Univerzita Komenského v Bratislave

Fakulta matematiky, fyziky a informatiky

Vizuálna podpora dokazovania čiastočnej správnosti programov pomocou Hoareovej metódy

bakalárska práca

2015

Jakub Pavčo

Univerzita Komenského v Bratislave

Fakulta matematiky, fyziky a informatiky

Vizuálna podpora dokazovania čiastočnej správnosti programov pomocou Hoareovej metódy

bakalárska práca

Študijný program: Informatika
Študijný odbor: 2508 Informatika
Školiace pracovisko: Katedra informatiky
Školiteľ: RNDr. Jana Katreniaková, PhD.

Bratislava, 2015
Jakub Pavčo

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Jakub Pavčo
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: 9.2.1. informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Vizuálna podpora dokazovania čiastočnej správnosti programov pomocou Hoareovej metódy
Visual support for proving partial correctness of programs using Hoare's method

Cieľ: Hoareova metóda pre dokazovanie čiastočnej správnosti štruktúrovaných programov
sa opiera o špecializovaný logický (inferenčný) systém nad induktívnymi formulami.
Cieľom bakalárskej práce je vytvoriť program na vizuálnu podporu dokazovania
čiastočnej správnosti programov pomocou Hoareovej metódy - vizualizácia rozdeľovania programu na vhodné programové segmenty a návrh inferenčných pravidiel, ktoré má používateľ pre jednotlivé segmenty použiť.

Vedúci: RNDr. Jana Katreniaková, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: doc. RNDr. Daniel Olejár, PhD.

Spôsob sprístupnenia elektronickej verzie práce:
bez obmedzenia

Dátum zadania: 24.10.2014

Dátum schválenia: 29.10.2014

doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Poďakovanie:
V prvom rade ďakujem vedúcej bakalárskej práce RNDr. Jane Katreniakovej PhD. za pomoc pri výbere témy. Taktiež za jej čas, námahu a rady pri konzultáciách. Ďalej ďakujem Tomášovi Paulíkovi za podporu počas písania, Milares a Monachusovi za pozvanie a možnosť písať prácu u nich doma a taktiež ďakujem ostatným priateľom a rodine, že mi dopomohli k napísaniu tejto práce.

Abstrakt

Hlavným cieľom tejto bakalárskej práce bolo vytvoriť aplikáciu na vizualizáciu dokazovania čiastočnej správnosti daného štruktúrovaného programu. Používaná je Hoareho metóda na dokazovanie čiastočnej správnosti. Prvá kapitola sa zaoberá teóriou, ktorá vysvetľuje Hoareho metódu. Druhá a tretia kapitola obsahujú opis nášho prístupu k vizualizácií a implementačné detaily. Posledná kapitola demonštruje využitie aplikáciena na konkrétnom príklade. Táto aplikácia by mala pomôcť študentom v lepšom porozumení Hoareho metódy počas ich štúdia teórie programovania.

Kľúčové slová: Hoareho metóda, čiastočná správnosť, vizualizácia

Abstract

The aim of this bachelor's thesis was to create an application to visualize proving of partial corectness for given structured program. Method used for proving program's correctness is Hoare's method. The first chapter of paper consists of theory explaining Hoare's method. In second and third chapters we describe our approach to visualization and implementation details of the application. Furthermore, the last champter demonstrates the application with an example of proving partial correctness. This application is supposed to help stundents to better understand Hoare's method in their studies of programming theory.

Keywords: Hoare method, partial correctness, visualization

Obsah

Zoznam obrázkov

Úvod

Každý človek sa stretáva denne s rôznymi problémami, pričom sa snaží ich vyriešiť čo najlepšie a najsprávnejšie. Rozhodnúť sa avšak málokedy býva jednoduché. Povedať s istotou či je dané riešenie to najlepšie a správne je niekedy zložitejšie ako vymyslieť samotné riešenie problému. S obdobnými ťažkosťami sa stretáme aj pri riešení matematických a algoritmických úloh. Skonštruovať algoritmus a vedieť o ňom s istotou prehlásiť, že nerobí chyby znamená, že sme našli bezchybné riešenie pre daný problém. A to je cieľom. Nájsť čo najlepšie riešenie. Vidíme, že motivácia za správnosťou algoritmov je zrejmá.

Ale ako vieme z istotou povedať, že alogritmus je správny? Môžeme program vytvoriť a následne ho testovať alebo môžeme formálne dokázať jeho správnosť. Avšak formálne dokazovať algoritmus, ktorý má niekoľko tisíc riadko kódu, je nesmierne zložité a časovo náročné. A často by bola vydaná námaha a čas zbytočná, pretože to nie je vždy úplne nutné. Aj preto sa v dnešnej dobe oveľa viac používa metóda testovania daného algoritmu a nerobí sa formálny dôkaz. Avšak pochopiť základné princípy dôkazu správností programov môže pomôcť pochopiť hlbšie súvislosti medzi danými príkazmi a vzťahmi v danom programe. Aj preto sme sa rozhodli vytvoriť aplikáciu, ktorá bude vizualizovať metódu na dôkaz správnosti.

V tejto práci sa budeme zaoberať najmä Hoareho metódov pre dokazovanie čiastočnej správnosti programov. Plánujeme rozdeliť prácu do štyroch častí. V prvej kapitole poskytneme dostatočné teoretické vedomosti pre pochopenie problematiky dôkazu čiastočnej správnosti, ako aj pochopenie Hoareho

metódy. V ďalšej kapitole sa chceme venovať samotnému návrhu vizualizácie a chceme opísať aplikáciu z dizajnérskeho hladiska. Tretia kapitola bude obsahovať náhľad k použitým technológiám a implementácií našej apikácie a v poslednej časti uvedieme príklad dôkazu čiastočnej správnosti pomocou našej aplikácie.

Veríme, že daná práca pomôže nielen študentom pri študovaní teórie programovania a dokazovaní programov pomocou Hoareho metódy ale komukoľvek, kto sa o túto problematiku zaujíma.

Kapitola 1

Úvod do dokazovania správnosti

V prvej časti kapitoly tejto práce si predstavíme programovací jazyk, v ktorom pracujeme pri dokazovaní čiastočnej správnosti pomocou Hoareho metódy. Následne si vysvetlíme samotnú Hoareho metódu. V poslednej časti si ukážeme konkrétny príklad dokazovania.

1.1 Programovací jazyk a jeho syntax

Programovací jazyk, ktorý budeme používať je veľmi jednoduchý ale dostatočne silný na to, aby sme s ním popísali aj komplikovanejšie programy z vyšších programovacích jazykov. Tento jazyk je štruktúrovaný, čiže samotný program je možné rozdeliť na jednotlivé štruktúry – menšie časti, ktoré samé o sebe dávajú zmysel. Dôležitým predpokladom je, že jazyk neobsahuje príkaz skoku.

1.1.1 Premenné

Používame tri typy premenných:

1. vstupné – hodnotami týchto premenných si inicializujeme pracovné premenné, ktoré následne používame ďalej. Označujeme ich $\overline{x}$ ={x_1, x_2,...,x_i} , kde i je prirodzené číslo

2. pracovné – v týchto premenných si ukladáme hodnoty počas behu programu. Označujeme ich $\overline{y}$ ={y_1, y_2,...,y_j} , kde j je prirodzené číslo

3. výstupné – do týchto premenných uložíme hodnoty ktoré tvoria výstup programu. Značíme ich $\overline{z}$={z_1, z_2,..., z_k} , kde k je prirodzené číslo

1.1.2 Príkazy

Program P je postupnosť príkazov st_1,st_2,...,st_i, kde i je prirodzené číslo. Náš jazyk obsahuje len šesť typov príkazov:

1. príkaz priradenia *a:= D*, kde *a* je premenná a *D* je konštanta alebo výraz zložený z konštánt, premenných a aritmetických operátorov, prípadne logických operátorov, ak priraďujeme viac premenných, tak sa priraďujú simultánne

2. zložený príkaz st_i:={ st_j; st_k;} kde st_j, st_k sú príkazy, ktoré sa vykonávajú za sebou

3. príkaz *begin [y]:=[x]* používa sa len raz a to na začiatku programu, pomocou vstupných premenných sa inicializujú pracovné premenné

4. príkaz *end [z]:=[y]* takisto sa používa len raz a to na konci programu, do výstupných za zapíše výstupná hodnota programu na základe hodnôt pracovných premenných

5. podmienkový príkaz *if p(y) then* st_i *else* st_j *fi* , kde *p(y)* je boolovský výraz zložený z premenných, konštánt, aritmetických a logických operátorov a operátorov porovnania a st_i,st_j je príkaz

6. príkaz cyklu *while p(y) do* st_j *od* kde *p(y)* je boolovský výraz ako v predchádzajúcom prípade, rovnako st_j je príkaz

1.1.3 Ukážka štrukturovaného programu

Tu môžeme vidieť ukážku štruktúrovaného programu, ktorý využíva príkazy priradenia, *begin*, *end* a príkaz cyklu. Program počíta odmocninu z čísla x. Prebraté z

begin$\{x \geq 0\}$
$[y_1, y_2, y_3]$:=[0,1,1];
while $y_2 \leq$ x do $[y_1, y_2, y_3]$:=$[y_1 + 1, y_2 + y_3 + 2, y_3 + 2]$ od;
[z]:=$[y_1]$;
end $\{z^2 \leq x < (z+1)^2\}$

1.1.4 Iný pohľad na program

Na program sa môžme pozerať ako na jeden príkaz, ktorý je zložený a skladá sa z viacerých dalších jednoduchých a zložených programov. Na predstavivosť môžme použiť príklad s kartónovými škatuľami rôznej veľkosti, ktoré sú naskladané do seba, prípadne vedľa seba v nejakej inej škatuli. Takto sa na program budeme pozerať aj pri vizuálizácií. Pre zjednodušenie pripustíme aby zložený príkaz mohol obsahovať aj viac ako dva príkazy. Náš program si rozložíme na menšie časti a tie budeme dokazovať. Iný pohľad na program je reprezentácia príkazov do stromového grafu. Koreň stromu budeme chápať ako *begin* S *end*, kde S je zložený príkaz. Každý ďalší príkaz je vrcholom a príkazy cyklu a podmienky obsahujú ďalšie príkazy. Príkazy priradenia sú listy. Viac si o tom povieme v kapitole Vizualizácia.

1.2 Dokazovanie správnosti

V tejto sekcii sa budeme venovať dokazovaniu čiastočnej správnosti programov pomocou Hoareho metódy. Program obohatíme o vstupné a výstupné podmienky a tým obmedzíme množinu vstupov s ktorými náš program vie pracovať. Pomocou výstupnej pdomienky charakterizujeme požadavky prog-

ramového výstupu pre hodnoty, ktoré splnili vstupnú podmienku. V ukážke štruktúrovaného programu sme mohli tieto podmienky vidieť za príkazmy *begin* a *end*. Takéto podmienky sa vyjadrujú v jazyku predikátovej logiky. Budeme ich označovať veľkými tlačenými písmenami a to konkrétne:

1. $P(\overline{x})$ – vstupná podmienka, ktorá musí platiť pre vektor vstupných hodnôt $\overline{x}$, dá sa povedať, že ohraničuje vstupné hodnoty
2. $Q(\overline{x},\overline{z})$ – výstupná podmienka, ktorá musí platiť v prípade zastavenia pre vstupné hodnoty $\overline{x}$ a výstupné hodnoty $\overline{z}$
3. $R_1, R_2, \ldots, R_n$,kde n je prirodzené číslo – pomocné podmienky ktoré budeme používať pri rozkladaní programu na menšie programové segmenty

Zavedieme si nasledovné definície prebraté z bakalárskej práce o vizualizácií Floydovej metódy. [2]

Definícia 1. *Program je čiastočne správny, ak pre všetky vstupné vektory* $\overline{x}$ *spĺňajúce vstupnú podmienku* $P(\overline{x})$ *platí, že v prípade zastavenia bude platná aj výstupná podmienka* $Q(\overline{x},\overline{z})$.

Definícia 2. *Program je úplne správny, ak pre všetky vstupné vektory* $\overline{x}$ *spĺňajúce vstupnú podmienku* $P(\overline{x})$ *platí, že program v konečnom čase zastaví a bude platná výstupná podmienka* $Q(\overline{x},\overline{z})$.

1.2.1 Metódy dokazovania čiastočnej správnosti a správnosti programov

Medzi najznámejšie metódy patria:
Floydova metóda - táto metóda sa opiera analýzu konečných výpočtových ciest v programe. Dá sa použiť na širokú triedu programov, ktorých abstrakciou sú štandardné schémy, ktoré obsahujú aj príkaz skoku. Podstatou je redukcia dôkazu na dokazovanie formúl špecifického jazyka. Používa sa tu

výpočtová indukcia.
Hoareho metóda - opiera sa o vytvorenie špecializovaného dokazovacieho systému nad takzvanými induktívnymi formulami. Bližšie si ju charakterizujeme v ďalšej časti.
Metóda intermitentov - na rozdiel od predošlých, ktoré boli navrhnuté na dokazovanie čiastočnej správnosti, sa používa na dôkaz úplnej správnosti. Princíp metódy spočíva v intermitentoch, ktoré na rozdiel od invariantov sú splnené len vtedy, ak výpočet daným bodom prejde aspoň raz.

1.3 Hoareho metóda

1.3.1 Induktívna formula

Induktívna formula $\{p\}S\{q\}$ popisuje vstupno výstupné charakteristiky programového segmentu S s tým, že vstupná podmienka je p a výstupná podmienka je q. Pod programovým segmentom rozumieme príkaz, ktorý môže byť aj zložený. Význam induktívnej formuly závisí od podmienok p, q a významu programu S.

Definícia 3. *Induktívna formula $\{p\}S\{q\}$ platí práve vtedy keď vstupné hodnoty programového segmentu S spĺňajú podmienku p a po zastavení segmentu S výstupné hodnoty vyhovujú podmienke q.*

Inak povedané induktívna formula je platná práve vtedy keď je programový segment S čiastočne správny vzhľadom k podmienka p a q. Ešte sa bližšie pozrieme na vlastnosti induktívnej formuly $\{p\}S\{q\}$. Uvažujme dve induktívne formuly:

1. $\{x = 0\}x := x + 1\{x = 1\}$.
2. $\{x = 0\}x := x + 1\{x > 0\}$.

Obidve formuly evidentne platia avšak ich rozdiel je v tom, že oproti prvej induktívnej formuly má druhá oveľa slabšiu (voľnejšiu) výstupnú podmienku vzhľadom na vstupnú podmienku a daný programový segment -

príkaz priradenia . Druhý pohľad na tieto dve fomule hovorí, že prvá formula má oveľa silnejšiu (prísnejšiu) podmienku vzhľadom na výstupnú podmienku a daný segment. Požiadavky na silnejšiu väzbu medzi vstupnou a výstupnou podmienkou za zavádzajú dvomi spôsobmi. Prvým je *najslabšia vstupná podmienka* podmienka k programu P a výstupnej podmienke q, ktorú označujeme $wp(P, q)$ a druhým je *najsilnejšia výstupná podmienka* pre vstupnú podmienku p a program P, ktorú označujeme $sp(P, q)$. V princípe nám označenie $wp(P, q)$ hovorí, že každá vstupná podmienka p, pre ktoré platí induktívna formula $\{p\}S\{q\}$, je silnejšia ako $wp(P, q)$. Tým charakterizuje množinu všetkých vstupných dát, pre ktoré sa program P buď nezastaví alebo sa zastaví a výstupná podmiena q je splnená. Podobne jde definované aj označenie $sp(P, q)$, ktoré znamená, že každá výstupná podmienka q, pre ktorú platí program $\{p\}S\{q\}$, je slabšia a ako $sp(P, q)$. Tým nám charakterizuje najmenšiu podmnožinu výstupných dát, pre ktoré sa program P pre vstupné hodnoty spĺňajúce p zastaví.Obšínejšie informácie o podmienkach sa dajú násť na tomto odkaze[3].
Pomocou týchto vzťahov vieme vyjadriť užšie prepojenie medzi vstupnou podmienkou p a výstupnou podmiekou q aj pre zložitejšie príkazy nášho štruktúrovaného jazyka. V tejto práci nebudú uvedené vyjadrenia $wp(P, q)$ a $sp(P, q)$ pre všetky typu príkazov ale vyjadríme si aspoň najslabšiu vstupnú podmienku pre príkaz priradenia – $wp(x := s, q) = q[x/s]$. Najslabšou vstupnou podmienkou je $q[x/s]$, ktorú dostaneme tým, že nahradíme všetky výskyty premennej x v q pravou stranou priraďovacieho príkazu. Ostatné príkazy a vyjadrenia môžeme nájsť v skriptách pre Úvod do teórie programovanie. [5] Dôležité je ale spomenúť, že inferenčný systém H, ktorý predstavíme v ďalšej časti, vychádza aj zo vzťahov medzi vstupnou a výstupnou podmienkou, ktoré sme si takto bližšie upresnili.

1.3.2 Inferenčný systém H

Hoareho metóda sa opiera o logický systém pre dokazovanie čiastočnej správnosti založenom na jazyku induktívnych formúl. Uvažujme formálny inferenčný

systém $H = (Ax, Pr)$ pre dokazovanie induktívnych formúl, pozostávajúci z axióm Ax a inferenčných pravidiel Pr. Tieto definície sú prebraté zo skrípt Základov teórie programovania. [5]
Množina axióm - Ax :

1. všetky platné formuly špecifického jazyka
2. axióma priradenia : $p(\overline{x},g(\overline{x},\overline{y}))\overline{y}:=g(\overline{x},\overline{y})p(\overline{x},\overline{y})$

Inferenčné pravidlá - Pr

1. pravidlo kompozície :
$$\frac{\{p\}\ S_1\ \{q\}, \{q\}\ S_2\ \{r\}}{\{p\}\ S_1;\ S_2\ \{r\}}$$
2. pravidlo alternatívy:
$$\frac{\{p \wedge b\}\ S_1\ \{q\}, \{p \wedge !b\}\ S_2\ \{q\}}{\{p\} if\ b\ then\ S_1\ else\ S_2 fi\ \{q\}}$$
3. pravidlo iterácie:
$$\frac{\{p \wedge b\}\ S\ \{p\}}{\{p\} while\ b\ do\ S\ od\ \{p \wedge !b\}}$$
4. pravidlo následku:
$$\frac{p \Rightarrow r,\ s \Rightarrow t,\ \{r\}\ S\ \{s\}}{\{p\}\ S\ \{t\}}$$

Uvedená verzia axiómy priamo vychádza z vyjadrenia najslabšej vstupnej podmienky pre príkaz priradenia. Z tejto skutočnosti a z definície pravdivosti induktívnej formuly intuitívne vyplýva, že axióma priradenia je vždy pravdivá. Inferenčné pravidlá pre zložené príkazy nám ukazujú, za akých podmienok vieme dokázať induktívnu formulu pre zložený príkaz ak máme dokázané jednotlivé zložky zloženého príkazu.
Na prvý pohľad sa môže zdať, že pravidlá sú príliš reštriktívne. Napríklad v pravidle alternatívy musia byť výstupné podmienky totožné. Avšak pravidlo následku umožnuje využívanie týchto pravidiel vo všeobecnosti. Toto pravidlo taktiež vychádza z úvah o slabších a silnejších podmienkach, ktorým sme sa venovali v predchádzajúcej časti.

Veta 1. *Nech P je štruktúrovaný program, p je vstupná a q je výstupná podmienka. Ak $H \vdash \{p\}\ P\ \{q\}$, potom je program čiastočne správny vzhľadom na špecifikácie dané dvojicou podmienok p,q.*

Táto veta nám hovorí, že vzhľadom na význam induktívnej formuly $\{p\}S\{q\}$, dokázaním tejto formuly Hoareho inferenčným systémom, vlastne dokážeme čiastočnú správnosť tohto programu.
Na záver treba ešte spomenúť dve vlastnosti systému H a to, že je zdravý a relatívne úplný. Dokázanie týchto vlastností je nad rozsah tejto práce. Uvádzame ich tu aby sme ukázali, že tento inferenčný systém je dobre navrhnutý, obsahuje len zdravé pravidlá a axiómu sú pravdivé.

1.3.3 Postup dokazovania

V zásade máme na výber dve možnosti postupu dokazovanie: zhora nadol a zdola nahor. Postup zhora nadol má výhodu, že začínam od vstupnej a výstupnej podmienky daného programu. Pomocou pravidla kompozície si viem "rozbiť"program P na menšie časti, na menšie programové segmenty. V praxi to znamená, že program $\{p\}S\{q\}$ viem pomocou pravidla kompozície rozdeliť na $\{p\}R_1\{r\}$ a $\{r\}R_2\{q\}$, kde R_1, R_2 sú programové segmenty a r je podmienka, ktorej pravdivostnú hodnotu vieme určiť. Ak dokážeme samostatne správnosť R_1 a R_2, tak vieme že pomocou kompozície máme dokázaný celý program. Takýmto spôsobom sa postupuje až kým neprídeme k príkazom priradenia, ktoré dokážeme pomocou axióm. Spätnými implikáciami je dôkaz hotový.
Postup zdola nahor sa líši od predchádzajúceho tým, že sa postupuje od príkazov, ktoré viem priamo dokázať . Začíname príkazmi priradenia, ktoré si viem dokázať pomocou axióm. Následne dokážem zložené príkazy, ktoré už majú dokázané všetky príkazy, ktoré v sebe obsahujú. Takto postupujem od najmenších segmentov až kým nemám dokázaný celý program.
Základným problémom, s ktorým sa tu stretávame aj v jednom aj v druhom postupe, je určenie podmienok. Či už podmienky q keď používame postup

zhora nadol alebo podmienok, ktoré máme použiť pri písaní axiómy, keď postupujem zdola nahor pri dokazovaní príkazu priradenia. Vzhľadom k tomu ako sú navrhnuté inferenčné pravidlá, je výhodné použiť nejakú podmienku, ktorá je platná v určitých úsekoch nášho programu a nemusíme navrhovať zakaždým novú podmienku. Tejto podmienke sa hovorí aj invariant.

Veta 2. *Invariant algoritmu I je podmienka, ktorá musí platiť v určitých okamihoch výpočtového procesu.*

Keď budeme invarinat spomínať budeme myslieť invariant cyklu, ktorý musí platiť pred a po každej iterácií cyklu alebo invariant algoritmu musí platiť po ceú dobu výpočtu algoritmu. Na nasledujúcom príklade uvidíme ako veľmi môže poznanie invariantu k danému algoritmu uľahčiť jeho dokazovanie. Avšak problém konštrukcie invarinatu k danému cyklu alebo k danému algoritmu je nerozhodnuteľným problémom. Aj preto bude určovanie podmienok v našej vizualizácií úplne na úžívateľovi.

1.3.4 Príklad

Tu uvedieme príklad dokazovania programu P:
Máme daný štruktúrovaný program. Dokážte jeho čiastočnú správnosť vzhľadom k vstupnej podmienke $p \equiv x \in \{0,1\}^+$ a výstupnej podmienke $q \equiv z = x^R$.

begin $[y_1, y_2, y_3] := [tail(x), head(x), \varepsilon]$
while $y_1 = \varepsilon$ **do**
    if $head(y_1) = y_2$ **then** $[y_1, y_3] := [tail(y_1), y_2.y_3]$
    else $[y_1, y_2, y_3] := [tail(y_1), 1 - y_2, y_2.y_3]$
    fi
od
end $[z] := [y_2.y_3]$

Program budeme dokazovať metódov zdola nahor:

1. $\{\varepsilon^R.head(x).tail(x) = x\}[y_1, y_2, y_3] := [tail(x), head(x), \varepsilon]\{y_3^R.y_2.y_1 = x\}$ axióma priradenia

2. $x \in \{0,1\}^+ \Rightarrow \varepsilon^R.head(x).tail(x) = x$
3. $\{x \in \{0,1\}^+\}[y_1, y_2, y_3] := [tail(x), head(x), \varepsilon]\{y_3^R.y_2.y_1 = x\}$ (1)+(2) pravidlo následku
4. $\{(y_2.y_3)^R.y_2.tail(y_1) = x\}[y_1, y_3] := [tail(y_1), y_2.y_3]\{y_3^R.y_2.y_1 = x\}$ axióma priradenia
5. $y_3^R.y_2.y_1 = x \ \wedge \ y_1 \neq \varepsilon \ \wedge head(y_1) = y_2 \Rightarrow (y_2.y_3)^R.y_2.tail(y_1) = x$
6. $\{y_3^R.y_2.y_1 = x \wedge y_1 \neq \varepsilon \wedge head(y_1) = y_2\}[y_1, y_3] := [tail(y_1), y_2.y_3]\{y_3^R.y_2.y_1 = x\}$ (4)+(5) pravidlo následku
7. $\{(y_2.y_3)^R.(1-y_2).tail(y_1) = x\}[y_1, y_2, y_3] := [tail(y_1), 1-y_2, y_2.y_3]\{y_3^R.y_2.y_1 = x\}$ pravidlo následku
8. $y_3^R.y_2.y_1 = x \wedge y_1 \neq \varepsilon \wedge \neg(head(y_1) = y_2) \Rightarrow (y_2.y_3)^R.(1-y_2).tail(y_1) = x$
9. $\{y_3^R.y_2.y_1 = x \wedge y_1 \neq \varepsilon \wedge \neg(head(y_1) = y_2)\}[y_1, y_2, y_3] := [t(y_1), 1 - y_2, y_2.y_3]\{y_3^R.y_2.y_1 = x\}$ (7)+(8)+pravidlo následku
10. $\{y_3^R.y_2.y_1 = x \wedge y_1 \neq \varepsilon\}$if $head(y_1) = y_2$ then ... else ... fi$\{y_3^R.y_2.y_1 = x\}$ (6)+(9)+ pravidlo alternatívy
11. $\{y_3^R.y_2.y_1 = x\}$while $y_1 \neq \varepsilon$ do .. od$\{y_3^R.y_2.y_1 = x \ \wedge \ \neg(y_1 \neq \varepsilon)\}$ (10)+pravidlo iterácie
12. $y_3^R.y_2.y_1 = x \ \wedge \neg(y_1 \neq \varepsilon) \Rightarrow y_2.y_3 = x^R$
13. $\{y_3^R.y_2.y_1 = x\}$while $y_1 \neq \varepsilon$ do ... od$\{y_2.y_3^R = x^R\}$ (11)+(12)+pravidlo následku
14. $\{y_2.y_3 = x^R\}[z] := [y_2.y_3]\{z = x^R\}$ axióma priradenia
15. $\{x \in \{0,1\}^+\}P\{z = x^R\}$ (3)+(13)+(14)+pravidlo kompozície

Týmto je dokázaná čiastočná správnosť daného programu.

Kapitola 2

Vizualizácia

V tejto kapitole sa budeme venovať spôsobu vizualizácie dôkazu správnosti tak, aby bola prehľadná, jednoduchá a dostatočne graficky interpretovala kroky v Hoareho metóde dokazovania čiastočnej správnosti. V predošlých častiach bolo cieľom vysvetliť teóriu, ktorá sa týka správnosti respektíve čiastočnej správnosti programov a ich dokazovania. Vysvetlili sme si Hoareho metódu, ktorú sme si následne na príklade ukázali.
Cielom tejto práce bolo navrhnúť a naprogramovať aplikáciu, ktorá by vizualizovala túto metódu a tým by užívateľovi čo najlepšie pomohla danej metóde porozumieť a naučiť sa ju používať. Snažili sme sa nájsť softvér, ktorý by približoval Hoareho metódu alebo by ju čiastočne vizualizoval ale našli sme len jeden zdroj, ktorý nám nebol až tak užitočný [1]. Ako sme v predchádzajúcek kapitole v závere predstavili dva spôsoby, ktorými sa dá používať Hoareho metóda, tak aj tu sme mali na výber, ktorú z nich chceme implementovať. Rozhodli sme sa implementovať metódu ktorá dokazuje daný program zdola nahor.
Ako bolo už spomenuté, program si vieme reprezentovať aj ako graf stromu. Táto predstava nám pomôže pri vizualizácií ako aj pri návrhu algoritmu na vizualizáciu o kotrom si povieme v 3. kapitole. Navrhneme dátový model reprezentujúci zadaný program. Model bude reprezentovaný obdĺžnikmi povkladanými do seba. Tieto obdĺžniky budú znázorňovať škatule. Najväčšia

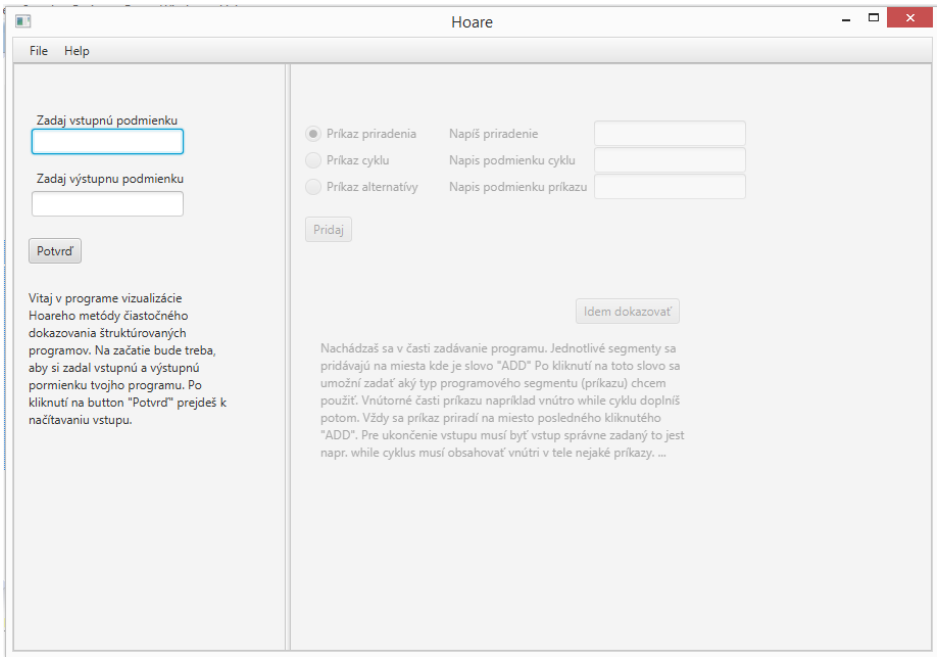
škatula je náš program a zároveň koreň stromu. Každý príkaz je škatuľa. Príkaz priradenia neobsahuje iné príkazy, takže v strome bude listom. Zložený príkaz je vrcholom v strome a jeho deti sú príkazy nachádzajúce sa v tele príkazu. Z pohľadu škatúľ to sú škatule, ktoré vidíme bezprostredne po otvorení škatule, ktorá predstavuje náš zložený príkaz. Program budeme znázorňovať pomocou farebných obdĺžnikov, ktoré budú za behu algoritmu meniť farbu. Naša aplikácia má samostatné okno, nepoužíva prehliadač, ktorý by ju zobrazoval na celú plochu obrazovky. Počas dôkazu sa menia grafické plochy v našej aplikácie. V nasledujúcich častiach si opíšeme čo obsahujú jednotlivé plochy.

2.1 Úvodná plocha

Na nasledujúcom obrázku môžeme vidieť ako vyzerá náša aplikácia po spustení.

V hornej lište obrázka 2.1 sa nachádza názov aplikácie a v pravom rohu krížik, ktorým sa aplikácia vypína. Pod ňou je umiestnené menu, ktoré je prístupné počas celého behu aplikácie a obsahuje dve položky: File, About. Položka File je určená pre prácu s daným súborom. Obsahuje len jednu možnosť a to je New. Táto možnosť je prístupná počas celej doby bežania nešej aplikácie a umožnuje začať dokazovať nový program bez toho, aby úžívateľ musel aplikáciu vypínať a zapínať. Položka About obsahuje informácie o autorovi aplikácie a jej verzii. Umiestnenie menu umožnuje jednotný prístup k určitým možnostiam aplikácie počas celého jej behu.

Na ľavej strane okna aplikácie sa nachádza textový opis, ktorý napovie ako začať s danou aplikáciou pracovať. Taktiež sú tu umiestnené dve vstupné polia pre zadanie vstupnej a výstupnej podmienky pre náš program, ktorý by sme chceli dokazovať. Obsahuje aj tlačidlo *Potvrd*, ktorým sa potvrdia zadané podmienky a aplikácia sa prepne do plochy, ktorá umožnuje načítavanie programu.



Obr. 2.1: Úvodná plocha

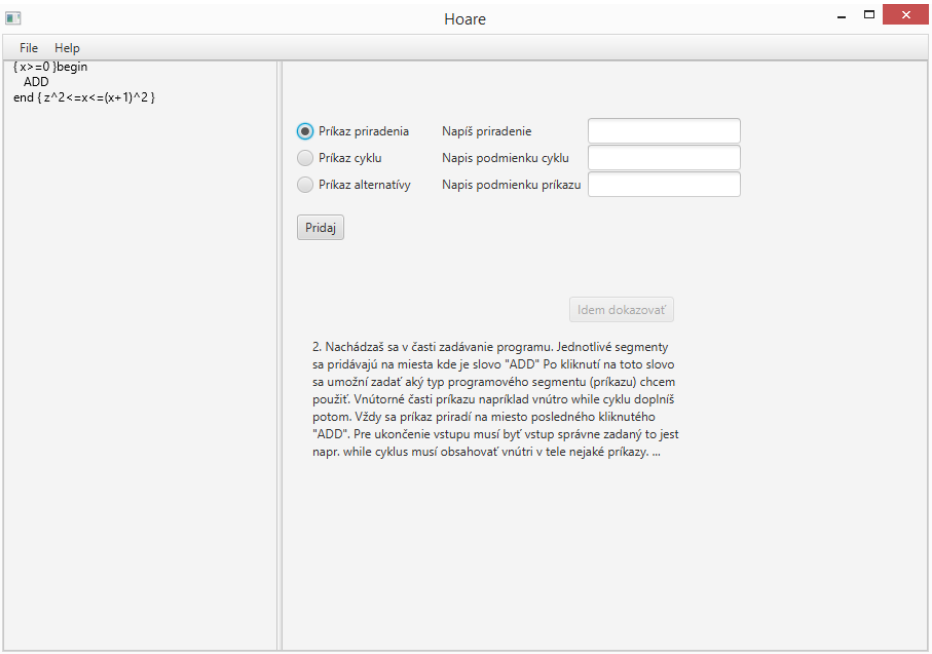
2.2 Plocha pre načítavanie programu

Pri rozmýšľaní ako bude vyriešené zadávanie vstupu do našej aplikácie sme mali prakticky dve možnosti.

Prvá možnosť bola načítavať vstup z textu ktorý nám zadal užívateľ. Tu by bola potreba vytvoriť parsovací algoritmus, ktorý by rozoznal príkazy, ktoré užívateľ zadal, skontroloval, či bol program zadaný správne a prípade chyby alebo nesprávneho zadania programu by užívateľa upozornil na jeho chybu. Úžívateľ by musel mať ukázané ako presne majú príkazy vyzerať a celý program by musel písať ručne.

Druhá možnosť, bolo vytvoriť užívateľské prostredie. Pomocou tohto prostredia by bolo možné zadávať program pomozou tlačitiel a menších textových vstupv. Jedna z výhod je, že nie je nutné každý príkaz celý písať. Aplikácia napíše kostru príkazu za užívateľa, čo menej zaťaží užívateľa. Oveľa väčším prínosom pre užívateľa je aj spôsob zadávania. Ten je nastavený tak, že

telo príkazu sa zadáva až po tom ako samotný príkaz. Napríklad, ak by sme chceli zadať príkaz cyklu, zapíšeme jeho podmienku. Aplikácia si kostru príkazu napíše sama a až potom zadávame telo príkazu, ktoré môže obsahovať iné príkazy. Benefit tohto spôsobu zadávania vstupného programu do aplikácie je jednoduchšie pochopenie reprezentácie štruktúrovaného programu ako stromu. Toto môže pomôcť k lepšiemu pochopeniu Hoareho metódy.

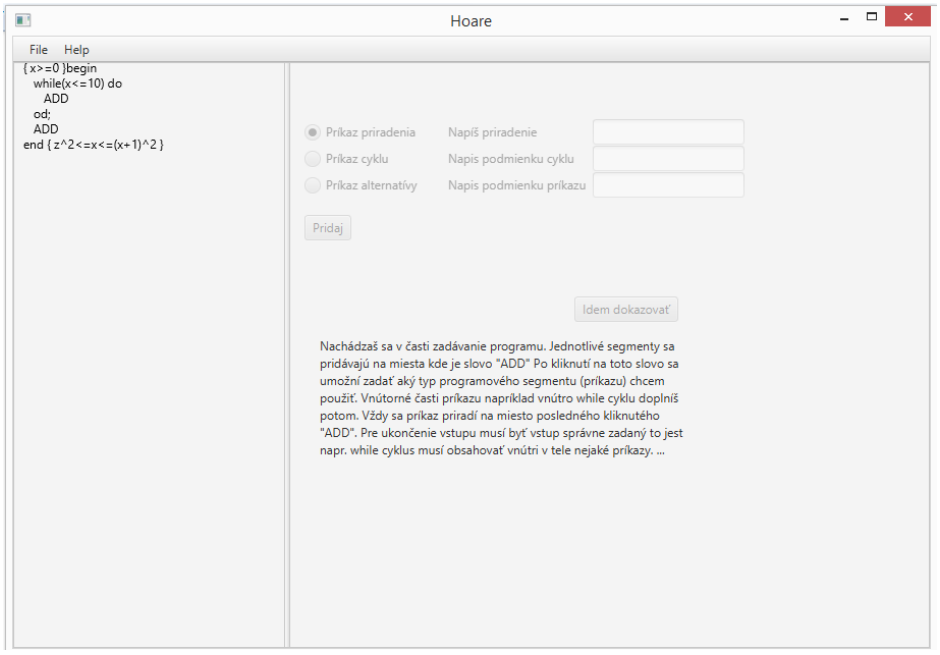


Obr. 2.2: Načítavanie vstupu

Prvá dôležitá vec, ktorú si na obrázku 2.2 treba všimnúť, je textový popis na pravej dolnej časti okna aplikácie. Tento krátky text slúži k tomu aby užívateľ pochopil ako zadávať program. Z prvého pohľadu to nemusí byť zrejmé. Po prečítaní informačného textu je zadávanie programu veľmi jednoduché, prehľadné a intuitívne. V ľavej časti môžeme vidieť program v takej forme, ako ho máme momentálne zadaný. Po každom zadanom príkaze sa sem príkaz pridá na príslušné miesto a táto časť sa prekreslí. Na začiatku

zadávania vstupu, ako môžeme vidieť na obrázku 2, obsahuje iba príkazy begin(vstupná podmienka), end(výstupná podmienka), a medzi nimi slovíčko *ADD*. Po kliknutí na slovíčko *ADD* sa aktivuje plochu číslo 2 na pravej strane okna.

Táto plocha obsahuje možnosť výberu jedného z troch príkazov : príkaz alternatívy, priradenia a cyklu. Po vybratí je potreba zadania ďalších údajov, ktoré treba na pridanie príkazu. V našom prípade to bude podmienka pre príkaz cyklu, podmienka pre príkaz alternatívy a samotný príkaz priradenia, ak sme si vybrali tento príkaz. Po vybratí a zadaní potrebných údajov je možné na tlačítko *Pridaj* klinúť a tým sa pridá príkaz do ľavej strany okna aplikácie.. Ak sa jedná o zložený príkaz tak pridá *ADD* aj na miesta kde sa nachádza telo príkazu, aby sme tam mohli pridať ďalšie príkazy. Pôvodné *ADD* posunie dole aby bola možnosť zadávať príkazy, ktoré budú nasledovať po našom pridanom príkaze.



Obr. 2.3: Vstup s príkazom cyklu

Na obrázku 2.3 môžme vidieť aplikáciu po zadaní príkazu cyklu do predtým

prázdneho programu. Ak by sme chceli ukončiť zadávanie vstupu do našej aplikácie, museli by sme doplniť ešte príkazy do tela cyklu. Vidíme, že sa tam nenachádza žiaden príkaz. Ak by sme tam doplnili príkaz priradenia, tak naša aplikácia by vstup vyhodnotila ako vhodný na dokazovanie čiastočnej správnosti a umožnila by nám kliknúť na tlačidlo "Koniec dokazovania". Toto je zároveň jediná podmienka a kontrola na to či je vstup správny. Podmienky v zložených príkazoch nijako nekontrolujeme a to z dôvodu, že podmienka ako taká môže byť veľmi rôznorodá a neexistuje presné kritérium, podľa ktorého by sme vedeli určiť čo je dobrá podmienka a čo nie. Čo sa týka kontroly priradenia, tak tam takisto nie je dôvod kontrolovať, či užívateľ naozaj zadal príkaz priradenia. Pretože ak by zadal niečo iné, tak náš program sa k tomu textu, ktorý tam napísal, bude stále správať akokeby to bol príkaz priradenia a nijako to funkcionalitu našej aplikácie neovplyvní. Takže naozaj nám kontrola, či každý zložený príkaz obsahuje v tele svojho príkazu iný príkaz, stačí. Ak to zadaný program spĺňa, je možnosť ukončiť vstup a prejsť k dokazovaniu.

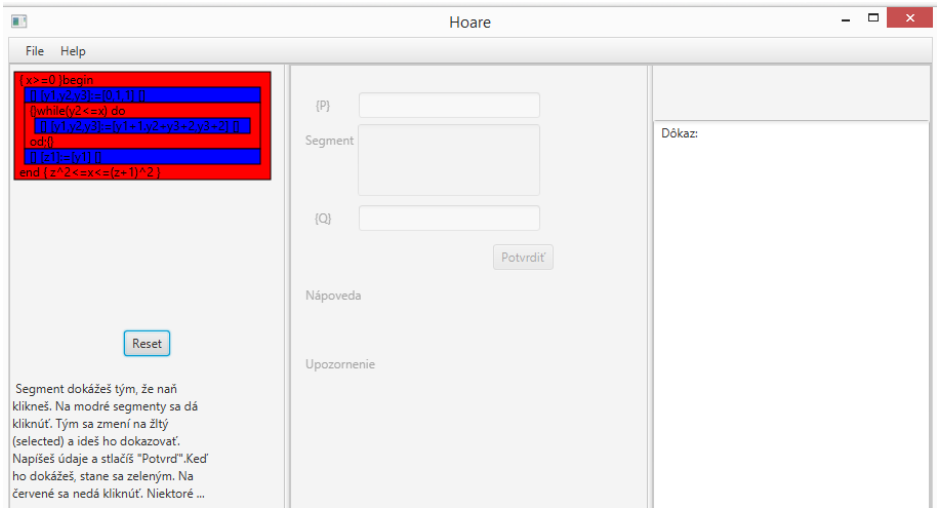
2.3 Plocha pre dokazovanie

Okno aplikácie sa po ukončení zmení. Na nasledujúcom obrázku môžeme vidieť v akom stave je aplikácia po zadaní programu.

Vidíme, že plocha okna aplikácie je rozdelená na tri časti. Vysvetlíme si vlastnosti a možnosti každej časti.

2.3.1 Sekcia vizualizácie

Na ľavej strane aplikácie je miesto na vizuálizáciu algoritmu. Navrhli sme prehľadnú, jednoduchú vizualizáciu na znázornenie v ktorej časti algoritmu dokazovani sa nachádzame.
Každý príkaz je reprezentovaný obdĺžnikom. Príkaz priradenia, ako najjednoduchší príkaz, obsahuje iba telo príkazu. Ostatné príkazy obsahujú okrem slovného vyjadrenia príkazu aj príkazy ktoré sa nachádzajú v ich tele.



Obr. 2.4: Dokazovanie

Napríklad príkaz cyklu, ktorý obsahuje príkaz priradenia, má vlastný obdĺžnik a vo svojom vnútri obsahuje menší obdĺžnik ktorý patrí príkazu priradenia. Toto zobrazenie zodpovedá predstave, ktorú sme už predstavili v prvej kapitole, kde sme program prirovnávali k škatuliam povkladaných do seba. Táto vizualizácia bude počas celého dokazovania zobrazovať v akom štádiu sa užívateľ nachádza, čo potrebuje ešte dokázať a čo práve dokazuje. Okrem toho obsahuje informácie o vstupných a výstupných podmienkach daného segmentu. Takže užívateľ vidí z čoho presne má vychádzať. Na označenie postupu v dôkaze sa používa nasledovné farebné označenie:

1. červená farba: programový segment nemám ešte dokázaný a ani ho nemôžem ísť dokazovať v tejto chvíli

2. modrá farba: programový segment nemám ešte dokázaný ale ak chcem, môžem ho ísť dokazovať

3. žltá farba: programový segment práve dokazujem

4. zelená farba: programový segment som už dokázal

O samotné prepínanie farieb sa stará aplikácia. Ak užívateľ klikne na modrý obdĺžnik, zapne sa sekcia pre dokazovanie a obdĺžnik sa stane žltým. V prípade, že sa rozhodne, že chce začať iným modrým obdĺžnikom, môže užívateľ znova kliknúť na iný modrý obdĺžnik. Tým pôvodný žltý obdĺžnik zmení späť na modrý a práve kliknutý modrý sa stane žltým. Po tom ako užívateľ dokáže daný programový segment, tak ten sa v našej vizualizácií zmení na zelený, zmenia sa vstupno výstupné podmienky daného segmentu a zároveň sa zmenia farby na modrú tým červeným programovým segmentom, ktoré už teraz môžme dokázať.
Zároveň sa tu nachádza tlačítko *Reset*, ktoré v prípade chyby, napríklad zle zadaných podmienko pri dokazovaní segmentu na začiatku nášho dôkazu, umožní dokazovať program celý od začiatku.

2.3.2 Sekcia dokazovania

Táto sekcie je rozdelená na dve časti: vrchnú a spodnú časť.
Spodná časť obsahuje len dva prvky a to nápovedu, varovanie. Obidve majú len jednu úlohu a to je napomôcť užívateľovi správne dokázať programovací segment, ktorý je momentálne vyznačený žltou farbou. Nápoveda obsahuje informácie ohľadom práve dokazovaného segmentu, rovnako ako inferenčného pravidla, ktoré by ma užívateľ použiť. Okrem toho je tam krátky návod ako sa orientovať v programe. Varovanie upozorňuje na čo by si mal dať užívateľ pozor, respektíve na čo by nemal zabudnúť skôr než potvrdí to čo napísal. Horná časť obsahuje textové polia, ktoré slúžia na zadanie dôkazu pre konkrétny segment. Táto časť sa sprístupní až keď je vybratý niektorý programovací segment zo sekcie Vizualizácia a je označený žltou farbou. Telo dôkazu tam naša aplikácia napíše automaticky, aby užívateľ nemusel všetko ručne prepisovať. Cieľom bolo aby užívateľ tejto aplikácie nestrácal čas prepisovaním častí ktoré už zadával. Po zadaní dôkazu do textových je možné dôkaz potvrdiť a tým je segment dokázaný.

2.3.3 Sekcia dôkazu

V poslednej sekcii sa nachádza len jeden prvok a to je textové pole. Do tohto poľa sa ukladá každý dôkaz, ktorý sme zadali počas nášho dôkazu čiastočnej správnosti. Po každom potvrdenom dôkaze zo Sekcie dokazovania sa pridá nový riadok s dôkazom. Cieľom je, aby mal užívateľ pred sebou aj textovú predlohu dôkazu a mohol si do nej dopisovať či už svoje poznámky alebo mierne úpravy. Po skončení dokazovania dokazovania je možnosť si text skopírovať a ďalej distribuovať.

Kapitola 3

Implementácia

V tejto kapitole sa budeme venovať implementácií navrhnutého modelu, ktorý som opísal v kapitole Vizualizácia. Podrobnejšie popíšeme použitý programovací jazyk, knižnice, dátové štruktúry, navrhové vzory a algoritmy, ktoré sme použili. Jedným z cieľov aplikácie bolo, aby fungovala na rôznych operačných systémoch a mohol ju tak používať každý.

3.1 Použité technológie

Rozhodli sme sa použiť programovací jazyk Java, ktorý spĺňa podmienku, aby užívateľ mohol používať aplikáciu bez ohľadu na operačný systém. Vzhľadom k výberu programovacieho jazyka je potrebné, aby užívateľ mal nainštalovanú Javu na svojom počítači. Taktiež sme v tomto jazyku už pracovali, takže sme s ním boli oboznámený. Naša aplikácia beží v samostatnom vlastnom okne, nie je potreba žiadneho iného podporného softvéru alebo knižníc. Okrem programovacieho jazyka sme potrebovali zvoliť vhodné grafické nástroje na vykreslenie našej vizualizácie a na tvorbu užívateľského rozhrania –GUI.

3.1.1 JavaFX

Výber grafickej knižnice úzko súvisel s výberom programovacieho jazyka. Prirodzene, po výbere jazyku Java sme siahli po jeho oficiálnych grafických balíčkoch. Na výber sme mali medzi knižnicami JavaFX a Swing. Ak by nespĺňala ani jedna z týchto knižníc nasledujúce nároky, hľadali by sme nejakú inú knižnicu. Prípadmne by sme zvažovali nad zmenou programovacieho jazyka a tým by sa nám otvorili nové možnosti. Požiadavky boli tieto:

1. Možnosť vytvoriť jednoduché GUI pre prácu s našimi dátovými štruktúrami.
2. Schoposť vykresliť vlastné grafické útvary aj s textom. Taktiež možnosť vyvolanie akcie po kliknutí.
3. Okamžité prekreslovanie obrázka po interakcii s užívateľom.
4. Dostatočná a dostupná dokumentácia pre uľahčenie práce s danou knižnicou.

Vzhľadom k tomu, že JavaFX nahradila Swing v roku 2014 ako oficiálna knižnica jazyku Java pre tvorbu GUI, tak sme sa rozhodli pre túto knižnicu. Okrem toho spĺňa všetky uvedené kritériá, ktoré sme požadovali a zistili sme, že je to odporúčaná knižnica [6]. Takže nebola potreba zvažovať iné možnosti. Pri tvorbe samotného GUI sme mali taktiež na výber. Mohli sme si zvoliť spôsob, že samy si budeme tvoriť jednotlivé tlačítka. Tie by sme následne ručne vkladali do okna našej aplikácie alebo sme mohli na to použiť inú aplikáciu na návrh GUI.

3.1.2 SceneBuilder a práca s ním

SceneBuilder je program plne zameraný na tvorbu užívateľského prostredia. Takmer všetku vizuálnu stránku našej aplikácie sme tvorili v ňom. SceneBuilder pracuje s FXML súbormi. FXML je jazyk, pre návrh formulárov, ktorý je odvodený z jazyka XML. Veľká výhoda bola rýchlosť tohto prístupu a

takisto fakt, že sme videli presný návrh ako naša aplikácia bude vyzerať.

3.2 Návrh aplikácie a rozloženie triedl

Ďalšou podstatnou časťou bolo aj navrhnúť vhodnú softvérovú štruktúru na ukladanie dát, zobrazenie dát užívateľi a spracovávanie vstupov. Pri tvorbe našej aplikácie sme vychádzali z návrhového vzoru MVC (Model-View-Controller). Ide o takú softvérovú architektúru, ktorá rozdeľuje dátový model aplikácie, užívateľské rozhranie a riadiacu logiku do troch nezávislých komponentov. Pre prípad našej aplikácie sme si rozdelili triedy do 2 balíčkov: view a model. O ich obsahu si povieme v nasledujúcich častiach.

3.2.1 Model

V prvej kapitole sme písali o možnosti reprezentácie štruktúrovaného programu ako stromu. Z tejto predstavy sme vychádzali aj keď sme tvorili dátový model našej aplikácie. Každý príkaz bude reprezentovaný jedným vrcholom v strome. Od každého príkazu požadujeme určité vlastnosti, ktoré sú rovanko platné pre všetky príkazy. V našom prípade to bolo, či už bol dokázaný, ako vyzerá jeho textová forma, či má nejaké príkazy v štruktúre programu pod sebou a tak ďalej. Zároveň potrebujeme aby sa v niektorých funkciách správali inak. Už len samotné vypísanie na obrazovku bude pre každý príkaz vyzerať inak. Riešením bolo vytvorenie abstraktnej triedy *Segment* z ktorej dedilo päť tried: *BeginEnd*, *Cyklus*, *Priradenie*, $IfThen$ a *Empty*. Každá táto trieda predstavovala jeden príkaz zo štruktúrovaných programov. Náš dátový model obsahoval len jeden objekt triedy *BeginEnd*. Táto trieda obsahovala *LinkedList* objektov triedy *Segment*. Podobne trieda *Cyklus* a trieda $IFThen$ obsahovala dva. Listy stromu tvorili objekty triedy *Priradenie*. Triedu *Empty* si vysvetlíme keď budeme opisovať zadávanie vstupu. Všetky tieto triedy boli súčasťou balíčka *model*. Okrem toho bola súčasťou tohto

balíčka aj trieda *Coordinates* a *FictionalFather*. Prvá slúžila na jednoduchšiu prácu s celočíselnými koordináciami, ktoré používame pri vykreslovaní a triedu *FictionalFather* si vysvetlíme neskôr.

3.2.2 View

Balíček tried obsahuje v prvom rade triedu *MainAppka*, ktorá obsahuje funkciu *Main*, ktorá sa spúšťa pri štarte našej aplikácie. Ďalej obsahuje triedu *MyText* ktorá dedí z Java triedy *Text* a triedy *MyRectangle*, ktorá dedí z triedy *Rectangle*. Ostatné súbory sú už spomínané tri FXML súbory upravované programom *SceneBuilder* a ich *Controller* triedy. Vytvorili sme rodičovský rámec RootLayout.fxml , ktorý obsahuje len vrchné menu, ktoré sa nemení počas celého behu aplikácie. Následne sme vytvorili dva grafické rámce Intro.fxml a Work1.fxml obsahujúce formulárové komponenty: tlačidlá, textové polia, menovky. Tieto dva grafické rámce sú vložené do rodičovského a náš program sa vie prepínať a raz zobraziť jeden a raz druhý. Tento návrh bol vytvorený kvôli možnosti oddeliť načítavanie vstupu od samotného dôkazu. Preto sa načítavanie odohráva v prvom rámci. Keďže počas dôkazu potrebuje mať užívateľ stále prehľad nad tým, čo už napísal a čo dokázal, rozhodli sme sa vizualizáciu dôkazu robiť iba v jednom rámci.
Ku každému rámcu je pripojená Controller trieda. Táto trieda má prístup k formulárovým komponentom daného vizuálneho rámca. Daný FXML súbor má odkaz na danú triedu ako na svoju Controller triedu. Každý komponent, ktorému sme priadali funkcionalitu, má jedinečné fx:id. Meno tohto fx:id sa nachádza v Controller triede aj so špeciálnym @FXML označením, vďaka ktorému Java vie, že sa jedná o komponent z FXML súboru. Takisto Controller triedy majú referenciu na našu *Main* triedu a tým vedia volať jej funkcie a upravovať náš dátový model.

3.3 Implementácia načítavania

Pri štarte našej aplikácie sa v prvom rade načíta GUI z nášho FXML súboru a priradíme si Controller triedu do premennej aby sme mohli pristupovať k jej funkciám. V úvode očakávame od užívateľa aby zadal vstupnú a výstupnú podmienku. Keďže tieto podmienky nevieme skontrolovať, či ich užívateľ zadal správne, tak sa žiadna kontrola nevykonáva. Po potvrdení sa inicializuje premenná *Program* segmentom *BeginEnd* obsahujúcim predtým zadané podmienky.Toto je zaroveň referencia na koreň nášho dátového stromu. Po inicializovaní sa užívateľovi zobrazí kostra programu obsahujúca príkazy *begin* a *end* spoločne s podmienkami.

3.3.1 Vypísanie na obrazovku

Na to aby sme mohli dať užívateľovi spätnú väzbu, bolo potrebné navrhnúť spôsob, akým sa bude už zadaný program vypisovať. Rozhodli sme sa použiť objekty našej triedy *MyText*, ktoré navyše obsahujú referenciu na svoj objekt triedy *Segment*, ktorý ich vytvoril. Objekty tejto triedy obsahujú ľubovoľný text, ktorý mu zadáma a vie sa vypísať do nášho okna na presne zadanú pozíciu. Taktiež je možné nastaviť tomuto objektu schopnosť vyvolať udalosť po kliknutí naň. Pre vypočítanie pozícií a textu, ktorý sa ma vypísať sme vytvorili funkciu *vypis* pre triedu *Segment*. Táto funkcia sa zavolá hneď na začiatku nášho programu po zadaní vstupnej a výstupnej podmienky. Argumenami tejto funkcie je pole objektov typu *Text* s názvom *Labels* a objekt triedy *Coordinates*, ktorý slúži na prepočítavanie pozície nasledujúceho výpisu. Funkcia sa volá rekurzívne, prechádza cez celý dátový strom príkazov, prepočítava si pozíciu vypisovania a ukladá si novovytvorené *MyText* objekty. Následne sa objekty z *Labels* pridá do grafického rámca *Intro* ktorý ich sám vypíše.

3.3.2 Pridanie príkazu

Na pridávanie sme využili našu pracovnú triedy *Empty*. Táto trieda takisto dedí z triedy *Segment*. Objekty tejto triedy môžu tvoriť len listy v našom dátovom strome. Nachádzajú na konci každého zoznamu Segmentov nejakého vrchola, ktorý obsahuje alebo bude obsahovať ďalšie príkazy ako deti. Objekty tejto triedy sa nám vypíšu ako text *ADD*. Samotný objekt obsahuje referenciu na svojho rodiča a má nastavenú možnosť vyvolať udalosť po kliknutí na jeho text na obrazovke. Po kliknutí na daný text sa uloží refrerencia na rodiča kliknutého segmentu a zavolá sa funkcia z našej Controller triedy pre daný grafický rámec a odblokuje pravá časť aplikácie, ktorá obsahuje možnosť výberu medzi tromi príkazmy, textové polia a tlačidlá. Po vyplnení a a potvrdení sa vytvoril nový objekt príslušnej triedy podľa zadaného príkazu a pridal sa do nášho dátového stromu skrze referenciu na rodiča, ktorú sme si uložili predtým. Dané komponenty určené pre pridanie príkazu sa automaticky zablokujú. Všetky objekty typu *Text* si následne vymažeme z grafického rámca, pretože bol pridaný nový príkaz, ktorý zmenil pozície všetkým nasledujúcim už vypísaním Segmentom. Opäť sa zavolá funkcia *vypis*, ktorá prepočíta pozície pre výpis nášmu celému programu a všetky objekty z *Labels* sa pridajú do grafického rámca. /newline Problém avšak nastal pri pridávaní ďalších príkazov do tela príkazu podmienky. Vzhľadom k tomu, že obsahoval dva zoznamy objektov *Segment* a to jeden pre splnenie podmienky a druhý pre nesplnenie. Bolo problematické rozlíšiť do ktorej časti príkazu podmienky chceme nový príkaz pridať. V obidvoch zoznamoch mal náš objekt *Empty* referenciu na rodiča, ktorý bol ten istý. Riešením bolo vytvorenie triedy *FictionalFather*, korá bola priradená jednému z objektov *Empty* ako jeho rodič. Tento objekt mal referenciu na náš podmienkivý príkaz, obsahoval funkcie *Add*, ktorá volala funkciu *AddB* jeho rodičovi, ktorú vedel zavolať len on. Tým sme vedeli rozlíšiť a pridávať nové príkazy tam, kde sme chceli.

3.3.3 Ukončenie

Ako sme už spomínali v predchádzajúcej kapitole pri opise vizualizácie, ukončiť dovolíme užívateľovi, až keď je vstup správne zadaný. Zistíme to pomocou funkcie $isVertig$, ktorá pre daný objekt triedy *Segment* zistí, či v jeho zozname detí sa nachádzajú aspoň dva príkazy. Vieme, že sa tam určite nachádza *EmptySegment* a takto zistíme, či obsahuje aj nejaký ďalší. Ak áno, rekurzívne sa voláme na všetky deti. Ak všetky deti vrátili hodnotu $true$ tak takú vrátim aj ja. Inak vrátim hodnotu $false$. Pomocou takéhoto rekurzívneho volania viem kedykoľvek zistiť, či môžem dovoliť ukončenie vstupu. Táto funkcia sa volá po každom pridaní pridání príkazu do nášho dátového stromu. Podľa hodnoty, ktorú vráti, sa odblokuje respektíve zablokuje tlačidlo $Idemdokazovať$. Po stlačení tohto tlačidla sa odstránia objekty triedy *Empty* z nášho dátového stromu a načíta druhý grafický rámec, ktorý slúži na vizualizáciu dôkazu a samotný dôkaz.

3.4 Implementácia Hoareho metódy

V kapitole Vizualizácia sme spomínali a na obrázkoch aj ukázali ako bude vyzerať vizualizácia nášho programu. Budeme používať obdĺžniky obsahujúce text príkazu a postup v dôkaze budeme značiť farbami. Po načítaní nášho grafickéhého rámca sú zablokované formulárové komponenty slúžiace na samotné písanie dôkazu. Podstatnou zložkou pri implementovaní dôkazu bolo rozhodnúť sa, či budeme dokazovať program zdola nahor alebo zhora nadol. Pri postupe zdora nahor vieme vždy jednoznačne určiť v ktorom kroku pri dokazovaní sa nachádzame. Postupujeme od najmenších príkazov až kým nedokážeme celý program. Problémom je určiť podmienky pre príkaz priradenia, ktorý sa môže nachádzať v tele zložitejších príkazov. Avšak príklady, ktoré by obsahovali viacero príkazov iterácie alebo podmienky v sebe, sú zbytočne komplikované na cvičenie dokazovania. Bežný užívateľ sa s nimi nestretne a preto sme toto nepovažovali za až tak veľké negatívum. Prínosom je čistota dôkazu. Samotný dôkaz smerom zdola nahor obsahuje len použité

inferenčné pravidlá, axiómy priradenia a programové segmenty. Nie je potreba, aby obsahoval dodatočné vysvetľujúce texty. Taktiež pri intrepretácií, ktorú sme navrhli, je výhodné použiť tento postup. Pridávanie jeednotlivých podmienok pre daný príklad je jasné, pretože sa tam pridávajú pri dôkaze samotného príkazu.

Postup zhora nadol je oproti predchádzajúcemu o niečo intuitívnejší. Pretože postupujeme od známych podmienok vstupu a výstupu. Avšak určenie podmienok pri dekompozícií na menšie programovacie segmenty sa môže ukázať oveľa komplikovanejšie. Taktiež to závisí od príkladu. Problémom pri implementácií je, že podmienky sa k danému príkazu pridávajú najprv. To znamená, že v jednom momente prideľujem podmienky viacerým segmentom, takže aj formulárové komponenty by sme museli navrhnúť inak. Možné riešenie by bolo určiť prvú podmienku pri dekompozícií.Následne dokázať segment pre, ktorý poznáme vstupnú a výstupnú podmienku a až potom určiť ďalšiu podmienku v dekompozícií. Avšak toto je na implementáciu značne náročné, najmä určiť na ktorý bojekt viem kliknúť a akú akciu to má vykonať. Taktiež je komplikovanejšie určiť v ktorej časti algotitmu sa nachádzame. Pretože daný segment nemôžem prehlásiť za dokázaný, ak nie sú dokázané príkazy, ktoré obsahuje. Pri zamýšľaní sa nad touto alternatívou prístupu sme zistili, že dané segmenty by sme potrebovali rozlíšiť oveľa viacej farbami ako len štyrmi.Ďalším možným negatívom je vetveni. Pri predošlom postupe ak sme videli v dôkaze axiómu priradenia, tak sme vedeli, že toto je vetva dôkazu ktorá sa čoskoro spojí s inou. Tu je hlavným problémom, že celý dôkaz sa vetví a možnosti ako pokračovať spočiatku pribúdajú. Okrem toho, k danému postupu sa často píšu komentáre, ktoré vysvetľujú dané kroky a nechceli sme užívateľa zaťažiť prílišným písaním.

Po zvážení uvedených dôvodov sme sa rozhodli pre metódu zdola nahor a tú sme aj implementovali.

3.4.1 Vykreslenie vizualizácie

Pri samotnom vykreslení sme narazili na problém. Pôvodne sme chceli vykresliť obdĺžniky, ktoré by sami obsahovali text vo vnútri a následne by sme ich vykreslili. Avšak naformátovať text vo vnútri obdĺžnika napríklad pre príkaz cyklu sa ukázalo ako veľmi zložité. Nenašli sme spôsob ako text naformátovať tak aby vnútri dostatočná medzera pre vizualzáciu príkazov z tela cyklu. Preto sme sa rozhodli, že našu vizualizáciu implementujeme pomocou už navrhnutého vypisovania textu a následne vykreslíme príslušné obdĺžniky pod danými textovými objektami. Vytvorili sme si triedu *MyRectangle*, ktorej sme dali stavovú premennú *counter* podľa ktorej vieme určiť v akom stave sa nachádza daný príkaz, ku ktorému prináleží daný obdĺžnik. Slúži to hlavne pre farebné odlíšenie daného obdĺžnika. Na to aby sme vykreslili naraz aj obdĺžniky aj text sme museli vypočítať rozmery a umiestnenie obdĺžnikov najprv. Dôvodom bolo, že ich nebudeme vykreslovať v takom poradí ako to budeme počítať. Na to sme vytvorili funkciu *calculateVisualization*, ktorá funguje podobne ako funkcia *vypis*. Ako argumenty má jeden objekt triedy *Coordinates* a aktuálnu šírku obdĺžnika, ktorý chceme vypísať. Ako argument vracia výšku obdĺžnika, ktorý máme vypísať. Tak si vie zistiť aký veľký má byť obdĺžnik pre zložitejší príkaz. Ak vieme výšku obdĺžnikov príkazov, ktoré vykresľujeme vnútri tohto obdĺžnika. Táto funkcia zároveň každému segmentu v našom dátovom strome priradá do premennej *stvorcek* daný odĺžnik. Následne zavoláme funkciu *visualize*, ktorá nám do grafického rámca pridá naše novvytvorené objekty *MyRectangle*. Prechádza náš dátový strom a najprv vykreslí daný obdĺžnik a potom sa rekurzívne zavolá na deti daného segmentu. Takto sa nám obdĺžniky budú prekrývať presne tak ako sme chceli a nestane sa, že by väčší obdĺžnik zakryl menší a nebolo by ten menší vidieť. Následne zavoláme funkciu *vypis* a pridáme aj textovú zložku pre vizualizáciu nášho dôkazu.

3.4.2 Dôkaz

Implementácia samotného dôkazu spočíva vo funkcii *canBeProven*. Táto fukcia sa zavolá hneď po vykreslení a volá sa rekurzívne na dátový strom. Pre každý vrchol zisťuje, či ho aktuálne môžem dokazovať. Aby som ho mohol dokazovať, musí byť ešte nedokázaný a musím mať všetky príkazy, ktoré obsahuje, dokázané. Pre každý vrchol, ktorý toto spĺňa nastavím farbu jeho obdĺžnika na modrú a zároveň premennú *counter* v $MyRectangle$ na hodnotu 1. Udalosť ktorá je vyvolaná kliknutím na daný obdĺžnik priamo závisí od honoty premennej *counter*. Iba v prípade hodnoty 1 je vyvolaná akcia. Akcie, ktoré sa vyvolajú kliknutím na modrý obdĺžnik:

1. Zmení sa predchádzajúci žltý obdĺžnik na modrý. Opäť sa volá funkcia $CanBeProven$, ktorá správne nastaví farbu pôvodnému žltému obdĺžniku

2. Odblokujú sa komponenty pre dokazovanie. Stredná časť, obsahujúca tlačidlá a textové polia sa odomkne a je možné do nej vpisovať. Obsahuje textové pole pre vstupnú podmienku P a výstupnú Q, samotné dokazovaný programový segment a tlačidlo na potvrdenie. Textové pole pre programový segment je už predvyplnené vďaka funkcii $getJadro$, ktorá sa volá na vrchol v strome, následne na jeho deti a vráti písomnú formu celého segmentu.

3. Aktualizujú sa výpisy. Náš kliknutý obdĺžnik zavolá funkciu jeho príslušného Segmentu, ktorý má v sebe uložené textové informácie o inferenčnom pravidle, ktoré treba použiť, o upozornení ktoré sa má vypísať. Samotné upozornenie sa môže pre rovnaký segment líšiť, keďže upozorňujeme aj na správne použitie pravidla následku, čiže vypisujeme implikácie zo vzťahov vstupných a výstupných premenných pre segmenty , ktoré daný segment obsahuje.

4. Pošle informácie objektu triedy $Work1Controller$, ktorá si údaj uloží a následne ho použije v prípade dokázania daného segmentu.

Po vyplnení všetkých polí na zadanie dôkazu a potvrdení tlačidlom sa vyvolá akcia, ktorá vykoná nasledovné funkcie.

1. Zmení hodnotu premmennej *counter* na 2 a farbu na zelenú vo vybranom obdĺžniku, zmení hodnotu $Proven$ v segmente nášho dátového stromu.

2. Zistí si $String$, ktorý sa nachádza v textovom poli obsahujúcom celý dôkaz, pridá na koniec časť dôkazu, ktorý sme práve napísali a uloží so do vlastnej premennej typu $String$.

3. Aktualizuje vstupné a výstupné premenné pre dané segmenty v našém modely. Priradí k danému segmentu hodnoty z textových polí P a Q a premaže vstupné a výstupné hodnoty segmentov, ktoré sa nachádzajú v dokazovanom segmente.

4. Opätovne načíta grafický rámec, kedže potrebujeme prekresliť vizualizáciu, kde bola aktualizovaná aj textová zložka. Volá sa funkcia $canBeProven$, ktorá nastaví farebné odlíšenie. Z premennej, ktorá obsahuje celý dôkaz sa daný dôkaz vloží do textového poľa.

V prípade chyby počas dokazovania je možné stlačiť tlačidlo $Reset$, ktoré rekurzívne prejde celý dátový strom, premaže všetky vstupné a vystupné hodnoty segmentov a nastaví hodnotu premenných $Proven$ na $false$. Následne sa prekreslí grafický rámec, dôkaz sa vymaže a užívateľ môže dokazovať odznova. V nasledujúcej kapitole si ukážeme funkcionalitu našej aplikácie na príklade.

Kapitola 4

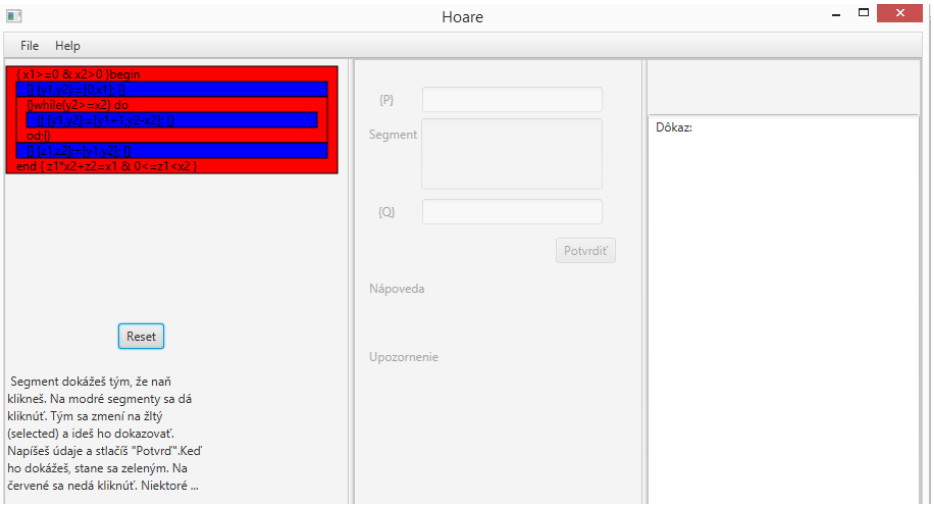
Ukážka vizualizácie

V tejto kapitole si prakticky ukážeme ako v našej aplikácií spracúvame a čiastočne dokazujeme správnosť zadaného programu. Nasledovný príklad je prebratý zo zbierky riešených úloh [4]:

$begin\{x_1 \geq 0 \wedge x_2 > 0\}$
$[y_1, y_2]$:=[0,x_1];
while $y_2 \geq x_2$ do $[y_1, y_2]$:=$[y_1 + 1, y_2 - x_2]$ od;
$[z_1, z_2]$:=$[y_1, y_2]$;
end $\{z_1 * x_2 + z_2 = x_1 \wedge 0 \leq z_2 < x_2\}$

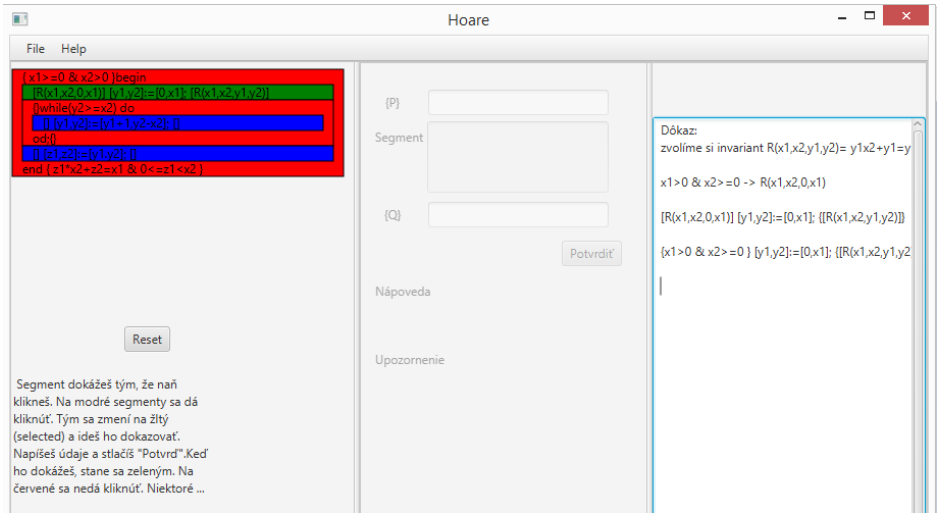
4.1 Dôkaz pomocou aplikácie

Pomocou aplikácie načítame náš program. Keďže sme načítanie detailnejšie opísali v kapitole Vizualizácia a preto ho nebudeme teraz opisovať pomocou obrázkov. Na 4.1 vidíme stav aplikácie po zadaní vstupu. Ešte sme nezačali nič dokazovať. Vidíme, že máme na výber pre začiatok nášho dôkazu. Bolo by vhodné ak by sme si určili invariant pomocou ktorého budeme daný program dokazovať.

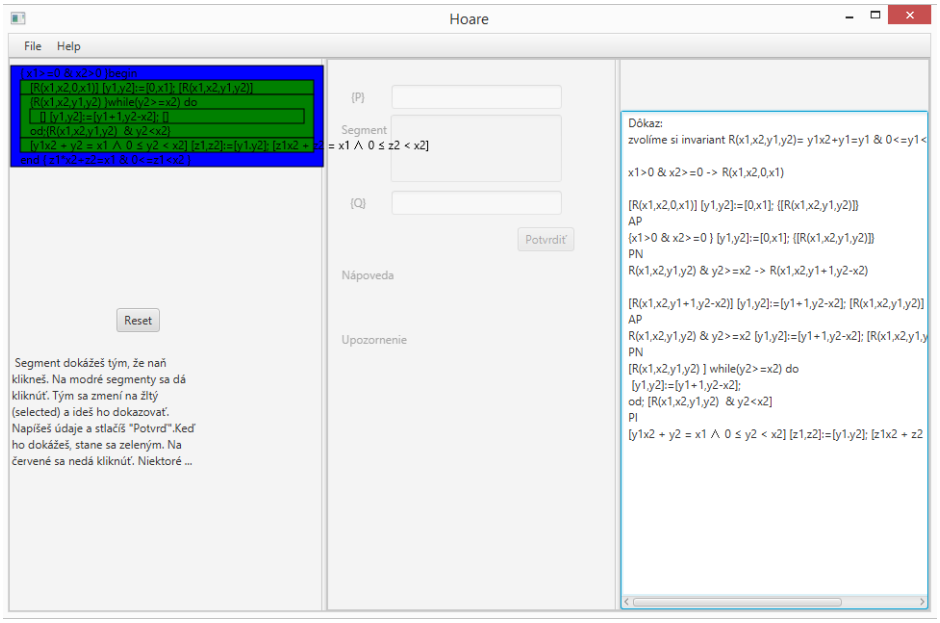


Obr. 4.1: Po načítaní

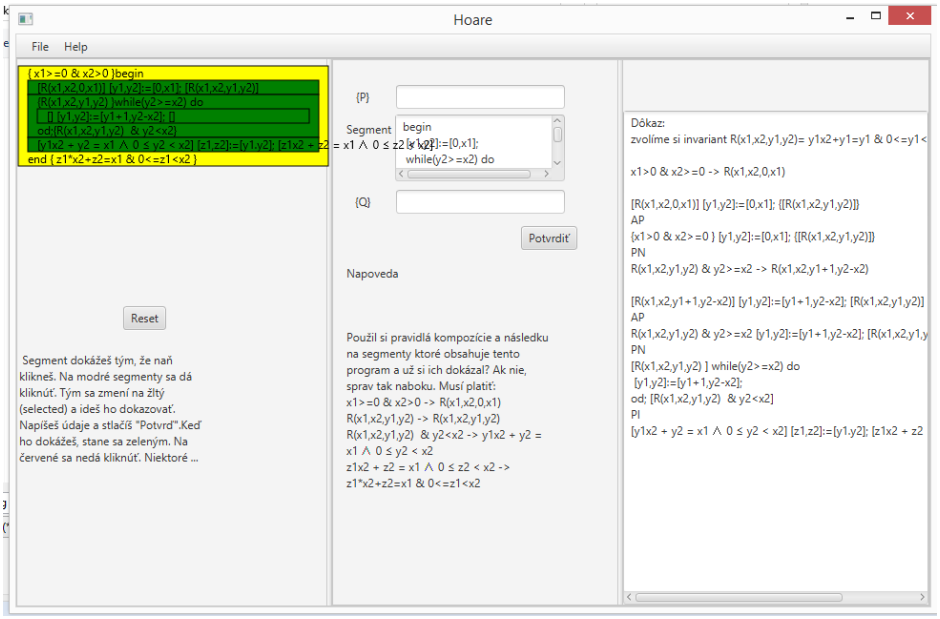
Zvolíme si invariant, ktorý budeme používať a zapíšeme jeho údaje do bočného textového poľa. Taktiež sme dokázali prvý príkaz priradenia pomocou axiómy priradenia a použili sme pravidlo následku pre vstupnú podmienku a danú axiómu. Stav po týchto krokoch vidíme na 4.2
Následne sme použili vzťahy vypývajúce z invariantu a dokázali príkaz priradenia vo vnútri cyklu, pomocou axiómy priradenia. Potom sme dokázali príkaz iterácie a príkaz priradenie na konic programu. Stav po týchto krokoch vidíme na 4.3. Program sa pred dokončením dôkazu pýta, či platia všetky uvedené vzťahy na to aby tam mohlo byť použité pravidlo následku medzi jednotlivými príkazmi, ktoré náš program obsahuje. Ak užívateľ nespájal programové segmenty postupne pomocou pravidla následku teraz je mu to pripomenuté aby tak spravil ako to vidíme na obrázku 4.4.
Na obrázku 4.5 vidíme program po dokončení dôkazu. Textovú časť si môže užívateľ ešte poupraviť do finálnej podoby. Ak chceme ísť dokazovať ďalší program, je potreba v menu zvoliť možnosť New File.



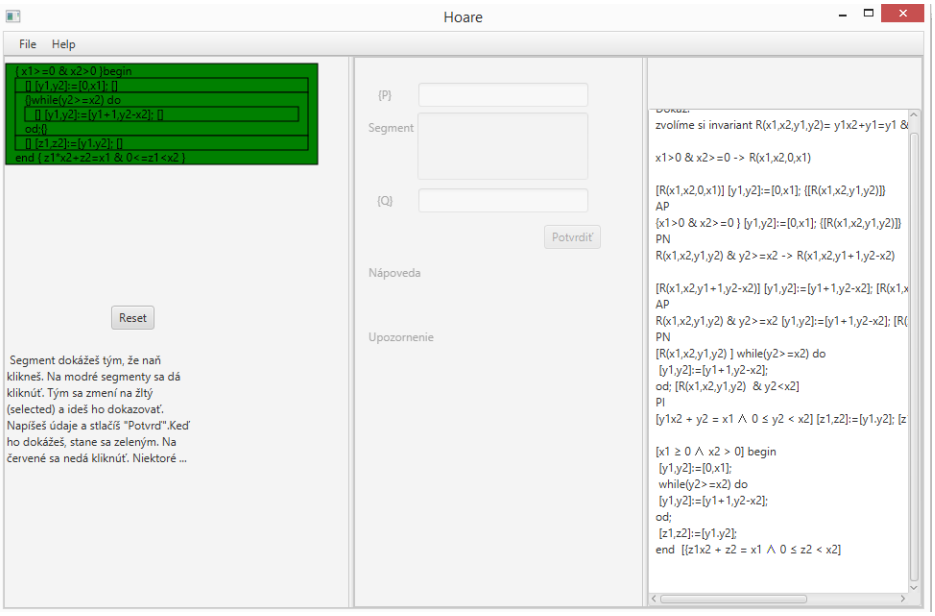
Obr. 4.2: Po dokázaní axómy



Obr. 4.3: V polke dôkazu



Obr. 4.4: Tesne pred dokázaním



Obr. 4.5: Dokončený dôkaz

Záver

Podarilo sa nám splniť hlavný cieľ tejto práce, čo bolo vytvorenie aplikácie pre vizualizáciu Hoareho metódy dokazovania čiastočnej správnosti.
Práca obsahuje teoretické základy v dostačujúcom rozsahu pre pochopenie danej metódy. Ďalej popisujeme náš prístup k vizualizácií, spôsob reprezentácie programov, možnosti v postupe dôkazu. Taktiež sa práca zaoberá implementačnými detailmy danej aplikácie. Pri vývoji aplikácie sme sa stretli s viacerými možnosťami, akým smerom chceme našu aplikáciu vyvíjať. Hlavným bolo voľba postupu dôkazu. Vybrali sme si postup zdola nahor a je možné, že postup zhora nadol neskôr implementujeme do našej aplikácie. Taktiež venujeme jednu kapitolu ukážke práce s aplikáciou pre lepšie porozumenie zo strany študentov.
Veríme, že daná aplikácia pomôže študentom pri ich štúdiách teoretickej informatiky.

Literatúra

[1] binaervarianz. Visualisation of hoare algorithm, 2012. Dostupné z `http://j-algo.binaervarianz.de/`.

[2] Martin Filek. Vizuálna podpora dokazovania správnosti programov pomocou floydovej metódy. Bakalárska práca, Univerzita Komenského v Bratislave, 2014.

[3] Mike Gordon and Helene Collavizza. Forward with hoare, 2010. Dostupné z `http://www.cl.cam.ac.uk/ mjcg/Hoare75/paper.pdf`.

[4] Ondrej Jombík. *Matematická teória programovania. Zbierka riešených úloh.* Fakulta matematiky, fyziky a informatiky, Univerzita Komenského, 2007.

[5] Igor Prívara. Základy matematickej teórie programovania. Fakulta matematiky, fyziky a informatiky, Univerzita Komenského v Bratislave, 2007.

[6] Jan Stafford. Moving from swing to javafx, 2013. Dostupné z `http://www.theserverside.com/feature/Pros-cons-of-moving-from-Swing-to-JavaFX-UI-tools-a-plus`.