Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Grafové editory pre prieskumník štruktúr logiky prvého rádu

Bakalárska práca

2026
Jakub Marček

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Grafové editory pre prieskumník štruktúr logiky prvého rádu

Bakalárska práca

Študijný program:	Aplikovaná informatika
Študijný odbor:	Informatika
Školiace pracovisko:	Katedra aplikovanej informatiky
Školiteľ:	Mgr. Ján Kľuka, PhD.

Bratislava, 2026
Jakub Marček

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Jakub Marček
Študijný program: aplikovaná informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Grafové editory pre prieskumník štruktúr logiky prvého rádu
Graph editors for the first-order logic structure explorer

Anotácia: V rámci série predchádzajúcich bakalárskych prác (Cifra, 2018; Baluch, 2020; Tóth, 2021) vznikla interaktívna webová aplikácia, ktorá umožňuje skúmať pravdivosť formúl a hodnoty termov v konkrétnych konečných štruktúrach pre jazyky logiky prvého rádu. Vnútorný návrh aplikácie však vznikol v čase, keď použité frameworky neposkytovali súčasné prostriedky, je neprehľadný a ďalšie úpravy sú prácne a prinášajú riziko vzniku ťažko odhaliteľných chýb. Náš bakalársky študent v súčasnosti pracuje na reimplementácii. Ďalším krokom vo vývoji je poskytnúť možnosť editovať interpretácie predikátov a funkcií v grafovej forme (orientované a bipartitné grafy, Hasseho diagramy a pod.)

Cieľ:
- Vytvoriť prehľad rôznych foriem grafovej reprezentácie relácií a funkcií v diskrétnej matematike.
- Vybrať vhodnú knižnicu pre jazyk TypeScript podporujúcu interaktívnu prácu s grafmi.
- Implementovať komponenty na zobrazovanie a interaktívnu úpravu vybraných grafových reprezentácií.
- Integrovať tieto komponenty do aplikácie Prieskumník štruktúr.

Literatúra:
Cifra, Milan. Prieskumník sémantiky logiky prvého rádu. Bakalárska práca. Bratislava : Univerzita Komenského, 2018.
Baluch, Miroslav. Prieskumník grafových štruktúr pre logiku prvého rádu. Bakalárska práca. Bratislava : Univerzita Komenského, 2020.
Tóth, Richard. Henkinova-Hintikkova hra v prieskumníku štruktúr. Bakalárska práca. Bratislava : Univerzita Komenského, 2021.
Bieliková, Silvia. Generovanie používateľského rozhrania z grafových dát. Bakalárska práca. Bratislava : Univerzita Komenského, 2024.

Vedúci: Mgr. Ján Kľuka, PhD.
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: doc. RNDr. Tatiana Jajcayová, PhD.

Dátum zadania: 28.10.2024

Dátum schválenia: 18.03.2025 doc. RNDr. Damas Gruska, PhD.
garant študijného programu

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

....

študent vedúci práce

Poďakovanie: Ďakujem.

Abstrakt

V práci sa zaoberáme vývojom nových nástrojov rozširujúcich existujúcu webovú aplikáciu Prieskumník štruktúr logiky prvého rádu. Hlavné rozšírenie spočíva v troch rôznych interaktívnych grafových editoroch, ktoré umožňujú vizualizovať a upravovať interpretácie predikátov a funkcií vo forme orientovaného grafu, bipartitného grafu a Hasseho diagramu. Zaoberáme sa návrhom, ako premeniť jednotlivé reprezentácie do prehľadnej interaktívnej podoby, a aj ich následnou implementáciou a integráciou do existujúcej aplikácie. Podobným spôsobom tiež predstavujeme niekoľko ďalších nástrojov a vylepšení. Novú verziu aplikácie, zahŕňajúcu všetku pridanú funkcionalitu, sme následne integrovali do Logického pracovného zošita používaného na predmete Logika pre informatikov. V závere sa venujeme vyhodnoteniu testovania použiteľnosti našich riešení, ktoré prebehlo so študentmi predmetu.

Kľúčové slová: logika prvého rádu, grafový editor, webová aplikácia

Abstract

TODO

Keywords: first-order logic, graph editor, web application

Obsah

Úvod

Možnosti využitia elektronických nástrojov pri výučbe sú v dnešnej dobe široké a oproti konvenčným učebným metódam dokážu poskytnúť interaktívnejší a pútavejší pohľad na skúmanú problematiku. Ich tvorba však predstavuje výzvu, ktorá vyžaduje nielen znalosť danej témy, ale aj schopnosť hľadať kreatívne a inovatívne riešení s ňou spojených problémov. V mojej práci sa budem zaoberať návrhom a implementáciou takýchto nástrojov so zámerom rozšíriť funkcionalitu existujúcej aplikácie.

Prvá verzia webovej aplikácie, z ktorej táto práca vychádza, vznikla v rámci bakalárskej práce Milana Cifru [6]. Jej úlohou bolo ponúknuť študentom predmetu Logika pre informatikov interaktívny nástroj na tvorbu štruktúr pre jazyky logiky prvého rádu, v ktorých je následne možné skúmať pravdivosť formúl a hodnoty termov. Neskôr bola aplikácia rozšírená v bakalárskej práci Richarda Tótha [14] o Henkinovu-Hintikkovu hru a v práci Miroslava Baluchu o jeden grafový pohľad so zámerom poskytnúť študentom alternatívny spôsob vizualizácie a manipulácie týchto štruktúr. Implementácia aplikácie sa však časom stala neaktuálnou a ťažko udržateľnou.

V bakalárskej práci Jozefa Filipa [7] bola aplikácia nanovo implementovaná s využitím aktuálnych technológií. Nová verzia však neponúka grafový ani iné pohľady poskytované pôvodnou aplikáciou. Jediný spôsob, akým je možné s ňou interagovať, je pomocou textových vstupov, ktoré sa často stávajú neprehľadnými, najmä pri väčších interpretáciách predikátových a funkčných symbolov. Taktiež neumožňujú jednoduchú vizualizáciu niektorých zaujímavých vlastností samotnej interpretácie, akou je napríklad to, že tvorí čiastočné usporiadanie.

Primárnym cieľom tejto práce je vytvoriť grafové nástroje, ktoré umožnia prehľadnejšiu manipuláciu a vizualizáciu interpretácií predikátov a funkcií. Nechceme však reimplementovať pôvodný grafový pohľad, keďže ten sa z dôvodu viacerých problémov ukázal byť medzi študentmi málo využívaný. Jednou zo zásadných zmien bude využitie troch rôznych grafových reprezentácií (orientovaný graf, bipartitný graf a Hasseho diagram), pričom každá z nich poskytne študentovi unikátny pohľad na tú istú interpretáciu. Taktiež umožníme vizualizáciu ďalších vzťahov v rámci štruktúry týkajúcich sa unárnych predikátov a indivíduových konštánt. Dôležitým aspektom bude poskytnúť rozhranie, ktoré je intuitívne, prehľadné a najmä jednoduché na použitie.

Prieskumník štruktúr však rozšírime aj o ďalšie nástroje. Jedným z nich bude re-

implementácia maticového a databázového pohľadu z prvej verzie aplikácie. Ďalej pribudne nástroj na jednoduchšie definovanie interpretácií funkcií pomocou rozboru prípadov. Tiež sa zameriame na rozšírenie umožňujúce zadávať dopyty na splniteľnosť otvorených formúl v rámci vytvorenej štruktúry.

V prvej kapitole zadefinujeme základné pojmy a v krátkosti predstavíme technológie používané ďalej v práci. Druhá kapitola slúži na bližšie predstavenie súvisiacich aplikácií spolu s rozborom ich nedostatkov. Na začiatku tretej kapitoly zadefinujeme požiadavky kladené na výslednú aplikáciu. Následne sa v návrhu pozrieme na spôsob, akým sme sa rozhodli stanovené ciele dosiahnuť. Postupne opíšeme návrh jednotlivých grafových editorov, maticového a databázového editora, editora interpretácií funkcií rozborom prípadov a novej časti aplikácie venujúcej sa dopytom na splniteľnosť otvorených formúl. Tiež opíšeme návrh stavu všetkých pridaných rozšírení. V štvrtej kapitole priblížime vybrané časti implementácie. Posledná, piata kapitola rozoberá výsledky testovania našej aplikácie študentmi predmetu Logika pre informatikov.

Kapitola 1

Základné pojmy

Táto kapitola slúži na oboznámenie čitateľa s pojmami a konceptmi používanými v ďalších častiach práce. Jednotlivé pojmy do potrebnej miery vysvetlíme a pri grafových reprezentáciách aj stručne naznačíme motiváciu za ich výberom, čo má pomôcť k lepšiemu pochopeniu ich potenciálnych prínosov.

1.1 Logika prvého rádu

Táto podkapitola vychádza najmä z prednášok a poznámok k predmetu Logika pre informatikov [9].

Logika prvého rádu je rodina formálnych jazykov, ktoré hovoria o objektoch a ich vlastnostiach. Symboly konkrétneho jazyka $\mathcal{L}$ logiky prvého rádu tvoria *indivíduové premenné* (napr. x, y, z), *logické symboly* ($\neg$, $\vee$, $\wedge$, $\rightarrow$, $\doteq$, $\exists$, $\forall$), *pomocné symboly* (ľavá zátvorka, pravá zátvorka a čiarka) a *mimologické symboly*, tie sa ďalej delia na *indivíduové konštanty* (napr. Alice, FMFI), *funkčné symboly* (napr. f, g) a *predikátové symboly* (napr. *matka*, *spieva*). Indivíduové premenné, logické symboly a pomocné symboly sú rovnaké naprieč všetkými jazykmi logiky prvého rádu, čo však jednotlivé jazyky odlišuje sú použité mimologické symboly.

Predikátové symboly používame na označenie vlastností alebo vzťahov. V konkrétnom jazyku $\mathcal{L}$ vždy majú (spolu s funkčnými symbolmi) pevne zvolený počet argumentov, ktorý určuje ich *arita*, niekedy označovaná horným indexom. Predikátové symboly s aritou 1 (unárne) zvyknú označovať nejakú vlastnosť alebo stav (napr. $zviera^1$, $svieti^1$), zatiaľ čo predikátové symboly s aritou 2 a viac (n-árne pre $n \geq 2$) opisujú určitý vzťah svojich argumentov (napr. $stavia^2$, $ukazuje^3$).

Ďalší zadefinujeme pojem *term*. Každá indivíduová premenná a indivíduová konštanta jazyka $\mathcal{L}$ je term. Taktiež ak f predstavuje funkčný symbol s aritou n a $t_1, \ldots, t_n$ sú termy, tak aj postupnosť symbolov $f(t_1, \ldots, t_n)$ je term.

Ďalej každá postupnosť symbolov $t_1 \doteq t_2$, kde t_1 a t_2 sú termy predstavuje *rovnostný*

atóm a každá postupnosť symbolov $P(t_1, \ldots, t_n)$, kde P je predikátový symbol s aritou n a $t_1, \ldots, t_n$ sú termy predstavuje *predikátový atóm*. Všetky rovnostné a predikátové atómy súhrnne nazývame *atomickými formulami*.

Nakoniec množinu všetkých *formúl* jazyka $\mathcal{L}$ definujeme takto: Každá atomická formula je zároveň aj formula. Ak A je formula, tak aj postupnosť symbolov $\neg A$ je formula. Ak A aj B sú formuly, tak aj postupnosti symbolov $(A \wedge B)$, $(A \vee B)$, $(A \to B)$ sú formuly. A napokom ak x je indivíduová premenná a A je formula, tak aj postupnosti symbolov $\exists x A$ a $\forall x A$ sú formuly.

Prvým dôležitým pojmom týkajúcim sa sémantiky logiky prvého rádu je *štruktúra*, ktorá sa vždy viaže na nejaký konkrétny jazyk. Označujeme ju dvojicou $\mathcal{M} = (D, i)$, kde D predstavuje ľubovoľnú neprázdnu množinu (doména) a i je zobrazenie (interpretačná funkcia). Úlohou interpretačnej funkcie je každej indivíduovej konštante priradiť prvok z D, každému funkčnému symbolu priradiť funkciu $f : D^n \to D$ a každému predikátovému symbolu priradiť podmnožinu množiny D^n, kde n v oboch prípadoch označuje aritu daného symbolu. Štruktúry sú dôležitou súčasťou skúmania vlastností jazyka logiky prvého rádu, keďže poskytujú jeho presný matematický model.

Tiež zadefinujeme niektoré ďalšie pojmy. *Ohodnotenie indivíduových premenných* je ľubovoľná funkcia $e : \mathcal{V}_\mathcal{L} \to D$, kde $\mathcal{V}_\mathcal{L}$ je množina indivíduových premenných. Potom *hodnota termu* t v štruktúre $\mathcal{M}$ pri ohodnotení premenných e je prvok z D označovaný $t^\mathcal{M}[e]$ definovaný nasledovne:

$$\begin{aligned} x^\mathcal{M}[e] &= e(x), \\ a^\mathcal{M}[e] &= i(a), \\ (f(t_1, \ldots, t_n))^\mathcal{M}[e] &= i(f)\left(t_1^\mathcal{M}[e], \ldots, t_n^\mathcal{M}[e]\right), \end{aligned}$$

pre všetky premenné x, všetky konštanty a, všetky funkčné symboly f s aritou n a všetky termy $t_1, \ldots, t_n$.

Relácia $\mathcal{M} \models X[e]$ (čítame *štruktúra* $\mathcal{M}$ *spĺňa formulu* X *pri ohodnotení* e) je definovaná induktívne takto:

$\mathcal{M} \models t_1 = t_2[e]$ vtt $t_1^\mathcal{M}[e] = t_2^\mathcal{M}[e]$,

$\mathcal{M} \models P(t_1, \ldots, t_n)[e]$ vtt $(t_1^\mathcal{M}[e], \ldots, t_n^\mathcal{M}[e]) \in i(P)$,

$\mathcal{M} \models \neg A[e]$ vtt $\mathcal{M} \not\models A[e]$,

$\mathcal{M} \models (A \wedge B)[e]$ vtt $\mathcal{M} \models A[e]$ a zároveň $\mathcal{M} \models B[e]$,

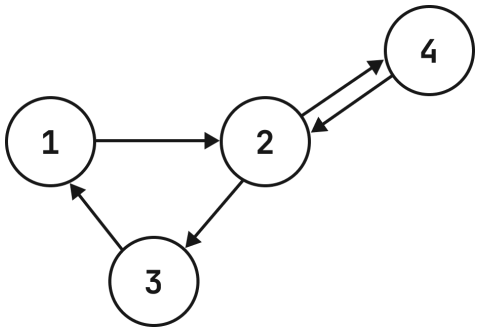
$\mathcal{M} \models (A \vee B)[e]$ vtt $\mathcal{M} \models A[e]$ alebo $\mathcal{M} \models B[e]$,

$\mathcal{M} \models (A \to B)[e]$ vtt $\mathcal{M} \not\models A[e]$ alebo $\mathcal{M} \models B[e]$,

$\mathcal{M} \models \exists x\, A[e]$ vtt pre nejaký prvok $m \in D$ platí $\mathcal{M} \models A[e(x/m)]$,

$\mathcal{M} \models \forall x\, A[e]$ vtt pre každý prvok $m \in D$ platí $\mathcal{M} \models A[e(x/m)]$,

pre všetky premenné x, všetky formuly A, B, všetky termy $t_1, \ldots, t_n$ a všetky predikátové symboly P s aritou n.



Obr. 1.1: Príklad vyobrazenia orientovaného grafu.

1.2 Grafové reprezentácie

V tejto podkapitole sú uvedené typy grafov použité v tejto práci spolu s krátkym vysvetlením možností ich použitia. Niektoré definície vychádzajú z Grimaldiho učebnice diskrétnej matematiky [8].

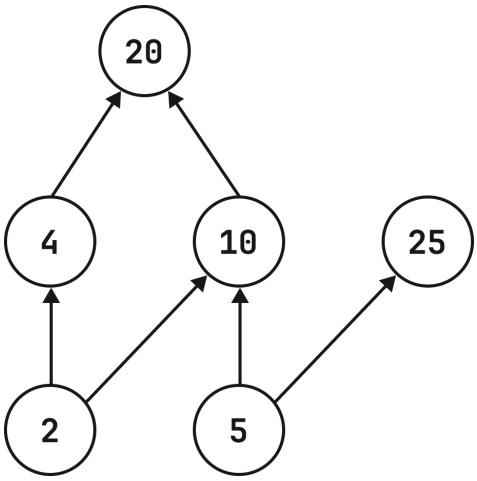
1.2.1 Orientovaný graf

Orientovaný graf G je daný neprázdnou konečnou množinou vrcholov V a množinou usporiadaných dvojíc $E \subseteq V \times V$, resp. hrán. V diagramoch sú vrcholy väčšinou znázornené ako kruhy, prípadne spolu s ich príslušným označením. Hrany sú zas vyobrazené pomocou šípok, ktorých smer určuje poradie vrcholov hrany (šípka vždy ukazuje na druhý vrchol v poradí). Majú mnohoraké využitie pri modelovaní vzťahov, ktoré vyžadujú zachovať nejaký smer, postupnosť úkonov alebo vyjadriť určitú hierarchiu. Príklad bežného diagramu jednoduchého orientovaného grafu možno vidieť na obrázku 1.1, kde $V = \{1, 2, 3, 4\}$ a $E = \{(1, 2), (2, 3), (3, 1), (2, 4), (4, 2)\}$.

1.2.2 Hasseho diagram

Pred opísaním Hasseho diagramu je najprv potrebné objasniť pojem čiastočného usporiadania, ktorý s ním veľmi úzko súvisí.

Podľa definície sa relácia $\leq$ na množine A nazýva *čiastočným usporiadaním*, ak je reflexívna, antisymetrická a tranzitívna. To znamená, že pre každé $x \in A$ musí platiť $x \leq x$ (prvok x je v relácii so sebou samým - reflexívna relácia), pre každé $x, y \in A$ platí $(x \leq y \wedge y \leq x) \Rightarrow x = y$ (ak je prvok x v relácii s prvkom y a y je v relácii s x, tak $x = y$ - antisymetrická relácia) a pre každé $x, y, z \in A$ platí $(x \leq y \wedge y \leq z) \Rightarrow x \leq z$ (ak je prvok x v relácii s prvkom y a y je v relácii s prvkom z, tak aj x je v relácii so z - tranzitívna relácia). Intuitívnejšie si ale tento pojem možno predstaviť ako dvojice prvkov, z ktorých prvý vždy predchádza druhému. Zvyčajne sa čiastočné usporiadanie označuje aj ako dvojica $(A, \leq)$.



Obr. 1.2: Príklad vyobrazenia Hasseho diagramu.

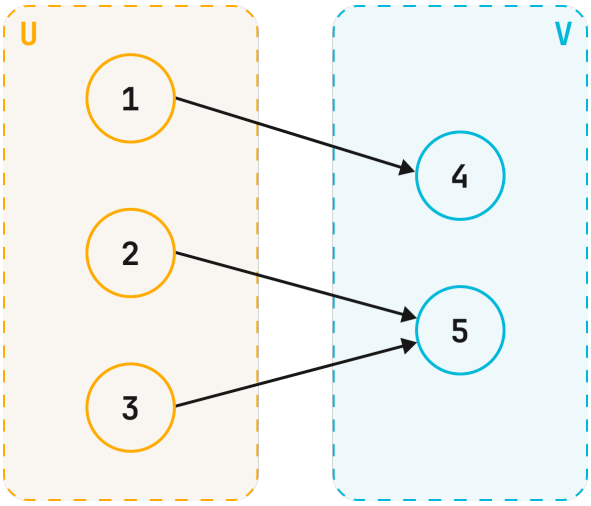
Hasseho diagram slúži na prehľadnejšiu grafickú reprezentáciu čiastočných usporiadaní, keďže zachytáva iba bezprostredné vzťahy, bez zbytočného kreslenia hrán, ktoré intuitívne vyplývajú z jeho vlastností. Pri jeho tvorbe však treba dbať na dodržiavanie určitých pravidiel:

- Každý prvok čiastočného usporiadania, teda prvok množiny A, je vrcholom v takomto grafe.
- Medzi ľubovoľnými prvkami $x, y \in A$ vytvárame hranu iba vtedy, keď $x \leq y$, $x \neq y$ a neexistuje taký prvok $z \in A$, rozdielny od x a y, pre ktorý platí $x \leq z \leq y$.
- Ak je medzi prvkami $x, y \in A$ hrana a v usporiadaní platí $x \leq y$, potom vrchol reprezentujúci prvok x kreslíme pod vrchol reprezentujúci prvok y.

Na obrázku 1.2 možno vidieť príklad Hasseho diagramu pre čiastočné usporiadanie $(A, \mathcal{R})$, kde $A = \{2, 4, 5, 10, 20, 25\}$ a $x \,\mathcal{R}\, y$, ak x delí y bezozvyšku. Dobré je podotknúť, že aj prvky 2 a 5 sú v relácii s prvkom 20, no hrany vyjadrujúce tento vzťah, ako bolo vyššie spomínané, nemusíme do diagramu explicitne zakresľovať.

1.2.3 Bipartitný graf

Bipartitný graf je taký graf G, ktorého vrcholy možno rozdeliť do dvoch disjunktných podmnožín U a V tak, že $U \cap V = \emptyset$ a zároveň každá hrana spája jeden vrchol z U s ďalším vrcholom z V. Vedia byť nápomocné pri hľadaní a skúmaní vzťahov (relácií a zobrazení) medzi entitami dvoch rôznych tried, ako napríklad medzi študentmi a učiteľmi, autormi a publikáciami alebo receptami a ingredienciami. Pri menšom množstve dát a vhodnom rozložení vrcholov umožňujú prehľadnú vizualizáciu prítomných vzťahov. Ukážku diagramu konkrétneho bipartitného grafu pre zobrazenie $f : \{1, 2, 3\} \to \{4, 5\}$, kde $f(1) = 4, f(2) = 5$ a $f(3) = 5$, so znázorneným rozdelením prvkov do množín U a



Obr. 1.3: Príklad vyobrazenia bipartitného grafu.

V (v tomto prípade si možno predstaviť definičný obor a obor hodnôt zobrazenia) je vidieť na obrázku 1.3.

1.3 Použité technológie

V tejto podkapitole sú uvedené kľúčové technológie použité pri implementácii.

1.3.1 React

React [10] je v súčasnosti jedna z najpopulárnejších JavaScriptových knižníc na tvorbu responzívnych webových aplikácií.

Základnú stavebnú jednotku tvorí komponent, čo je JavaScriptová funkcia, ktorá produkuje určitý strom elementov jazyka HTML, ďalej označovaný ako markup. Každý takýto komponent potom predstavuje konkrétnu časť používateľského rozhrania s vlastnou funkcionalitou a vzhľadom.

Vo väčšine prípadov sa markup komponentov zapisuje vo forme JavaScript XML (JSX), čo je syntaktické rozšírenie jazyka JavaScript. JSX je veľmi podobný značkovaciemu jazyku HTML, no umožňuje používanie vlastných značiek, ako aj jednoduchšiu manipuláciu s dynamickým obsahom. Toto je pre React kľúčové, keďže je tým umožnené skladať viaceré reaktívne komponenty do určitej hierarchie, ktorá reprezentuje celú aplikáciu.

1.3.2 TypeScript

TypeScript [11] je programovací jazyk s podporou pre statické typy s konfigurovateľnou mierou typovej kontroly. Funguje iba ako syntaktické rozšírenie JavaScriptu, ktoré sa

pri kompilácii naspäť zmení na čistý JavaScript. Práve táto vlastnosť zapríčiňuje jeho širokú kompatibilitu s už existujúcim ekosystémom.

Statické typovanie pomáha odhaľovať chyby už v skorých fázach vývoja, zlepšuje čitateľnosť aj udržateľnosť kódu a vývojárskym prostrediam umožňuje integráciu bohatšej funkcionality, ako napríklad pokročilejšie automatické dopĺňanie na základe kontextu alebo vyhľadávanie definícií. Svojimi prínosmi teda dokáže priamo aj nepriamo uľahčiť a spríjemniť prácu samotného programátora.

Jazyk poskytuje možnosť definovať rozhrania pomocou kľúčového slova `interface`, ktoré upresňujú zamýšľaný tvar objektov. Kľúčové slovo `type` zas umožňuje vytváranie typových aliasov, či už primitívnych alebo komplexných typov. Okrem samotných typov podporuje typovú inferenciu, čo pomáha s prehľadnosťou programu, keďže často nie je potrebné tvar objektu definovať manuálne.

1.3.3 React Flow

React Flow [1] je populárna knižnica na tvorbu grafových editorov, ktorá je súčasťou väčšieho projektu xyflow [3]. Zatiaľ čo React Flow je určený pre React, xyflow ponúka plnú kompatibilitu aj s niektorými inými podobnými knižnicami. Jej veľkou výhodou je široká škála možností prispôsobenia jednotlivých aspektov grafu, ako aj ich jednoduchá implementácia.

Každý vrchol predstavujú jeho atribúty a príslušný React komponent. Atribúty musia byť minimálne tvorené pozíciou a unikátnym identifikátorom. Ďalej však môžeme pridávať vlastné atribúty, ako aj komponenty, ktoré sa dajú prispôsobiť špecifickým potrebám našej aplikácie. Na veľmi podobnom princípe sa dajú konfigurovať aj hrany grafu, tie sú tiež tvorené komponentmi a spolu s vrcholmi tvoria základnú štruktúru každého grafu.

Knižnica tiež poskytuje niektorú rozšírenú funkcionalitu, akou je napríklad centrovanie grafov, uloženie a obnovenie stavu grafu alebo konfigurovateľný panel s nástrojmi.

1.3.4 Redux

Redux [2] je knižnica na správu stavu JavaScriptových aplikácií, často používaná spolu s knižnicou React na zjednodušenie tvorby komplexných používateľských rozhraní. Stav aplikácie je reprezentovaný iba ako obyčajný objekt, ktorý predstavuje centrálne miesto pre ukladanie a čítanie dát, čím výrazne uľahčuje ich zdieľanie naprieč komponentmi.

Stav v Reduxe nie je možné meniť priamo. Namiesto toho sa zmeny vykonávajú pomocou akcií a reducerov. Akcie sú objekty popisujúce konkrétnu udalosť v aplikácii. V prípade potreby môžu obsahovať aj doplňujúce údaje (payload). Tieto akcie ďalej spracúvajú reducery. To sú funkcie, ktoré na základe aktuálneho stavu a prijatej akcie určia nový stav aplikácie. Pre ich správne fungovanie je dôležité dbať na to, aby

reducer pre rovnaký stav a rovnakú akciu vždy vrátil rovnaký výsledok, nemali by sa preto používať napríklad na generovanie náhodných hodnôt. Aplikácia môže obsahovať viacero reducerov, čo umožňuje rozdeliť logiku na menšie, nezávislé časti.

Na získavanie konkrétnych dát zo stavu sa používajú selektory. Ide o funkcie, ktoré na základe súčasného globálneho stavu vyberajú alebo odvodzujú potrebné údaje, čím zjednodušujú a sprehľadňujú prácu s dátami.

Kapitola 2

Súvisiace práce

V tejto kapitole sa podrobnejšie venujeme aplikáciám Prieskumník štruktúr a Prieskumník grafových štruktúr, ktoré sú pre túto prácu kľúčové. Postupne vysvetlíme, aké problémy sa snažia vyriešiť, ako fungujú a aké sú ich nedostatky. V závere kapitoly stručne predstavíme aplikáciu Logický pracovný zošit, do ktorej sú spomínané aplikácie integrované.

2.1 Prieskumník štruktúr

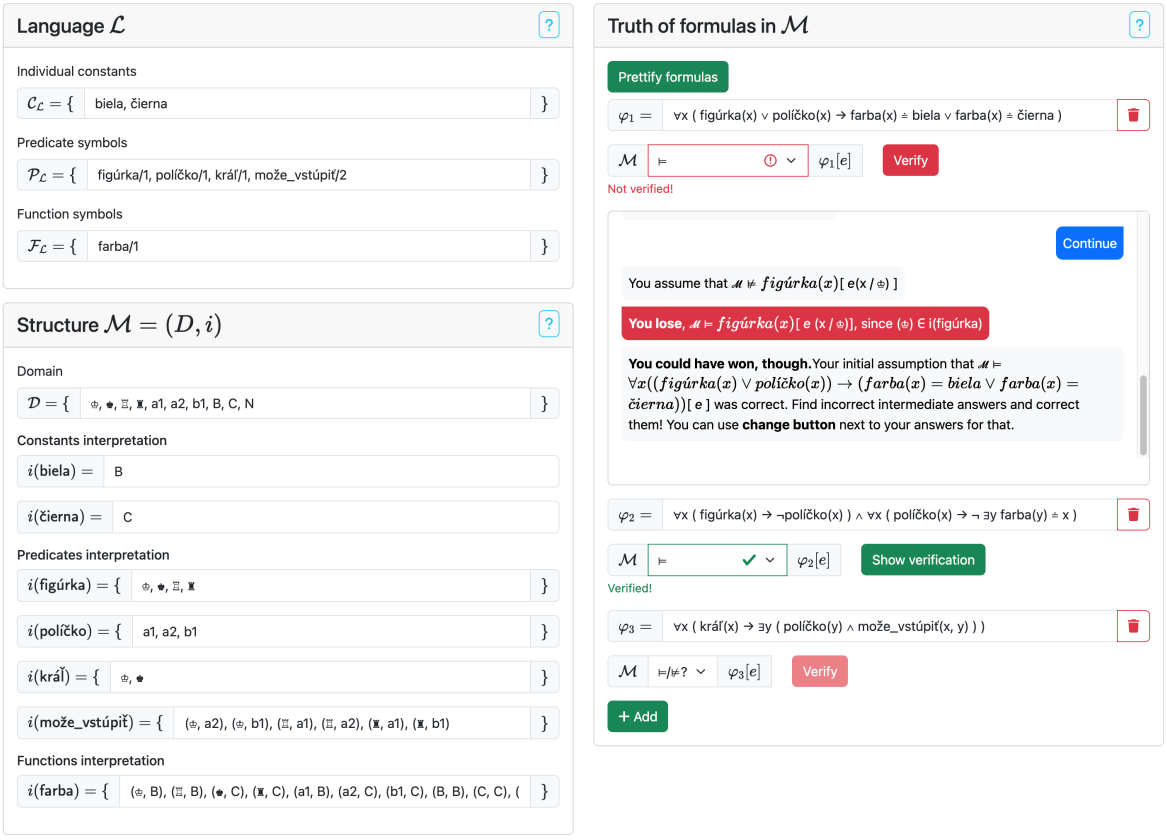
Prieskumník štruktúr je jednou z webových aplikácií používaných na predmete Logika pre informatikov. Jej primárnou úlohou je umožniť študentom interaktívne vytvárať vlastný jazyk a štruktúru logiky prvého rádu. V takto definovanej štruktúre je ďalej možná tvorba formúl a analýza ich pravdivosti s rýchlou spätnou väzbou.

Pôvodná aplikácia vznikla v rámci viacerých bakalárskych prác. Konkrétne ide o prácu Milana Cifru [6], ktorá sa zaoberá samotným vývojom Prieskumníka štruktúr, prácu Richarda Tótha [14], rozširujúcu prieskumník o Henkinovu-Hintikkovu hru a prácu Miroslava Baluchu [5], ktorá priniesla alternatívny grafový pohľad. Túto poslednú prácu ešte bližšie rozoberieme v nasledujúcej podkapitole 2.2.

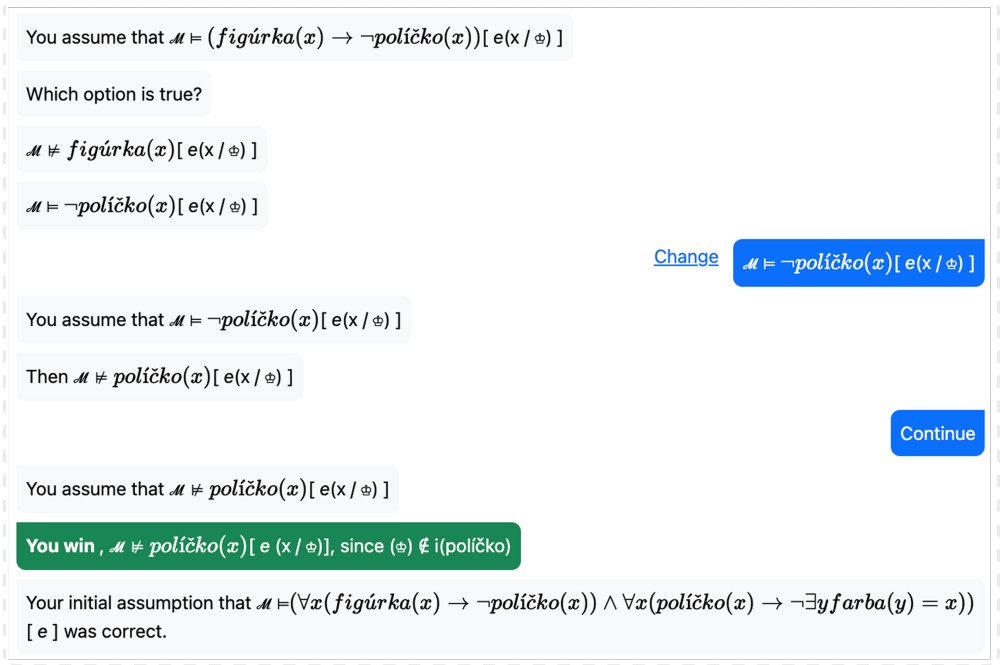
Implementácia aplikácie sa však časom stala neaktuálnou a nejednotnou, keďže bola výsledkom viacerých bakalárskych prác využívajúcich rôzne prístupy k použitým technológiám. To spôsobilo jej ťažkú udržateľnosť, ako aj náročnosť rozširovania o ďalšiu funkcionalitu.

Cieľom bakalárskej práce Jozefa Filipa [7] bolo vytvoriť novú, jednotnú verziu pôvodnej aplikácie bez grafového pohľadu a s novým spôsobom využitia Henkinovej-Hintikkovej hry. Túto verziu možno vidieť na obrázku 2.1.

V ľavej hornej časti obrázka sa nachádzajú textové vstupy na definovanie mimologických symbolov. V prípade predikátových a funkčných symbolov sa definuje aj ich arita vo formáte *symbol/arita*. Pod definíciou jazyka logiky prvého rádu sa definuje



Obr. 2.1: Používateľské rozhranie reimplentovanej verzie Prieskumníka štruktúr.



Obr. 2.2: Príklad dialógu Henkinovej-Hintikkovej hry.

samotná štruktúra, čo vyžaduje zadanie domény a určenie interpretácie jednotlivých mimologických symbolov. Aplikácia poskytuje iba jeden, tzv. množinový pohľad na interpretáciu predikátových a funkčných symbolov, ktorý je opäť realizovaný pomocou textového vstupu.

V pravej časti aplikácie používateľ definuje formuly. Po vytvorení validného jazyka a štruktúry môže skúmať pravdivosť týchto formúl v danej štruktúre. Najprv však musí sám odhadnúť a zvoliť, či je formula pravdivá, alebo nie. Svoj odhad si následne overí prostredníctvom Henkinovej-Hintikkovej hry. Tá funguje na princípe dialógu, v ktorom používateľ postupne dostáva otázky o pravdivosti čoraz menších podformúl pôvodnej formuly. Keď je už podformula dostatočne jednoduchá, hra používateľovi odhalí správnosť jeho predpokladu spolu s vysvetlením, prečo bol alebo nebol správny. Príklad dialógu je zobrazený na obrázku 2.2.

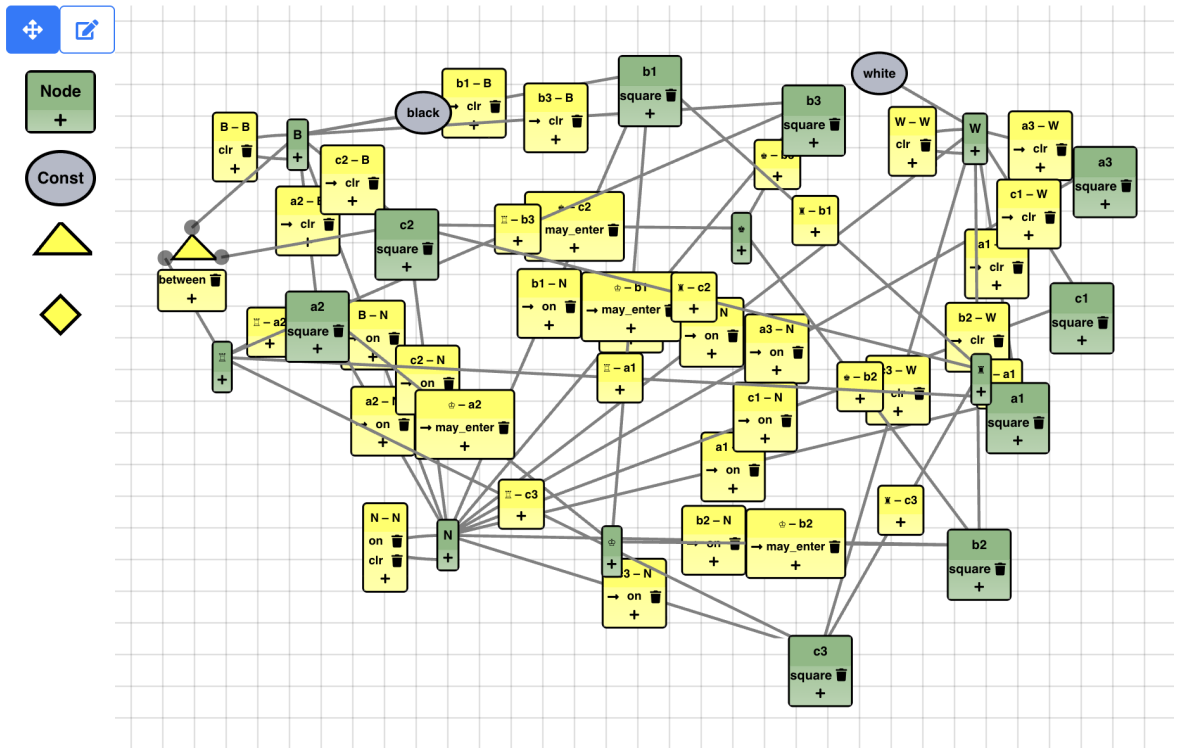
Asi najväčším problémom novej aplikácie v porovnaní s predošlou je nedostatok spôsobov vizualizácie a tvorby interpretácií predikátových a funkčných symbolov. Množinová, resp. textová reprezentácia sa už pri pomerne malých interpretáciách stáva neprehľadnou a ťažko editovateľnou, keďže vyžaduje dodržiavanie správnej syntaxe pri zápise. Predošlá verzia tento problém čiastočne riešila pridaním dvoch ďalších alternatívnych pohľadov. Konkrétne išlo o tzv. maticový a databázový pohľad, v ktorých bolo možné pomocou zaškrtávania alebo výberu zo zoznamu určiť, ktoré prvky sa majú v interpretácii vyskytnúť.

2.2 Prieskumník grafových štruktúr

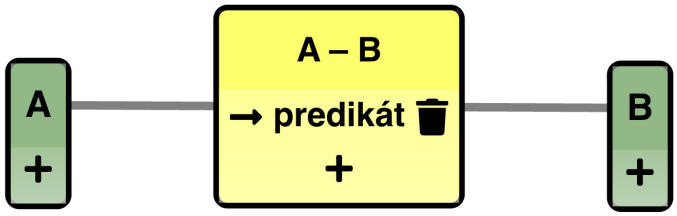
Úlohou bakalárskej práce Miroslava Baluchu [5] bolo rozšíriť aplikáciu Prieskumník štruktúr o nový pohľad na štruktúry logiky prvého rádu vo forme grafového editora. V jednom grafe sa mu podarilo pomocou rôznych typov vrcholov znázorniť konštanty, prvky domény a pomocou hrán aj vzťahy medzi nimi. Príklad zobrazenia konkrétnej štruktúry je na obrázku 2.3.

S grafom možno interagovať v dvoch režimoch. Prvým je pohybový režim, ktorý umožňuje iba mazanie a zmenu pozície jednotlivých objektov. V druhom, editovacom režime sú už sprístupnené funkcie ako vytváranie predikátových a funkčných symbolov, tvorba nových vzťahov alebo premenovávanie prvkov domény. Tlačidlá, pomocou ktorých sa medzi režimami prepína, sú viditeľné v ľavej hornej časti obrázka. Pod týmito tlačidlami sú umiestnené jednotlivé typy vrcholov. Ich potiahnutím do priestoru grafu sa vytvorí nová inštancia príslušného typu. Zelené vrcholy reprezentujú prvky domény a sivé konštanty. Žlté vrcholy pomáhajú pri tvorbe ternárnych a kvaternárnych vzťahov.

Binárny vzťah medzi prvkami domény je znázornený pomocou hrany spolu s jej popisom (viď obrázok 2.4). Do popisu možno pridávať alebo odoberať predikátové a



Obr. 2.3: Grafový editor v predošlej implementácii Prieskumníka štruktúr



Obr. 2.4: Príklad znázornenia binárneho vzťahu.

funkčné symboly, ktoré majú tento vzťah obsahovať vo svojej interpretácii. Pomocou šípky pri názve predikátu sa určuje smer vzťahu. Pri ternárnych a kvaternárnych vzťahoch je nutné najprv spojiť všetky prvky domény s príslušným vrcholom, pri ktorom sa následne dá meniť popis už spomínaným spôsobom.

Ako sa však časom ukázalo, implementácia mala niekoľko zásadných problémov. Hlavným z nich bola snaha nahradiť takmer celú funkcionalitu Prieskumníka štruktúr, čo na viacerých miestach viedlo ku kompromisom znižujúcim prehľadnosť aj intuitívnosť rozhrania. Azda najväčším vinníkom zlej prehľadnosti, resp. čitateľnosti, je spôsob zobrazovania binárnych vzťahov. Tých sa v štruktúrach používaných na predmete vyskytuje pomerne veľa, čo v kombinácii s ich rozsiahlou vizualizáciou vedie k prekrývaniu ostatných prvkov a výraznému zníženiu schopnosti orientácie sa v zobrazení.

Návrh niektorých ovládacích prvkov tiež pôsobil vo výsledku ťažkopádne, ako napríklad oddelenie editovania a presúvania do dvoch samostatných režimov. Ďalším proble-

mom je nedostatok zaujímavých a unikátnych funkcií v porovnaní s textovými vstupmi, ktoré by ho spravili pre študentov atraktívnejším. Spomínané nedostatky sa prejavili aj na jeho nízkej popularite na predmete, kde sa prakticky vôbec nevyužíval.

2.3 Logický pracovný zošit

Vývojom tejto webovej aplikácie sa vo svojej bakalárskej práci zaoberal Matej Mok [12]. Pred jej vznikom boli nástroje na výučbu matematickej logiky na predmete dostupné iba v podobe samostatných aplikácií, čo sa však s ich rastúcim počtom stávalo nepraktickým. Logický pracovný zošit tieto nástroje integruje do jedného celku, čím učiteľom predmetu umožňuje vytvárať ucelenejšie cvičenia a žiakom uľahčuje ich vypracovávanie, ako aj odovzdávanie. Ďalšou výhodou pracovného zošita je priebežné ukladanie práce žiakov, vďaka čomu sa k nej môžu kedykoľvek vrátiť. Učitelia majú zároveň k uloženým prácam prístup, čo im dovoľuje ich prezerať a pridávať komentáre.

Kapitola 3

Návrh

V tejto kapitole najprv zadefinujeme požiadavky kladené na výslednú aplikáciu. Následne predstavíme návrh jednotlivých riešení spolu s priblížením postupu, ktorým sme k ním dospeli. V závere sa zameriame na návrh stavu nových častí aplikácie a jeho zakomponovanie do už existujúceho riešenia.

3.1 Požiadavky

Požiadavky na rozšírenie aplikácie sa postupne vyvíjali a rozširovali, čo viedlo k pridaniu pomerne veľkého množstva novej funkcionality nad rámec grafových editorov. V tejto podkapitole opíšeme všetky požiadavky a vysvetlíme dôvod ich zaradenia.

3.1.1 Zámer rozšírenia aplikácie

Hlavným cieľom tejto práce je zlepšiť použiteľnosť existujúcej aplikácie Prieskumník štruktúr, používanej na predmete Logika pre informatikov. Ako bolo už bližšie spomenuté v podkapitole venovanej práve tomuto nástroju 2.1, súčasná verzia poskytuje iba veľmi obmedzený spôsob vizualizácie a manipulácie s interpretáciami predikátových a funkčných symbolov.

Primárnym zlepšením v tejto oblasti má byť rozšírenie nástroja o grafové editory, ktoré používateľom pomôžu pri vizualizácii a vytváraní binárnych predikátov a unárnych funkcií logiky prvého rádu. Konkrétne sa má jednať o orientovaný graf, bipartitný graf a Hasseho diagram. Každý z nich bol zvolený pre svoje unikátne vlastnosti, ktoré môžu byť nápomocné v rozličných situáciach.

Aplikáciu by sme však radi posunuli ďalej aj v iných smeroch, či už integráciou ďalších editorov, pridaním nových funkcionalít alebo vylepšením tých existujúcich.

3.1.2 Požiadavky na grafové editory

Každý z grafov by mal z hľadiska funkčnosti slúžiť ako plnohodnotná alternatíva k už existujúcemu množinovému pohľadu. Musí teda zobrazovať jednotlivé usporiadané dvojice, resp. vzťahy, ktoré konkrétna interpretácia opisuje a prvky domény, medzi ktorými tieto vzťahy vznikajú. Vzťahy sa tiež musia dať vytvárať a mazať.

Vyobrazenie grafov by malo byť prispôsobené konkrétnym vlastnostiam každej reprezentácie. Bipartitný graf musí jasne znázorňovať rozdelenie prvkov do dvoch skupín, pričom vytváranie nových hrán je dovolené iba z jednej z nich. Hasseho diagram by zas mal poskytnúť možnosť automatického vytvorenia hierarchie typickej pre túto reprezentáciu. Tiež by sa malo dbať na to, aby používateľ mohol vytvárať iba vzťahy spĺňajúce podmienky čiastočného usporiadania (bližšie opísané v sekcii 1.2.2).

Oproti množinovému pohľadu navyše pribudnú vylepšenia zamerané na prehľadnejšie zobrazenie ďalších vzťahov v štruktúre. Konkrétne by mali byť nejakým spôsobom vizualizované unárne predikáty, keďže tie vedia poskytnúť nápomocnú informáciu o vlastnostiach jednotlivých prvkov. Všetky unárne predikáty sa však vždy nemusia zobrazovať. Používateľ si z nich bude môcť vybrať a na základe príslušnosti prvkov domény k ich interpretáciám bude možné tieto prvky filtrovať, pričom prvky nevyskytujúce sa v žiadnej z vybraných interpretácií sa v grafe nezobrazia. Okrem toho by sa mali vhodným spôsobom zobrazovať aj konštanty, ktorými sú jednotlivé prvky domény označené.

3.1.3 Ďalšie požiadavky

Maticový a databázový editor: Jednou z pridaných požiadaviek je reimplementácia maticového a databázového editora z predošlej verzie Prieskumníka štruktúr. Ich účel je podobný ako pri grafových reprezentáciách, no okrem poskytnutia alternatívneho pohľadu dokážu reprezentovať aj iné než iba binárne predikáty a unárne funkcie. Pri týchto editoroch sa taktiež vyžaduje možnosť zobrazovať unárne predikáty a využívať ich na filtrovanie.

Editor interpretácií funkcií rozborom prípadov: Ďalšie vylepšenie má za cieľ uľahčiť tvorbu interpretácií funkčných symbolov. To býva často zdĺhavé, keďže doposiaľ spomínané riešenia vyžadujú explicitné definovanie všetkých prípadov. Dá sa to však vylepšiť nástrojom, ktorý umožní definovať podmienky na argumenty funkcie spolu s príslušnými hodnotami, ktoré má funkcia pri ich splnení nadobúdať. Požiadavkou je preto vytvorenie nástroja založeného na takomto princípe.

Dopyty: Toto rozšírenie na rozdiel od ostatných neslúži na tvorbu interpretácií. Malo by predstavovať samostatnú sekciu aplikácie určenú na definovanie otvorených formúl a následné získanie všetkých ohodnotení, pri ktorých sú tieto formuly splniteľné. Výsledok takéhoto dopytu by sa mal vhodne zobraziť v podobe tabuľky.

Obr. 3.1: Návrh spoločného rozhrania editorov.

Refaktorizácia častí aplikácie: Aplikácia si v aktuálnej verzii ukladá jednotlivé časti stavu primárne v textovej podobe, keďže to bol doteraz jediný spôsob reprezentácie dát. Pri našich nových požiadavkách je to však obmedzujúce a bude potrebné dáta ukladať štruktúrovanejším spôsobom. Taktiež logika zodpovedná za generovanie Henkinovej-Hintikkovej hry by mala v určitých situáciách fungovať na náhodnom výbere, no v súčasnej verzii tomu tak nie je. Požaduje sa preto oprava aj tohto správania.

3.2 Spoločné rozhranie editorov

Na obrázku 3.1 možno vidieť návrh rozhrania, do ktorého sú jednotlivé editory vkladané. Jeho cieľom je poskytnúť konzistentné rozhranie spoločných ovládacích prvkov naprieč všetkými druhmi editorov. Dokopy sa skladá z nasledujúcich komponentov:

Hlavička editora: V ľavej časti hlavičky je zobrazený názov predikátového alebo funkčného symbolu spolu s názvom aktuálneho editora. V pravej časti sa nachádza tlačidlo, po ktorého stlačení sa zobrazí zoznam všetkých kompatibilných editorov. Výber konkrétneho editora z ponuky zobrazenie automaticky prepne.

Panel unárnych predikátov: Tento komponent tvorí najdôležitejšiu časť celého rozhrania. Ústredným prvkom je horizontálny zoznam zaškrtávacích tlačidiel reprezentujúcich jednotlivé unárne predikáty, pričom každému z nich je priradená vlastná farba. Ako aj neskôr v návrhu uvidíme, kompatibilné editory túto farbu rôzne využívajú pri ich vizualizácii. Tlačidlo s dvoma fajkami, umiestnené naľavo, slúži na hromadné zaškrtnutie všetkých unárnych predikátov. Úplne vľavo sa nachádza zaškrtávacie tlačidlo označené písmenom „D“, tzv. doménové, ktorého funkcia je podrobnejšie vysvetlená v

časti 3.2.1 venovanej filtrovaniu.

Filter prvkov domény: Tlačidlo na rozbalenie filtrov je umiestnené napravo od panelu unárnych predikátov. Po jeho stlačení sa zobrazia zaškrtávacie tlačidlá reprezentujúce prvky domény. Odškrtnutím konkrétneho prvku indikujeme, že sa nemá nachádzať v zobrazení editora.

Chybový banner: Tento komponent sa zobrazí iba v prípade, že v interpretácii nastala chyba. V ľavej časti je vypísaná chybová správa. Ak sa navyše jedná o chybu, ktorú je možné opraviť, na pravej strane zároveň pribudne tlačidlo na jej opravu.

Zobrazenie editora: Do tejto časti je vložený komponent zvoleného editora.

3.2.1 Filtrovanie

Každý editor podporujúci filtrovanie nejakým spôsobom zobrazuje prvky domény. To, aké prvky budú zobrazené, ovplyvňuje viacero faktorov, ktoré sme počas vývoja menili. V jednom z prvých návrhov sa vždy zobrazovala celá doména. Neskôr sme toto správanie zmenili tak, aby boli zobrazené iba tie prvky, ktoré sa zároveň nachádzajú v interpretáciách zvolených unárnych predikátov. Videli sme však potenciálne využitie v oboch prístupoch, a preto sme sa ich rozhodli skombinovať pridaním tlačidla domény. Po jeho zaškrtnutí sa vždy zobrazuje celá doména, v opačnom prípade sa zobrazenie riadi interpretáciami unárnych predikátov.

Komponent filtra prvkov domény je separátny. Ideou za ním je umožniť viac selektívne filtrovanie. Výsledná zobrazená doména predstavuje prienik prvkov vybraných pomocou unárnych predikátov a prvkov zvolených vo filtri domény.

3.3 Grafové editory

Táto podkapitola sa zameriava na návrh editorov pre tri zvolené grafové reprezentácie, konkrétne ide o orientovaný graf, bipartitný graf a Hasseho diagram. Súčasťou je aj návrh jednotlivých komponentov grafu a spôsobu akým s nimi bude používateľ iteragovať.

3.3.1 Orientovaný graf

Hlavnou úlohou pri návrhu tejto grafovej reprezentácie bolo určiť, akým spôsobom sa budú jednotlivé informácie zobrazovať a ako s nimi bude používateľ následne interagovať. Tieto rozhodnutia sú kľúčové, keďže tvoria základ aj ostatných grafových editorov.

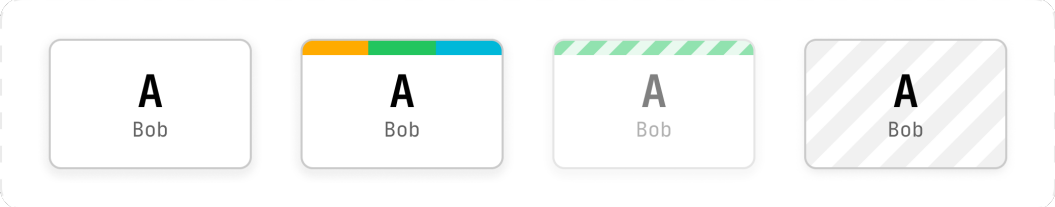
V samotnom grafe prvky domény prirodzene predstavujú vrcholy, zatiaľ čo vzťahy sú reprezentované pomocou hrán. Konštanty a unárne predikáty som sa taktiež roz-

hodol zakomponovať ako súčasť vrcholov, keďže ide o informácie prislúchajúce prvkom domény. Výzvou, ktorú takéto riešenie však prináša, je zachovať rovnováhu medzi množstvom informácií obsiahnutých vo vrcholoch a ich prehľadnosťou.

Prístup, ktorý som pri návrhu vrcholov zvolil, možno vidieť na obrázku 3.2. Najväčší typografický dôraz sa kladie na samotný prvok domény (na obrázku veľké písmeno A). Pod ním sa nachádzajú konštanty reprezentujúce tento prvok (na obrázku nápis Bob). Na druhom vrchole zľava možno vidieť panel predstavujúci unárne predikáty, do interpretácie ktorých daný prvok patrí, pričom každá farba symbolizuje konkrétny predikát. Práve použitie farieb namiesto iných alternatív, ako napríklad nápisov s názvom unárneho predikátu, má pomôcť s celkovou prehľadnosťou vrcholov.

Pri takomto prístupe sme však narazili na problém s dostupnosťou. Môj pôvodný výber totiž obsahoval aj také kombinácie farieb, ktoré by pre ľudí s poruchami farebného videnia neboli dostatočne rozlíšiteľné. Pri výbere vhodnej palety aj pre takéto prípady sme sa inšpirovali článkom Qualitative Color Schemes [13]. Ten obsahuje niekoľko príkladov vhodných farebných paliet spolu s návodom, ako s nimi zaobchádzať.

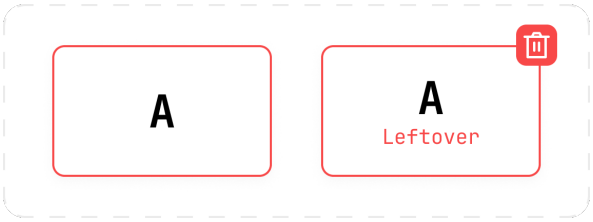
Posledné dva vrcholy na obrázku 3.2 sa aktivujú iba v prípade, keď používateľ prejde kurzorom na zaškrtávacie tlačidlo unárneho predikátu. Ak by zaškrtnutie tohto tlačidla znamenalo pridanie vrcholu do zobrazenia, objaví sa jeho čiastočne priehľadná verzia (druhá sprava). Ak by naopak zaškrtnutie viedlo k odobraniu daného vrcholu, je nahradený jeho vyšrafovanou verziou (prvá sprava). Cieľom je používateľovi jasnejšie naznačiť, aký vplyv bude mať jeho voľba na výsledné zobrazenie grafu.



Obr. 3.2: Návrh vizuálu vrcholov.

Ďalej sme sa rozhodli pomocou vrcholov vyjadriť aj určité chybové stavy. Tie môžu nastať v dvoch prípadoch. Jedným z nich je situácia, keď interpretácia obsahuje prvok, ktorý nepatrí do domény. Vrchol reprezentujúci takýto prvok je zobrazený v pravej časti obrázka 3.3. Okrem vizuálnej informácie v podobe červeného orámovania a nápisu poskytuje aj možnosť chybný prvok odstrániť kliknutím na tlačidlo koša, čím sa interpretácia opraví. Druhý prípad nastáva pri interpretáciách funkčných symbolov. Vtedy je vrcholom v ľavej časti obrázka označený prvok, ktorý nemá priradenú hodnotu alebo ich má priradených viac.

Dôležitou funkcionalitou grafu je aj spôsob vytvárania hrán. Na rozdiel od predošlej implementácie sme sa rozhodli nerozdeľovať ovládanie na dva režimy (pohybový a editovací). To si však vyžaduje, aby bola do vrcholu naraz zakomponovaná možnosť



Obr. 3.3: Návrh vizuálu chybových vrcholov.

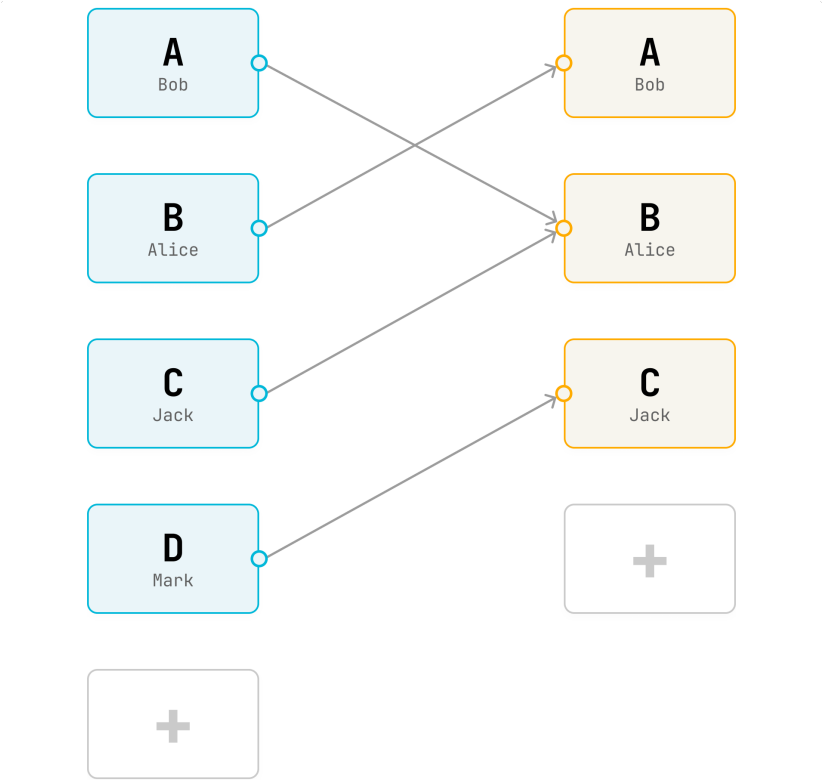
jeho presúvania, ako aj tvorby novej hrany. Žiadne z riešení ponúkaných knižnicou nám nevyhovovalo, preto sme sa rozhodli pre vlastný prístup, pri ktorom je vrchol rozdelený na dve časti – jednu určenú na vytváranie hrán a druhú na jeho presúvanie. Vytváranie hrany iniciujeme umiestením kurzora nad určitú oblasť po obvode vrchola. Používateľovi je táto oblasť naznačená jej stmavením a zmenou typu kurzora. Následným potiahnutím z tejto oblasti nad iný vrchol a pustením sa proces vytvárania ukončí. Počas vytvárania sa hrana zobrazuje červenou alebo zelenou farbou podľa toho, či je jej vytvorenie validné. Posúvanie vrcholu je umožnené na zvyšnej ploche, teda okolo jeho stredu.

3.3.2 Bipartitný graf

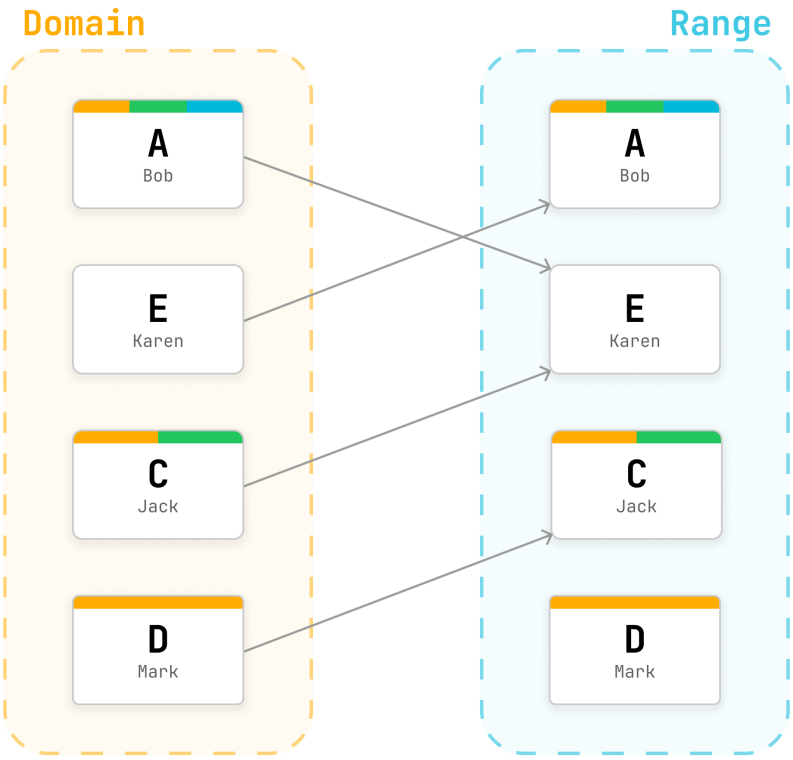
Prvý návrh tejto reprezentácie možno vidieť na obrázku 3.4. Hrany reprezentujú šípky smerujúce zľava doprava, pričom ide o jediný smer, v ktorom sa dajú vytvárať. Vrcholy sú rozdelené do dvoch stĺpcov, čo je naznačené aj ich rozdielnou farbou. Každý stĺpec reprezentuje ľubovoľnú podmnožinu prvkov danej domény. Používateľovi je tiež umožnené meniť poradie vrcholov, ale iba v rámci jeho skupiny, presúvanie medzi nimi nie je dovolené. Posledný, menej výrazný vrchol sa líši od ostatných, pretože slúži ako tlačidlo na pridávanie ďalších prvkov do príslušného stĺpca. Vytváranie hrán sa realizuje pomocou krúžkov na okraji vrchola potiahnutím z počiatočného na koncový vrchol. V tomto prípade je takéto riešenie v poriadku (narozdiel od orientovaného grafu), keďže smerovanie hrany je vždy jednoznačné, a preto nebude spôsobovať zbytočné prelínanie s počiatočným vrcholom.

Uvedené riešenie nám však nevyhovovalo, a preto sme sa rozhodli na ňom ešte zapracovať v druhom, finálnom návrhu, ktorý je vyobrazený na obrázku 3.5.

Jednou zo zmien je zjednotenie funkcionality a vizuálu vrcholov podľa vzoru orientovaného grafu. Príslušnosť vrcholov do skupiny je naznačená pomocou dvoch kontajnerov s rozdielnou farbou a názvom danej skupiny. Pre jednoduchosť sme sa tiež rozhodli odstrániť tlačidlá na pridávanie ďalších prvkov do skupiny. V dôsledku toho obe časti vždy obsahujú rovnakú podmnožinu domény, ovládanú iba pomocou filtrov. Týmito úpravami sa nám podarilo dosiahnuť konzistentnejšie správanie naprieč grafovými editormi. Všetky ostatné funkcie sme ponechali.



Obr. 3.4: Prvý návrh vizuálu pre bipartitný graf.

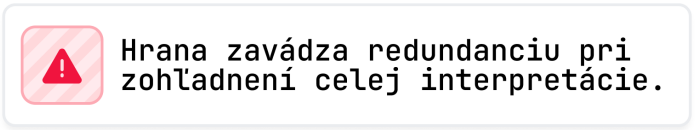


Obr. 3.5: Druhý návrh vizuálu pre bipartitný graf.

3.3.3 Hasseho diagram

Spôsob fungovania vrcholov a hrán grafu zostal v zásade nezmenený. Pri tejto reprezentácii sme sa však zamerali najmä na overovanie, či interpretácia binárneho predikátu skutočne tvorí čiastočné usporiadanie, čo je nevyhnutný predpoklad pre zmysluplnosť tejto reprezentácie.

Pri vytváraní nových hrán sa preto kontroluje, či by ich pridaním ostali zachované všetky podmienky čiastočného usporiadania. Ak by tomu tak nebolo, používateľ je už počas vytvárania hrany upozornený varovnou správou v spodnej časti rozhrania grafu s vysvetlením, akú vlastnosť by tým porušil. Príklad takejto správy možno vidieť na obrázku 3.6.



Obr. 3.6: Príklad varovnej správy Hasseho diagramu.

Ďalšou požiadavkou bolo automatické generovanie vertikálnej hierarchie vrcholov, typickej pre túto reprezentáciu. Rozhodli sme sa však, že nebudeme používateľa akokoľvek obmedzovať za účelom dodržiavania tejto hierarchie, ako tomu bolo v bipartitnom grafe, ale poskytneme mu dobrý základ rozmiestnenia vrcholov, ktorý si môže ďalej prispôsobiť podľa vlastných predstáv. Samotné generovanie preto prebieha iba pri prvom otvorení grafu alebo na vyžiadanie používateľa stlačením tlačidla na paneli nástrojov.

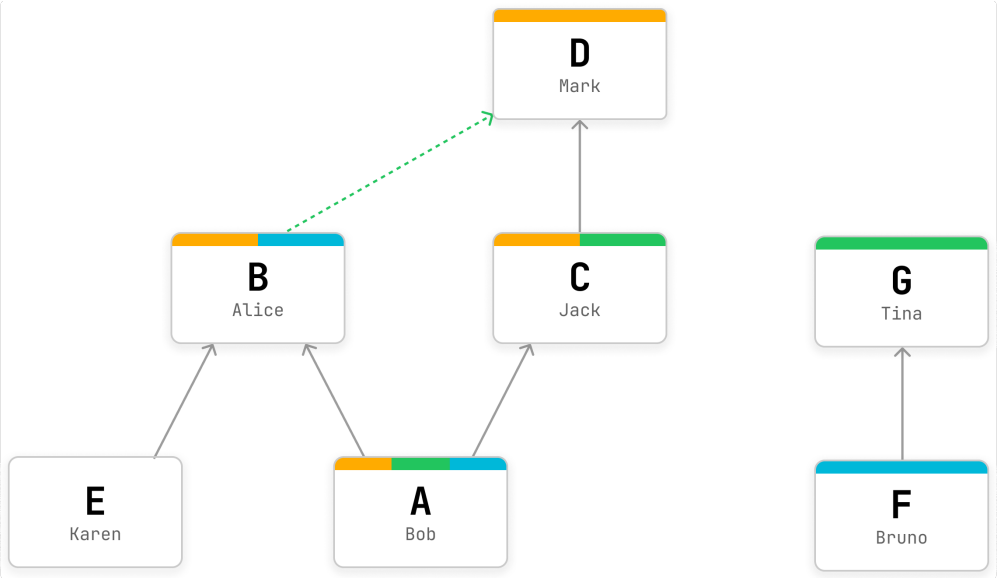
Môj návrh je zobrazený na obrázku 3.7. Viditeľná je jasná hierarchia vrcholov s orientovanými hranami smerujúcimi nahor. Zelenou prerušovanou čiarou je používateľovi naznačená korektnosť hrany, ktorú sa práve snaží pridať (tu je hrana v poriadku). V opačnom prípade by bola hrana zvýraznená červenou farbou a používateľovi by sa zobrazila už spomínaná varovná správa.

3.4 Tabuľkové editory

V tejto podkapitole opíšeme návrh ďalších dvoch typov editorov. Oba sú unikátne tým, že svoje zobrazenie prispôsobujú jednak tomu, či reprezentujú interpretáciu predikátového alebo funkčného symbolu, ako aj arite daného symbolu.

3.4.1 Maticový editor

Návrh editora reprezentujúceho konkrétny binárny predikát je znázornený na obrázku 3.8. Horizontálnu aj vertikálnu hlavičku tabuľky tvoria prvky domény, v tomto prípade sú to veľké písmená. Príslušnosť prvkov k interpretáciám unárnych predikátov



Obr. 3.7: Návrh vizuálu pre Hasseho diagram.

je vizualizovaná podobne ako pri grafoch, teda tiež pomocou farieb. Pre malú veľkosť dostupného priestoru bolo však v tomto prípade potrebné zvoliť kompaktnejšie riešenie v podobe krúžkov.

Každá usporiadaná dvojica prvkov domény je reprezentovaná zaškrtávacím políčkom, ktorým je možné jednoducho zaznačiť, či patrí alebo nepatrí do interpretácie. Editor tiež podporuje zobrazenie interpretácií unárnych predikátov. V takom prípade sa zobrazí iba horizontálna hlavička, keďže je potrebné zaškrtávať konkrétny prvok.

Pri reprezentáciách unárnych a binárnych funkčných symbolov sú zaškrtávacie políčka nahradené textovými vstupmi, pretože okrem argumentu funkcie sa vyžaduje určiť aj hodnotu, ktorú má funkcia nadobúdať. Inak je jeho rozloženie v zásade identické. Obsah týchto textových vstupov je potrebné aj validovať, aby sa overilo, že používateľ skutočne zadáva výhradne prvky domény. V prípade chyby sa príslušné pole zvýrazní červenou farbou a používateľovi sa na chybovom banneri zobrazí vysvetlenie problému.

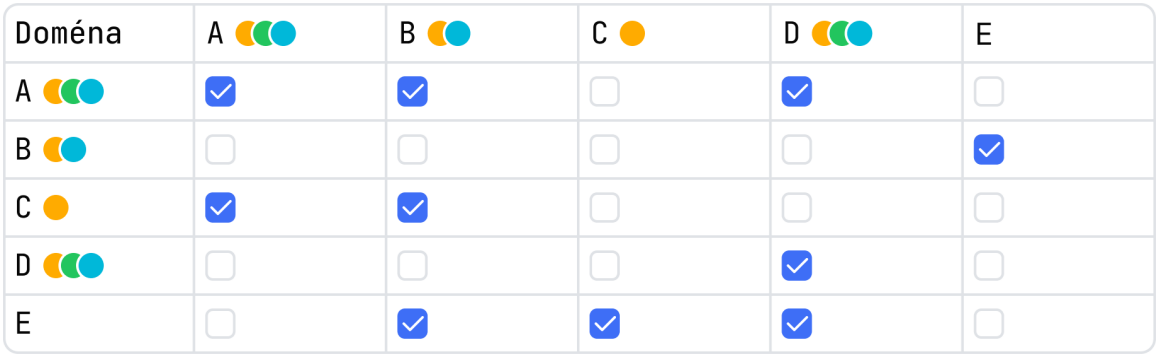
3.4.2 Databázový editor

Príklad editora reprezentujúceho konkrétny ternárny predikát možno vidieť na obrázku 3.9. V tejto reprezentácii sú jednotlivé trojice prvkov zoskupené do riadkov. Na rozdiel od predošlého editora funguje vždy na báze textových vstupov. Príslušnosť prvkov k interpretáciám unárnych predikátov je vyznačená priamo pri nich. Posledný riadok slúži na pridávanie nových trojíc. K pridaniu však dôjde iba vtedy, ak sú správne vyplnené všetky vstupy v riadku. Tlačidlo s ikonou koša slúži na odstránenie príslušnej trojice.

Pri reprezentáciách funkčných symbolov funguje rozhranie opäť trochu iným spôsobom. Najprv sú automaticky generované všetky n-tice prvkov domény, kde n zodpovedá

arite funkčného symbolu. Tieto n-tice sa následne zobrazia v tabuľke a ku každej z nich je sprava priradený jeden ďalší textový vstup, ktorý určuje, akú hodnotu má funkcia nadobúdať pre príslušnú n-ticu. Takýto prístup je zvolený kvôli tomu, že používateľ musí vždy poskytnúť celú definíciu funkcie. Z tohto dôvodu nie je potrebné ani vhodné, aby musel jednotlivé n-tice pridávať manuálne.

Databázový editor je špecifický aj tým, že sa dá ľahko prispôsobiť ľubovoľnej arite, keďže tá priamo súvisí s počtom stĺpcov tabuľky. Pri interpretáciách funkčných symbolov sme však rozhodli obmedziť jeho použiteľnosť maximálnou veľkosťou arity 5. Totižto generované tabuľky by už pri vyšších aritách boli príliš veľké a prakticky nepoužiteľné.



Obr. 3.8: Návrh rozhrania pre maticový editor.

Obr. 3.9: Návrh rozhrania pre databázový editor.

3.5 Editor interpretácií funkcií rozborom prípadov

Zámerom tohto editora je umožniť definovať interpretácie funkčných symbolov pomocou rozboru prípadov. Príklad takejto definície pre funkciu s dvoma argumentmi (3.1)

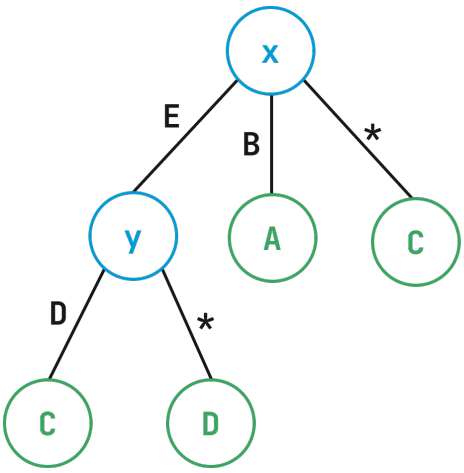
je uvedený nižšie.

$$f(x,y) = \begin{cases} \text{A}, & \text{ak } x = \text{B} \\ \text{C}, & \text{ak } x = \text{E a } y = \text{D} \\ \text{B}, & \text{inak} \end{cases} \tag{3.1}$$

Jednotlivé prípady predstavujú rôzne špecifické podmienky. Napríklad prvá sa vzťahuje iba na argument x, zatiaľ čo v druhej sa zohľadňuje aj y. Aby bola definícia úplná, je potrebné špecifikovať aj prípad, keď nie je splnená žiadna z uvedených podmienok. V príklade je tento prípad označený ako „inak“.

Keďže editor by mal fungovať pre ľubovoľný počet argumentov, bolo v prvom rade potrebné zvoliť vhodnú dátovú reprezentáciu takejto štruktúry. Rozhodli sa pre všeobecný strom, ktorého príklad možno vidieť na obrázku 3.10. Uzly stromu buď predstavujú argumenty funkcie (označené modrou), alebo konkrétne hodnoty (označené zelenou). Uzol reprezentujúci argument môže mať potomkov, pričom hrany z takéhoto uzla vyjadrujú podmienky pre daný argument. Naopak, uzly reprezentujúce hodnoty sú vždy listy stromu.

Každá cesta od koreňa k listu teda určuje konkrétnu kombináciu podmienok vedúcich k výslednej hodnote. Hrany označené hviezdičkou predstavujú všetky ostatné prípady, teda si ich možno predstaviť aj ako zvyšné prvky domény, ktoré neurčujú žiadnu konkrétnu podmienku vychádzajúcu z daného uzla.



Obr. 3.10: Všeobecný strom pre Editor interpretácií funkcií rozborom prípadov.

Takýto strom nám umožňuje jednoznačne reprezentovať interpretáciu funkčných symbolov, a to aj pomocou relatívne malého množstva dát. V porovnaní s ostatnými editormi to predstavuje veľkú výhodu, keďže tie vyžadujú explicitné definovanie celej interpretácie.

Ďalej bolo potrebné pre editor navrhnúť vhodné používateľské rozhranie, ktoré dovolí tvorbu takýchto stromov. Počas jeho vývoja sme prešli viacerými rôznymi návrhmi.

Pre poskytnutie lepšej predstavy o tom, čo viedlo k finálnemu návrhu, sú ďalej jednotlivé verzie v krátkosti popísané.

Prvý návrh

Prvý spôsob reprezentácie možno vidieť na obrázku 3.11. Jednotlivé riadky reprezentujú konkrétne cesty v strome od koreňa až po list, pričom na vzor matematického zápisu je hodnota v liste uvedená ako prvá. Všetky editovateľné časti sú reprezentované ako textové vstupy. Sprísnenie podmienky, resp. pridanie ďalšieho argumentu, sa realizuje pomocou príslušných tlačidiel v každom riadku. V kontexte stromovej reprezentácie je to rovnaké ako pridať hranu s uzlom reprezentujúcim argument. Vymazať konkrétny riadok je možné pomocou tlačidla s ikonou koša.

Na tejto reprezentácii nám však nevyhovovalo používateľské rozhranie, pretože sa nám nezdalo byť dostatočne intuitívne. Napríklad nebolo zrejmé, ako by mali byť označované jednotlivé argumenty, čo by bolo najmä pre nového používateľa dosť mätúce. Zároveň rozhranie neumožňovalo vykonávať niektoré dôležité úkony, ako napríklad pridávanie nových podmienok.

Druhý návrh

Druhý návrh je realizovaný tak, aby veľmi blízko reprezentoval samotnú stromovú štruktúru. Konkrétny príklad možno vidieť na obrázku 3.12. Uzly reprezentujúce argumenty sú znázornené slovom „switch“ spolu s textovým vstupom na označenie argumentu. Pre každý takýto uzol sa následne definujú podmienky, ktoré sú znázornené slovom „case“ s príslušným textovým vstupom pre jej hodnotu. Podmienka znázornená slovom „default“ označuje všetky ostatné prvky domény a pridáva sa automaticky, keďže vždy musí mať určenú hodnotu. Presunutím kurzora na konkrétnu podmienku sa zobrazí menu. V ňom je pomocou tlačidla s ikonou koša možné danú podmienku vymazať, druhé tlačidlo slúži na pridanie novej podmienky. Po jeho stlačení je ešte potrebné určiť, či podmienka bude obsahovať priamo hodnotu, alebo ďalší argument funkcie.

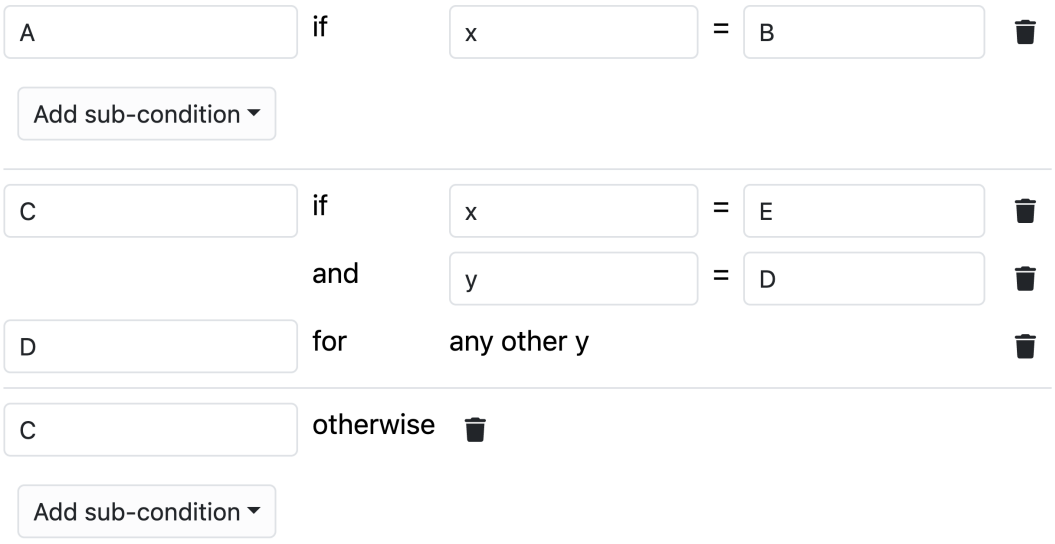
V tejto verzii bola tiež pridaná validácia jednotlivých vstupov. Napríklad sa kontroluje, či používateľ nezadal viac podmienok s rovnakou hodnotou, alebo či nebol ten istý argument použitý už v niektorom rodičovskom uzle.

Dôvod, prečo sme nezostali pri tejto reprezentácii, bol najmä ten, že jej vizuálna podoba nezodpovedala matematickej definícii. Napriek tomu nám však poskytla veľmi dobrý základ, z ktorého sme mohli vychádzať pri ďalšom návrhu.

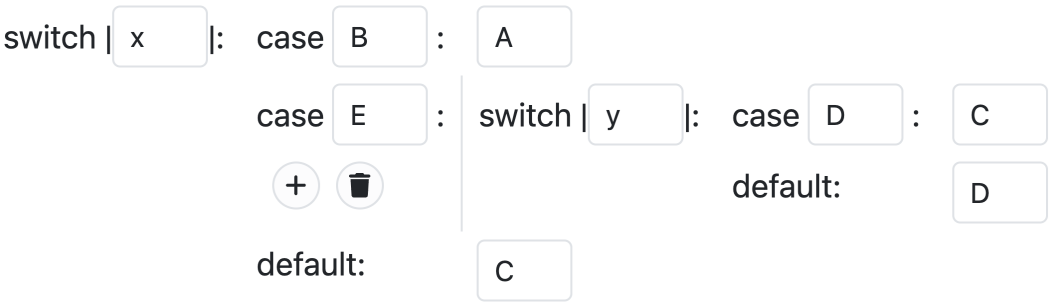
Tretí návrh

Pri finálnom návrhu (obr. 3.13) sme sa snažili, aby čo najvernejšie zodpovedal matematickému zápisu. Spôsob rozdelenia jednotlivých prípadov na riadky je podobný ako pri prvom návrhu. Rozdiel je však v tom, že tlačidlo na bližšiu špecifikáciu podmienky sa nachádza na pravej strane príslušného riadku. Po jeho stlačení sa zobrazí výber argumentov, ktoré sa v podmienke ešte nevyskytujú. Po výbere sa automaticky pridá nová podmienka s predvyplneným textovým vstupom podľa zvoleného argumentu. Pridanie novej podmienky sa realizuje prostredníctvom menej výrazných, čiastočne priehľadných riadkov. Tie sa v kontexte stromu nachádzajú všade tam, kde je ešte možné pridať ďalšiu podmienku k uzlu reprezentujúcemu argument. Jednotlivé vstupy sú taktiež validované, podobne ako pri druhom návrhu.

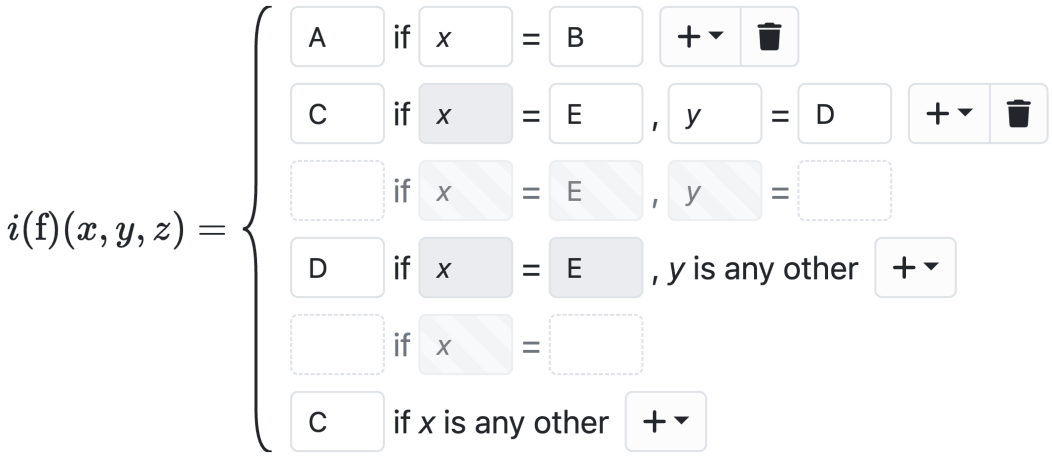
Pri takomto spôsobe reprezentácie stromu je však často potrebné zobrazovať jeden uzol reprezentujúci argument naprieč viacerými riadkami. Preto sme sa rozhodli povoliť úpravu tohto argumentu iba v tom riadku, v ktorom sa vyskytuje po prvýkrát. V ostatných riadkoch je príslušný textový vstup deaktivovaný.



Obr. 3.11: Prvý návrh pre Editor interpretácií funkcií rozborom prípadov.



Obr. 3.12: Druhý návrh pre Editor interpretácií funkcií rozborom prípadov.



Obr. 3.13: Tretí návrh pre Editor interpretácií funkcií rozborom prípadov.

3.6 Dopyty

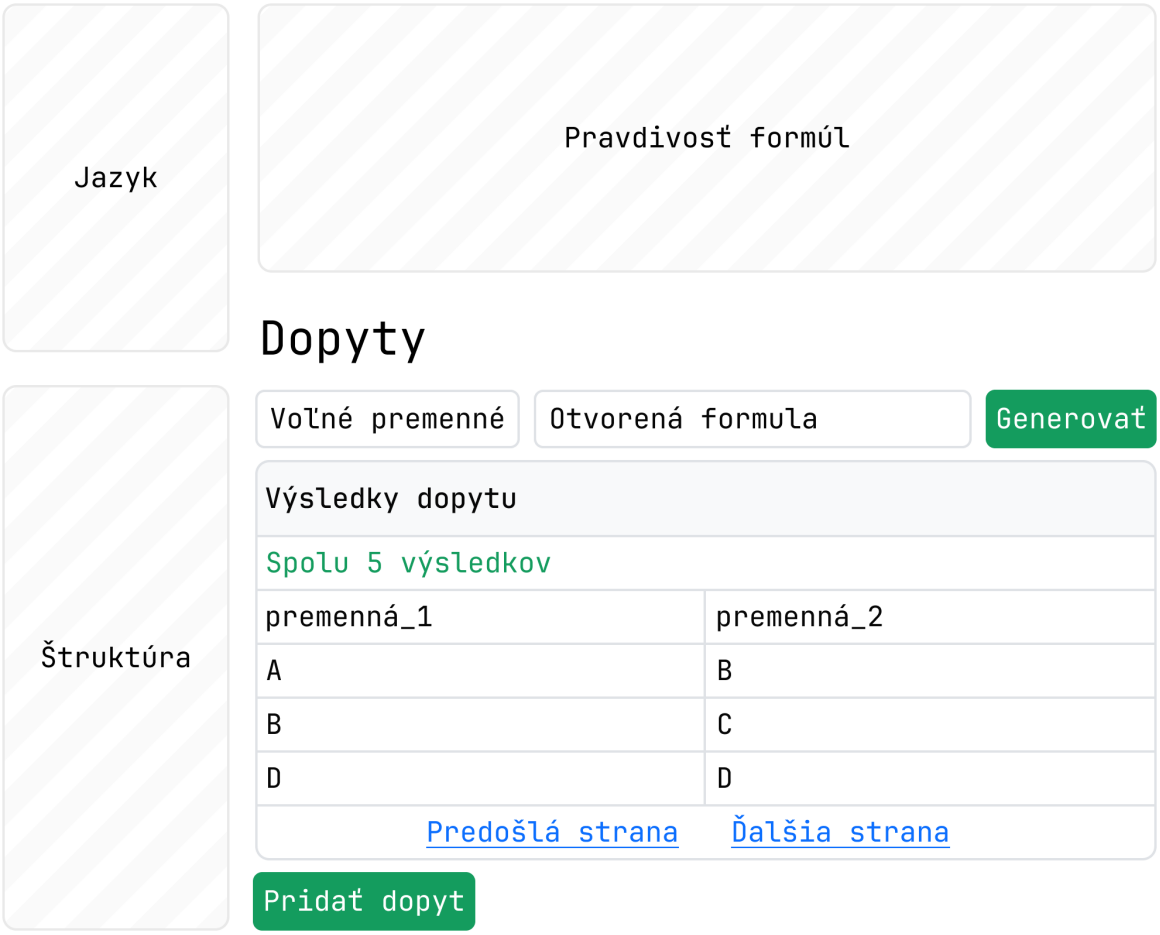
Tento komponent neslúži ako editor, ale reprezentuje samostatnú časť aplikácie, v ktorej je možné definovať dopyty na splniteľnosť otvorených formúl logiky prvého rádu. Návrh rozhrania a umiestnenia v aplikácii možno vidieť na obrázku 3.14.

V kontexte aplikácie má komponent štandardné rozhranie. Na samom vrchu sa nachádza jeho názov, pod ním sú definované dopyty. Pridávať nové, prázdne dopyty je možné pomocou tlačidla v dolnej časti. Ďalej sa pozrieme na rozhranie konkrétneho dopytu, ten je na obrázku zobrazený v strede.

Na definovanie dopytu je najprv potrebné určiť voľné premenné, podľa ktorých sa budú generovať ohodnotenia. Tie sa zadávajú do prvého textového vstupu zľava a musia byť oddelené čiarkou. Druhý textový vstup hneď vedľa slúži na definovanie otvorenej formuly, ktorú chceme vyhodnotiť. Oba vstupy sú validované. Pri premenných sa kontroluje správny formát, ako aj to, či skutočne patria medzi voľné premenné danej formuly. Ak nie, používateľovi sa pod vstupom zobrazí chybová správa s informáciou o konkrétnych premenných, ktoré podmienku nespĺňajú. Validácia formuly využíva už existujúce riešenie používané v aplikácii. Bolo ho však potrebné doplniť tak, aby zohľadňovalo aj používateľom definované premenné.

Ak sú oba vstupy vyplnené správne, používateľ môže stlačiť tlačidlo úplne napravo. Tým sa dopyt vyhodnotí, teda sa vygenerujú všetky možné ohodnotenia, ktoré spĺňajú danú otvorenú formulu. Následne sa zobrazí počet výsledkov spolu s ich zoznamom. Každý riadok predstavuje jedno ohodnotenie a stĺpce zodpovedajú jednotlivým premenným. Zoznam je navyše stránkovaný, takže aj pri väčšom množstve ohodnotení sa zobrazí iba obmedzený počet výsledkov. Prechádzanie strán zoznamu je umožnené tlačidlami na jeho spodku.

Problém, ktorý môže nastať je, ak používateľ po vygenerovaní výsledkov nejakým spôsobom upraví štruktúru. To totiž často znamená, že vygenerované ohodnotenia už



Obr. 3.14: Návrh rozhrania a umiestnenia pre komponent dopytov.

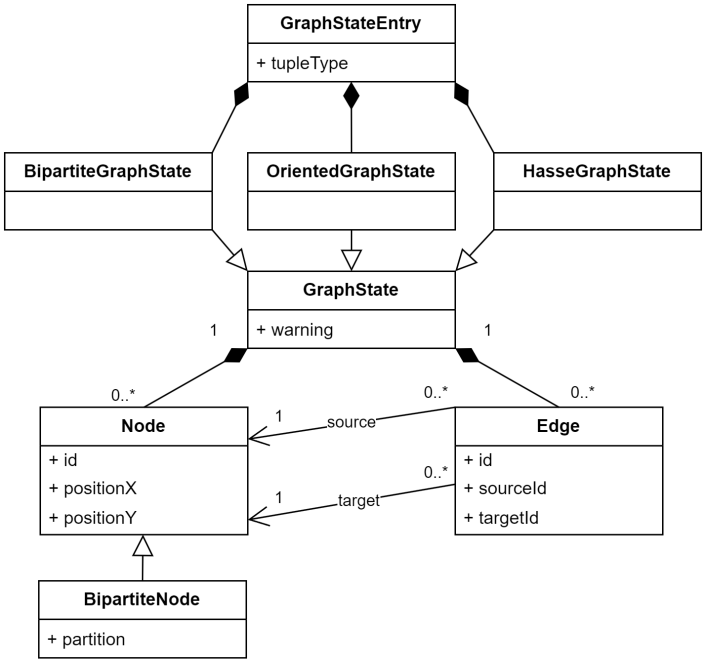
nie sú aktuálne.

Jedným z riešení bolo pri každej zmene štruktúry všetky existujúce výsledky skryť. Na ich opätovné zobrazenie by bolo potrebné opäť stlačiť tlačidlo na generovanie. Táto možnosť nám však nevyhovovala, pretože sme chceli, aby boli výsledky stále k dispozícii. Pre používateľa môžu byť totiž nápomocné, najmä počas úprav samotnej štruktúry.

Nakoniec sme sa rozhodli zvoliť prístup, pri ktorom sa výsledky po upravení štruktúry neskryjú, ale sú označené ako potenciálne neaktuálne pomocou varovnej správy zobrazenej v hlavičke výsledkovej tabuľky. Na odstránenie tohto varovania ich používateľ musí opäť vygenerovať.

3.7 Nový stav aplikácie

V tejto podkapitole sú predstavené návrhy stavov nových častí aplikácie. Všetky opísané riešenia sú integrované do centrálneho stavu aplikácie spolu s jeho existujúcimi časťami. Tam, kde je to vhodné, sú návrhy navyše doplnené aj diagramom znázorňujúcim ich dátový model.



Obr. 3.15: Triedny diagram pre grafové editory.

Textový pohľad

V pôvodnej implementácii je stav organizovaný do viacerých logických celkov zodpovedajúcich jednotlivým častiam aplikácie, spomedzi nich sú pre nás dôležité najmä *language* a *structure*. Oba uchovávajú väčšinu svojich dát výhradne v textovej podobe. Teda napríklad pri štruktúre je interpretácia konkrétneho predikátu reprezentovaná ako text, ktorý používateľ zadal do príslušného textového vstupu.

Tento spôsob reprezentácie by bol však pri implementácii našich rozšírení veľmi nepraktický a neefektívny, keďže by si vyžadoval neustále premieňanie medzi textovou a štruktúrovanou formou interpretácie používanou v jednotlivých editoroch. Na druhej strane, pre správne fungovanie textových vstupov je potrebné, aby bol aj naďalej ukladaný ich presný obsah, keďže používateľ môže zadať aj syntakticky nesprávne vstupy, ktoré sa nedajú priamo previesť do štruktúrovanej formy.

Rozhodli sme sa preto zaviesť novú časť stavu dedikovanú ukladaniu obsahu všetkých textových vstupov spomínaných častí aplikácie. Každý takýto vstup je reprezentovaný pomocou záznamu, ktorý obsahuje okrem samotného textového reťazca aj informáciu o type reprezentovaných dát (napr. či ide o indivíduové konštanty alebo interpretáciu funkčného symbolu). Jednotlivé záznamy sú unikátne identifikované typom reprezentovaných dát a v prípade interpretácií symbolov konštánt, predikátov a funkcií aj ich názvom.

V pôvodných častiach stavu sú všetky textové reprezentácie nahradené vhodnou štruktúrovanou formou. Napríklad v prípade interpretácií predikátov ide o zoznam jednotlivých n-tíc prvkov domény, kde n je jeho arita.

Grafové editory

V tejto časti stavu sa ukladajú dáta jednotlivých grafových reprezentácií. Návrh dátového modelu je znázornený na obrázku 3.15. Každý binárny predikát a unárna funkcia predstavuje jeden záznam v podobe `GraphStateEntry`, pričom jednotlivé záznamy sú jednoznačne identifikované kombináciou názvu a typu príslušného symbolu.

Každý `GraphStateEntry` obsahuje stav (`GraphState`) pre jednotlivé grafové typy. Tento stav zahŕňa zoznam vrcholov (`Node`) a hrán (`Edge`). Hrana je navyše asociovaná s dvojicou vrcholov, pričom jeden je počiatočný (`source`) a druhý koncový (`target`). Vrchol bipartitného grafu (`BipartiteNode`) rozširuje `Node` o atribút `partition`, ktorý určuje jeho príslušnosť k jednej z dvoch skupín.

Maticový a databázový editor

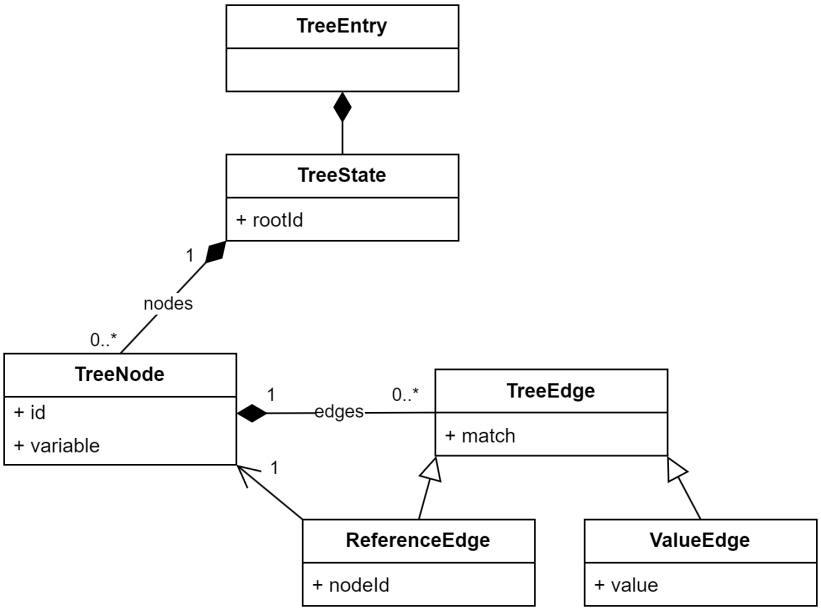
Rovnako ako pri grafových editoroch je záznam o ich stave jednoznačne identifikovaný pomocou názvu a typu predikátového alebo funkčného symbolu. V prípade databázového editora je stav ukladaný ako zoznam hodnôt zadaných do textových vstupov pre jednotlivé n-tice. Naopak maticový editor pre každú n-ticu zaznamenáva buď stav zaškrtávacieho políčka, alebo hodnotu textového vstupu, v závislosti od typu reprezentácie.

Editor interpretácií funkcií rozborom prípadov

Stav tejto reprezentácie je veľmi podobný stromovej štruktúre opísanej v podkapitole 3.5. Návrh dátového modelu editora možno vidieť na obrázku 3.16. Každý funkčný symbol zodpovedá jednému `TreeEntry` záznamu, ktorý je jednoznačne identifikovaný podľa názvu symbolu. Takýto záznam obsahuje stav editora (`TreeState`), v ktorom sú uložené jednotlivé uzly (`TreeNode`) spolu s identifikátor koreňového uzla. Každý uzol je zas tvorený zoznamom z neho vychádzajúcich hrán (`TreeEdge`). Abstraktný typ `TreeEdge` špecializujú dva typy hrán. Jednou z nich je referenčná hrana (`ReferenceEdge`), ktorá si uchováva identifikátor koncového uzla, teda ďalšej inštancie `TreeNode`. Druhou je hodnotová hrana (`ValueEdge`), tá priamo ukladá samotnú hodnotu.

Dopyty

Táto časť stavu má obsahovať zoznam jednotlivých dopytov. Každý dopyt sa skladá z textovej reprezentácie premenných a formuly.



Obr. 3.16: Triedny diagram pre editor interpretácií funkcií.

Spoločné rozhranie editorov

V tejto časti sa ukladá informácia o tom, ktorý editor je aktuálne otvorený a o jednotlivých nastaveniach filtrovania. Konkrétne sa jedná o stav zaškrtávacieho tlačidla domény, názvy zaškrtnutých unárnych predikátov ako aj zoznam prvkov domény označených v komponente filtra domény. Záznam o tomto stave je opäť jednoznačne identifikovaný rovnakým spôsobom, ako bolo už spomenuté v opisoch predošlých stavov.

Kapitola 4

Implementácia

V tejto kapitole sa zameriame na vybrané časti implementácie. Tie majú priblížiť hlavné problémy riešené počas vývoja a zároveň vysvetliť zvolené prístupy a ich realizáciu pomocou použitých technológií. Cieľom je taktiež poskytnúť základný prehľad implementácie, ktorý uľahčí orientáciu v systéme pri jeho prípadnom budúcom rozšírení.

4.1 Použité technológie

Nová verzia pôvodnej aplikácie je plne implementovaná v jazyku TypeScript. Používateľské rozhranie je vytvorené pomocou knižnice React a na správu globálneho stavu sa používa knižnica Redux (všetky sú opísané v podkapitole 1.3). Prirodzene sme sa teda rozhodli pokračovať v používaní týchto technológií aj pri tvorbe našich rozšírení. TypeScript sa ukázal ako veľmi nápomocný aj počas oboznamovania sa s existujúcou implementáciou, keďže vďaka definovaným typom bolo výrazne jednoduchšie porozumieť očakávaným tvarom objektov alebo argumentov funkcií.

Tiež bolo potrebné vybrať knižnicu, pomocou ktorej budú implementované grafové editory. V kontexte už spomenutých technológií sme ako najvhodnejšie riešenie zvolili knižnicu React Flow. Na základe môjho prieskumu išlo o jedinú knižnicu, ktorá podporovala už spomenuté technológie a zároveň poskytovala dostatočné množstvo konfigurácie na implementáciu všetkých navrhnutých funkcionalít. Veľkou výhodou bola aj kvalitná dokumentácia doplnená sadou interaktívnych príkladov, ktoré výrazne uľahčovali objavovanie všetkých dostupných možností.

4.2 Integrácia nových nástrojov

V tejto podkapitole priblížime spôsob, akým sme predstavené nástroje zakomponovali do už existujúcej aplikácie.

4.2.1 Organizácia

Súborovú štruktúru pôvodnej aplikácie sme v zásade nemenili. Držali sme sa stanovenej konvencie, podľa ktorej je každá funkcionalita organizovaná v priečinku `src/features`. Zdrojové súbory pre jednotlivé nástroje preto ukladáme do samostatných podpriečinkov pomenovaných podľa daného nástroja. Obsah týchto priečinkov je rôzny, no každý obsahuje aspoň hlavný React komponent pre daný nástroj a súbor definujúci jeho Redux stav. V rámci stavu sú spravidla implementované reducery a potrebné selektory.

4.2.2 Integrácia komponentov editorov

Predošlá verzia aplikácie používala na zobrazovanie interpretácií textové vstupy, tie boli obalené v komponente `InterpretationInput` a následne používané naprieč aplikáciou. Pre náš účel však potrebujeme riešenie, ktoré bude vedieť vybrať a zobraziť vhodný komponent na základe aktuálne zvoleného editora.

Tento problém rieši nový komponent `InterpretationEditor`. Ten zaobaľuje všetky editory vrátane pôvodného. Jedným z jeho vstupných parametrov je názov reprezentovaného symbolu. Na základe tohto názvu získa zo stavu typ editora, ktorý sa má zobraziť. Ak ide o textový editor, zobrazuje sa pôvodný komponent `InterpretationInput`. V opačnom prípade sa vykreslí nový komponent `DrawerEditor`.

`DrawerEditor` integruje spoločné rozhranie editorov pozostávajúce z ďalších troch častí. Jednou z nich je komponent panela nástrojov `EditorToolbar`. Ten implementuje rozhranie filtrovania. Ďalším je `EditorError`, predstavujúci chybový banner. Treťou časťou je samotný komponent editora, ktorý je určený na základe získaného typu.

4.2.3 Synchronizácia stavu

Pri synchronizácii stavu jednotlivých editorov je potrebné, aby prebiehala automaticky. Teda každá zmena vykonaná v jednom editore sa musí okamžite premietnuť aj do stavu ostatných editorov. Jednotlivé editory však reprezentujú interpretácie symbolov vo svojom stave rôznym spôsobom, čo predstavuje problém.

Riešením je zaviesť jednotné miesto a formát ukladania interpretácií, ktorý budú všetky editory zdieľať a vedieť spracovať. Na tento účel sme sa rozhodli použiť interpretácie uložené v rámci stavu štruktúry. V nasledujúcej časti popíšeme spôsob, akým je implementované zdieľanie týchto dát medzi jednotlivými editormi.

Aktualizácia interpretácie konkrétneho symbolu uloženého v stave štruktúry sa realizuje pomocou príslušných akcií:

- `updateInterpretationPredicates`: akcia na aktualizáciu predikátu
- `updateInterpretationFunctions`: akcia na aktualizáciu funkcie

```
1 extraReducers(builder) {
2   builder.addCase(updateInterpretationPredicates, (state,
       action) => {
3     if (action.meta.source === "databaseView") return;
4
5     const { key, value } = action.payload;
6     syncInterpretation(key, "predicate", value, state);
7   });
8 }
```

Ukážka programu 4.1: Extra reducer používaný pri aktualizácii databázového editora.

Tieto akcie okrem svojho typu nesú aj názov daného symbolu (atribút `key`) a samotnú aktualizovanú interpretáciu (atribút `value`).

Každý editor má v súvislosti s týmito akciami dve úlohy. Prvou je vyvolať príslušnú akciu v momente, keď sa v jeho pohľade nejakým spôsobom zmení samotná interpretácia. To vyžaduje okrem iného aj konverziu medzi jeho internou reprezentáciou a formátom používaným v štruktúre.

Druhou je reagovať na tieto akcie vo svojom vlastnom reduceri a na základe informácií, ktoré nesú, aktualizovať svoj stav. Na tento účel bolo potrebné využiť mechanizmus knižnice Redux, ktorý umožňuje jednému reduceru reagovať na akcie iného. V tomto prípade teda reducery editorov reagujú na spomínané akcie štruktúry.

Ukážka takéhoto reducera (4.1) demonštruje spracovanie zmeny interpretácie predikátu v databázovom editore. Reducer je definovaný v rámci funkcie `extraReducers`, ktorá dostáva ako argument objekt `builder`. Pomocou metódy `addCase` sa určí akcia (prvý argument), pri ktorej sa má príslušný reducer zavolať (druhý argument).

V tele reducera sa najprv prostredníctvom atribútu `meta.source` kontroluje pôvod akcie. Ten sa vždy nastavuje pri jej vyvolaní. Je to dôležité najmä z toho dôvodu, aby databázový editor zbytočne nespracovával akcie, ktoré sám vyvolal. Následne sa volá funkcia `syncInterpretation`, implementujúca logiku prevádzania interpretácie do podoby používanej daným editorom.

4.3 Implementácia grafových editorov

Táto podkapitola približuje jednotlivé časti implementácie grafových editorov. Postupne prejdeme štruktúrou stavu, spôsobom jeho využitia a nakoniec sa pozrieme na integráciu komponentov knižnice React Flow.

4.3.1 Štruktúra stavu

Stav grafov predstavuje obyčajný objekt, v ktorom každý záznam uchováva stav jednotlivých reprezentácií pre každý binárny predikát alebo unárnu funkciu. Kľúče sú kombináciou názvu a typu príslušného symbolu. Hodnota asociovaná s konkrétnym kľúčom je opäť objekt s dvoma atribútmi. Prvým je `tupleType`, ten predstavuje typ symbolu, a teda môže nadobúdať hodnoty `'predicate'` alebo `'function'`. Druhým je atribút `state`, ktorý je ďalej členený podľa jednotlivých reprezentácií na `state.oriented`, `state.bipartite` a `state.hasse`. Konkrétny stav je tvorený atribútmi `nodes` a `edges`, tie predstavujú zoznam vrcholov a hrán.

Tvar konkrétneho vrcholu alebo hrany je určený samotnou knižnicou a nie je odporúčané ho priamo rozširovať o vlastné atribúty. Na tento účel je vyčlenený atribút `data`, ktorého tvar možno ľubovoľne špecifikovať. V prípade vrcholov ho využívame na zaznamenanie momentálneho variantu ich zobrazenia, pomocou atribútov `error`, `ghost`, `hatched` a `leftover` (bližšie opísané v časti venovanej orientovanému grafu 3.3.1). Všetky z nich sú voliteľné a typu `boolean`. Vrchol bipartitného grafu obsahuje ešte aj atribút `origin`, vyjadrujúci jeho príslušnosť do skupiny. Môže nadobúdať hodnoty `'domain'` alebo `'range'`.

4.3.2 Práca so stavom

Jednotlivé dáta zo stavu sú komponentom grafov sprístupnené prostredníctvom selektorov. To dovoľuje oddeliť transformáciu dát od implementácie používateľského rozhrania. Jednotlivé selektory je navyše možné navzájom kombinovať, čo uľahčuje ich organizáciu a znižuje duplicitu logiky.

Napríklad úlohou selektora `selectNodes` je pre konkrétnu reprezentáciu získať zoznam vrcholov, aplikovať na ne aktuálne doménové filtre a výsledok vrátiť v podobe zoznamu. Ako vstupné argumenty prijíma typ symbolu interpretácie, jeho názov a aj typ samotnej reprezentácie. Na základe týchto informácií dokáže zo stavu vybrať príslušné vrcholy. Pre získanie domény po aplikovaní všetkých filtrov sa používa ďalší selektor `selectRelevantDomain`. Nakoniec sa ešte zo zoznamu vrcholov odstránia tie, ktoré nereprezentujú prvok nachádzajúci sa vo vyfiltrovanej doméne.

Na podobnom princípe funguje aj selektor `selectEdges`, ktorého úlohou je vrátiť zoznam hrán priamo použiteľný pri zobrazení grafu. Napríklad pri Hasseho diagrame je v selektore potrebné odstrániť nadbytočné hrany (podľa pravidiel opísaných v sekcii 1.2.2), keďže v jeho stave sú explicitne uložené všetky hrany reprezentujúce interpretáciu.

Aktualizácia stavu prebieha prostredníctvom reducerov a k nim prislúchajúcich akcií. Zmeny týkajúce sa stavu vrcholov a hrán sú reprezentované akciami `nodesChanged` a `edgesChanged`. Zmena môže zahŕňať napríklad posunutie vrcholu alebo označenie

hrany. Pri vytvorení novej hrany je vyvolaná akcia `nodesConnected`, ktorej súčasťou sú aj identifikátory oboch vrcholov. Akcia `leftoverDeleted` slúži na odstránenie chybného vrcholu. Vyvoláva sa pri kliknutí na tlačidlo koša príslušného vrchola, popísaného v sekcii venovanej návrhu vrcholov 3.3.1.

Dôležitou akciou je aj `syncGraphView`, slúžiaca na inicializáciu stavu všetkých grafových reprezentácií všetkých binárnych predikátov a unárnych funkcií. Používa sa najmä pri synchronizácii grafových editorov so samotnou štruktúrou.

4.3.3 Komponenty

Hlavný komponent, ktorý obaluje komponenty konkrétnych reprezentácií je `GraphView`. Jeho úlohou je na základe poskytnutého typu grafu vykresliť správny komponent reprezentácie. Zobrazenie každej reprezentácie je implementované v rámci jej vlastného komponentu, konkrétne ide o `OrientedGraph`, `BipartiteGraph` a `HasseDiagram`. Tieto komponenty následne obaľujú komponent `ReactFlow` poskytovaný knižnicou, ktorý sa využíva na vykreslenie samotného rozhrania grafu.

Úlohou jednotlivých komponentov je získavať potrebné dáta na vykreslenie grafu zo stavu príslušnej reprezentácie, ako sú napríklad vrcholy a hrany, prostredníctvom vyššie spomínaných selektorov. Zároveň sa v nich definuje konfigurácia komponentu `ReactFlow`, špecifická pre daný typ grafu. Príkladom možnosti, ktorú knižnica pri konfigurácii ponúka, je funkcia `isValidConnection`. Po jej špecifikovaní ju knižnica volá pri každom pokuse o vytvorenie novej hrany, čo poskytuje možnosť implementovania vlastnej validačnej logiky pre vytváranie hrán. Táto funkcionalita sa využíva napríklad v komponente `HasseDiagram`, kde sa kontroluje, či by pridaním hrany ostali zachované všetky podmienky čiastočného usporiadania.

Menšie komponenty `PredicateNode` a `DirectEdge` predstavujú konkrétne vrcholy a hrany rozhrania grafu. O ich vykresľovanie sa stará samotná knižnica a taktiež je ich potrebné definovať v rámci konfigurácie. Pomocou komponentu `PredicateNode` je implementovaná napríklad funkcionalita vytvárania hrán alebo zobrazovania panela s farbami unárnych predikátov. Obe sú bližšie popísané v rámci časti venovanej návrhu orientovaného grafu 3.3.1.

4.4 Refaktorizácia stavu pôvodnej aplikácie

Ako bolo uvedené v podkapitole venovanej návrhu nového stavu aplikácie 3.7, časti stavu predstavujúce štruktúru a jazyk uchovávali výhradne obsah svojich textových vstupov. Každý z týchto vstupov však predstavuje odlišný typ dát, ktorý si vyžaduje špecifický formát zápisu, ako aj vlastnú validáciu a prevod do štruktúrovanej podoby. Z toho dôvodu bol každý typ textového vstupu asociovaný s príslušnou akciou na jeho

```
predicate_interpretation: {
    parse: (value) => parseTuples(value),
    toText: (struct) =>
      struct.map((tuple) => `(${tuple.join(",")})`).join(", "),
    validate: selectValidatedPredicate,
    syncActionCreator: updateInterpretationPredicates,
    payloadType: "key",
}
```

Ukážka programu 4.2: Konfiguračný objekt popisujúci interpretáciu predikátov.

aktualizáciu a selektorom. V tomto selektore prebiehala ich premena na štruktúrovanú podobu spolu so syntaktickou a sémantickou validáciou.

Prvým krokom pri refaktorizácii bolo v pôvodných častiach stavu nahradiť textové reprezentácie ich štruktúrovanou formou a následne týmto zmenám prispôsobiť aj príslušné selektory. To si vyžadovalo zo selektorov odstrániť časť zodpovedajúcu za premenu z textovej na štruktúrovanú podobu spolu so syntaktickou validáciou. Sémantická kontrola v selektoroch ostala zachovaná. Zároveň bolo potrebné upraviť akcie a k nim prislúchajúce reducery tak, aby namiesto textových vstupov pracovali priamo so štruktúrovanými dátami.

Ďalším krokom bolo vytvorenie novej časti stavu určenej na ukladanie obsahu jednotlivých textových vstupov. Na aktualizáciu stavu konkrétneho textového vstupu sa používa funkcia `updateTextView`. Jej úlohou je v prvom rade na základe typu textového vstupu vykonať syntaktickú kontrolu vstupného reťazca a následne vyvolať akciu `textViewChanged`, ktorá zabezpečuje aktualizáciu obsahu textového vstupu, ako aj prípadnej chybovej správy s ním spojenej. Ak je reťazec syntakticky v poriadku, nasleduje jeho transformácia do štruktúrovanej podoby a vyvolanie jednej z refaktorizovaných akcií zodpovedných za aktualizáciu príslušného typu dát.

Keďže textové vstupy nie sú jediným spôsobom, ako meniť interpretáciu predikátového alebo funkčného symbolu, je potrebné ich obsah v tomto prípade aktualizovať aj pri každej inde vyvolanej zmene. To je riešené rovnakým spôsobom, ako bolo už opísané v časti venovanej integrácii komponentov editorov 4.2.2. Teda pomocou extra reducera, ktorý reaguje na vyvolanie akcie predstavujúcej aktualizáciu interpretácie. Tieto akcie však nesú údaje v štruktúrovanej podobe a je ich pred použitím potrebné vhodne previesť do textovej formy.

Na konfiguráciu toho, ako správne spracovať jednotlivé typy dát, sme pre každý zadefinovali popisný objekt. Príklad takéhoto objektu pre interpretáciu predikátov je uvedený v ukážke 4.2. Jeho jednotlivé atribúty sú využívané v už opísaných častiach implementácie textových vstupov.

Konkrétne atribútom `parse` špecifikujeme funkciu používanú pri konverzii z textovej na štruktúrovanú formu. Naopak atribút `toText` určuje funkciu na konverziu štruktúrovanej formy do textovej. Atribút `validate` reprezentuje selektor, pomocou ktorého je možné zistiť, či je štruktúrovaná reprezentácia sémanticky validná, čo sa používa v komponentoch textových vstupov na získanie kompletnej chybovej správy. Ďalej `syncActionCreator` predstavuje akciu, na ktorej vyvolanie je potrebné reagovať pri synchronizácii textového vstupu so štruktúrovanou častou stavu. Posledný atribút `payloadType` určuje, či je nutné pri použití selektora `validate` alebo akcie `syncActionCreator` špecifikovať aj identifikátor (kľúč). V prípade interpretácií symbolov individúových konštánt, predikátov a funkcií tento kľúč predstavuje ich názov. Naopak pri typoch dát vyskytujúcich sa v stave iba raz (napr. doména) nie je nutné uvádzať žiadny identifikátor.

4.5 Integrácia s Logickým pracovným zošitom

Naša aplikácia je do Logického pracovného zošita integrovaná ako knižnica, ktorá poskytuje rozhranie pozostávajúce z funkcie `prepare`, používanej pri inicializácii vloženej aplikácie, a komponentu `AppComponent`, predstavujúceho koreňový komponent používateľského rozhrania. Požiadavky na ich fungovanie boli implementované už v rámci predošlej bakalárskej práce [7]. Pre zakomponovanie našich rozšírení bolo potrebné doplniť už existujúcu funkcionalitu ukladania a načítania stavu vloženej aplikácie, realizovanú pomocou dvoch funkcií.

Na exportovanie momentálneho stavu sa používa funkcia `exportAppState`. Jej úlohou je vrátiť objekt, ktorý Logický pracovný zošit následne prevedie do formátu JSON a uloží. Keďže samotný Redux stav predstavuje tiež iba obyčajný objekt, v pôvodnej implementácii bol jednoducho exportovaný celý stav aplikácie. To však v našom prípade nie je potrebné. Konkrétne stav maticového editora, databázového editora a textového pohľadu sme sa rozhodli vôbec neukladať, keďže ho možno odvodiť z interpretácií uložených v stave štruktúry. Z rovnakého dôvodu sa aj pri grafových editoroch ukladá iba pozícia jednotlivých vrcholov, pričom informácie o hranách sa opäť odvádzajú zo stavu štruktúry. Týmito optimalizáciami sa nám podarilo výrazne znížiť objem ukladaných dát. Ostatné spomínané rozšírenia ukladajú celý svoj stav.

Pri otvorení aplikácie môže byť dostupný počiatočný stav (ak bol predtým uložený), jeho načítanie je zabezpečené funkciou `importAppState`. Proces načítania sme sa rozhodli rozšíriť o krok validácie vstupných dát, ktorý je realizovaný knižnicou Zod [4]. Tá umožňuje definovať schémy opisujúce očakávanú štruktúru objektov. Každú exportovanú časť stavu bolo potrebné pomocou takejto schémy opísať a spojiť do jednej centrálnej schémy `serializedAppStateSchema`. Na overenie tvaru konkrétneho objektu

na základe tejto schémy poskytuje knižnica metódu `parse`. Ak vstupné dáta nespĺňajú požadovaný tvar, metóda vráti chybovú hlášku. V takomto prípade sa používateľovi zobrazí varovanie v rozhraní aplikácie, informujúce o tom, že niektoré časti stavu sa nepodarilo načítať. V prípade, že validácia prebehla úspešne, sú jednotlivé časti priamo načítané alebo odvodené.

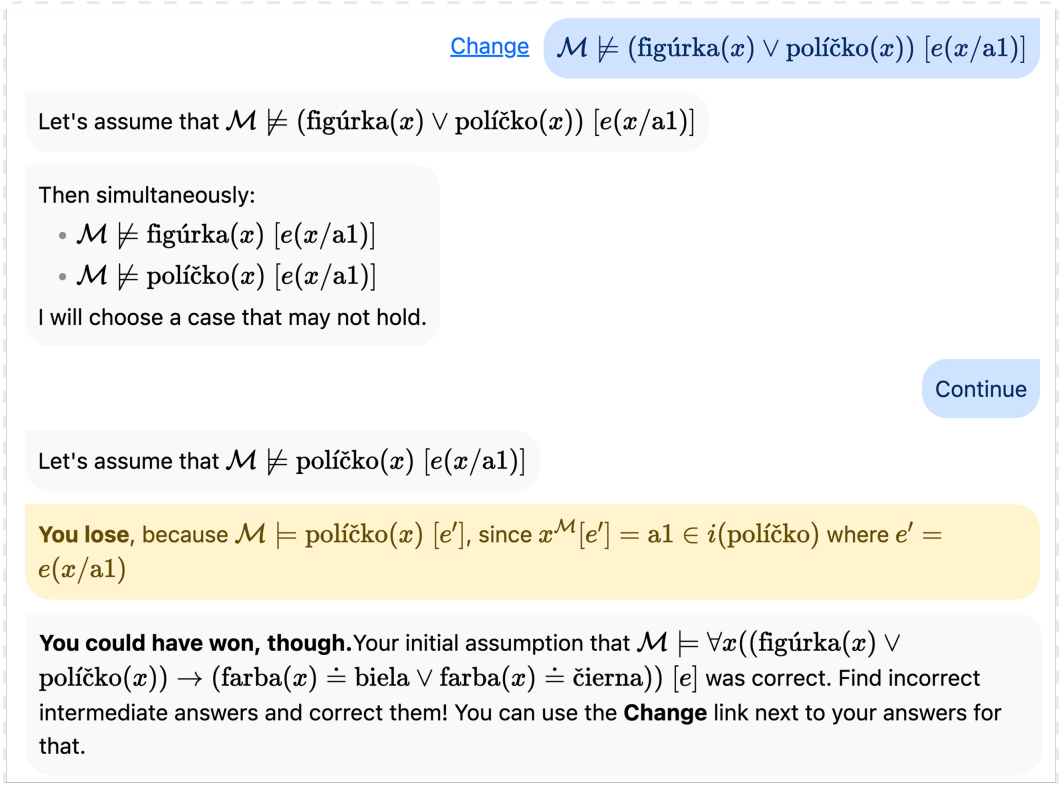
4.6 Vylepšenia Henkinovej-Hintikkovej hry

Reimplementovaná verzia Henkinovej-Hintikkovej hry spočiatku neobsahovala všetku požadovanú funkcionalitu. Kľúčovou zmenou jej fungovania bolo zavedenie znáhodnenia v situáciách, keď si hra vyberá podformulu alebo ohodnotenie otvorenej formuly. Prvé pokusy o implementáciu tejto funkcionality však odhalili viaceré nedostatky.

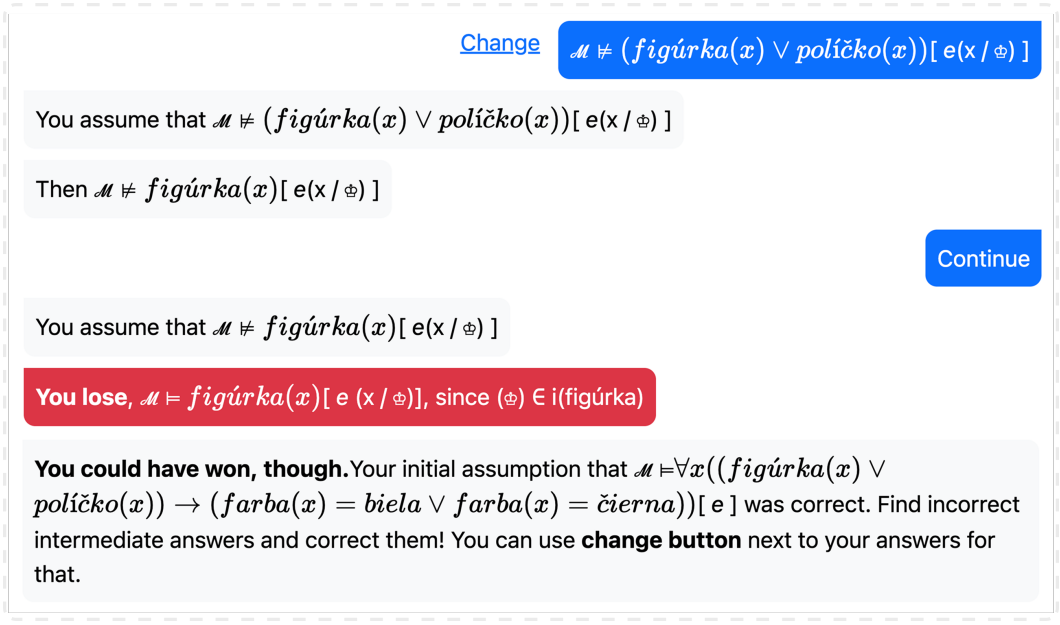
Niektoré z nich spôsobovali výrazné spomalenie a zhoršenie responzívnosti celej aplikácie pri formulách s väčším počtom vnorených kvantifikátorov, čo negatívne ovplyvňovalo používateľský zážitok. Iné zas v určitých okrajových prípadoch viedli až k pádu samotnej aplikácie. Preto bolo najskôr potrebné výrazne refaktorizovať existujúcu implementáciu, čo zároveň prispelo k lepšiemu pochopeniu jej fungovania.

Keďže hra implementuje väčšinu svojej logiky v selektoroch, zmeny zahŕňali najmä ich sprehľadnenie, odstránenie zbytočných častí, ako aj úpravu komponentov, ktoré ich využívajú. Ukázalo sa tiež, že selektory zodpovedné za správne resetovanie hry pri zmene štruktúry nepočítali so všetkými možnými situáciami, v ktorých sa hra môže nachádzať, a bolo potrebné tieto prípady vhodne ošetriť. Dokopy moje úpravy priniesli pri zložitejších formulách až desaťnásobné zlepšenie výkonu, čo je pre používateľa značne citeľné.

Po spomínaných zmenách sa nám pomerne jednoducho podarilo implementovať aj samotné znáhodnenie. Následne sme sa hru rozhodli doplniť o ďalšie vylepšenia. Jedným z nich bolo zjednotenie spôsobu výpisu matematických výrazov v dialógu hry. Pôvodná verzia kombinovala používanie Unicode znakov s prehľadnejším zápisom pomocou KaTeX-u, pričom tieto zápisy boli generované naprieč viacerými zdrojovými súbormi. Všetka logika generovania zápisov bola preto presunutá do súboru `messageBubbleFactories.ts`, ktorý obsahuje funkcie na generovanie rôznych typov výrazov výhradne vo formáte KaTeX, využívaných jednotlivými komponentmi hry. Okrem toho bolo pridaných viacero menších vylepšení používateľského rozhrania, ako aj samotného dialógu hry. Porovnanie rozhrania novej verzie s predchádzajúcou na krátkej časti hry možno vidieť na obrázku 4.1.



(a) Nová verzia



(b) Pôvodná verzia

Obr. 4.1: Porovnanie nového a pôvodného rozhrania Henkinovej-Hintikkovej hry.

Kapitola 5

Testovanie

TODO

Záver

Podarilo sa nám vytvoriť grafové editory obohacujúce aplikáciu Prieskumník štruktúr o nové možnosti tvorby interpretácií predikátov a funkcií. Snažili sme sa pri ich návrhu dosiahnuť v prvom rade prehľadnú a zároveň prispôsobiteľnú vizualizáciu. Zaoberali sme sa nápadmi, ako premeniť rôzne grafové reprezentácie do interaktívnej podoby a spríjemniť ich používanie rôznymi vylepšeniami. Postupne sme tiež pridávali ďalšie užitočné nástroje. Reimplementovali sme maticový a databázový editor z predchádzajúcej verzie aplikácie a prispôsobili ich našim špecifickým požiadavkám. Editorom interpretácií funkcií rozborom prípadov sme uľahčili tvorbu veľkých definícií funkcií a pridali sme aj novú sekciu aplikácie pre definovanie dopytov na splniteľnosť formúl.

Ďalej sme popísali integráciu jednotlivých riešení do existujúcej aplikácie, zabezpečili jej kompatibilitu s Logickým pracovným zošitom a nakoniec sa zamerali na vylepšenia súvisiace s Henkinovou-Hintikkovou hrou.

Aplikáciu sa podarilo dostať pomerne skoro do stavu, v ktorom mohla byť nasadená do výučby. Študenti mali preto počas celého semestra príležitosť využívať postupne pribúdajúcu funkcionalitu pri riešení cvičení a domácich úloh. V závere výučby sme jedno z cvičení využili na podrobnejšie testovanie nových nástrojov a následne sme prostredníctvom dotazníka zbierali spätnú väzbu od študentov.

Pri jednotlivých nástrojoch je však určite stále priestor na zlepšenie. Jedným z potenciálnych nápadov je pri zobrazení bipartitného grafu umožniť separátne filtrovanie zobrazenej domény oboch skupín. V súčasnosti je totiž množina vrcholov v oboch skupinách identická, čo často neposkytuje dostatočnú flexibilitu na dosiahnutie uspokojivého zobrazenia. Navrhnuté riešenie by sa malo dať využiť aj v maticovom editore na separátne filtrovanie stĺpcov a riadkov.

Rozhranie editora interpretácií funkcií môže zas novým používateľom prísť na prvý pohľad neintuitívne, čo by sa dalo vylepšiť zmenou jeho rozhrania, najmä v oblasti pridávania novej podmienky. Ďalším potenciálnym vylepšením je umožniť konfiguráciu farebnej palety používanej pri vizualizácii unárnych predikátov alebo optimalizovať implementáciu Henkinovej-Hintikkovej hry zmenou spôsobu vyhodnocovania formúl.

Vývojom tejto aplikácie som sa naučil používať nové technológie a prehĺbil znalosť tých, s ktorými som už mal predošlé skúsenosti. Asi najväčšou výzvou z tohto hľadiska bola pre mňa práca s knižnicou Redux, keďže som s ňou nikdy predtým nepracoval a pri

implementovaní riešení bolo jej hlbšie porozumenie kľúčové. Cenná bola aj skúsenosť pocitu zodpovednosti za vykonávanými zmenami počas toho, ako bola aplikácia už nasadená do výučby, keďže som nechcel študentom ani učiteľom spôsobiť zbytočné komplikácie prípadnými nedostatkami.

Literatúra

[1] *React Flow: A library for building node-based UIs*, 2026. Dostupné z `https://reactflow.dev`.

[2] *Redux: A JS library for predictable and maintainable global state management*, 2026. Dostupné z `https://redux.js.org`.

[3] *XFlow: React libraries for node-based UIs*, 2026. Dostupné z `https://xyflow.dev`.

[4] *Zod: TypeScript-first schema validation with static type inference*, 2026. Dostupné z `https://zod.dev`.

[5] Miroslav Baluch. Prieskumník grafových štruktúr pre logiku prvého rádu. Bakalárska práca, Univerzita Komenského v Bratislave, Fakulta matematiky, fyzika a informatiky, 2020.

[6] Milan Cifra. Prieskumník sémantiky logiky prvého rádu. Bakalárska práca, Univerzita Komenského v Bratislave, Fakulta matematiky, fyzika a informatiky, 2018.

[7] Jozef Filip. Reimplementácia prieskumníka štruktúr pre logiku prvého rádu. Bakalárska práca, Univerzita Komenského v Bratislave, Fakulta matematiky, fyzika a informatiky, 2025.

[8] R.P. Grimaldi. *Discrete and combinatorial mathematics: an applied introduction*. Addison-Wesley Pub. Co., 1985.

[9] Ján Kľuka, Ján Mazák, and Jozef Šiška. Logika pre informatikov a Úvod do matematickej logiky: Poznámky z prednášok. 2025.

[10] Meta Platforms, Inc. *React: A JavaScript library for building user interfaces*, 2026. Dostupné z `https://react.dev`.

[11] Microsoft. *TypeScript: Typed JavaScript at Any Scale*, 2026. Dostupné z `https://www.typescriptlang.org`.

[12] Matej Mok. Interaktívny pracovný zošit pre výučbu logiky pre informatikov. Bakalárska práca, Univerzita Komenského v Bratislave, Fakulta matematiky, fyzika a informatiky, 2022.

[13] Paul Tol. Qualitative color schemes. *Paul Tol's notes. Colour schemes and templates*, 2021.

[14] Richard Tóth. Henkinova-hintikkova hra v prieskumníku štruktúr. Bakalárska práca, Univerzita Komenského v Bratislave, Fakulta matematiky, fyzika a informatiky, 2021.