce na projektu složitá a místy se zdála beznadějnou (často bylo potřeba nějakou část začít psát odznova), hodně jsem se při ní naučil, dostal jsem náměty na budoucí práci a získal zkušenosti týkající se organizace práce na větším projektu. Celkově tedy projekt eru hodnotím kladně.

Seznam použité literatury

[BD97] John Baez a James Dolan. "Higher-Dimensional Algebra III: n-Categories and the Algebra of Opetopes". In: (pros. 1997). DOI: https://doi.org/10.1006/aima.1997.1695.

[Fina] Eric Finster. *Opetopic - Documentation: Universal Properties*. URL: http://opetopic.net/docs/categories/uprops.

[Finb] Eric Finster. *Type Theory and the Opetopes*. URL: https://ncatlab.org/nlab/files/FinsterTypesAndOpetopes2012.pdf.

[Koc+10] Joachim Kock et al. "Polynomial functors and opetopes". In: *Advances in Mathematics* 224.6 (2010), s. 2690–2737. ISSN: 0001-8708. DOI: https://doi.org/10.1016/j.aim.2010.02.012. URL: https://www.sciencedirect.com/science/article/pii/S0001870810000769.

Seznam obrázků