VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Automatické nalezení lineárních vztahů mezi proměnnými programu

Automatic Discovery of Linear Relations among Variables of a Program

2013 Tomáš Navrátil

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Tomáš Navrátil**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Automatické nalezení lineárních vztahů mezi proměnnými programu
Automatic Discovery of Linear Relations Among Variables of a Program

Zásady pro vypracování:

Cílem práce je implementovat algoritmus pro automatické zjišťování lineárních vztahů (tj. vztahů popsaných lineárními rovnicemi a nerovnicemi) mezi hodnotami proměnných programu. Tato problematika spadá do oblasti statické analýzy programů, konkrétně pak do oblasti analýzy založené na tzv. abstraktní interpretaci.

1. Nastudujte algoritmus popsaný v článku Cousot, Halbwachs (1978) a z literatury nastudujte související problematiku (abstraktní interpretace, algoritmy pro práci s konvexními mnohostěny).
2. Implementujte algoritmus popsaný v Cousot, Halbwachs (1978) ve Vámi zvoleném programovacím jazyce. Vytvořený program dostane jako vstup popis libovolného programu ve Vámi navržené syntaxi a jako výstup vydá pro každý bod zadaného programu seznam všech zjištěných lineálních vztahů mezi proměnnými programu v tomto bodě.
3. Vytvořený program otestujte na vhodně zvolených příkladech.

Seznam doporučené odborné literatury:

P. Cousot, N. Halbwachs - Automatic Discovery of Linear Restraints among Variables of a Program. In 5th ACM SIGPLAN-SIGACT Symp. of Principles of Programming Languages (POPL), pp. 84-97, 1978.

Další literatura podle pokynů vedoucího práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Zdeněk Sawa, Ph.D.**

Datum zadání: 18.11.2011
Datum odevzdání: 07.05.2013

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA * Fakulta elektrotechniky a informatiky * OSTRAVA *

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Poděkování

Mé poděkování si zaslouží hlavně pan Ing. Zdeněk Sawa, Ph.D. za odborné vedení v průběhu celé práce. Dále bych chtěl poděkovat rodičům za povzbuzování a trpělivost v době, kdy tato práce vznikala.

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....

Abstrakt

Cílem této práce je navrhnout a následně implementovat algoritmus pro automatické zjišťování lineárních vztahů mezi hodnotami proměnných ve vstupním programu. Lineární vztahy jsou popsány lineárními rovnicemi a nerovnicemi, které mohou v *n*-rozměrném prostoru tvořit matematický objekt zvaný mnohostěn, který se stane základem pro výpočty použité v návrhu. Vstupní program bude načten ze vstupního textového souboru. Výstup pak bude soustava lineárních omezení pro každý bod vstupního programu. Proměnné lineárních omezení budou proměnné vstupního programu. Problematika spadá do oblasti statické analýzy programu, konkrétně do jedné z jejích metod – abstraktní interpretace.

Klíčová slova

statická analýza, abstraktní interpretace, mnohostěn, polyhedron, lineární omezení, frame, konvexní obálka, Laneryho metoda, flowchart, program

Abstract

The aim of this work is to design and to implement an algorithm for automatic discovery of linear restraints among variables of a program. Linear restraints are described by linear equations and inequalities, which can form a mathematical object named Polyhedron in *n*-dimensional space. This Polyhedron will be used as an essential object for computations in this design. Input program will be read from a text file. Output will be linear restraints for each node of an input program. The variables of linear restraints will be variables of the input program. The topic of this work belongs to subarea of Static Analysis of programs called Abstract interpretation.

Keywords

static analysis, abstract interpretation, polyhedron, linear restraints, frame, convex hull, Lanery's method, flowchart, program

Seznam obrázků

Obsah

1 Úvod

Cílem této práce je implementovat algoritmus, který je popsán v článku [1]. Článek popisuje postupy a matematické základy pro řešení problému automatického nalezení lineárních vztahů mezi proměnnými v programu. Tato problematika spadá do oblasti statické analýzy zdrojového kódu.

Lineární vztahy mezi proměnnými v programu lze popsat pomocí soustavy lineárních rovnic a nerovnic. Tyto lineární vztahy se také někdy nazývají lineární omezení. Například soustava tří lineárních omezení:

$$a \geq 0; a \leq 10; 2a + b = 3$$

popisuje vztah mezi dvěma proměnnými a a b. První dvě nerovnice znamenají, že proměnná a může nabývat hodnoty od nuly do deseti a třetí rovnice znamená, že hodnotu proměnné b lze získat z hodnoty proměnné a rovnicí

$$b = 3 - 2a$$

Lze vidět, že poslední rovnice reprezentuje lineární vztah mezi oběma proměnnými, který platí pro určitý bod tohoto programu. Tímto bodem je obecně myšleno takové místo programu, kde se mění hodnoty proměnných. Například za příkazem, který přiřazuje do proměnné nějakou hodnotu.

Program, který je cílem této implementační práce, dostane jako svůj vstup zdrojový kód nějakého programu. Nad tímto zdrojovým kódem provede analýzu, jejímž výstupem budou lineární omezení, která budou popisovat vztahy mezi proměnnými v programu. Tyto omezení pak budou předložena jako výstup analýzy pro kontrolu nebo další zpracování.

Provedená analýza může například ukázat, že hodnota proměnné může v některých případech nabývat hodnot v intervalu, který přesahuje limity jejího datového typu. V jiném případě se může zjistit, že hodnota indexu pole může přesahovat velikost tohoto pole. Jde tedy vidět, že jde o problémy, které nemusí být na první pohled ze zdrojového kódu programu zřejmé a můžou způsobit nechtěné chování programu.

Obecně se programy provádějící tento druh analýzy zdrojového kódu řadí do oblasti nazvané statická analýza programu. Statická analýza se zabývá vyhodnocením běhu programu ještě před jeho přeložením nebo spuštěním. Je důležité ji provádět u programů (systémů), které musí po nasazení správně a bezchybně fungovat. Tyto systémy jsou charakteristické tím, že jakákoli neočekávaná a neošetřená chyba, která se v nich vyskytne, má za následek ztrátu velkého množství vynaložených finančních prostředků nebo v horším případě újmu na lidském zdraví. Příklad takového systému je složitější výrobní linka pro přesnou montáž. Často se jedná o systémy pracující v reálném čase tzv. Real-Time a v extrémních podmínkách.

Tento dokument je strukturovaný do následujících kapitol. V Kapitole 2 jsou uvedeny důležité teoretické základy nutné k pochopení hlavních, zde použitých algoritmů. V Kapitole 3 je proveden návrh celého programu. Kapitola 4 popisuje implementační detaily. Kapitola 5 popisuje použití programu. Kapitola 6 popisuje testování programu a v Kapitole 7 je provedeno zhodnocení výsledků. Nakonec Kapitola 8 obsahuje seznam použité literatury.

2 Teoretický úvod

V této kapitole bude provedeno seznámení s důležitými pojmy a postupy nutnými pro pochopení návrhu a implementace programu pro automatické nalezení lineárních vztahů mezi proměnnými v programu.

Poznámka: místo věty „Automatické nalezení lineárních vztahů mezi proměnnými v programu“ bude dále v textu někdy použito jen slovo analýza, která bude vždy znamenat uvedenou větu, pokud nebude uvedeno jinak.

2.1 Statická analýza

Statická analýza provádí vyhodnocení programu ne na základě jeho spouštění, ale na základě informací obsažených ve zdrojovém kódu programu. Jinak řečeno, program provádějící statickou analýzu načítá zdrojový kód jiného programu a snaží se ho určitým způsobem analyzovat. Cíl analýzy je ověření správné funkce programu ještě předtím, než bude program spuštěn.

Programy pro statickou analýzu pracují tak, že načtou zdrojový kód programu do vnitřní reprezentace v paměti, nad kterou provedou nějaký druh analýzy. Přesně tento postup provádí i program, který je cílem této implementační práce.

Obecně existuje několik metod jak provádět statickou analýzu programu. Jedna z těchto metod se nazývá abstraktní interpretace, která je také použita v této práci.

2.1.1 Abstraktní interpretace

Cíl abstraktní interpretace je nalezení a popsání všech možných reálných průchodů daným programem. Jde o nerozhodnutelný problém, viz [4]. Dosud existující algoritmy pro provedení abstraktní interpretace totiž naleznou pouze určitou *over*-aproximaci všech možných reálných průchodů. Stručný text vysvětlující základní principy abstraktní interpretace, jehož autor je Patrick Cousot je k nalezení v této webové prezentaci [4].

Abstraktní interpretace se dá použít mnoha způsoby. Může například analyzovat, zda hodnoty proměnných programu jsou vždy v mezích daných jejich datovým typem (např. typy `int` nebo `float` v jazyku C++). Další zajímavé použití je ověření, zda proměnná představující index pole, může nabývat hodnoty vetší, něž je velikost tohoto pole.

V článku [2] v Kapitole 2 je uveden příklad abstraktní interpretace pomocí abstraktních hodnot. Princip této interpretace spočívá v nahrazení, neboli abstrakci, konkrétních hodnot celých čísel jejich intervaly. Znamená to například, že se podmnožina celých čísel $\{-1,1,5,12\}$ nahradí uzavřeným intervalem $[-1,12]$. Je zřejmé, že se tím ztratí informace o konkrétních číslech původní množiny. Například číslo 7 patří do abstraktního intervalu, ale již nepatří do původní množiny čísel. V dalších kapitolách v [2] jsou pak popsány základní vlastnosti tohoto typu abstraktní interpretace.

Cílem této práce je provést abstraktní interpretaci vyhledáváním lineárních vztahů mezi proměnnými v programu. Tato úloha je pouze zobecnění interpretace pomocí intervalů z odstavce výše. Místo intervalů jsou hodnoty proměnných popsány lineárními omezeními, které zároveň určují vztahy mezi těmito proměnnými.

2.2 Lineární omezení

Jak již bylo v úvodu vysvětleno, hledané lineární vztahy mezi proměnnými v programu jsou reprezentovány pomocí soustavy lineárních rovnic a nerovnic, které mohou být nazvány lineární omezení.

Základní myšlenkou následujících definic a potažmo celé zde probírané problematiky je představit si řešení soustav lineárních rovnic nebo nerovnic v obecně *n*-rozměrném prostoru (v soustavě souřadnic o *n* osách). Přitom *n* představuje počet proměnných ve vstupním programu, který je určen pro analýzu. Je-li tedy ve vstupním programu celkem pět proměnných, budou všechny výpočty probíhat v prostoru o pěti rozměrech.

Pro účely celé této práce budou v následujícím textu pro reprezentaci čísel v *n*-rozměném prostoru použity množiny čísel reálných $\boldsymbol{R}^n$ (popř. pouze $\boldsymbol{R}$) nebo racionálních $\boldsymbol{Q}^n$ (popř. pouze $\boldsymbol{Q}$). Tyto množiny čísel tvoří v algebraickém smyslu těleso. Aby nedocházelo k nejasnostem, budou obě množiny značeny jako těleso $\boldsymbol{F}^n$ (popř. pouze $\boldsymbol{F}$).

2.2.1 Lineární rovnice

Obecně lze soustavu *m* lineárních rovnic o *n* neznámých zapsat ve tvaru:

$$\begin{aligned} a_{11}x_1 + \cdots + a_{1n}x_n &= b_1 \\ \vdots \quad \cdots \quad \vdots &\quad \vdots \\ a_{m1}x_1 + \cdots + a_{mn}x_n &= b_m \end{aligned}$$

Kde jednotlivé prvky mají následující význam:

- $a_{ij}: i = 1, \dots, m; j = 1, \dots, n$ se nazývají koeficienty soustavy a $a_{ij} \in \boldsymbol{F}$
- $b_i: i = 1, \dots, m$ se nazývají pravé strany soustavy a $b_i \in \boldsymbol{F}$
- $x_j: j = 1, \dots, n$ jsou proměnné soustavy

Řešení jedné rovnice v *n*-rozměrném prostoru tvoří hyperrovinu, kterou si lze představit jako objekt o *n-1* rozměrech, který dělí prostor o *n* rozměrech na dvě části. Pro představu lze uvést tyto příklady:

- *1*-rozměrný prostor (např. přímka) je rozdělen bodem na dvě části. Na jednu část přímky ležící od bodu napravo a druhou ležící napravo. Bod je *0*-rozměrný.
- *2*- rozměrný prostor (2D plocha) je rozdělen přímkou na dvě části. Přímka je *1*-rozměrný objekt.
- *3*-rozměrný prostor (3D prostor) je rozdělen 2D plochou na dvě části. 2D plocha je *2*-rozměrný objekt. Této ploše se říká rovina.
- Prostor o více než třech rozměrech je dělen hyperrovinou, která je vždy o jeden rozměr prostoru menší.

Obecně lze pojem hyperrovina použít pro prostory všech rozměrů.

Řešením soustavy rovnic výše je pak průnik řešení (hyperrovin) všech rovnic soustavy.

2.2.2 Lineární nerovnice

Podobně jako v předchozím případě, soustavu *m* lineárních nerovnic o *n* neznámých lze zapsat v následujícím tvaru:

$$\begin{matrix} a_{11}x_1 + \cdots + a_{1n}x_n & \leq b_1 \\ \vdots \quad \cdots \quad \vdots & \vdots \\ a_{m1}x_1 + \cdots + a_{mn}x_n & \leq b_m \end{matrix}$$

Kde prvky a_{ij}, b_i *a* x_j mají stejný význam jako u soustavy lineárních rovnic.

Řešením neostré nerovnice v *n*-rozměrném prostoru je *uzavřený* poloprostor. Je *uzavřený*, protože v něm leží i hraniční body, díky použití nerovnosti. Celý *n*-rozměrný prostor je tedy rozdělen na dva poloprostory. V jednom poloprostoru jsou všechny body řešením nerovnice a v druhém není žádný bod řešení nerovnice. Řešením soustavy nerovnic výše je potom průnik všech poloprostorů, které jsou řešením každé nerovnice. Tímto průnikem vznikne matematický objekt, kterému se říká *uzavřený mnohostěn*.

Konvexní množina

Libovolná množina bodů *M* v *n*-rozměrném prostoru je konvexní právě tehdy, existuje-li pro každé dva body $x_1 \in M$ a $x_2 \in M$ číslo λ pro které platí:

$$\lambda x_1 + (1 - \lambda)x_2 \in M \wedge 0 \leq \lambda \leq 1 \qquad (1)$$

Což znamená, že mezi každými dvěma body v *M* existuje úsečka, která leží celá v *M* (každý bod úsečky leží v *M*).

Dále v této práci se budou uvažovat pouze *uzavřené konvexní mnohostěny*.

Poznámka: V [1] se pro výraz *mnohostěn* používá slovo *polyhedron*. Jde o doslovný překlad z anglického jazyka. V následujícím textu se bude využívat obou těchto slov a bude tím myšlen ten samý objekt.

2.3 Mnohostěn

Výše uvedené definice lineárních omezení a jejich reprezentace v *n*-rozměrném prostoru jsou základy pro všechny výpočty popsané v [1] a v této práci. Je důležité uvést, že pro zjednodušení výpočtů se budou uvažovat pouze neostré nerovnosti typu ≤.

Nerovnice navíc představují jakési zobecnění rovnic, protože každou rovnici lze chápat jako dvě neostré nerovnice s obrácenými znaménky. Tedy rovnici:

$$a - 2b = 4$$

lze zapsat jako dvě nerovnice:

$$a - 2b \leq 4; -a + 2b \leq -4$$

Pro výpočty se budou používat pouze nerovnice typu ≤ a pro reprezentaci výsledků se možné dvojice nerovnic převedou na rovnice.

Nyní bude vysvětlena reprezentace lineárních nerovnic v *n*-rozměrném prostoru na konkrétním příkladu. Celá situace je znázorněna na Obrázku 1.

Je dána soustava tří nerovnic:

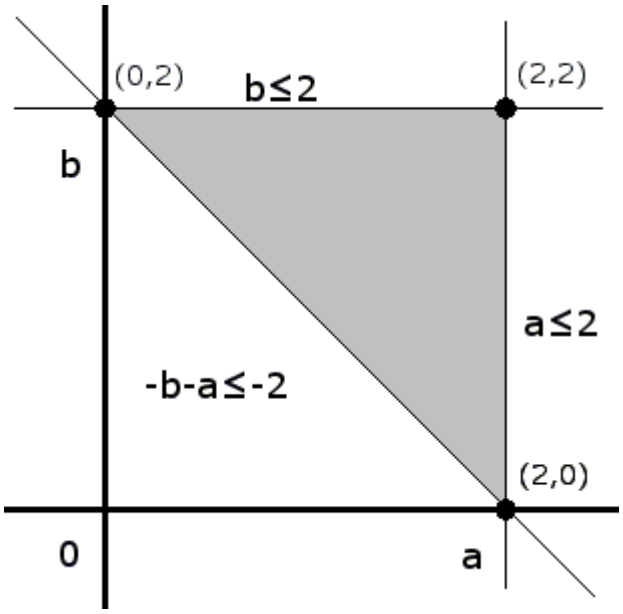
$$b \leq 2; a \leq 2; -b - a \leq -2$$

Proměnné a a b si lze představit jako proměnné vstupního programu pro analýzu. Nerovnice pak jako lineární vztahy mezi těmito proměnnými, které jsou výsledek analýzy. Proměnné jsou dvě. Znamená to tedy, že se množina řešení vyskytuje ve dvourozměrném prostoru. Dle definice soustavy nerovnic výše má:

- nerovnice $b \leq 2$ poloprostor řešení od přímky popsané touto nerovnicí (na Obrázku 1) směrem dolů
- nerovnice $a \leq 2$ poloprostor řešení směrem od přímky doleva
- nerovnice $-b - a \leq -2$ poloprostor řešení diagonálně od přímky doprava a nahoru

Pokud se provede průnik těchto poloprostorů, vznikne mnohostěn, v tomto případě trojúhelník, ohraničený všemi třemi nerovnicemi, který je na Obrázku 1 vyznačen šedě.

Na tento problém se lze také dívat z opačné strany, tedy, tak že každý mnohostěn neboli *polyhedron* lze popsat jako průnik poloprostorů řešení jednotlivých nerovnic ze soustavy lineárních nerovnic. Jinak řečeno, mnohostěn je možno reprezentovat, popisovat, soustavou lineárních rovnic a nerovnic.



Obrázek 1: Reprezentace mnohostěnu

2.3.1 Vrcholová reprezentace mnohostěnu

Pro účely této práce je důležitá jiná reprezentace mnohostěnu. Jde o tzv. vrcholovou reprezentaci. Mnohostěn je popsán množinou vrcholů. Na Obrázku 1 je možno vidět příklad této reprezentace. Jde jednoduše o tři body (0,2), (2,0) a (2,2) představující vrcholy výsledného mnohostěnu.

Poznámka: Pro označení vrcholové reprezentace se někdy používá cizí slovo *frame*. Neexistuje pro něj doslovný český překlad. Nepřesný překlad je rám, tedy množina bodů ohraničující daný mnohostěn. Slovo frame bude používáno dále v textu.

Vrcholová reprezentace mnohostěnu je popsána detailně v [1] v Kapitole 3.2. Zde je důležité poznamenat, že vrcholy (body) představují jen jednu omezenou možnost jak mnohostěn, popsat. Je to z toho důvodu, že výsledný mnohostěn může být v daném prostoru uzavřený nekonečný objekt. Některé jeho hrany popsané rovnicemi mohou být protáhnuty v jednom nebo v obou směrech do nekonečna. Popsat takovýto mnohostěn pouze pomocí bodů by potom nebylo možné, protože by ležely v nekonečnu. Proto je frame mnohostěnu popsán třemi množinami. Množinou vrcholů, polopřímek a přímek.

2.3.1.1 Vrchol

V [1] je pro vrchol v *n*-rozměrném prostoru používán název *vertex*. Vrchol je speciální typ bodu mnohostěnu. Každý neprázdný mnohostěn v prostoru $\boldsymbol{F}^n$ je tvořen nekonečnou množinou bodů. Vrchol je potom takový bod, který není konvexní kombinací ostatních vrcholů. Pro každé dva vrcholy platí vztah (1) výše. Vrcholů je v mnohostěnu konečně mnoho.

Poznámka: Dále v textu se bude pro vrchol používat označení bod a bude tím myšleno totéž, pokud nebude uvedeno jinak.

Bod je v *n*-rozměrném prostoru popsán *n*-ticí souřadnic. Prvky *n*-tice jsou seřazeny dle pevně dané posloupnosti proměnných. Pro představu každá proměnná pak představuje osu v souřadnicové soustavě v *n*-rozměrném prostoru (viz Obrázek 1 osy *a* a *b*) .

Má-li program tři proměnné *a*, *b*, *c* a každý bod bude ve tvaru *(a,b,c)*, potom konkrétní bod *(1,2,3)* znamená následující *(a=1,b=2,c=3)*. Prostor je přitom trojrozměrný s osami *a*, *b* a *c*. Počátek soustavy souřadnic je bod $(0,0,0)$. Stejná logika pojmenování prvků platí i dále u polopřímek a přímek.

2.3.1.2 Polopřímka

V [1] je pro polopřímku používán název *ray*. Polopřímka je vektor v *n*-rozměrném prostoru udávající směr, ve kterém se nachází množina bodů mnohostěnu. Je-li mnohostěn definován jen jedním bodem *s* a polopřímkou *r* potom do něj patří bod *s* a všechny body ležící v prostoru od tohoto bodu ve směru, který určuje polopřímka *r*.

2.3.1.3 Přímka

V [1] je pro přímku používán název *line*. Přímku si lze v *n*-rozměrném prostoru představit jako dva navzájem opačné vektory (polopřímky z Podkapitoly 2.3.1.2) určující směry, ve kterých se nachází množina bodů mnohostěnu. Je-li mnohostěn definován jen jedním bodem *s* a přímkou *d* potom do něj patří bod *s* a všechny body ležící v prostoru od tohoto bodu ve směrech, které určuje přímka *d*. Např. přímka je popsána vektorem $(0,1)$ a určuje tak dva směry dané polopřímkami $(0,1)$ a $(0,-1)$.

Uzavřený konvexní mnohostěn *M* v *n*-rozměrném prostoru čísel $\boldsymbol{F}^n$ lze formálně definovat:

- množinou vrcholů $S = \{s_1, \dots, s_\sigma\}$, kde $\sigma \geq 1$
- množinou polopřímek $R = \{r_1, \dots, r_\rho\}$, kde $\rho \geq 0$
- množinou přímek $D = \{d_1, \dots, d_\delta\}$, kde $\delta \geq 0$

Dále pak existencí čísel:

- $\lambda_1, \dots, \lambda_\sigma \in [0,1]$, kde $\sum_{i=1}^{\sigma} \lambda_i = 1$
- $\mu_1, \dots, \mu_\rho \in \boldsymbol{F}^+$
- $\upsilon_1, \dots, \upsilon_\delta \in \boldsymbol{F}$

Potom lze každý bod *x* (nejen vrchol) patřící mnohostěnu *M*, tj. $x \in M$ získat následujícím vztahem:

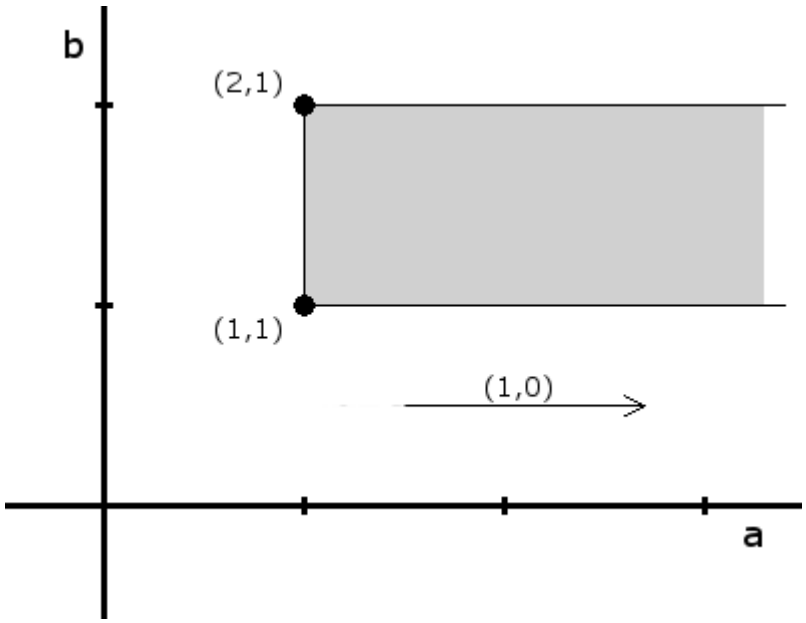
$$x = \sum_{i=1}^{\sigma} \lambda_i s_i + \sum_{j=1}^{\rho} \mu_j r_j + \sum_{k=1}^{\delta} \upsilon_k d_k \quad (2)$$

Ze vztahu (2) plyne, že pokud mnohostěn neobsahuje žádný vrchol, není tím ani splněna podmínka:

$$\lambda_i \in [0,1] \wedge \sum_{i=1}^{\sigma} \lambda_i = 1$$

Bod *x* není potom definován, což znamená, že mnohostěn je prázdný objekt.

Obrázek 2 ukazuje příklad mnohostěnu popsaného pomocí množiny dvou bodů $\{(1,1); (2,1)\}$ a jedné polopřímky $\{(1,0)\}$. Polopřímka (označená šipkou) označuje vektor udávající směr protáhnutí obou bodů do kladného nekonečna v ose *a*.



Obrázek 2: Polopřímka mnohostěnu

2.3.2 Zvláštní případy mnohostěnu

Dosud se předpokládalo, že mnohostěn je tvořen neprázdnou množinou omezení a vrcholovou reprezentací, která obsahuje množinu bodů, polopřímek a přímek. Nyní budou vysvětleny zvláštní případy mnohostěnu. Jsou to prázdný mnohostěn a mnohostěn pokrývající celý *n*-rozměrný prostor.

Mnohostěn je prázdný právě tehdy, když jeho frame neobsahuje žádný bod (vrchol). Když neexistuje žádný bod, ale pouze polopřímky nebo přímky není možné popsat pomocí vztahu (2), kde se jakýkoli bod mnohostěnu v prostoru nachází, protože polopřímky a přímky udávají pouze směr a ne pozici v prostoru, kterou udává bod.

Naopak mnohostěn pokrývající celý *n*-rozměrný prostor, je takový mnohostěn, jehož soustava lineárních omezení je prázdná. Z hlediska analýzy lineárních vztahů mezi proměnnými v programu to znamená, že každá proměnná v programu (je jich celkem *n* v *n*-rozměrném prostoru) může nabývat jakékoli hodnoty.

2.3.3 Převody mezi reprezentacemi mnohostěnu

V předchozím odstavci byl popsán mnohostěn a dva způsoby jak jej popsat neboli reprezentovat. Šlo o reprezentaci pomocí lineárních omezení a reprezentaci pomocí frame. Každý mnohostěn uchovávaný v programu je nutné reprezentovat oběma reprezentacemi najednou. Je to z toho důvodu, že některé algoritmy, které jsou součástí této práce, se lépe a efektivněji provádí pouze nad jednou reprezentací. Po skončení takového algoritmu se pak provede převod do reprezentace druhé.

2.3.3.1 Převod z frame na lineární omezení

V tomto případě je mnohostěn reprezentován pouze pomocí jeho frame, který se převede na soustavu lineárních omezení. K dispozici jsou množiny bodů, polopřímek a přímek frame mnohostěnu. Převod je podrobně popsán v [1] v odstavci 3.3.1.1 a provádí se v následujících dvou krocích:

1. Vytvoření iniciální soustavy rovnic z libovolného bodu.
2. Zakomponování ostatních prvků frame do vzniklé soustavy v pořadí body, polopřímky a přímky. Zakomponováním každého prvku vznikne vždy nová soustava lineárních omezení.

Má-li program např. dvě proměnné a a b a bod (1,2), potom v prvním kroku vznikne iniciální soustava rovnic $a = 1; b = 2$.

Ve druhém kroku pak existují jiná pravidla pro zakomponování bodů, polopřímek a přímek do aktuální soustavy. Obecně operace zakomponování nového prvku do aktuální soustavy do ní vloží sloupec s novou proměnnou, který se následně musí eliminovat, protože jde pouze o pomocnou proměnnou. Po zakomponování posledního prvku do soustavy a eliminaci pomocné proměnné vznikne výsledná soustava lineárních omezení.

2.3.3.2 Převod z lineárních omezení na frame

V tomto případě je mnohostěn reprezentován pomocí soustavy lineárních omezení, která se převedou na objekt frame, tedy na množiny bodů, polopřímek a přímek. Převod je podrobně popsán v [1] v odstavci 3.4.

Pro převod se používají postupy známé z lineárního programování, konkrétně výpočet pomocí simplexové metody. Více o simplexové metodě viz [5].

Úkolem simplexové metody je nalézt optimální řešení soustavy nerovnic. V případě této práce toto řešení představuje jeden vrchol výsledného mnohostěnu. Další vrcholy mnohostěnu se naleznou vyhledáním jiného optimálního řešení opět pomocí simplexové metody.

2.3.4 Zjednodušení reprezentací mnohostěnu

Oba převody mezi reprezentacemi mají nevýhodu v tom, že výsledná lineární omezení nebo frame obsahují redundantní lineární omezení nebo prvky frame.

Převod na lineární omezení může například vygenerovat následující nerovnice:

$$a + b \leq 4; a \leq 10; a \leq 5$$

Ihned lze vidět, že omezení $a \leq 10$ je zbytečné a je již obsaženo v omezení $a \leq 5$.

Jiným příkladem může být objekt frame daný body $\{(1,1); (1,2); (2,1)\}$ a polopřímkou $\{(1,0)\}$. Viz Obrázek 2. Bod $(1,2)$ leží ve směru polopřímky $(1,0)$ vedené z bodu $(1,1)$. Bod $(1,2)$ je zbytečný, protože ze znalosti bodu a směru v prostoru, je možné vypočítat každý bod v tomto směru od tohoto bodu.

Zjednodušení výsledné reprezentace se vždy provádí pomocí druhé reprezentace. Tedy po převodu z frame na lineární omezení se provede zjednodušení lineárních omezení pomocí objektu frame. U převodu lineárních omezení na frame se provede zjednodušení frame lineárními omezeními.

2.4 Reprezentace programu

Předchozí Kapitola čtenáře seznámila s problematikou lineárních omezení reprezentující hledané vztahy mezi proměnnými v programu. Tato kapitola čtenáře seznámí s vnitřní reprezentací zdrojového programu v paměti počítače, nad kterou se bude provádět samotná analýza lineárních vztahů mezi proměnnými v programu.

Vstup do analýzy bude zdrojový kód programu v navrženém jednoduchém programovacím jazyce, nad kterým program, který je cílem této práce implementovat, provede analýzu a jako výstup vydá lineární omezení týkající se proměnných v programu.

Ve Výpisu 1 níže je uveden program v jednoduchém programovacím jazyce, používající dvě proměnné *a* a *b* deklarované za klíčovým slovem `var` a jednu funkci obsahující výpočet, která má název `main`. Tento program je možno použít jako vstup do programu, který nad ním provede analýzu.

Výstupem analýzy jsou pak informace obsažené v bílých obdélnících. Ty představují jakási tvrzení o proměnných v programu v určitém jeho místě, které je vyznačeno šipkou směrem do programu. Tyto tvrzení jsou vlastně hledaná lineární omezení – tedy výstup analýzy.

Výpis 1: příklad programu (testovací případ tc6_4.txt)

```
var a,b = 1;
function main(){
                    b=1
      a = 2*b + 1;
                    a=3; b=1
      while(b ≤ 3){
            a = a + 2;
            b = b + 1;
                    a-2b=3; b≥1; b≤4
      }

      if(a == 9) b = 0;
      else a = 5;
                    a=5;
}
```

Na příkladu lze vidět, že např. za prvním přiřazením `a = 2*b + 1;` bude proměnná *a* vždy nabývat hodnoty 3 a proměnná *b* hodnoty 1.

Trochu složitější situace je u smyčky `while`, pro kterou platí ne hned na první pohled patrná lineární omezení:

$$a - 2b = 3;\ b \geq 1;\ b \leq 4$$

která omezují proměnnou *b* na hodnoty od jedné do čtyř, a přitom lze říci, že pro hodnotu proměnné *a* v každé iteraci smyčky platí vztah:

$$a = 3 - 2b$$

Takto si lze tedy představit analýzu vstupního programu s možnou interpretací výsledků. Nyní bude věnována pozornost vnitřní reprezentaci programu v paměti počítače.

2.4.1 Orientovaný graf

Na příkladu ve Výpisu 1 lze vidět, že lineární omezení budou při analýze postupně vznikat pro různá místa výpočtu programu směrem od jeho začátku k jeho ukončení. Nabízí se tedy možnost reprezentovat vstupní program jako orientovaný graf s jedním počátečním, více koncovými a dalšími uzly představující výpočet programu. Dle [1] je tato reprezentace vhodná pro účely této práce.

Orientovaný graf je dvojice $G = (V, H)$ kde:

- V jsou vrcholy grafu.
- H jsou hrany grafu. Každá hrana je uspořádaná dvojice. Má svůj počátek v uzlu, ze kterého vychází a konec v uzlu do kterého vede. Je značena šipkou.
- Přitom platí že $V \cap H = 0$.

2.4.1.1 Uzly grafu

Uzly grafu představují konstrukce (příkazy, podmínky, větvení, smyčky, …) vstupního programu. Celkem graf obsahuje 6 typů uzlů, které jsou popsány v následujících odstavcích a vyobrazeny na Obrázku 3.

Startovní uzel

Je první uzel grafu a taky jediný tohoto typu, který se v něm nachází. Označuje začátek výpočtu, tedy funkci `main` ve Výpisu 1. Má jednu výstupní hranu. Obsah uzlu jsou inicializační hodnoty proměnných – pouze u těch proměnných, které byly inicializovány při jejich deklaraci za klíčovým slovem `var`.

Koncový uzel

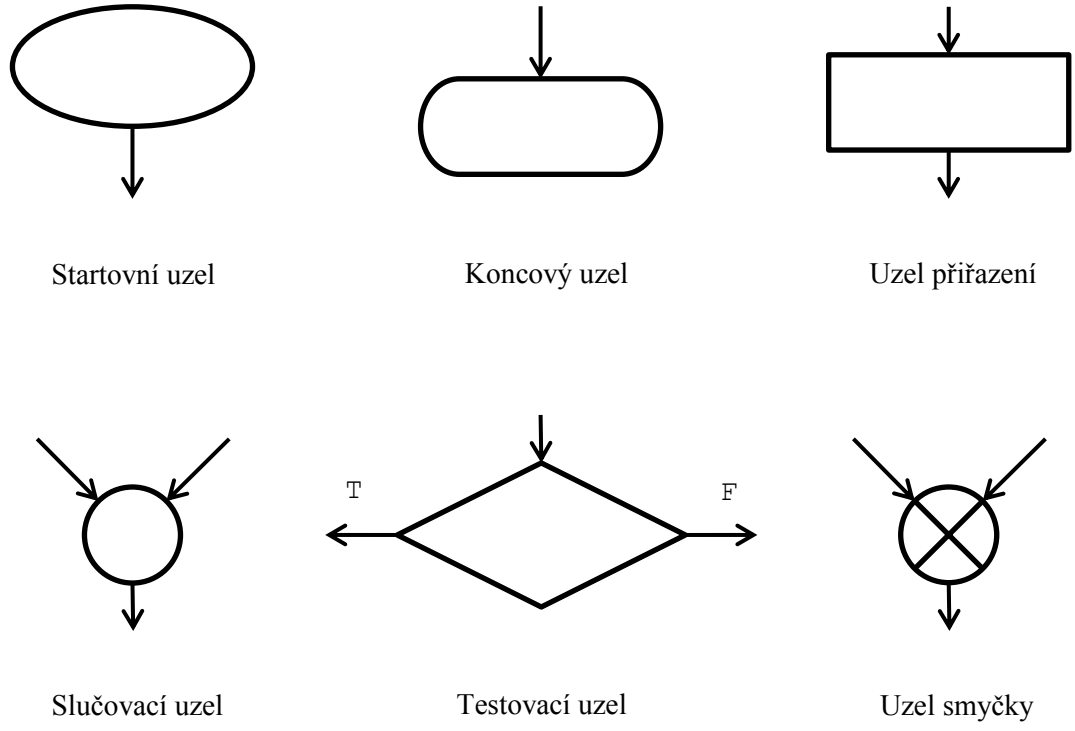
Označuje ukončení zpracovávání programu. Je to poslední uzel pro danou cestu grafem. V grafu se může vyskytovat vícekrát. Má pouze jednu vstupní hranu, a žádnou výstupní. Tento uzel je definován pouze svým typem a nepotřebuje v sobě uchovávat další informace.

Uzel přiřazení

Je asociován s každým přiřazením ve vstupním programu. Má jednu vstupní a jednu výstupní hranu. Jeho obsah je informace popisující přiřazení. Např. `a = a + 2` z Výpisu 1.

Testovací uzel

Je asociován s každou podmínkou v programu (`if`, `for`, `while`). Má jednu vstupní hranu a dvě výstupní hrany – jedna označuje tok v případě, že je podmínka *pravda* a druhá když je podmínka *nepravda*. Jeho obsah je informace popisující testovací podmínku.



Obrázek 3: Uzly orientovaného grafu

Slučovací uzel

Tento uzel představuje slučování cest např. za blokem příkazu `if`. Po vykonání obou větví v příkazu `if` se musí cesty v programu opět spojit. Slučovací uzel tedy spojuje více vstupních hran do jedné výstupní hrany. Tento uzel je definován pouze svým typem a nepotřebuje v sobě uchovávat další informace.

Uzel smyčky

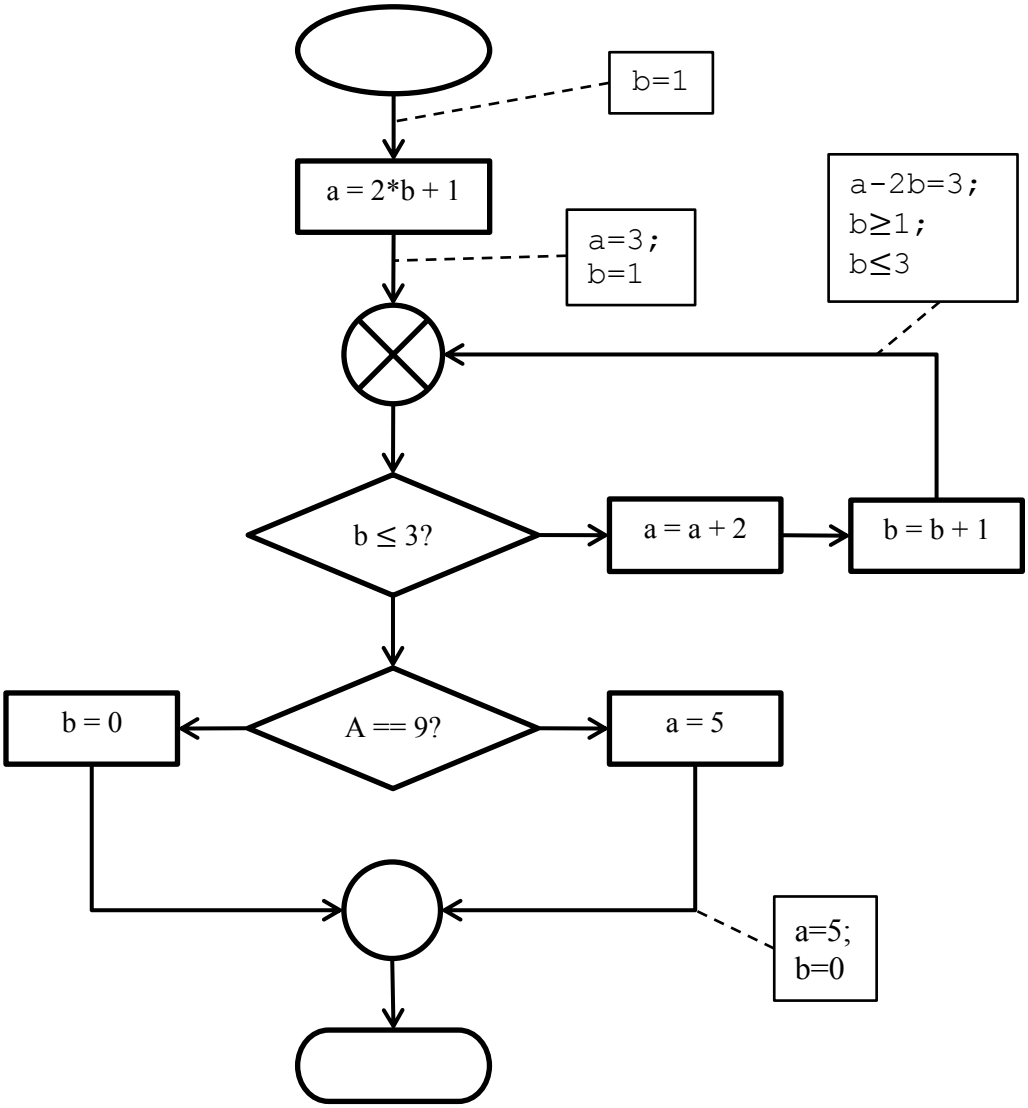
Je speciální typ slučovacího uzlu pro slučování větví přicházející do uzlu z venku smyčky a větví přicházející do něj zevnitř této smyčky. Každý cyklus ve vstupním programu by měl obsahovat jeden uzel smyčky v grafové reprezentaci. Tento uzel je definován pouze svým typem a nepotřebuje v sobě uchovávat další informace.

2.4.1.2 Hrany grafu

V grafu existuje pouze jeden typ orientované hrany a zároveň platí, že každá hrana grafu nese informaci o hledaných lineárních vztazích mezi proměnnými v programu. Tedy, každé hraně jsou přiřazena lineární omezení reprezentovaná soustavou lineárních rovnic a nerovnic a objekt frame. Přitom reprezentace pomocí frame je pouze pomocná reprezentace pro některé druhy výpočtů.

Nyní jsou známy všechny typy uzlů a úloha orientovaných hran v grafu. Pro kompletnost je na Obrázku 4 uveden orientovaný graf ze zdrojového kódu z Výpisu 1, ve kterém jsou také znázorněny lineární omezení přiřazená některým jeho hranám.

Poznámka: Tuto formu grafové reprezentace programu lze také nazvat anglickým slovem *flowchart*. Český překlad by mohl být diagram řídícího toku neboli popis toku algoritmu. Slovo *flowchart* se využívá hlavně v [1] a bude se využívat i v tomto textu.



Obrázek 4: Příklad grafové reprezentace

2.4.2 Převod do grafové reprezentace

Pro převedení vstupního programu do jeho vnitřní grafové reprezentace v paměti budou využity poznatky z teorie překladačů.

Překladač převádí jazyk srozumitelný programátorovi na jazyk srozumitelný stroji (strojový kód), na kterém program běží. Překladač lze rozdělit na dvě části – přední část (nazývaná anglicky *frontend*) a zadní část (nazývaná anglicky *backend*).

Pro převedení vstupního programu do grafové reprezentace je nutné navrhnout přední část překladače, která vstupní program převede do formy abstraktního syntaktického stromu (AST – Abstract Syntax Tree), ze které se potom vytvoří samotný graf.

Lexikální analyzátor funguje jako konečný automat, který načítá vstupní zdrojový kód programu po znacích, které skládá do lexémů (čísla, názvy proměnných, operátory,…) a předává je vyšší vrstvě – syntaktickému analyzátoru.

Syntaktický analyzátor provádí syntaktickou analýzu a převedení programu do AST. Podle načtených lexémů kontroluje, zda je vstupní zdrojový kód správně zapsán. Např. kontroluje použití středníku za každým příkazem, pokud je v daném jazyce středník vyžadován. Syntaktický analyzátor musí tedy obsahovat nějaký popis syntaxe zdrojového kódu programu. Tento popis se nazývá bezkontextová gramatika definovaná jako čtveřice $B = (\Pi, \Sigma, S, P)$, kde:

- Π je konečná množina neterminálních souborů
- Σ je konečná množina terminálních symbolů neboli abeceda. A Platí $\Pi \cap \Sigma = 0$
- $S \in \Pi$ je počáteční neterminální symbol
- P je konečná množina pravidel ve tvaru $A \rightarrow \beta$, kde
  - $A \in \Pi$, je neterminální symbol
  - $\beta \in (\Pi \cup \Sigma) *$ je řetězec složený z neterminálních a terminálních symbolů

Nakonec sémantický analyzátor ověřuje správnost jazykových konstrukcí, procházením AST. Např. kontroluje datové typy proměnných, doplňuje AST o jiné informace a převádí AST do mezikódu.

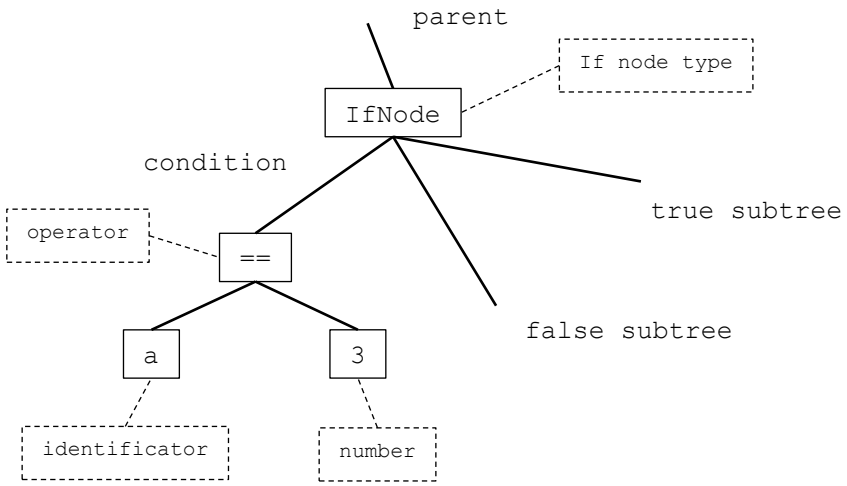
2.4.3 Abstraktní syntaktický strom

AST je stromová reprezentace zdrojového kódu vstupního programu, vzniklá v syntaktickém analyzátoru, aplikováním pravidel gramatiky na lexémy, vrácené lexikálním analyzátorem. Jeho vnitřní uzly jsou příkazy programu (`if`, `while`, …), názvy funkcí a operátory ($\leq, \geq$, ...). Vnější uzly (listy) jsou pak operandy (názvy proměnných nebo čísla). Zdrojový program se tedy načte do podoby AST a až v této podobě se s ním dále pracuje. Přesně tak tomu bude i v rámci této práce.

Pro příklad je ve Výpisu 2 uvedena podmínka `if`, ze které je na Obrázku 5 vytvořen AST.

Výpis 2: podmínka if

```
if(a == 3) {//true part}
else {//false part}
```



Obrázek 5: Abstraktní syntaktický strom

Lze vidět, že uzel `IfNode` samotný je součást nějaké stromové struktury (celého programu) má totiž definovaného rodiče `parent`. Dále se uzel skládá z ukazatele na podstrom podmínky `condition`, která je složena z proměnné `identificator` *a* a čísla `number` 3. Nakonec `IfNode` obsahuje ukazatele na podstromy typu *pravda* a *nepravda* (`true subtree, else subtree`).

Součástí návrhu tohoto projektu je návrh přední části překladače. Nejprve bude navržen konečný automat pro lexikální analyzátor. Pak bude následovat návrh bezkontextové gramatiky pro syntaktický analyzátor. Přitom syntaktický analyzátor bude provádět syntaktickou i sémantickou analýzu. Jeho výstupem bude vnitřní forma programu ve tvaru AST. Proto bude navržena i hierarchie tříd popisující uzly AST. AST se navazujícím modulem programu převede na vnitřní reprezentaci programu pomocí orientovaného grafu, který je popsán v Kapitole 2.4.1. Místo do mezikódu se tedy reprezentace programu pomocí AST převede do reprezentace pomocí grafu.

2.5 Analýza

V předchozí kapitole byla popsána vnitřní paměťová reprezentace vstupního programu pomocí orientovaného grafu a byl uveden postup, jak se k této reprezentaci dostat pomocí přední části překladače programovacího jazyka. V této kapitole bude vysvětleno, jakým způsobem se provede celá analýza nad grafovou reprezentací.

Na program, který je vstupem analýzy jsou kladeny dva důležité požadavky:

1. Počet proměnných musí být znám před začátkem výpočtu, proto program používá deklarační část pro deklarování všech proměnných. Tento požadavek je důležitý toho důvodu, že výpočty v analýze se musí provádět nad mnohostěny nacházející se ve stejném *n*-rozměrném prostoru.
2. Program používá jeden datový typ proměnných. Je to kvůli tomu, že výpočty v analýze se musí dělat nad stejným typem dat.

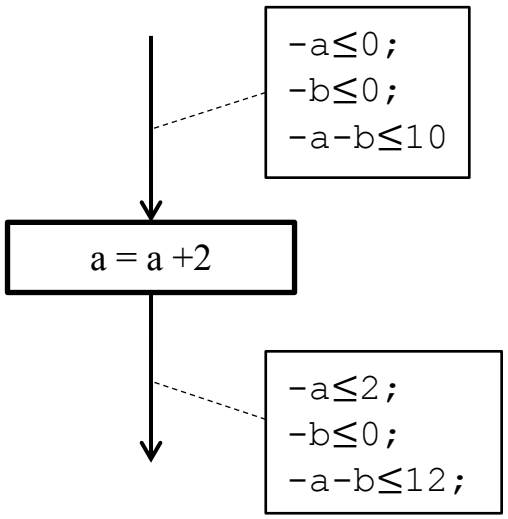
2.5.1 Transformace v uzlech grafu

Transformace v uzlu je výpočet převádějící lineární omezení ze vstupní hrany (hran) uzlu na lineární omezení, která se přiřadí výstupní hraně (hranám) uzlu na základě obsahu tohoto uzlu. Na Obrázku 6 je příklad této transformace pro uzel přiřazení. Na tomto příkladu lze vidět, jak se výstupní omezení změnila v závislosti na přiřazení $a = a + 2$.

Transformace se provede jen tehdy, pokud mnohostěn na vstupní hraně popř. mnohostěny na všech vstupních hranách nejsou prázdné.

Transformace v uzlech grafu tedy tvoří základ provedení celé analýzy. Pro každý typ uzlu je v [1] popsán postup, který je nutno implementovat.

Zde je nutné připomenout, že hrany grafu obsahují obě reprezentace mnohostěnu, tedy vrcholovou a pomocí lineárních omezení. Vrcholová reprezentace je pouze pomocná. Je nutné ji udržovat kvůli efektivitě některých transformací. Některé transformace se rychleji a efektivněji provedou pomocí vrcholové reprezentace a jiné pomocí reprezentace lineárními omezeními. Nyní bude následovat vysvětlení principů transformací ve všech typech uzlů. Matematické základy pro tyto transformace jsou uvedeny v [1] v Kapitole 4.



Obrázek 6: Uzel přiřazení

2.5.1.1 Startovní uzel

Transformace ve startovním uzlu má za úkol vytvořit mnohostěn, který bude asociovaný s výstupní hranou tohoto uzlu.

Pokud není žádná proměnná v deklarační části inicializovaná, tento mnohostěn bude pokrývat celý *n*-rozměrný prostor. Nebude obsahovat žádná lineární omezení a jeho frame bude obsahovat bod z počátku souřadnicové soustavy a množinu přímek, jejíž počet je dán počtem proměnných. Každý vektor přímky bude mít hodnotu jedna na místě reprezentující danou proměnnou a ostatní prvky *n*-tice vektoru budou nulové. Z definice frame v odstavci 2.3.1 vyplývá, že každá přímka určuje dva směry na ose soustavy souřadnic, ve kterých se nachází body popisovaného mnohostěnu. Jinak řečeno, tato počáteční reprezentace frame znamená, že neinicializované proměnné můžou nabývat jakékoli hodnoty na začátku výpočtu. Používá-li program dvě proměnné *a* a *b*, které nejsou inicializovány, bude platit:

- lineární omezení: $\emptyset$
- frame:
  - množina bodů $\{(0,0)\}$
  - množina přímek $\{(1,0);(0,1)\}$

Pokud bude některá proměnná inicializována, soustava lineárních omezení bude obsahovat rovnici přiřazující inicializační hodnotu do dané proměnné. V objektu frame se hodnota dosadí do vektoru bodu na místo určené inicializovanou proměnnou a množina přímek nebude obsahovat přímku obsahující jedničku na místě inicializované proměnné. Pokud tedy bude při deklaraci inicializována proměnná *a* na hodnotu 5 bude reprezentace následující:

- lineární omezení: $a = 5$
- frame:
  - množina bodů $\{(5,0)\}$
  - množina přímek $\{(0,1)\}$

Proměnná b tedy není inicializována, což by mělo být zřejmé z existence přímky, která má na místě proměnné b hodnotu jedna. Proměnná a nabývá pouze hodnoty pět, ale proměnná b může mít jakoukoli hodnotu.

2.5.1.2 Koncový uzel

Je jediný uzel, pro který není definována žádná transformace, protože neobsahuje žádnou výstupní hranu, které by se přiřadil vypočtený mnohostěn. Na jeho vstupní hraně lze najít lineární vztahy platné na konci výpočtu v dané cestě grafem.

2.5.1.3 Uzel přiřazení

Přiřazení mohou být lineární a nelineární. Lineární přiřazení se dále dělí na invertibilní a neinvertibilní. Přiřazení je lineární právě tehdy lze-li jej zapsat ve tvaru:

$$x_{i_0} = \sum_{i=1}^{n} (a_i \cdot x_i) + b$$

Kde:

- x_{i_0} je proměnná na levé straně přiřazení
- $a_i \in \boldsymbol{F}^n$ je vektor koeficientů
- x_i jsou proměnné
- $b \in \boldsymbol{F}^n$ je koeficient nezávislý na žádné proměnné – konstanta

Invertibilní lineární přiřazení obsahuje proměnnou z levé strany přiřazení také ve výrazu na pravé straně přiřazení (příslušný koeficient a_i není nula). Neinvertibilní přiřazení naopak neobsahuje proměnnou z levé strany přiřazení ve výrazu na pravé straně. Přiřazení je nelineární právě tehdy, když není lineární. Přiřazení navíc nemá žádné vedlejší účinky. Mění pouze hodnotu proměnné na levé straně přiřazení.

Prvním typem přiřazení, se kterým je možné se setkat, je přiřazení nelineární. Jelikož se tato práce zabývá pouze lineárními vztahy, bude mít nelineární přiřazení zásadní vliv na hodnotu proměnné na levé straně. Dojde totiž ke ztrátě informace o hodnotách této proměnné a jejích vztazích k ostatním proměnným. Např. obsahem uzlu je nelineární přiřazení $a = a * b$. Soustava lineárních omezení na vstupní hraně uzlu je:

$$a \leq 10; -a \leq 0; a + b \leq 1$$

Potom po transformaci v uzlu se jeho výstupní hraně přiřadí pouze lineární omezení $b \leq 1$. Proměnná a tedy může nabývat od této chvíle jakékoli hodnoty a nemusí se tedy objevit v soustavě lineárních omezení. Princip transformace spočívá v eliminaci proměnné a ze soustavy. Po transformaci se do objektu frame ze vstupní hrany přidá přímka, která má hodnotu jedna na místě indexu eliminované proměnné, jinak hodnoty nula. Touto přímkou se rozumí stejně jako v případě neinicializované proměnné ve startovním uzlu, že hodnota proměnné může nabývat jakékoli hodnoty. Tento frame se pak zjednoduší a přiřadí se výstupní hraně uzlu. Pokud jsou indexy proměnných v příkladu výše $a = 0; b = 1$ bude mít nová přímka tvar $(1,0)$.

Lineární omezení a frame se v případě lineárního přiřazení vypočítají odděleně.

Pokud je přiřazení lineární. Prvky frame (body, polopřímky a přímky) ze vstupní hrany se postupně dosadí do přiřazení a takto se z nich vypočítají nové prvky frame, které se uloží do výstupní hrany.

Tímto se budou měnit pouze hodnoty proměnné na levé straně přiřazení. Je-li $a = b + 1$ přiřazení a (1,2) je bod frame na vstupní hraně potom nově vypočtený bod bude (3,2).

Přiřazení je neinvertibilní tehdy, když se proměnná z levé strany přiřazení nenachází ve výrazu na právě straně přiřazení (např. $a = 2 * b + 1$). Tento typ přiřazení je charakteristický tím, že do proměnné na levé straně přiřazuje zcela novou hodnotu. Z tohoto důvodu se musí ze vstupních lineárních omezení eliminovat proměnná z levé strany přiřazení, čímž se ztratí podobně jako u nelineárního přiřazení informace o aktuálním oboru hodnot této proměnné, ale ihned se zase získá tím, že se celé přiřazení přidá do výstupní soustavy omezení jako nová rovnice. Pokud je např. přiřazení v uzlu $a = 5 * b + 8$ a soustava lineárních omezení je

$$a \leq 10; -a \leq 0; a + b \leq 1$$

Potom po eliminaci proměnné a vznikne pouze jedna nerovnice $b \leq 1$, ke které se přidá přiřazení v uzlu. Tedy lineární omezení, která budou reprezentovat mnohostěn ve výstupní hraně uzlu jsou:

$$b \leq 1; a - 5 * b = 8$$

Přiřazení je invertibilní, když se proměnná z levé strany přiřazení nachází i ve výrazu na pravé straně přiřazení (např. $a = a + 2$). Znamená to, že do proměnné na levé straně se přiřazuje hodnota závislá na předchozí hodnotě této proměnné. V tomto případě se z pravé strany přiřazení vyjádří tato proměnná a dosadí se do celé soustavy lineárních omezení ze vstupní hrany.

Výsledná lineární omezení se pak uloží do výstupní hrany uzlu.

2.5.1.4 Testovací uzel

Pokud je testovací podmínka nelineární, zkopíruje se mnohostěn ze vstupní hrany na obě výstupní hrany. Stejně jako v uzlu přiřazení i zde se neberou nelineární podmínky v potaz. Algoritmy transformace z [1] totiž nedovolují ověřit pravdivost nelineární podmínky.

Na lineární podmínku jakkoli složitě zapsanou se lze dívat dvěma způsoby. Jako na podmínku typu rovnost nebo nerovnost.

Podmínku typu rovnost si lze představit jako hyperrovinu v prostoru. Pokud tato hyperrovina protíná mnohostěn ze vstupní hrany uzlu, přiřadí se do výstupní hrany typu *pravda* všechny body mnohostěnu, které protíná tato hyperrovina. Do výstupní hrany typu *nepravda* se vloží celý vstupní mnohostěn, tedy i s body v hyperrovině, což plyne z požadavku, aby výsledný mnohostěn byl vždy uzavřený konvexní objekt. Tento požadavek by tedy byl porušen, pokud by ve výstupní hraně typu *nepravda* byl mnohostěn neobsahující tu svou část, kterou protíná hyperrovina podmínky.

Pokud hyperrovina mnohostěn neprotíná, bude mnohostěn ve výstupní hraně typu pravda prázdný a výstupní hraně typu nepravda bude přiřazen celý vstupní mnohostěn.

Pokud mnohostěn celý leží v hyperrovině, bude mnohostěn ve výstupní hraně typu nepravda prázdný a výstupní hraně typu pravda bude přiřazen celý vstupní mnohostěn.

Podmínka typu nerovnost představuje opět hyperrovinu. Ovšem v tomto případě hyperrovina dělí prostor na dva poloprostory. V jednom poloprostoru má nerovnice podmínky řešení a ve druhém nemá (viz definice soustavy lineárních nerovnic). Tedy pokud hyperrovina protíná vstupní mnohostěn, půlí ho na dvě části, z nichž jedna (v poloprostoru, kde má nerovnice řešení) bude přiřazena na výstupní

hranu typu *pravda* a druhá na výstupní hranu typu *nepravda*. Část mnohostěnu ležící v hyperrovině bude součástí obou výstupních mnohostěnů.

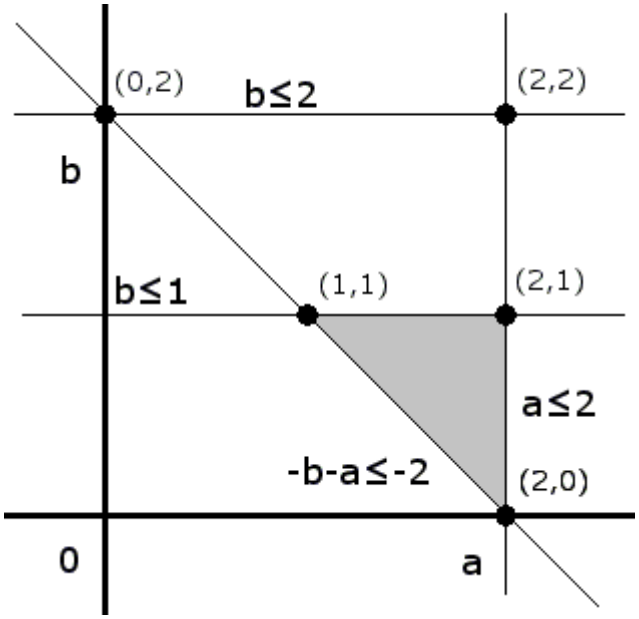
Důležité je zde poznamenat, že každá podmínka typu nerovnost musí být převedena na neostrou nerovnost typu $\leq$. Je to z toho důvodu, že pro všechny výpočty v programu se používá pouze tohoto typu nerovnosti (lineární omezení typu nerovnost jsou reprezentovány pouze tímto typem nerovnosti). Toto zjednodušení je následek požadavku, aby výsledný mnohostěn byl vždy uzavřený konvexní objekt.

Příklad je uveden na Obrázku 7. Testovací podmínka je ve tvaru $b \leq 1$ je uvedena ve Výpisu 3.Tedy:

Výpis 3: Příklad testovací podmínky

```
if(b ≤ 1){…}
else {…}
```

Dle Obrázku 7 se mnohostěn nachází ve dvourozměrném prostoru. Hyperrovina je přímka popsaná rovnicí $b = 1$. Ovšem podmínka je nerovnost, proto poloprostor řešení nerovnice je od této přímky směrem dolů. Tato přímka navíc rozděluje mnohostěn na dvě oblasti. Oblast mnohostěnu nacházející se v poloprostoru řešení nerovnice (šedá oblast) bude přiřazena do výstupní hrany typu *pravda* testovacího uzlu a zbylá oblast bude přiřazena do výstupní hrany typu *nepravda*.



Obrázek 7: Transformace v testovacím uzlu

Reprezentace pomocí frame bude následující:

- Výstupní hrana typu *pravda*
  - Množina bodů $\{(1,1); (2,1); (2,0)\}$
- Výstupní hrana typu *nepravda*
  - Množina bodů $\{(0,2); (2,2); (2,1); (1,1)\}$

Dále platí, že když nebude hyperrovina popsaná nerovnicí podmínky protínat mnohostěn, bude výsledek přiřazený oběma výstupním hranám záležet na tom, jestli se mnohostěn nachází

v poloprostoru řešení nerovnice nebo ne. Pokud ano, mnohostěn ze vstupu bude celý přiřazen na výstup typu *pravda*, výstup *nepravda* bude prázdný a pokud ne tak naopak.

Nakonec je nutno převést výsledné reprezentace pomocí frame na reprezentace lineárními omezeními.

2.5.1.5 Slučovací uzel

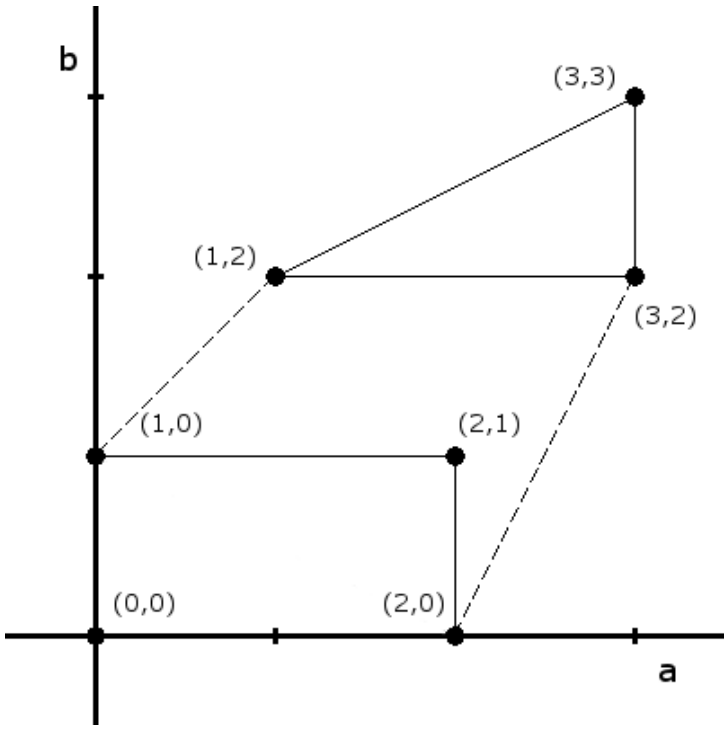
Tento uzel slučuje více vstupních toků do jednoho výstupního toku. Jinak řečeno, musí sloučit dva a více mnohostěnů ze vstupních hran do jednoho mnohostěnu ve výstupní hraně. Jelikož jsou mnohostěny objekty v *n*-rozměrném prostoru, sloučení představuje vytvoření konvexního obalu těchto mnohostěnů. Konvexní obal mnohostěnů se nejlépe provede, sjednocením množin vrcholů ze všech vstupních hran grafu což je jednoduchá a rychlá množinová operace. Výsledek se pak převede na reprezentaci pomocí lineárních omezení.

Sloučení dvou mnohostěnů ilustruje Obrázek 8. Vstupním hranám jsou přiřazeny mnohostěny, jejichž vrcholové reprezentace jsou:

- množina bodů prvního mnohostěnu: $\{(0,0); (2,0); (2,1); (1,0)\}$
- množina bodů druhého mnohostěnu: $\{(1,2); (3,2); (3,3)\}$

Přitom na pořadí mnohostěnů nezáleží. Výsledný frame mnohostěnu vznikne sjednocením množin vrcholů obou vstupních mnohostěnů. V tomto případě se budou sjednocovat pouze množiny bodů. Sjednocení je naznačeno čárkovanou čarou. Platí tedy:

- množina bodů: $\{(0,0); (2,0); (2,1); (1,0); (1,2); (3,2); (3,3)\}$



Obrázek 8: Transformace ve slučovacím uzlu

Ovšem bod (2,1) prvního mnohostěnu leží vevnitř nového mnohostěnu. Je tedy zbytečný a je nutné jej eliminovat a nakonec převést reprezentaci frame na lineární omezení. Výstupní frame tedy bude tvořen množinou bodů: $\{(0,0); (2,0); (1,0); (1,2); (3,2); (3,3)\}$.

Jak lze na Obrázku 8 dále vidět, výsledný mnohostěn obsahuje i oblasti, které nebyly součástí vstupních mnohostěnů. Tímto způsobem se tedy do výpočtu zanáší určitá nepřesnost, jež je ovšem nezbytná a plynoucí z požadavku, aby mnohostěn byl vždy uzavřený konvexní objekt.

2.5.1.6 Uzel smyčky

Uzel smyčky je speciální typ slučovacího uzlu. Má více vstupních hran a jednu hranu výstupní. Přitom se v grafové reprezentaci může vyskytovat pouze v cyklech grafu, které reprezentují smyčky ve vstupním programu.

Předpoklad pro následující úvahu je alespoň jeden uzel přiřazení v cyklu programu, které v každé iteraci mění hodnotu proměnné na levé straně přiřazení na hodnotu jinou, než byla v předchozí iteraci. Pokud by nebyl definován uzel smyčky, byl by na jeho místě uzel slučovací. Jelikož by se mnohostěn na vstupní hraně slučovacího uzlu stále měnil (díky onomu přiřazení) byl by i výstupní mnohostěn vypočítaný transformací v každé iteraci jiný. Pokud by navíc byla smyčka nekonečná, nikdy by nedošlo ke stabilizaci transformace. Analýza lineárních vztahů by se tedy zacyklila stejně jako vstupní program po spuštění. Uzel smyčky má zabránit tomuto chování. Navíc má stabilizovat transformaci v uzlu jak nejrychleji to je možné.

Výpočet transformace v tomto uzlu využívá aplikaci tzv. operátoru *widening* ∇ z teorie abstraktní interpretace. Více o operátoru *widening* lze nalézt v [3] v Kapitole 7. Teoreticky dokáže tento operátor nalézt aproximaci horní hranice striktně rostoucí posloupnosti a tu vydat jako výsledek, se kterým se dá následně dále pracovat. Jako příklad lze uvést podprogram ve Výpisu 4.

Výpis 4: Podprogram pro operaci *widening*

```
x = 1; y = 5;
while(true){
      x = x + 1;
      y = y - 2;
}
```

Jakmile výpočet operátoru *widening* zjistí, že hodnoty proměnných x a y jsou striktně rostoucí (proměnná y je záporně rostoucí), vypočítá následující intervaly pro obě proměnné:

$$x \in [1, \infty]; y \in [-\infty, 5]$$

Přitom vyhodnocení operátoru *widening* proběhne v konečném počtu kroků i za předpokladu, že smyčka programu je nekonečná.

Toto byla pouze teorie. V praxi je nutné výpočet operátoru *widening* přizpůsobit situaci a vhodně jej navrhnout a implementovat. To se také děje v této práci.

Transformace v tomto uzlu nejprve provede sloučení mnohostěnů obou vstupních hran jako v případě slučovacího uzlu. Tento nový mnohostěn představuje aktuální stav smyčky. Na výstupní hraně uzlu smyčky je pak uložen mnohostěn představující stav smyčky z minulé iterace.

Na základě znalostí aktuálního a předchozího mnohostěnu se provede odstranění zbytečných lineárních omezení, jež se mění v každé iteraci smyčky a zůstanou tak omezení obecně pro smyčku platná v každé iteraci. Odstranění zbytečných omezení se provede tak, že se do nerovnic předchozího mnohostěnu dosadí body z frame aktuálního mnohostěnu. Každé omezení, které je po dosazení všech bodů frame platné bude součástí výstupních omezení. Výsledná omezení se převedou na frame a celý mnohostěn se uloží do výstupní hrany uzlu.

Celou situaci si lze představit na následujícím příkladu, kdy vstupy jsou:

- lineární omezení z přechozí iterace $b \geq 0; a \leq 1; a + b = 4$ přiřazená výstupní hraně uzlu
- množina bodů z objektu frame vypočtená transformací v aktuální iteraci $\{(1,3); (2,2); (3,1)\}$

Po dosazení bodů do rovnic platí, že pro všechny body není platné pouze omezení $a \leq 1$, které je platné pouze pro bod (1,3). Výstupní omezení pro tento uzel tedy budou $b \geq 0; a + b = 4$.

Zde je nutno poznamenat, že tato implementace použití operátoru *widening* pro mnohostěny, jak je popsána v [1] v Kapitole 4.5 je jedna z jeho prvních implementací. Jelikož spočívá pouze v odstranění zbytečných lineárních omezení nelze očekávat, že ve všech případech vydá přesnou aproximaci výsledků. Výsledek záleží totiž pouze na tom, jaká lineární omezení přišla do uzlu smyčky zevnitř cyklu programu. Pro složitější cykly často vydává *over*-aproximaci hodnot proměnných. Složitější cyklus obsahuje více vnořených cyklů, více podmínek a různé typy přiřazení.

Novější implementace operátoru *widening* používají pro přesná vyhodnocení smyček složité heuristiky, čímž se zlepšuje odhad výsledných lineárních omezení pro smyčky. Více o porovnání různých implementací operátorů *widening* je možné najít v [6].

I v této implementaci lze ovšem určitým způsobem zlepšit odhad výsledných lineárních omezení pro smyčku. Obvyklé je aplikovat operátor *widening* až tehdy když je objevena celá smyčka tzn., všechny vstupní hrany obsahují mnohostěny. Je to logické, protože kdyby nebyl mnohostěn na hraně vycházející zevnitř smyčky, nemělo by smysl operátor aplikovat. Ovšem lze udělat následující optimalizaci. Potom co je objevena celá smyčka, počká se s aplikací operátoru ještě několik iterací, ve kterých se bude provádět transformace jako ve slučovacím uzlu. Tím bude zajištěno získání více lineárních omezení (více informací) zevnitř smyčky a na ty se později aplikuje operátor *widening*. Doba čekání, není přesně dána. Záleží na složitosti smyčky. Proto v této práci bude ponecháno rozhodnutí o počtu čekacích iterací uživateli.

2.5.2 Provedení analýzy

V předchozím odstavci byl popsán jednotkový krok celé analýzy, tedy transformace. S touto znalostí lze již popsat postup celého jejího provedení.

Analýza lineárních vztahů mezi proměnnými v programu spočívá v provádění transformací ve všech uzlech grafu v iteracích. Každá iterace začíná vždy ve startovním uzlu, ze kterého se grafem po orientovaných hranách postupuje do všech koncových uzlů. Tyto transformace se provádějí tak dlouho, dokud nedojde k jejich stabilizaci.

Stabilizace transformací neboli stabilizace analýzy je stav, kdy transformace v každém uzlu v aktuální iteraci vypočetla stejná lineární omezení jako transformace v předchozí iteraci. Jinak řečeno, transformace nevypočítaly žádná nová lineární omezení a neučinily by tak ani v další iteraci. Analýzu je tedy možno ukončit a prezentovat výsledky – zobrazit lineární omezení, která jsou asociována se všemi hranami grafu.

3 Analýza a návrh

V Kapitole 2 byly popsány základní funkce programu pro automatické nalezení lineárních vztahů mezi proměnnými v programu. Celá Kapitola 2 je tedy souhrn požadavků na výsledný program. Tato kapitola popisuje analýzu a návrh programu dle těchto požadavků.

První pohled na program je z hlediska jeho činnosti. Následující tři body charakterizují činnost programu:

1. načtení vstupního programu do vnitřní reprezentace v paměti pomocí grafu
2. nalezení lineárních vztahů mezi proměnnými v programu. Provádí se nad grafovou reprezentací
3. zobrazení lineárních vztahů mezi proměnnými v programu v každém jeho místě

Analýza vstupního programu je poměrně časově a paměťově náročný proces. Vliv na paměťovou a časovou náročnost má velikost vstupního programu a počet použitých proměnných. Je tedy nutné na tyto omezení pamatovat hned při návrhu.

Časovou náročnost ovlivňuje

- Velikost vstupního programu. Čím větší je orientovaný graf vytvořený ze vstupního programu tím déle trvá ho projít ze startovního uzlu do všech koncových uzlů.
- Čekání na ustálení analýzy. Graf se bude procházet tak dlouho, dokud se jednotlivé mnohostěny v jeho hranách nebudou měnit. Nejdéle se čeká na ustálení mnohostěny ve smyčkách.
- Počet proměnných vstupního programu udává rozměr (dimenzi) prostoru výpočtů. Tedy čím více proměnných tím déle budou trvat jednotlivé transformace.

Paměťovou náročnost ovlivňuje

- Velikost vstupního programu. Čím více řádků kódu tím více bytů zabírá orientovaný graf v paměti.
- Počet proměnných vstupního programu.

Časová a paměťová náročnost jsou důvody pro zvolení jazyka C++, který programátorovi umožňuje provádět efektivnější správu paměti a jemnější řízení kódu.

3.1 Reprezentace čísel

Vstupní program používá pouze jeden typ čísel pro výpočty a žádné jiné datové typy. Přitom není možné používat vestavěné reálné datové typy jazyka C++ jako jsou `double` a `float`, protože jsou omezeny svou velikostí v paměti a tím i svou přesností ve výpočtech. Matematické algoritmy jsou obecně náchylné právě na ztrátu přesnosti výpočtů. Proto byla pro reprezentaci čísel vybrána externí knihovna GMP – *The GNU Multiple Precision Arthmetic Library,* jež je psána v jazyku C a obsahuje objektovou nadstavbu pro jazyk C++.

GMP je knihovna poskytující datové celočíselné, reálné a racionální typy pracující v libovolné přesnosti. Znamená to, že velikost uloženého čísla zde závisí pouze na velikosti dostupné paměti a výpočty jsou vždy přesné.

Pro účely této práce byla vybrána racionální čísla, tedy čísla ve tvaru zlomku, kde čitatel i jmenovatel jsou celé číslo. Třída poskytující tento typ se v GMP nazývá `mpq_class`.

3.2 Mnohostěn

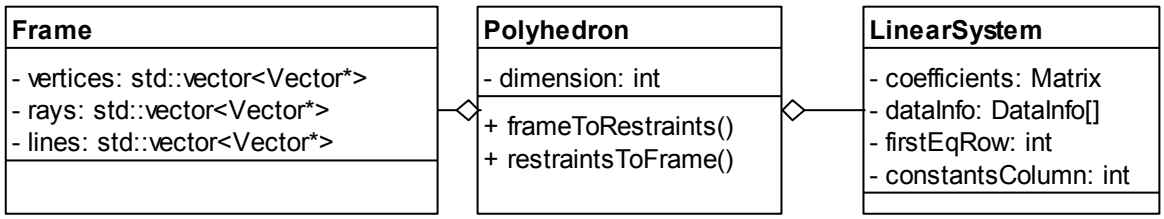
Mnohostěn je objekt, který bude přiřazen každé hraně orientovaného grafu. V Kapitole 2 bylo uvedeno, že je reprezentován pomocí lineárních omezení a objektu frame. Mnohostěn je definován ve třídě `Polyhedron`. Ta obsahuje informaci o rozměru prostoru, ve kterém se mnohostěn nachází a další dvě vlastnosti jsou třídy:

- `LinearSystem` – reprezentace lineárních omezení
- `Frame` – vrcholová reprezentace

Nakonec třída osahuje dvě metody pro převod mezi oběma reprezentacemi:

- `restratitsToFrame()` – převede lineární omezení na objekt frame
- `frameToRestraints()` – převede objekt frame na lineární omezení

Třídní diagram objektu `Polyhedron` je uveden na Obrázku 9.



Obrázek 9: Diagram tříd pro polyhedron

3.2.1 Lineární omezení

Lineární omezení jsou reprezentována třídou `LinearSystem`. Jde o soustavu lineárních rovnic a nerovnic. Nejvhodnější reprezentace takovéto smíšené soustavy je pomocí matice, kde prvních n řádků jsou nerovnice typu $\leq$ a ostatní řádky až do konce jsou rovnice. Tento přístup tedy vyžaduje použití proměnné indikující první rovnici v soustavě. Její jméno je `firstEqRow` Samotná matice je reprezentována třídou `Matrix` a jméno proměnné je `coefficients`.

Každý sloupec matice představuje jednu proměnnou ze vstupního programu. Další strukturou ve třídě `LinearSystem` je tedy pole popisující sloupce matice `DataInfo`, ve kterém jsou uloženy identifikace proměnných (každá proměnná má přiřazen unikátní identifikátor a jméno, které se mapují na sloupec matice). Každá soustava má také sloupec pravých stran jehož poloha v matici je uložena v proměnné `constantsColumn`. Např. soustava:

$$2a \leq 1; 5b \leq 12; a + b = 6$$

Bude reprezentována takto:

- `coefficients` $= \begin{bmatrix} 2 & 0 & 1 \\ 0 & 5 & 12 \\ 1 & 1 & 6 \end{bmatrix}$
- `firstEqRow = 2`

- `constantsColumn = 2`
- první sloupec je proměnná *a*, druhý sloupec je proměnná *b*. Tato informace je uložena ve struktuře `dataInfo`

Třída `Matrix` představuje matici koeficientů `coefficients` ve třídě `LinearSystem`. Data jsou uložena ve dvourozměrném poli dat typu `mpq_class`, která je poskytována knihovnou GMP. Třída `Matrix` je dále doplněna o funkce pro manipulaci s maticemi (odstranění/přidání řádku nebo sloupce, násobení vektorem, atd.)

3.2.1.1 Eliminace proměnné ze soustavy lineárních omezení

Eliminace proměnné ze soustavy je důležitá operace používaná v této práci. Používá se v algoritmech převodů mezi reprezentacemi mnohostěnu a v transformaci v uzlu přiřazení. Princip eliminace spočívá v přepočítání soustavy lineárních omezení tak, aby na konci vznikla soustava omezení, která má nuly ve sloupci proměnné určené k eliminaci. Potom je možné tento sloupec ze soustavy bez obav odstranit, protože neexistují vztahy mezi touto proměnnou a ostatními proměnnými v soustavě. Metoda popsaná dále předpokládá, že soustava lineárních omezení je tvořena pouze nerovnicemi. Před začátkem výpočtu je nutné případné rovnice v soustavě převést na nerovnice.

Vstupní soustava je popsána vztahem $Ax \leq B$. Sloupec, který se má eliminovat je označen c_0. Výstupní soustava s eliminovaným sloupcem c_0 bude pak ve tvaru $Dx \leq E$. Přitom jedna nerovnice obou soustav je popsána vztahy $ax \leq b$ resp. $dx \leq e$, kde:

- *a* a *d* jsou vektory představující jeden řádek matice *A* resp. *D*
- *b* a *e* jsou pravé strany nerovnic
- *x* je vektor proměnných

Postup eliminace je pak následující:

- každá nerovnice $ax \leq b$ kde $a_{c_0} = 0$ je součástí nové soustavy nerovnic. Stane se tedy nerovnicí $dx \leq e$. Tato rovnice již obsahuje nulu v eliminovaném sloupci
- každé dvě nerovnice $a_1 x \leq b_1$ a $a_2 x \leq b_2$, pro které platí $a_{1,c_0} \cdot a_{2,c_0} < 0$ budou přepočítány na jednu nerovnici $dx \leq e$ dle vztahu:

$$|a_{1,c_0}| \cdot a_2 + |a_{2,c_0}| \cdot a_1 \leq |a_{1,c_0}| \cdot b_2 + |a_{2,c_0}| \cdot b_1$$

  tedy vektor *d* a číslo *e* jsou dány vztahy:

$$d = |a_{1,c_0}| \cdot a_2 + |a_{2,c_0}| \cdot a_1$$

$$e = |a_{1,c_0}| \cdot b_2 + |a_{2,c_0}| \cdot b_1$$

Postupy definovanými v obou bodech výše byla vytvořena soustava lineárních omezení $Dx \leq E$ obsahující nuly ve sloupci c_0. Tento sloupec může být nyní odstraněn ze soustavy.

Této metodě eliminace proměnné ze soustavy nerovnic se říká projekce podle sloupce c_0. Je implementována v metodě `LinearSystem::project(int c0)`.

Jako příklad, nechť je dána následující soustava nerovnic:

$$a + b \leq 1; 2a + c \leq 3; 2b - 3c \leq 2$$

Požadavek je eliminace proměnné c, která je ve třetím sloupci soustavy. První nerovnice neobsahuje proměnnou c. V maticové reprezentaci soustavy je hodnota nula na jejím místě. První nerovnice bude tedy součástí výsledku. Potom zbývají dvě poslední nerovnice, kde pro koeficienty proměnné c platí $1 * (-3) < 0$. Z těchto dvou nerovnic tedy vznikne jedna nová výpočtem (proměnné d a e jsou použity z definice výše):

- levá strana nerovnice:

$$d = |a_{1,c_0}| \cdot a_2 + |a_{2,c_0}| \cdot a_1 = |1| \cdot (0,2,3) + |-3| \cdot (2,0,1) = (6,2,0)$$

- pravá strana nerovnice:

$$e = |a_{1,c_0}| \cdot b_2 + |a_{2,c_0}| \cdot b_1 = |1| \cdot 2 + |-3| \cdot 3 = 11$$

Výsledná soustava lineárních omezení bude:

$$a + b \leq 1; 6a + 2b \leq 11$$

Proměnná c se již v soustavě nevyskytuje a v její maticové reprezentaci jsou ve třetím sloupci samé nuly. Algoritmem pro úpravu matice lze tento nulový sloupec odstranit.

Pseudokód algoritmu eliminace se nachází v Příloze I pod názvem Pseudokód LS-ELIMIMINATE.

3.2.2 Frame

Vrcholová reprezentace mnohostěnu je definována ve třídě `Frame`. Frame obsahuje tři množiny – množinu bodů (`vertices`), polopřímek (`rays`) a přímek (`lines`). Každý prvek z každé množiny je vektor. Proto byla vytvořena třída `Vector` pomocí které se uchovávají prvky ve všech množinách. `Vector` své data uchovává v jednorozměrném poli typu `mpq_class`.

Samotné množiny prvků jsou reprezentovány parametrizovaným datovým typem `std::vector<Vector>` ze standardní knihovny C++. Dále třída `Frame` obsahuje metody pro manipulaci s prvky jednotlivých množin, např. sjednocení dvou množin bodů.

3.2.3 Převody mezi reprezentacemi mnohostěnu

Přítomnost převodů mezi reprezentacemi mnohostěnu je nutná z toho důvodu, že některé algoritmy v programu dokáží vypočítat pouze jednu reprezentaci. Druhá reprezentace se z ní pak získá převodem.

Převod frame na lineární omezení je implementován ve třídě `FrameToRestraints`.

Převod lineárních omezení na frame je implementován ve třídě `RestraintsToFrane`.

3.2.4 Zjednodušení reprezentací mnohostěnu

Algoritmy převodů mezi reprezentacemi mnohostěnů vytvářejí lineární omezení nebo frame, které nejsou minimální. Jednotlivé reprezentace je nutné udržovat minimální hlavně kvůli složitosti výpočtů nad jednotlivými reprezentacemi. Platí zde, že čím více je v soustavě omezení nebo čím více má frame prvků, tím jsou výpočty časově náročnější.

V popisu algoritmů se dále objevuje pojem saturace neostré nerovnice prvkem frame. Prvek frame (bod, polopřímka nebo přímka) saturuje neostrou nerovnici tehdy, pokud po jeho dosazení si budou její pravá a levá strana rovny. Např. je dána nerovnice $2 \cdot a + 3 \cdot b \leq 5$. Bod $(1,1)$ tuto nerovnici saturuje, zatímco bod $(0,1)$ ne. Ověřuje-li se saturace nerovnice polopřímkou a přímkou je nutno

nebrat v úvahu konstantní člen nerovnice. Nerovnice se tedy upraví do tvaru $2 \cdot a + 3 \cdot b \leq 0$ a dosadí se do ní polopřímky a přímky. Např. polopřímka (-3,2) saturuje nerovnici, zatímco polopřímka (1,0) ji nesaturuje.

3.2.4.1 Zjednodušení soustavy lineárních omezení

Implementace je provedena ve třídě `FtRSimplification`.

Zjednodušení se provede pomocí objektu frame. Postupuje se v následujících krocích:

1. Pro každou nerovnici soustavy se zjistí jakými body a polopřímkami je saturována. Informace o saturacích se uloží do prvků pole, typu `ItemSaturations`, jehož velikost `inequalitiesCount` je počet nerovnic.

   structure ItemSaturations {
   Integer rowId;
   BitSet saturations;
   };

   ItemSaturations vertexSaturations[inequalitiesCount];
   ItemSaturations raySaturations[inequalitiesCount];

   Přitom typ `BitSet` představuje bitové pole, kde každý bit odpovídá bodu nebo polopřímce. Hodnota bitu 0 říká, že bod nebo polopřímka nesaturuje nerovnici danou proměnnou `rowId`. Hodnota 1 znamená opak.

2. Nerovnice, která není nikdy saturována žádným bodem, je nepotřebná a může se z lineárních omezení odstranit. Pozná se to tak, že všechny prvky pole `vertexSaturations` mají v proměnné `saturations` všechny hodnoty bitů nastaveny na 0. Pseudokód této funkce je uveden v Příloze I pod názvem Pseudokód SIMPLIFY-RESTRAINTS-IRRELEVANT-1.
3. Nerovnost, která je saturována všemi body a polopřímkami představuje rovnici. Pole `vertexSaturations` a `raySaturations` mají v proměných `saturations` všechny bity nastaveny na 1. Pseudokód této funkce je uveden v Příloze I pod názvem Pseudokód SIMPLIFY-RESTRAINTS-EQUALITIES-2.
4. Pro každé dvě nerovnice, které nejsou rovnice (platí i pro rovnice nalezené v bodě 2) platí:
   a. Pokud jsou saturovány stejnými body a polopřímkami, jednu z nich je možno eliminovat.
   b. Pokud je první nerovnice saturována podmnožinou bodů a polopřímek druhé nerovnice, potom je možno první nerovnici eliminovat – jinak řečeno první nerovnice je saturována všemi prvky jako druhá nerovnice. První nerovnici není tedy třeba udržovat.

   Pseudokód této funkce je uveden v Příloze I pod názvem Pseudokód SIMPLIFY-RESTRAINTS-REDUNDANT-3.

Pseudokód zjednodušení lineárních omezení pomocí objektu frame je uveden v Příloze I pod názvem Pseudokód SIMPLIFY-RESTRAINTS-4.

3.2.4.2 Zjednodušení objektu frame

Implementace je provedena ve třídě `RtFSimplification`.

Zjednodušení objektu frame se provede pomocí lineárních omezení. Toto zjednodušení je duální ke zjednodušení, které je popsané v předchozím odstavci. Postupuje se v těchto krocích:

1. Pro každý bod z frame se zjistí, jaké nerovnice saturuje. Pro každou polopřímku se zjistí, jaké nerovnice saturuje a jaké rovnice splňuje. Výsledky se uloží do prvku pole, typu `ItemSaturations`.

   structure ItemSaturations {
   Vector item;
   BitSet saturations;
   };

   ItemSaturations rowSaturationsVerticesIneq[verticesCount];
   ItemSaturations rowSaturationsRaysIneq[raysCount];
   ItemSaturations rowSaturationsRaysAll[raysCount];

   Opět se využije struktury `BitSet`, kde každý bit odpovídá lineární rovnici nebo nerovnici v soustavě lineárních omezení. Hodnota 0 pak znamená, že prvek frame uložený v proměnné `item` nesaturuje danou nerovnici. Hodnota 1 pak znamená opak.

2. Bod nebo polopřímka, které nesaturují žádnou nerovnici, jsou zbytečné. Tedy všechny prvky polí `rowSaturationsVerticesIneq` a `rowSaturationsRaysIneq` mají v proměnné `saturations` všechny hodnoty bitů nastaveny na 0. Pseudokód této funkce je uveden v Příloze I pod názvem Pseudokód SIMPLIFY-FRAME-IRRELEVANT-1.
3. Polopřímka saturující všechna omezení je přímka. Všechny prvky pole `rowSaturationsRaysAll` mají v proměnné `saturations` ve všech bitech nastaveny na 1. Pseudokód této funkce je uveden v Příloze I pod názvem Pseudokód SIMPLIFY-FRAME-LINES-2.
4. Nakonec se najdou všechny zbytečné prvky (body nebo polopřímky), jejichž saturace jsou již obsaženy v jiných prvcích.
   a. Existují-li dva body nebo dvě polopřímky saturující stejné nerovnice je jeden bod nebo jedna polopřímka zbytečná.
   b. Pokud pro některé dva body platí, že jeden bod saturuje podmnožinu nerovnic, které saturuje druhý bod je první bod zbytečný. To samé platí o každých dvou polopřímkách.

   Pseudokód této funkce je uveden v Příloze I pod názvem Pseudokód SIMPLIFY-FRAME-REDUNDANT-3. Je uveden pouze pro ověření dvojic bodů. Pro ověření dvojic polopřímek stačí dosadit místo pole `rowSaturationsVerticesIneq` pole `rowSaturationsRaysIneq`.

Pseudokód zjednodušení frame pomocí lineárních omezení je uveden v Příloze I pod názvem Pseudokód SIMPLIFY-FRAME-4.

3.3 Vstupní jednotka programu

V předchozích odstavcích byly popsány základní datové typy a třídy používající se ve výpočtech programu. Tento odstavec poskytuje pohled na tzv. *vstupní jednotku* programu, která má za úkol načtení vstupního textového souboru s programem do vnitřní reprezentace pomocí orientovaného grafu (objektu flowchart) v paměti počítače.

Funkce vstupní jednotky jsou následující a budou postupně popisovány dále v textu:

1. kontrola syntaxe a sémantiky programu - třídy `LexicalAnalyzer`, `SyntaxAnalyzer`
2. načtení programu do AST - třídy `LexicalAnalyzer`, `ProgramParser`
3. převedení AST do seznamu instrukcí - třída `FlowchartCreator`
4. převedení seznamu instrukcí do grafové reprezentace – třídy `FlowchartCreator`, `Flowchart`

3.3.1 Kontrola syntaxe a sémantiky

Vstupní program je kontrolován v syntaktickém analyzátoru, který je definován ve třídě `SyntaxAnalyzer`, která využívá třídu `LexicalAnalyzer` pro načítání jednotlivých lexémů ze zdrojového kódu programu.

3.3.1.1 Lexikální analyzátor

Lexikální analyzátor pracuje jako konečný automat. Základní operace je přečtení jednoho znaku ze souboru a jeho zpracování v automatu. Podle toho jaký znak byl načten, se změní stav automatu. Pokud je aktuální stav koncový lexikální automat vrátí nadřazené vrstvě:

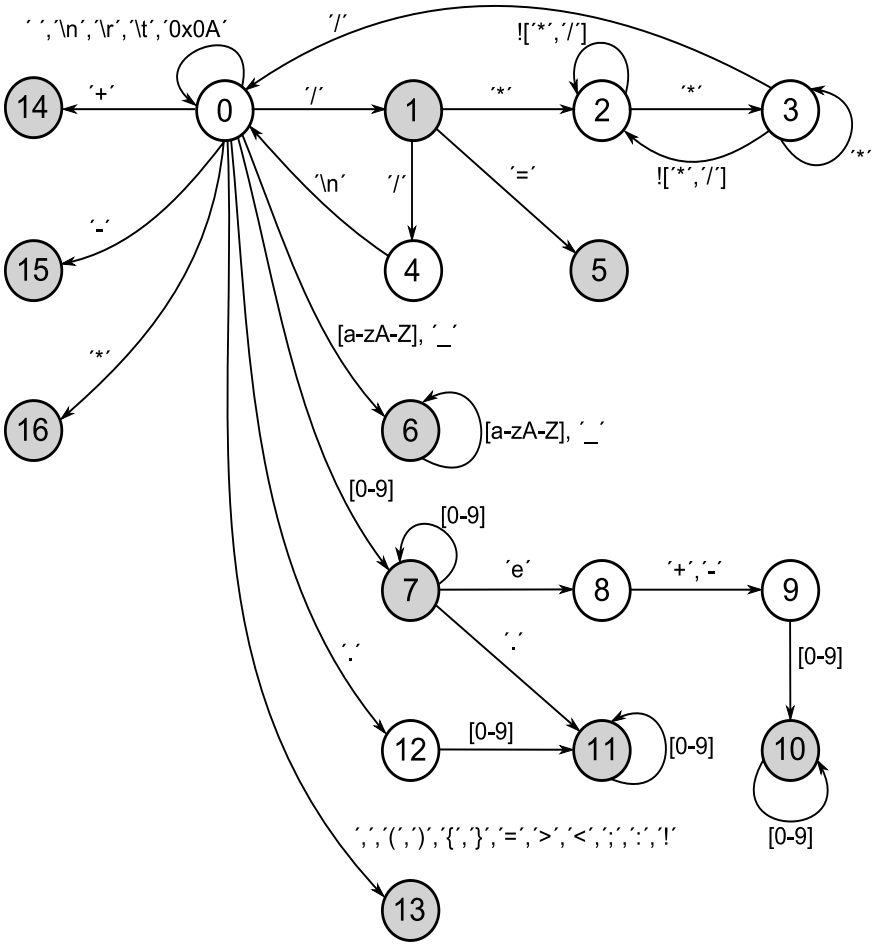
- Informaci typu načteného lexému.
- Pokud bude charakter načtených dat číslo, nebo název proměnné, budou hodnota čísla nebo jméno proměnné poskytnuty lexikálním analyzátorem taktéž.

Jelikož se v práci používá knihovna GMP pro reprezentaci čísel, konkrétně třída `mpq_class`, která obsahuje konstruktor pro vytvoření vnitřní reprezentace čísla z řetězce, bude lexikální analyzátor zjednodušen. Zjednodušení spočívá v tom, že pokud bude načteno číslo ze vstupního souboru, bude toto číslo vráceno jako řetězec. Odpadne tedy přepočítávání vstupního čísla na zabudované typy programovacího jazyka – což by bylo i nepraktické pokud by uživatel zadal ve vstupním programu číslo větší než jakýkoli zabudovaný číselný typ.

Konečný automat, který bude použitý pro lexikální analyzátor, je na Obrázku 10. Stav nula je počáteční a koncové stavy jsou vyplněny šedou barvou. Jednotlivé přechody jsou označeny znaky, pro které se přechody provedou.

Lexikální analyzátor bude obsahovat dvě základní funkce:

- `LexicalElement nextToken()` – načte lexém ze vstupního souboru a vrátí jeho typ v proměnné typu `LexicalElement`. Tato funkce bude implementovat konečný automat popsaný níže.
- `String getString()` – vrátí hodnotu načteného lexému – pokud to daný typ `LexicalElement` umožňuje. Například jméno načteného identifikátoru, nebo číslo.



Obrázek 10: Automat lexikálního analyzátoru

3.3.1.2 Gramatika

Nyní bude popsána gramatika obsahující pravidla bezkontextového jazyka, ve kterém je zapsán vstupní program pro analýzu. Gramatika je využívána ve třídách `SyntaxAnalyzer` a `ProgramParser`.

Jazyk popsaný gramatikou musí být pro uživatele dobře pochopitelný a musí být jednoduchý. Požadavky na gramatiku jsou tyto:

- vstupní proměnné se deklarují na začátku programu za klíčovým slovem `var`
- existuje jedna výpočetní funkce s názvem `main`, udávající začátek programu
- přiřazení a testy nemají vedlejší efekty. V přiřazení se tedy mění pouze hodnota proměnné na levé straně a v testu se nemění žádná proměnná
- obsahuje smyčky

Pravidla gramatiky jsou uvedeny v následujícím ve Výpisu 5. Tučně jsou vyznačeny neterminální symboly gramatiky a pravé strany pravidel jsou odděleny pomocí znaku ‘|’.

Výpis 5: Bezkontextová gramatika pro vstupní jednotku programu

1. **program** → **programItem program** | ε
2. **programItem** → **function** | **declaration**
3. **declaration** → ”var” **declarationList**
4. **declarationList** → **declarationAssign nextDeclaration** ‘;’
5. **declarationAssign** → Identificator ‘=’ **expression** ‘;’ | ‘,’ **declarationAssign** | ε
6. **nextDeclaration** → **declarationAssign nextDeclaration**
7. **function** → “main” ‘(’ ‘)’ **statementList**
8. **statementList** → ‘{’ **command nextCommand** ’} ’
9. **nextCommand** → **command nextCommand** | ε
10. **command** → Identificator ‘=' **commmandExpression** | “++“ identificator | “--“ identificator | identificator “+= “ **commmandExpression** | identificator “-= “ **commmandExpression** | identificator “/= “ **commmandExpression** | Identificator “*= “ **commmandExpression** | identificator “++ “ | Identificator “-- “ | “if“ '(' **condition** ')' **command commandElse** | “while“ '(' **condition** ')' **command** | “for“ '(' **command** ';' **condition** ';' **command** ';' ')' **command** | **statementList**
11. **commandExpression** → **expression** ‘;’
12. **commandElse** → “else“ **command** | ε
13. **condition** → **expression operator expression**
14. **operator** → “==“ | “!=“ | “<=“ | “>=“ | “>“ | “<“
15. **expresstion** → **term nextExpression** | '-' **term nextExpression**
16. **nextExpression** → '+' **term nextExpression** | '-' **term nextExpression** | ε
17. **term** → **factor nextTerm**
18. **nextTerm** → '*' **factor nextTerm** | '/' **factor nextTerm** | ε
19. **factor** → identificator | number | '(' **expression** ')"

Aby měl uživatel co nejméně problémů s pochopením jazyka vstupního programu, jsou zde použité konstrukty jazyka stejné jako v jazycích *C* a *Java* (stuktury `for`, `while`, `goto`) nebo jako v jazyku Pascal (`var`, `function`).

3.3.1.3 Syntaktický analyzátor

Kontroluje správnou syntaxi vstupního programu. Každé pravidlo gramatiky představuje funkci v syntaktickém analyzátoru. Kontrola se provádí rekurzivním voláním funkcí dle typu lexému získaného voláním lexikálního analyzátoru. Této metodě zpracovávání vstupního programu se říká rekurzivní sestup. Chyba se najde tak, že lexikální analyzátor vrátí lexém, který není aktuálním stavem gramatiky očekáván, tzn. v aktuální funkci neexistuje volání funkce pro případ načteného lexému. Potom dojde k ukončení zpracovávání programu.

Další úkol syntaktického analyzátoru je kontrola, zda každá proměnná v programu byla deklarována v deklarační části za klíčovým slovem `var`.

3.3.2 Uložení informací o proměnných vstupního programu

Informace o proměnných ze vstupního zdrojového kódu programu se používají v téměř ve všech algoritmech analýzy. Jsou načteny ve třídě `SyntaxAnalyzer`. Objekt, který bude udržovat tyto informace, je tedy třeba vhodně navrhnout. Pro tento účel vhodný návrhový vzor *Singleton* – jedináček. Ten dovoluje vytvořit pouze jednu instanci daného objektu, kterou budou ostatní objekty používat. Ve všech třídách, kde se bude objekt s proměnnými používat, bude deklarován jako soukromá vlastnost.

Informace o proměnných jsou uchovány v objektu typu `ProgramInfo`.

Objekt obsahuje:

- ukazatel do AST na první instrukci programu – kořenový uzel AST
- seznam všech dostupných proměnných ve vstupním programu. Každá proměnná bude charakterizována:
    - Jménem.
    - Unikátním celočíselným identifikátorem (indexem) v rozmezí 0 až *n*-1, kde *n* je počet proměnných. Důležité je, že toto pořadí určuje pozici proměnné v objektech lineárních omezení a frame mnohostěnu a nebude se nikdy měnit.
    - Podstromem AST, který v deklaraci představuje inicializaci této proměnné (přiřazení). Nebo jen uzlem deklarace pokud nebyl uzel inicializován.

Životní cyklus tohoto objektu je následující

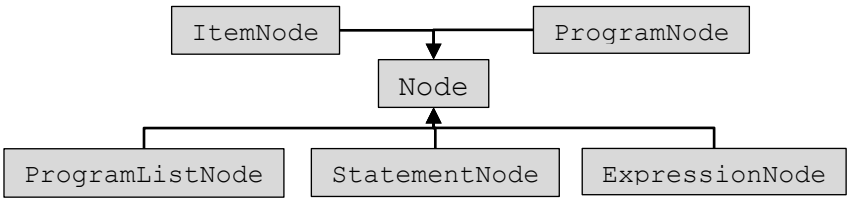
1. V objektu typu `SyntaxAnalyzer` se objekt vytvoří a začnou se do něj ukládat proměnné z deklarační části programu.
2. V objektu typu `ProgramParser` se ke každé proměnné přiřadí podstrom abstraktního syntaktického stromu charakterizující deklaraci proměnné.
3. V analýze objektu flowchart se při různých operacích objekt používá. Vyhledávání proměnné je implementováno pomocí jména i jejího indexu.
4. Objekt zanikne při ukončování programu.

3.3.3 Načtení programu do AST

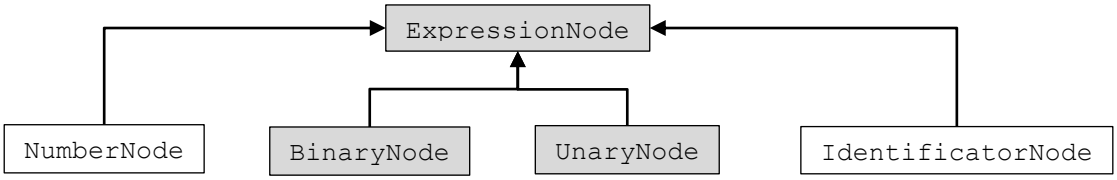
Potom co byla zkontrolována syntaxe programu a byly načteny informace o proměnných, je možné přejít k dalšímu kroku ve *vstupní jednotce*, kterým je načtení zdrojového kódu programu do formy abstraktního syntaktického stromu. Tento úkol provádí *programový parser* definovaný ve třídě `ProgramParser`.

`ProgramParser` používá pro vytvoření AST pravidla gramatiky. Každé pravidlo gramatiky představuje funkci, která vytvoří a vrátí uzel nebo podstrom konečného AST. Stejně jako u syntaktického analyzátoru, i zde se program pracuje metodou rekurzivního sestupu.

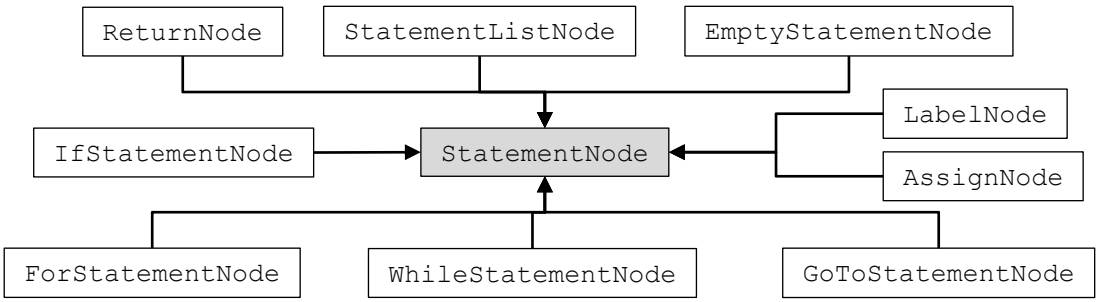
Na Obrázcích 11 až 15 je uvedena celá hierarchie uzlů AST (šedě jsou zobrazeny abstraktní třídy), které se používají v této práci. Instance uzlů, které jsou nejníže v hierarchii AST, se nazývají listy programu (např. `IdentificatorNode`, `NumberNode`). Všechny uzly nad nimi potom zapojují tyto listy do větších podstromů. První volaná funkce program parseru – první pravidlo gramatiky – tedy vrátí hodnotu až jako poslední a bude to kořenový uzel celého stromu, tedy výchozí bod pro procházení celého AST. Třídy AST jsou definovány v souboru *AstNodes.h*.



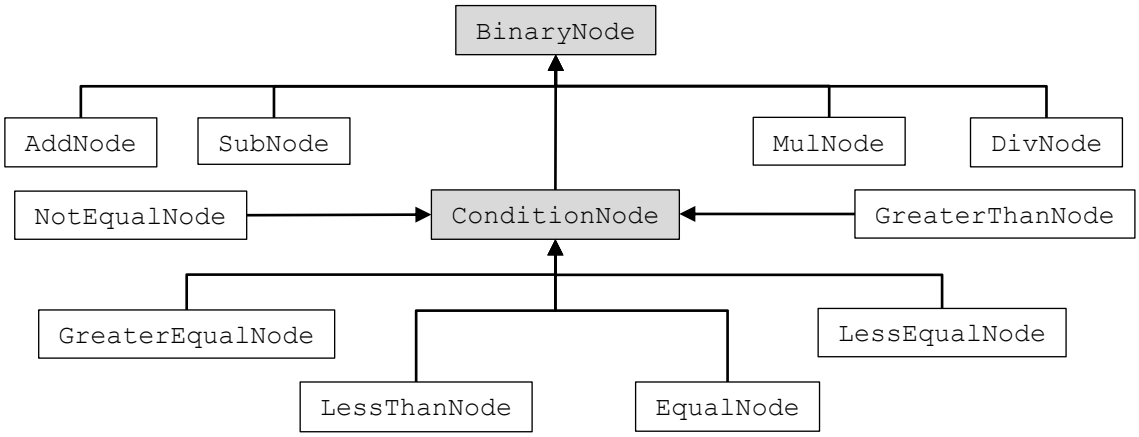
Obrázek 11: Diagram tříd AST - hlavní abstraktní třídy



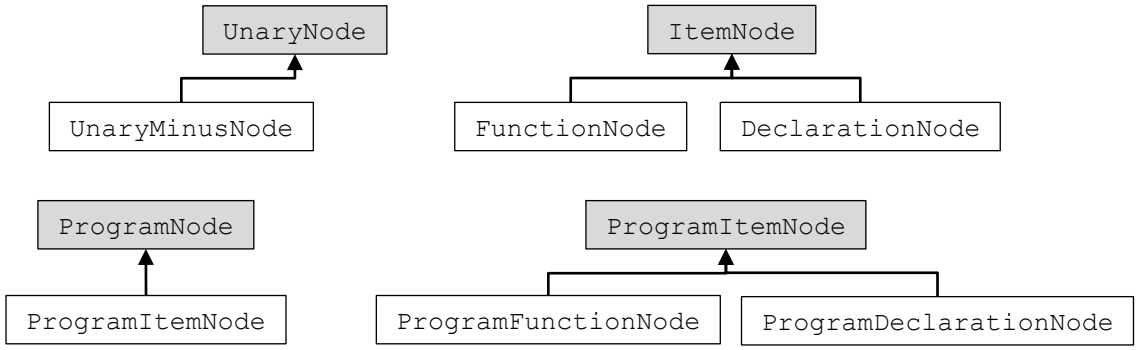
Obrázek 12: Diagram tříd AST - třídy reprezentující výrazy



Obrázek 13: Diagram tříd AST - třídy reprezentující konstrukce jazyka



Obrázek 14: Diagram tříd AST - třídy reprezentující operace a podmínky



Obrázek 15: Diagram tříd AST - třídy reprezentující unární operátor programové části

3.3.4 Převod AST na seznam instrukcí

Dosud byl uveden postup načtení zdrojového programu do vnitřní formy abstraktního syntaktického stromu pomocí přední části překladače programovacího jazyka. Nyní ve třetím kroku *vstupní jednotka* programu provede převod AST do seznamu instrukcí, ze kterého ve čtvrtém kroku provede vytvoření vnitřní reprezentace pomocí orientovaného grafu.

Přímý převod AST – graf nebude použitý, protože obě reprezentace jsou od sebe hodně odlišné. Zatímco AST je čistá stromová struktura, graf představuje tok programu. Převod by byl problematický hlavně při zpracovávání vnořených bloků pro příkazy `if`, `for` a `while`, kdy by bylo nutné se pohybovat celou strukturou vnořeného bloku, aby se našla následující instrukce za tímto blokem. Algoritmus takového převodu by byl zbytečně složitý a obtížný.

Použití seznamu instrukcí není primární požadavek této práce, ale je odvozený z požadavku pro převod vstupního programu na orientovaný graf.

3.3.4.1 Seznam instrukcí

Seznam instrukcí si lze představit jako program napsaný v jazyce assembler. Podobnost je zde pouze v tom, že instrukce zapsané v takovémto seznamu se provádějí jedna za druhou, tak jak jsou uloženy v seznamu nebo se provádějí skoky. Převodem do seznamu instrukcí bude tedy změněna reprezentace programu ze stromové struktury do obyčejného seznamu instrukcí jdoucích za sebou.

Jelikož je reprezentace pomocí seznamu instrukcí pouze dočasná mezivrstva v převodu, měla by určitým způsobem zjednodušit vstupní reprezentaci pomocí AST, tak aby bylo možno provést převod na graf jednodušeji a efektivněji.

Instrukce jsou definovány v souboru *Instructions.h* a převod je pak implementován ve třídě `FlowchartCreator`. Třída `Instruction` definuje jednu instrukci programu. Její vlastnosti jsou:

- identifikační číslo udávajícím pořadí v seznamu instrukcí
- odkaz na další instrukci v pořadí
- skok instrukce – využije se pouze pro instrukce skoku. Bude obsahovat jméno návěští, na které má instrukce skočit
- typ instrukce
- obsah instrukce

Ve Výpisu 6 je uveden příklad vstupního programu a ve Výpisu 7 potom je ho reprezentace pomocí seznamu instrukcí.

Výpis 6: Příklad programu (testovací případ tc6_5.txt)

```
var a = 0, b = 1, i;
function main(){
      a = 2*b + 5;
      if(a == 6) b++
      else b--

      for(i = 0; i < 10; i++;) a = b + i;
}
```

Výpis 7: Seznam instrukcí pro program z Výpisu 6

```
1: a = 2*b + 5
2: [!(a == 6)] goto label-if
3: b = b + 1
4: goto label-else
5: label-if:
6: b = b - 1
7: label-else:
8: i = 0
9: label-for:
10: [!(i < 10)] goto label-endfor
11: a = b + i
12: i = i + 1
13: goto label-for
14: label-endfor:
15: return
```

Typ instrukce určuje její obsah podobně jak je to uzlů v objektu flowchart:

- **Přiřazení** – reprezentuje přiřazení ve vstupním programu. Obsahuje abstraktní syntaktický podstrom tohoto přiřazení.
- **Podmínka** – reprezentuje jakoukoli podmínku ve vstupním programu, tedy ne jen podmínku typu `if` ale in podmínky ve `for` a `while` cyklech. Obsahuje abstraktní syntaktický podstrom této podmínky.
- **Skok** – reprezentuje skok v programu. Skok je vytvořen ze všech smyček, příkazu `if` a příkazu `goto` ve vstupním programu. Obsahem instrukce je název návěští `label`, na které se má skočit.
- **Návěští** – reprezentuje návěští `label` v programu. Obsahem instrukce je pouze jméno návěští.
- **Návrat** – reprezentuje instrukci pro návrat z funkce.

Nyní bude uveden postup převodu AST na seznam instrukcí. Bude se provádět pomocí rekurzivního procházení AST. Pro každý typ řídícího uzlu z AST bude ve funkci převodu existovat výpočet, který jej převede na instrukci nebo skupinu instrukcí.

Nejprve se v AST vyhledá uzel funkce `main` představující začátek výpočtu. Od tohoto uzlu se začnou zpracovávat jednotlivé konstrukce AST vstupního programu, které jsou jeho podstromy. Jsou to:

- přiřazení
- podmínka `if`
- cykly `for` a `while`
- skok `goto` a návěští `label`
- struktura `return`

Postup jak se jednotlivé konstrukce AST převedou na instrukce je uveden dále.

Přiřazení
Vytvoří se instrukce přiřazení, do které se vloží AST tohoto přiřazení a zapojí se za poslední instrukci v seznamu instrukcí

Podmínka `if`

Pokud řídící struktura `if` neobsahuje část `else`, vzniknou následující instrukce:

1) instrukce podmínky s instrukcí skoku na konec bloku `if`
2) sada instrukcí bloku `if`
3) instrukce návěští za blokem `if`

Nejprve se vytvoří instrukce 1). Obsah podmínky bude negace podmínky z AST. Instrukce se potom zapojí za poslední instrukci v seznamu instrukcí. Následuje rekurzivní zpracování těla `if`, vznikne tak 2). Celá sada se pak zapojí za 1). Za poslední instrukci z 2) se pak vytvoří instance s návěštím 3)

Pokud řídící struktura `if` obsahuje i část `else` doplní se předcházející tři instrukce těmito instrukcemi:

4) instrukcí skoku za blok `else`
5) sada instrukcí pro blok `else`
6) instrukci návěští za celý blok `if-else`

Instrukce 1) a 2) se vytvoří stejně jako v předcházejícím případě. Pak se vytvoří instrukce skoku za blok `else` 4) a zapojí se za instrukci 2). Za instrukci 4) se zapojí instrukce 3). Poté se rekurzivně zpracuje blok `else` 5) a zapojí se za instrukci 3). Nakonec se vytvoří instrukce návěští 6) a zapojí se za poslední instrukcí z 5)

Cyklus `for`

Cyklus `for` lze rozdělit na několik instrukcí:

1) instrukce počátečního přiřazení – inicializace proměnné cyklu
2) instrukce návěští udávající start smyčky v seznamu instrukcí
3) instrukce podmínky `for` s instrukcí skoku na návěští za konec cyklu `for`
4) sada instrukcí cyklu
5) instrukce pro změnu hodnoty proměnné cyklu
6) instrukce skoku na začátek cyklu
7) instrukce návěští nacházející se za koncem cyklu – od které se pokračuje dál ve vykonávání programu

Nejprve je vytvořena instrukce počátečního přiřazení 1) a zapojena za poslední instrukci v seznamu. Její obsah je AST přiřazení. Následuje instrukce návěští 2) zapojená za instrukcí 1). Za ní se vytvoří a připojí instrukce podmínky 3) se jménem návěští za konec cyklu `for` – tedy 7). Potom dojde k rekurzivnímu zpracování těla bloku `for` – vytvoří se sada instrukcí 4). Za 4) se zapojí instrukce přiřazení pro změnu hodnoty proměnné cyklu 5). Za 5) se zapojí instrukce skoku na začátek cyklu 6) – skok na návěští 2. Nakonec se vytvoří instrukce návěští za koncem cyklu 7) a zapojí se za instrukci 6).

Cyklus `while`

Cyklus `while` se rozdělí do těchto instrukcí:

1) instrukce návěští udávající start smyčky
2) instrukce podmínky `while` s instrukcí skoku na návěští za konec cyklu
3) sada instrukcí cyklu
4) instrukce skoku na začátek cyklu
5) instrukce návěští za koncem cyklu

Nejprve je vytvořena instrukce 1) a je zapojena za poslední instrukcí v seznamu instrukcí. Potom se vytvoří instrukce 2) a zapojí se za 1). Následuje rekurzivní zpracování těla smyčky, ze kterého vznikne sada instrukcí 3), která se zapojí za 2). Instrukce 4) a 5) se vytvoří a zapojí stejně jako instrukce 6) a 7) v případě `for` cyklu

Struktury `goto` a `label`
Vytvoří se vždy jedna instrukce typu skok v případě `goto` a jedna instrukce typu návěští v případě `label`.

Struktura `return`
Vytvoří se jedna instrukce ukončující vykonávání celého programu v daném místě seznamu instrukcí

3.3.5 Převod seznamu instrukcí na orientovaný graf

Posledním krokem *vstupní jednotky* programu je převedení seznamu instrukcí do vnitřní reprezentace pomocí orientovaného grafu (objektu flowchart) nad kterým se bude provádět automatické vyhledávání vztahů mezi proměnnými v programu.

Převod je implementovaný ve funkci `FlowchartCreator::create()`. Tato funkce vytvoří ze seznamu instrukcí novou instanci objektu `Flowchart`.

Než bude popsán algoritmus převodu, je nutné si popsat třídu reprezentující orientovaný graf, tedy třídu `Flowhchart`. Třída obsahuje:

- Seznam hran. Hrana je definována ve třídě `Arc`.
- Seznam uzlů. Uzly jsou definovány v abstraktní třídě `Vertex` a jejích podtřídách.
- Ukazatel na první uzel grafu.
- Metodu `process()`, která provede celou analýzu.

Diagram tříd pro třídu `Flowchart` a všechny její spolupracující třídy je uveden na Obrázku 16.

3.3.5.1 Hrany grafu

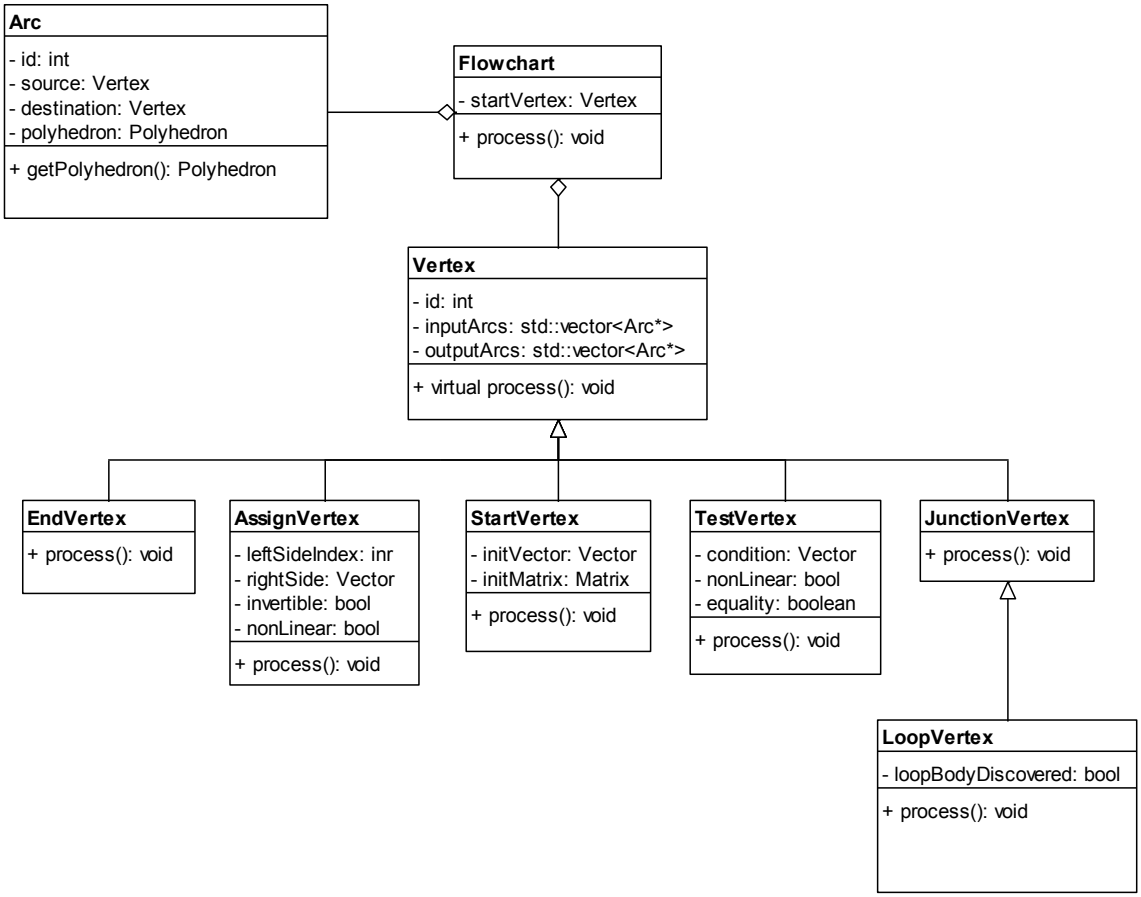
Třída `Arc` obsahuje definici hrany grafu. Každá hrana má tyto vlastnosti:

- unikátní identifikátor v rámci všech hran `id`
- ukazatel na vstupní uzel `source`
- ukazatel na výstupní uzel `destination`
- objekt reprezentující mnohostěn `polyhedron`

3.3.5.2 Uzly grafu

Jak již bylo v Kapitole 2 napsáno, graf obsahuje celkem 6 typů uzlů. Přitom je definován jeden abstraktní předek všech uzlů ve třídě `Vertex`, která obsahuje:

- unikátní identifikátor uzlu `id`
- seznam vstupních hran `inputArcs`
- seznam výstupních hran `outputArcs`
- virtuální metodu `process()`, kterou budou implementovat potomci třídy `Vertex`. Tato metoda představuje výpočet jednotkového kroku analýzy, tedy transformaci v uzlu.



Obrázek 16: Diagram tříd pro objekt Flowchart

Každý konkrétní uzel zděděný od tohoto uzlu pak může obsahovat další informace nutné k provedení transformace v metodě `process()`.

Samotný převod seznamu instrukcí na graf se provede tak, že se budou postupně procházet všechny instrukce počínaje první instrukcí. Ta obsahuje odkaz na další instrukci, atd. Přitom instrukce přiřazení, podmínky a návratu reprezentují uzly přiřazení, podmínky a ukončení v grafu. Instrukce skok a návěští budou vyžadovat uložení informace o místě skoku nebo návěští, aby se mohl správně vytvořit slučovací uzel. Uzel smyčky se najde až po vytvoření celého grafu.

3.3.5.3 Reprezentace přiřazení a podmínek

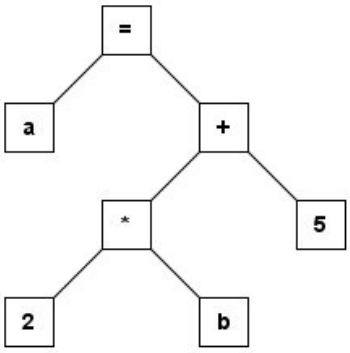
Doposud se s přirazením nebo podmínkou ve vstupním programu pracovalo jako s podstromem AST vstupního programu. Dokonce i každá instrukce tohoto typu obsahuje ukazatele na konkrétní podstrom přiřazení nebo podmínky v AST. V grafové reprezentaci se již s AST nepracuje. V transformacích v uzlech je efektivnější pracovat s přiřazeními nebo podmínkami jako s vektory. Je to jednoduše z toho důvodu, že prvky množin objektu frame (body, polopřímky, přímky) jsou také vektory a každý řádek matice reprezentující lineární omezení lze považovat za vektor. A právě uzel je místo, kde všechny tyto vektory přicházejí do styku. Proto podstromy AST pro uzly přiřazení a testu jsou převedeny na vektorovou reprezentaci. Převod bude součástí převodu ze seznamu instrukcí na objekt flowchart.

Princip převodu tedy spočívá v převedení podstromu AST přiřazení nebo podmínky do vektorové reprezentace ve tvaru:

$$(a_1 x_1, \dots, a_n x_n, r)$$

- a_i je vektor koeficientů proměnných v přiřazení nebo podmínce
- $x_i = \begin{cases} 0, & proměnná\ i\ není\ součástí\ přiřazení\ nebo\ podmínky \\ 1, & proměnná\ i\ je\ součástí\ přiřazení\ nebo\ podmínky \end{cases}$
- r je konstanta v přiřazení nebo podmínky – nenáleží tedy žádné proměnné s indexem i

Jako příklad bude uvedeno přiřazení $a = 2 * b + 5$ z programu z přechozí kapitoly. Pro úplnost program, ve kterém je toto přiřazení obsahuje pouze tři proměnné a, b a i, s indexy $(a = 0; b = 1; i = 2)$. AST pro tento typ přiřazení je uveden na následujícím Obrázku 17.



Obrázek 17: AST přiřazení

Jeho vektorová reprezentace potom bude:

$$(a_1 a; a_2 b; a_3 i; r) = (1 \cdot 0; -2 \cdot 1; 0 \cdot 0; -5) = (0; 2; 0; 5)$$

Nyní, když je známa struktura grafu, jeho uzlů a hran a je popsána reprezentace obsahu přiřazení a podmínky v uzlech je možné popsat, jak se vytvoří celý objekt flowchart ze seznamu instrukcí. Popis bude založen na jednotlivých uzlech grafu a jejich vztahu k instrukcím, ze kterých se vytvoří. Pro přehlednost je doporučeno sledovat diagram tříd na Obrázku 16.

3.3.5.4 Startovní uzel

Je vždy první uzel grafu. Vytvoří se ještě před procházením seznamu instrukcí. Jeho obsahem jsou:

- **Inicializační vektor** – (IV) `initVector` - je to pole typu `bool` jehož velikost je počet proměnných programu. Hodnota `true` nebo `false` na dané pozici udává informaci o tom, zda proměnná s daným indexem byla při deklaraci inicializována nebo ne.
- **Inicializační matice** – (IM) – `initMatrix` - obsahuje koeficienty přiřazení každé proměnné, která byla inicializována při deklaraci. Struktura matice bude následující. Bude obsahovat tolik sloupců kolik je proměnných v programu plus jeden sloupec udávající hodnotu na pravé straně přiřazení při inicializaci.

Pokud bude proměnná *i* inicializována, bude v inicializačním vektoru na indexu, který patří proměnné, hodnota `true`. V inicializační matici potom bude ve sloupci, jehož číslo odpovídá indexu proměnné, hodnota 1 a v posledním sloupci hodnota, na kterou byla proměnná inicializována. V rámci příkladu ve Výpisu 6 budou *IV* a *IM* vypadat takto(indexy proměnných jsou $a = 0, b = 1, i = 2$):

$$IV = (1,1,0), IM = \begin{pmatrix} 1 & 0 & 0\ 0 \\ 0 & 1 & 0\ 1 \\ 0 & 0 & 0\ 0 \end{pmatrix}$$

Proměnné *a* patří první sloupec matice. Je zde hodnota jedna, což znamená, že proměnná *a* bude inicializována na hodnotu v posledním sloupci matice, což je nula. Pro proměnnou *b* platí to samé. Proměnná *i* nemá ve třetím sloupci hodnotu jedna a není tedy inicializována.

3.3.5.5 Uzel přiřazení

Bude vytvořen z každé instrukce přiřazení a zapojen novou hranou za předcházející instrukci. Jeho obsahem bude:

- index proměnné na levé straně přiřazení `leftSideIndex`
- vektor udávající koeficienty přiřazení `rightSide`
- informace o tom zda je přiřazení invertibilní `invertible`
- informace o tom zda je přiřazení nelineární `nonLinear`

Informace o tom zda je přiřazení invertibilní se získá tak, že se najde hodnota ve vektoru přiřazení na pozici o hodnotě indexu na levé straně přiřazení. Pokud je hodnota ve vektoru na této pozici jiná od nuly, jde o invertibilní přirazení. Jinak jde o neinvertibilní přiřazení.

Příklad invertibilního přiřazení je $i = i + 1$. Proměnná i je na obou stranách přiřazení.

3.3.5.6 Uzel podmínky

Bude vytvořen z každé instrukce podmínky a zapojen novou hranou za předcházející instrukci. Jeho obsahem bude:

- vektor udávající koeficienty podmínky `condition`
- informace o tom zda je podmínka nelineární `nonLinear`
- informace o tom zda je podmínka typu rovnost `equality`

Vektor udávající koeficienty podmínky vznikne normalizováním podmínky do tvaru $(a_1x_1, \dots, a_nx_n, b)$. Zda je podmínka typu rovnost, se pozná z typu AST uzlu `ConditionNode`:

- pokud je typ AST uzlu rovnost, jde o rovnost
- pokud jde o nějaký typ nerovnosti, bude tato nerovnost přepočítána na nerovnost typu $ax \leq b$, dle článku v kapitole

(Ne)lineární podmínka se zjistí stejným způsobem jako v případě uzlu přiřazení.

3.3.5.7 Slučovací uzel

Vznikne z instrukce typu návěští. Slučuje více vstupních toků. Obsah uzlu není nijak definován. Jde pouze o uzel slučující více vstupních hran do jedné výstupní hrany.

Při vytváření slučovacího uzlu může nastat několik situací:

1. Uzel, který obsahuje instrukci skoku, nebyl v době vytvoření slučovacího uzlu ještě zpracován, nenachází tedy se v seznamu uzlů provádějící skoky a není tedy možné zapojit ho k tomuto uzlu. Tento uzel se tedy přidá do seznamu uzlů, které jsou cíli skoků a zapojí se později. Tato situace nastane tehdy, pokud instrukce typu `label` předchází instrukci typu `goto`.
2. Instrukce návěští patří cyklům `for` nebo `while` – v tomto případě jde o uzel, jehož výstupní hrana vede dovnitř smyčky. Jedna jeho vstupní hrana vede zevnitř smyčky a druhá vstupní hrana vede z programu do smyčky. Tento uzel bude vytvořen pro instrukce `for` 2) a instrukci `while` 1), které jsou uvedeny výše.
3. Instrukce návěští patří struktuře `if` – v tomto případě bude jeden vstupní uzel odpovídat poslednímu uzlu bloku `if`. Druhý vstupní uzel bude odpovídat vzdálenému uzlu podmínky `if`, který je uložený v seznamu uzlů, ze kterých se provádějí skoky. Oba uzly se tedy zapojí hranou do tohoto slučovacího uzlu.

3.3.5.8 Koncový uzel

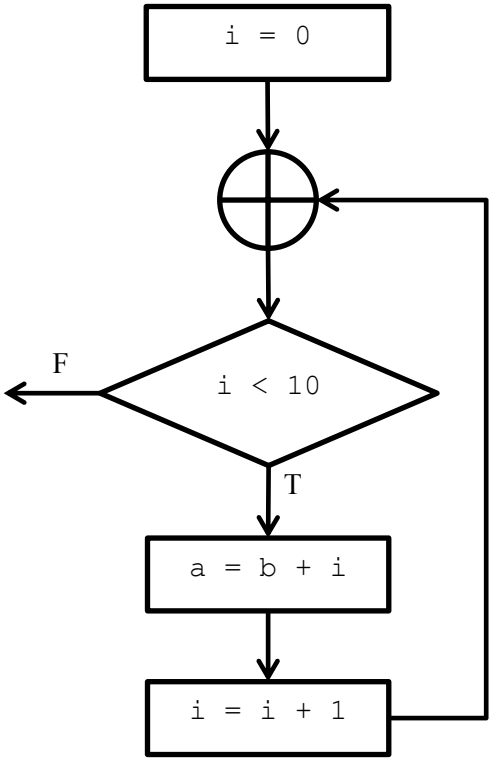
Vytvoří se, pokud se při převodu narazí na instrukci typu návrat. Neobsahuje žádné další informace.

3.3.5.9 Uzel smyčky

Jak již bylo řečeno dříve, uzel smyčky se hledá až po vytvoření objektu flowchart pomocí algoritmu prohledávání do hloubky. Jelikož algoritmu prohledávání grafu do hloubky se věnuje mnoho publikací (na internetu nebo knižních), nebude zde podrobně vysvětlen. Hledání využívá rekurzivní verzi algoritmu:

1. všechny uzly se označí jako nenavštívené
2. do zásobníku se vloží startovní uzel objektu flowchart
3. zavolá se rekurzivní metoda prohledávání:
    a. pokud je zásobník prázdný, algoritmus se ukončí
    b. ze zásobníku se vybere uzel z jeho vrcholu a jeho stav se změní na otevřený.
    c. zjistí se všichni sousedé tohoto uzlu:
        i. soused, který je ve stavu nenavštívený, se vloží do zásobníku a provede se rekurzivní volání
        ii. soused, který je ve stavu otevřený a jeho typ je slučovací uzel je budoucí uzel smyčky. Uložíme jej do seznamu uzlů, které budou uzly smyčky
4. po skončení rekurzivní fáze jsou všechny uzly ze seznamu uzlů, které jsou smyčky nahrazeny uzly smyček.

Na Obrázku 18 je příklad grafové reprezentace cyklu `for` z programu ve Výpisu 6 a seznamu instrukcí ve Výpisu 7.



Obrázek 18: Cyklus for v grafové reprezentaci

V tomto okamžiku je vytvořena vnitřní reprezentace vstupního programu v paměti, objekt flowchart. Převodem seznamu instrukcí na flowchart a vyhledáním uzlů smyčky končí úloha *vstupní jednotky* programu. Nyní nic nebrání tomu, aby se provedla analýza.

3.4 Analýza - transformace v uzlech

Po načtení vstupního zdrojového kódu programu do vnitřní reprezentace pomocí orientovaného grafu je možné přistoupit k samotné analýze, tedy k automatickému vyhledávání lineárních vztahů mezi proměnnými v programu.

V Kapitole 2 bylo vysvětleno, že transformace v uzlech grafu jsou jednotkovou operací analýzy. Transformace se pro každý uzel ve třídě `Vertex` provede v metodě `Vertex::process()`.

Ve třídě `Flowchart` je definována metoda `Flowchart::process()`, která provede celou analýzu. Analýza může končit dvěma způsoby:

- Analýza dokončena. Dojde k ustálení všech transformací ve všech uzlech.
- Analýza nedokončena. Po konečném počtu iterací, který je definován jako konstanta ve třídě Flowchart, nedošlo ke stabilizaci transformací. Stále existují uzly, které v transformaci vypočítávají novou reprezentaci mnohostěnu v každé iteraci. I tak se program ukončí a budou vypsány výsledky s informací, že nedošlo k úplné stabilizaci. Toto násilné ukončení programu je nutné kvůli tomu, aby program provádějící analýzu skončil v konečném čase. Konstanta udávající počet iterací, je ale dostačující i pro velké vstupní programy.

V následujících odstavcích bude pro každý typ uzlu grafu uveden navržený popis transformace dle algoritmů popsaných v [1]. Půjde o vysvětlení algoritmu, který se provede v metodě

`Vertex::process()` každého typu uzlu. V kapitole 2 lze najít jednoduchá vysvětlení transformací na příkladech.

Aby byl zdrojový kód programu přehlednější, jsou jednotlivé transformace definovány v samostatných třídách. Každá třída obsahuje metodu `transform()`, která se zavolá v metodě `Vertex::process()` pro daný typ uzlu. U každého vysvětlovaného uzlu budou uvedeny jména transformačních tříd.

3.4.1 Startovní uzel

Implementace transformace: třída `StartTransform`
Spuštění transformace: `StartTransform::transform()`
Metoda `StartTransform::transform()` je volána v metodě: `StartVertex::proces()`

Tento uzel neobsahuje žádnou vstupní hranu. Jeho obsahem jsou pouze inicializační vektor *IV* a inicializační matice *IM*. Transformace se tedy provádí pouze s pomocí *IV* a *IM*. Mnohostěn ve výstupní hraně tohoto uzlu je potom definován jedním z následujících dvou způsobů. Předpokládá se počet proměnných programu *n* a výpočet se provádí v *n*-rozměrném prostou nad tělesem $\boldsymbol{F}^n$:

- pokud je každý prvek inicializačního vektoru roven hodnotě `false`, inicializační matice je nepotřebná a platí:
    - systém lineárních omezení je prázdný
    - frame obsahuje:
        - bod počátku v $\boldsymbol{F}^n$ tedy $s = (0, \dots, 0)$
        - žádnou polopřímku
        - přímky $d_1, \dots d_n$ takové, že pro každé $i = 1..n, d_{ii} = 1 \; a \; d_{ij} = 0, pro \; j = 1..n \; a \; j \neq i$

    Tato inicializace říká, že mnohostěn je definován každým bodem prostoru $\boldsymbol{F}^n$. Tedy existuje bod v počátku soustavy souřadnic, od kterého se ve všech směrech daných přímkami dá vypočítat jakýkoli bod prostoru.

- Pokud prvek inicializačního vektoru obsahuje na nějaké pozici *i* hodnotu `true`, znamená to, že proměnná s indexem *i* byla inicializována. Hodnota, na kterou byla inicializována, se pak nachází na *i*-tém řádku a posledním sloupci inicializační matice. Potom platí:
    - frame obsahuje:
        - bod v $\boldsymbol{F}^n$ jež má na pozici *i* hodnotu inicializační proměnné
        - žádnou polopřímku
        - z definice přímek z předchozího bodu chybí přímka d_i
    - po vytvoření frame výše se tento převede na lineární omezení

Pseudokód algoritmu lze najít v příloze I pod názvem Pseudokód START-1.

3.4.2 Uzel přiřazení

Implementace transformace: třída `AssignTransform`
Spuštění transformace: `AssignTransform::transform()`
Metoda `AssignTransform::transform()` je volána v metodě: `AssignVertex::proces()`

Tento uzel obsahuje jednu vstupní hranu a jednu výstupní hranu. Vstup transformace bude:

- obsah uzlu – přiřazení ze vstupního programu
- mnohostěn ze vstupní hrany uzlu

Obsah uzlu tvoří:

- index proměnné na levé straně přiřazení - `leftSideIndex`
- vektor přiřazení z výrazu na pravé straně ve tvaru - `rightSide`
- informace o tom zda je přiřazení nelineární - `nonLinear`
- informace o tom zda je přiřazení invertibilní – `invertible`

Výstup transformace bude mnohostěn, který se přiřadí výstupní hraně uzlu.

3.4.2.1 Nelineární přiřazení

Jelikož se v této práci zkoumají lineární vztahy, efekt nelineárního přiřazení bude mít za následek ztrátu informace o hodnotách proměnné na levé straně přiřazení v soustavě lineárních omezení. Transformace se pak provádí následovně. Na levé straně přiřazení je proměnná s indexem i_0. Z lineárních omezení je potom eliminován sloupec matice patřící proměnné i_0. V případě frame, se přidá přímka obsahující jedničku na pozici i_0 a jinde nuly. Celý frame se pak pomocí nových lineárních omezení zjednoduší. Výsledný frame a lineární omezení pak budou představovat reprezentaci mnohostěnu přiřazeného k výstupní hraně uzlu přiřazení.

Pseudokód algoritmu lze najít v příloze I pod názvem Pseudokód ASSIGN-NONLINEAR-1.

3.4.2.2 Lineární přiřazení

Výraz na pravé straně lineárního přiřazení je definován jako $\sum_{i=1}^{n}(a_i x_i) + b$, kde:

- n je počet proměnných programu
- x_i jsou proměnné programu
- a_i jsou koeficienty, kterými jsou násobeny proměnné programu
- b je konstanta nepřiřazená žádné proměnné

Dále jsou definovány:

- i_0 jako koeficient proměnné na levé straně přiřazení
- frame vstupního polyhedronu je dán množinami bodů *S*, polopřímek *R* a přímek *D*
- frame výstupního polyhedronu je dán množinami bodů $\acute{S}$, polopřímek $\acute{R}$ a přímek $\acute{D}$

Je-li například přiřazení ve tvaru $c = 2 * b - 10$ a v programu jsou použity pouze dvě proměnné *c* a *b* s indexy $c = 0$ a $b = 1$ potom:

- $n = 2$
- $x_0 = c$ a $x_1 = b$
- $a_0 = 0$ a $a_1 = 2$ ($a_0 = 0$ protože proměnná c není na pravé straně přiřazení)

- $i_0 = 0$

Potom je možné vypočítat frame výstupního mnohostěnu následovně:

- pro každý bod $\acute{s}_\iota$ z množiny bodů $\acute{S} = (\acute{s}_1, \dots, \acute{s}_\sigma)$ platí:
  - pokud $i = i_0 : \acute{s}_\iota = as_i + b$
  - pokud $i \neq i_0 : \acute{s}_\iota = s_i$
- pro každou polopřímku $\acute{r}_j$ z množiny polopřímek $\acute{R} = (\acute{r}_1, \dots, \acute{r}_\rho)$ platí:
  - pokud $j = i_0 : \acute{r}_j = ar_j$
  - pokud $j \neq i_0 : \acute{r}_j = r_j$
- pro každou přímku $\acute{d}_k$ z množiny přímek $\acute{D} = (\acute{d}_1, \dots, \acute{d}_\delta)$ platí:
  - pokud $k = i_0 : \acute{d}_k = ad_k$
  - pokud $k \neq i_0 : \acute{d}_k = d_k$

Pseudokód algoritmu lze najít v příloze I pod názvem Pseudokód ASSIGN-FRAME-2.

Tímto byl vypočítán frame výstupního mnohostěnu. Nyní je nutné přepočítat lineární omezení ze vstupního mnohostěnu na lineární omezení, která budou součástí výstupního mnohostěnu. Přitom přiřazení se dělí na invertibilní a neinvertibilní.

3.4.2.2.1 Invertibilní přiřazení

Invertibilní přiřazení je takové přiřazení, ve kterém se proměnná z levé strany, tedy ta do které se přiřazuje, nachází i ve výrazu na straně pravé. Například přiřazení pro proměnné *c* a *b*:

$$c = 3 * b + c + 5$$

je invertibilní lineární přiřazení, protože proměnná *c* je i ve výrazu na pravé straně. Smysl následujícího výpočtu je v tom, že stará hodnota proměnné, do které se přiřazuje, je použita pro výpočet nové hodnoty této proměnné. Nová lineární omezení se získají takto:

1. proměnná na levé straně se označí, aby se její název nepletl s tou samou proměnnou na pravé straně

$$\bar{c} = 3 * b + c + 5$$

2. z pravé strany se vyjádří proměnná, která byla původně na levé straně

$$c = \bar{c} - 3 * b - 5$$

3. nyní se vyjádřená proměnná dosadí do původních lineárních omezení z mnohostěnu na vstupní hraně uzlu, čímž vzniknou nová lineární omezení, která budou reprezentovat mnohostěn ve výstupní hraně uzlu.

Pseudokód algoritmu lze najít v příloze I pod názvem Pseudokód ASSIGN-INVERTIBLE-3.

3.4.2.2.2 Neinvertibilní přiřazení

Je přiřazení, ve kterém se proměnná na levé straně nenachází ve výrazu na straně pravé. Předpokládá se, že index proměnné na levé straně je i_0. Postup zpracování takovéhoto přiřazení je následující:

1. ze vstupních lineárních omezení se eliminuje sloupec i_0 představující proměnnou na levé straně přiřazení
2. přiřazení se přidá jako rovnice k omezením vzniklým v předchozím bodu. Tím byly získány lineární omezení pro výstupní polyhedron.

Pseudokód algoritmu lze najít v Příloze I pod názvem Pseudokód ASSIGN-NONINVERTIBLE-4.

Nyní jsou známy všechny výpočty prováděné v uzlu přiřazení. Pseudokód převodu lze najít v Příloze I pod názvem Pseudokód ASSIGN-5.

3.4.3 Uzel podmínky

Implementace transformace: třída `TestTransform`
Spuštění transformace: `TestTransform::transform()`
Metoda `TestTransform::transform()` je volána v metodě: `TestVertex::proces()`

Tento uzel má jednu vstupní hranu a dvě výstupní hrany. Vstup transformace budou:

- obsah uzlu – podmínka ze vstupního programu
- mnohostěn ze vstupní hrany uzlu

Výstupem transformace budou dva mnohostěny. Jeden mnohostěn představuje tu část vstupního prostoru (popřípadě vstupního mnohostěnu), pro kterou je podmínka pravdivá a druhý tu část vstupního prostoru pro který je podmínka nepravdivá.

Obsah uzlu tvoří:

- vektor podmínky - `condition`
- informace o tom zda je podmínka nelineární - `nonLinear`
- informace o tom zda je přiřazení typu rovnost - `equality`

Pro vysvětlení transformace v tomto uzlu je nutné definovat:

- $P = (S, R, D, A, B)$ je vstupní mnohostěn.
- $P_T = (S_T, R_T, D_T, A_T, B_T)$ je mnohostěn asociovaný s výstupní hranou reprezentující výstup typu pravda tohoto uzlu.
- $P_F = (S_F, R_F, D_F, A_F, B_F)$ je mnohostěn asociovaný s výstupní hranou reprezentující výstup typu nepravda tohoto uzlu.

Kde:

- S, S_T, S_F jsou množiny bodů frame mnohostěnů
- R, R_T, R_F jsou množiny polopřímek frame mnohostěnů
- D, D_T, D_F jsou množiny přímek frame mnohostěnů
- A, A_T, A_F jsou matice koeficientů lineárních omezení a B, B_T, B_F jsou konstanty pravých stran lineárních omezení. Zde je nutné poznamenat, že skutečná maticová reprezentace lineárních omezení v tomto programu obsahuje koeficienty matice a sloupec pravých stran v jedné matici. Pro účely vysvětlení transformace je nutné je ale od sebe oddělit

3.4.3.1 Nelineární podmínka

Je první typ podmínky, která může nastat. Jelikož se tato analýza zabývá lineárními vztahy, nebudou brány nelineární vztahy v potaz, protože pro ně nejsou popsány transformace. Výsledné mnohostěny pak budou kopií vstupního mnohostěnu, tedy:

$$P_T = P_F = P$$

3.4.3.2 Podmínka typu rovnost

Vektor podmínky (`condition`) je zapsán ve tvaru (ax^T, b), kde:

- a je vektor koeficientů podmínky, které jsou v součinu s proměnnými
- x^T je vektor představující proměnné programu
- b je konstanta, která není koeficientem žádné proměnné

Například, program se třemi proměnnými a jejich indexy $(x_1 = 0; x_2 = 1; x_3 = 2)$ obsahuje tuto podmínku (v části `if`, `while`, nebo `for`):

$$x_1 + 3 \cdot x_2 + 2 = 4 \cdot x_3 + 8$$

Ve třídě `FlowchartCreator` dojde k vytvoření vektoru podmínky tak, že se nejprve všechny prvky převedou z pravé strany na stranu levou, tedy:

$$x_1 + 3 \cdot x_2 - 4 \cdot x_3 - 6 = 0$$

A následně se ze všech koeficientů vytvoří vektor, přitom musí platit, že konstanta bude vždy poslední prvek vektoru. Výsledný vektor podmínky pak bude:

$$(1,3,-4,-6)$$

Poznámka: Zde je důležité si uvědomit, že vektor podmínky nereprezentuje prostor o jednu dimenzi větší než dimenze vstupního mnohostěnu, ale je to jen zápis podmínky, potřebný pro výpočty v tomto uzlu. Pokud jsou tedy v programu tři proměnné tak vektor podmínky bude mít vždy velikost čtyři a dimenze prostoru, ve kterém se provádějí výpočty, zůstane vždy tři.

V Kapitole 2.5.1.4 bylo napsáno, že podmínka představuje rovnici hyperroviny, která dělí prostor, ve kterém se nachází mnohostěn ze vstupní hrany, na dvě části. Rovnice hyperroviny z příkladu výše je:

$$x_1 + 3 \cdot x_2 - 4 \cdot x_3 = 6$$

Podle toho v jakém vztahu jsou hyperrovina H a vstupní mnohostěn P lze rozlišit několik případů výpočtů:

- hyperrovina protíná mnohostěn:
  - pokud je mnohostěn celý obsažený v hyperrovině platí pro výstupní mnohostěny:
    - $P_T = P$
    - $P_F = \emptyset$
  - pokud hyperrovina protíná jen část mnohostěnu, platí:
    - $P_T = P \cap H$
    - $P_F = P$
- hyperrovina vede mimo mnohostěn. Potom platí:
  - $P_T = \emptyset$ a $P_F = P$

Poznámka: hyperrovina H je řešení rovnice podmínky, která je obsahem testovacího uzlu. Obě slova jsou tedy v tomto případě zaměnitelná. Pro přehlednost se bude v některých případech používat slova podmínka a jinde slova hyperrovina nebo značení H. Je srozumitelnější říci, že vstup splňuje podmínku než, že celý mnohostěn leží v hyperrovině.

Nyní je možno přistoupit k návrhu transformace v uzlu. Nejprve je nutné zjistit, zda vstupní mnohostěn splňuje podmínku v uzlu, tzn. celý se nachází v hyperrovině H. Zjistí se to tak, že se všechny prvky frame vstupního mnohostěnu dosadí do rovnice podmínky. Pokud pro všechny body, polopřímky a přímky rovnice platí, transformaci lze ukončit výstupem $P_T = P$ a $P_F = \emptyset$.

Pseudokód pro zjištění pravdivosti podmínky pro celý vstupní frame vypadá takto:

```
function conditionIsTrue(Frame inputFrame, Vector condition){
    if(all inputFrame items comply condition) return true;
    return false;
}
```

Pokud je funkce výše vyhodnocena jako nepravdivá je nutno vypočítat prvky $P \cap H$. Znamená to, že mnohostěn buď leží mimo hyperrovinu a tedy $P \cap H = \emptyset$, nebo hyperrovina protíná mnohostěn. Potom $P \cap H \neq \emptyset$. Prvky frame $P \cap H$ lze získat následujícími osmi výpočty:

1) Bod s splňující podmínku $ax^T = b$. Tedy $as = b$. Bod je součástí H.
2) Polopřímka r splňující podmínku $ax^T = 0$. Tedy $ar = 0$. Polopřímka je vektor udávající směr a proto musí být porovnána se směrem vektoru hyperroviny H (Který je dán právě $ax^T = 0$). Pokud rovnice platí je r součástí H.
3) bod s ležící na úsečce vedené mezi dvěma sousedními body s_1a s_2 vstupního frame protínající rovnici danou podmínkou uzlu $ax = b$. Existence bodu se ověří následovně:

$$\lambda = \frac{b - as_2}{as_1 - as_2}; kdy\ 0 \leq \lambda \leq 1$$

Pokud takové λ existuje lze bod získat vztahem:

$$s = \lambda s_1 + (1 - \lambda)s_2$$

Kde λ představuje posunutí na úsečce od bodu s_1 (λ =0) směrem k bodu s_2 (λ =1). Pokud je tedy např. $\lambda = 0{,}5$ leží nový bod s přesně uprostřed přímky mezi dvěma body s_1 a s_2. Na Obrázku 7 jde o body (1,1) a (2,1)

4) bod s ležící na polopřímce vedené z bodu $s_1$ve směru polopřímky r_1. Existence bodu se ověří následovně:

$$\mu = \frac{b - as_1}{ar_1}; kdy\ \mu \geq 0$$

Pokud takové μ existuje, bod je dán vztahem:

$$s = s_1 + \mu r_1$$

Kde μ představuje vzdálenost bodu s od bodu s_1 ve směru polopřímky r_1. Pokud $\mu = 0$, pak $s = s_1$.

5) bod *s* ležící na přímce vedené z bodu $s_1$ve směru přímky d_1. Bod je určen pomocí:

$$\nu = \frac{b - as_1}{ad_1}$$

a vypočítá se takto:

$$s = s_1 + \nu d_1$$

Tyto dva vztahy platí pro všechny dvojice bodů a přímek ve frame vstupního mnohostěnu. Proměnná ν pak představuje vzdálenost bodu *s* od bodu s_1 v obou směrech přímky *d*.

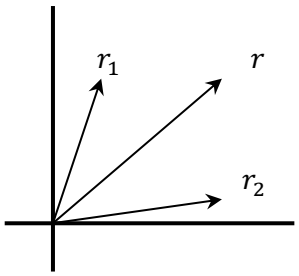
6) polopřímka *r*, která vznikne jako kombinace dvou přímek r_1a r_2. Existence polopřímky se ověří pomocí:

$$\mu = \frac{ar_1}{ar_2}; kdy\ \mu \geq 0$$

Pokud takové μ existuje polopřímka je dána vztahem:

$$r = r_1 + \mu r_2$$

Jde tedy o polopřímku *r* udávající vektor, který má stejný směr jako rovnice hyperroviny, která se nachází mezi dvěma vektory r_1a r_2. Celou situaci ukazuje Obrázek 19.



Obrázek 19: Kombinace dvou vektorů

7) polopřímka *r* vzniklá jako součet polopřímky r_1a vektoru lineárně závislého na přímce d_1. Tedy

$$r = r_1 - \left(\frac{ar_1}{ad_1}\right) d_1$$

Platí pro všechny dvojice polopřímek a přímek ze vstupního frame mnohostěnu. Jde o stejný případ jako je na Obrázku 19 s tím rozdílem, že místo druhé polopřímky je zde přímka.

8) přímka *d* vzniklá jako součet přímky d_1 a vektoru lineárně závislého na přímce d_2. Je dána vztahem

$$d = d_1 - \left(\frac{ad_1}{ad_2}\right) d_2$$

Provede se pro všechny dvojice přímek ze vstupního frame mnohostěnu. Opět jde o stejný případ jako na Obrázku 19 ovšem zde jde o dvě přímky.

Pseudokód pro výpočet $P \cap H$ je uveden v Příloze I pod názvem Pseudokód TEST-INTERSECTION-1.

3.4.3.3 Podmínka typu nerovnost

Nerovnost je zapsána v normalizovaném tvaru jako $ax^T \leq b$. Jak již bylo řečeno dříve, všechny ostatní typy nerovností se musí přepočítat na tento typ nerovnosti a uložit v tomto tvaru do podmínky nerovnosti v uzlu. Například podmínka:

$$x_1 + 3 \cdot x_2 + 2 > 4 \cdot x_3 + 8$$

bude mít normalizovaný tvar:

$$-x_1 - 3 \cdot x_2 + 4 \cdot x_3 \leq -6$$

Ostrá nerovnost se navíc převedla na nerovnost neostrou, aby se zaručilo, že výsledné mnohostěny budou uzavřené objekty.

Pro výpočet podmínky typu nerovnost se využije hyperrovina H jako při výpočtu podmínky typu rovnost. H se z podmínky nerovnosti získá záměnou znaménka na rovnost tedy $ax^T = b$. Tato hyperrovina opět dělí n-rozměrný prostor na dva poloprostory. Navíc v případě nerovnosti platí, že jeden poloprostor je řešením nerovnosti a druhý není. Viz Obrázek 7. Pak lze rozlišit následující možnosti ověření splnitelnosti podmínky:

1) hyperrovina leží mimo vstupní mnohostěn:
    - vstupní mnohostěn leží v poloprostoru řešení podmínky:
        - $P_T = P$
        - $P_F = \emptyset$
    - vstupní mnohostěn neleží v poloprostoru řešení podmínky:
        - $P_T = \emptyset$
        - $P_F = P$
2) hyperrovina H protíná mnohostěn. Dělí tedy mnohostěn na dvě části. Potom platí (mnohostěny P_t a P_f jsou definovány dále):
    - $P_T = (P \cap H) \cup P_t$
    - $P_F = (P \cap H) \cup P_f$

V případě 1) se H nachází v prostoru mimo vstupní mnohostěn. Pokud mnohostěn leží v poloprostoru řešení nerovnice podmínky, zkopíruje se celý vstupní mnohostěn na výstup typu pravda P_T. Výstup typu nepravda P_F bude prázdný. Volná interpretace tohoto případu je, že hodnoty všech proměnných v aktuálním místě programu leží právě v prostoru vyhovující podmínce. Pokud vstupní mnohostěn neleží v poloprostoru řešení nerovnice podmínky, je situace opačná.

V případě 2) je situace složitější. Nejprve je nutné si uvědomit, že část vstupního mnohostěnu ležícího v H $(P \cap H)$ je součástí obou výstupních mnohostěnů, aby se zaručilo, že výsledný mnohostěn bude uzavřený objekt. Prvky $P \cap H$ se vypočítají pomocí výpočtů v bodech 1) až 8) z předchozí Podkapitoly.

Další problém je nalezení P_t a P_f kdy:

- P_t je část vstupního mnohostěnu ležící v poloprostoru řešení nerovnice dané podmínkou uzlu
- P_f je část vstupního mnohostěnu, který neleží v poloprostoru řešení nerovnice dané podmínkou uzlu

Frame mnohostěnu P_t je definován:

- množinou bodů S_t, do které patří ty body vstupního mnohostěnu splňujících podmínku nerovnosti $ax^T < b$. Tedy:

$$S_t = \{\forall s \in S: as < b\}$$

- množinou polopřímek R_t, do které patří polopřímky a přímky splňující následující podmínky:

$$R_t = \{\forall r \in R: ar < 0\} \cup \{\forall d \in D: ad < 0\} \cup \{\forall -d \in D: ad > 0\}$$

- množinou přímek D_t, do které patří všechny přímky z $P \cap H$.

Ve výpočtech výše se používají pouze ostré nerovnosti. Je to z toho důvodu, že prvky frame, které vyhovují čistým rovnostem, jsou již obsaženy v $P \cap H$.

Frame mnohostěnu P_f s množinami bodů S_f, polopřímek R_f a přímek D_f se získá obdobně jako frame P_t s tím rozdílem, že jednotlivé body a polopřímky vstupního mnohostěnu musí vyhovovat opačným typům nerovnic. Množina přímek pak bude $D_f = D_t$.

Množina polopřímek R_t/R_f je navíc tvořena i vektory přímek vstupního mnohostěnu. Platí totiž, že pokud d je vektor přímky nějakého frame potom d a $-d$ jsou vektory polopřímek tohoto frame. V testování podmínky se tedy musí objevit i přímky.

Polopřímky a přímky frame se porovnávají s nulou na pravé straně podmínky, protože jde o vektory. Porovnávají se tedy směry vektorů polopřímek a přímek se směrem vektoru hyperroviny.

Naopak body se musí porovnávat s konstantou na pravé straně, protože udávají konkrétní místo v n-rozměrném prostoru vzhledem k H.

Pseudokód zpracování podmínky typu nerovnost je uveden v Příloze I pod názvem Pseudokód TEST-INEQUALITY-2.

Pseudokód transformace v testovacím uzlu je uveden v Příloze I pod názvem Pseudokód TEST-3.

Výstupní mnohostěny P_T a P_F nejsou minimální a je třeba je vůči lineárním omezením zjednodušit.

3.4.4 Slučovací uzel

Implementace transformace: třída `JunctionTransform`
Spuštění transformace: `JunctionTransform::transform()`
Metoda `JunctionTransform::transform()` je volána v metodě: `JunctionVertex::proces()`

Slučovací uzel má více vstupních hran a jednu hranu výstupní. Podstata transformace spočívá ve sloučení, matematicky vytvoření konvexního obalu, mnohostěnů na vstupních hranách. Výsledný konvexní obal vstupních mnohostěnů se přiřadí do výstupní hrany.

Uzel neobsahuje žádná speciální data nutná k transformaci. Transformace se tedy zúčastní pouze mnohostěny na vstupních hranách.

Než bude popsán postup transformace, je nutné definovat:

- mnohostěn přiřazený k jedné vstupní hraně $P_1 = (S_1, R_1, D_1, A_1, B_1)$

- mnohostěn přiřazený k druhé vstupní hraně $P_2 = (S_2, R_2, D_2, A_2, B_2)$
- výstupní mnohostěn $P = (S, R, D, A, B)$, který bude přiřazený výstupní hraně. Vznikne jako konvexní obal vstupních mnohostěnů.

Operace vytvoření konvexního obalu je asociativní a proto ji stačí ukázat na dvou vstupních mnohostěnech P_1 a P_2.

Transformace nejprve vytvoří konvexní obal z prvků frame obou vstupních mnohostěnů P_1 a P_2, která je definována jako následující sjednocení:

$$S = S_1 \cup S_2, R = R_1 \cup R_2, D = D_1 \cup D_2$$

V druhém kroku je nutné získat z výsledného frame (S, R, D) systém lineárních omezení A a B. Nejsnadněji se jeví frame přímo konvertovat na lineární omezení. Ovšem toto je drahá operace a existuje elegantnější a levnější způsob jak to udělat.

Výsledná výstupní lineární omezení lze totiž vypočítat z obou vstupních mnohostěnů. Z jednoho vstupního mnohostěnu se použije reprezentace lineárních omezení a z druhého reprezentace pomocí frame. Pro příklad (výběr může být i opačný):

- z P_1 se použije systém omezení (A_1, B_1) zapsaných ve tvaru $A_1 x \leq B_1$
- z P_2 se použije frame (S_2, R_2, D_2)

Potom se všechny prvky frame (S_2, R_2, D_2) jednotlivě v pořadí body, polopřímky a přímky začnou začleňovat do nerovnic omezení $A_1 x \leq B_1$. Po začlenění posledního prvku jsou vytvořena výsledná lineární omezení $Ax \leq B$.

Třetí krok transformace spočívá ve zjednodušení výsledného frame (S, R, D) za použití výše vypočítaných omezení $Ax \leq B$. Je to z toho důvodu, že sjednocení prvků frame může obsahovat zbytečné prvky. Tímto je pak dokončena transformace ve slučovacím uzlu.

Pseudokód převodu je uveden v Příloze I pod názvem Pseudokód JUNCTION-1.

3.4.5 Uzel smyčky

Implementace transformace: třída `LoopJunctionTransform`
Spuštění transformace: `LoopJunctionTransform::transform()`
Metoda `LoopJunctionTransform::transform()` je volána v metodě:
`LoopVertex::proces()`

Uzel smyčky je speciální typ slučovacího uzlu, který se používá pouze v cyklech grafu, které vzniknou ze smyček ve vstupním programu.

Transformace v tomto uzlu je rozdělena do dvou kroků. V prvním kroku se provede stejný výpočet jako v uzlu slučovacím. Vypočítá se konvexní obálka všech vstupních mnohostěnů. A výsledný mnohostěn se použije pro druhý krok transformace.

V Kapitole 2.5.1.6 je popsán princip druhého kroku transformace s jednoduchým příkladem. Cílem je na základě objevení směru růstu hodnot proměnných ve smyčce vybrat ta lineární omezení, která nejlépe vystihují vztahy mezi proměnnými programu uvnitř smyčky v jakékoli její iteraci a ostatní eliminovat. Pokud se tak stane, dojde také ke stabilizaci transformace v uzlu.

Pro další vysvětlení druhého kroku transformace je nutné definovat:

- mnohostěn $P_0 = (S_0, R_0, D_0, A_0, B_0)$, který vznikl operací sjednocení dvou vstupních mnohostěnů pomocí transformace ve slučovacím uzlu, nebo jde o mnohostěn z jedné vstupní hrany v případě, že mnohostěn v druhé vstupní hraně je prázdný
- výstupní mnohostěn $P_{n-1} = (S_{n-1}, R_{n-1}, D_{n-1}, A_{n-1}, B_{n-1})$ vypočtený v předchozí transformaci tohoto uzlu
- mnohostěn $P_n = (S_n, R_n, D_n, A_n, B_n)$ vzniklý aktuální transformací v tomto uzlu

Poznámka: Důležité je si uvědomit, že v druhém kroku transformace (aplikací operátoru *widening*) nedochází k vypočítávání nových prvků frame nebo lineárních omezení P_n, P_{n-1} nebo P_0. Dochází pouze k eliminaci některých lineárních omezení mnohostěnu P_0.

Aby se aplikace operátoru *widening* mohla s úspěchem provést, je nejprve nutné, aby byl P_0 opravdu konvexní obálkou obou vstupních mnohostěnů. Toto nelze zaručit při prvním průchodu grafem, protože cyklus grafu není zatím objeven a tedy vstupní hrana vedoucí ze smyčky do tohoto uzlu obsahuje prázdný mnohostěn. Obecně nemá smysl transformaci provádět, když je tento mnohostěn prázdný. V tomto případě platí $P_n = P_0$.

Pokud platí, že P_0 je sjednocení obou vstupních mnohostěnů, může se přejít k samotné aplikaci operátoru *widening,* kterou lze matematicky zapsat $P_n = P_{n-1} \nabla P_0$:

- do každého omezení ze soustavy $A_0 x \leq B_0$ se za x dosadí všechny body S_{n-1},. Je nutné, aby omezení byly ve tvaru neostrých nerovností $\leq$
- pokud některá nerovnice po dosazení alespoň jednoho bodu frame neplatí, bude eliminována. Neobjeví se tedy mezi výstupními omezeními v $A_n x \leq B_n$.

Nakonec je nutné převést výslednou reprezentaci lineárních omezení na frame pomocí převodní funkce.

3.4.5.1 Poznámka k použití operátoru widening

Operátor *widening* rozšiřuje interval hodnot proměnných na největší možnou velikost. Znamená to, že v místě za smyčkou se neobjeví omezení udávající přesnou hodnotu proměnné, jaká byla v její poslední iteraci. Objeví se zde omezení, že hodnota je menší případně větší než poslední konkrétní hodnota. Ve Výpisu 8 je uveden příklad.

Výpis 8: Příklad pro operátor *widening*

```
x = 0;
while(x < 10){
      x++;
}
```

Za celou smyčkou (tedy za }), se objeví omezení $x \geq 10$ a ne očekávané omezení $x = 10$. Toto je vlastnost operátoru *widening*. Za smyčkou tedy dojde k určité over aproximaci hodnot. Tento problém by se dal řešit použitím operátoru *narrowing*, který má opačnou úlohu než operátor *widening*. Snaží se zúžit interval možných hodnot proměnné. V této práci nebylo použití operátoru *narrowing* požadováno a proto není implementován.

Pro zlepšení odhadu operátoru *widening* je možné několik iterací čekat předtím než se bude aplikovat v uzlu smyčky. Počet čekání je možno zadat jako argument programu. Pokud není tento parametr zadán, počet iterací čekání bude standardně nula a operátor se aplikuje, až bude objevena celá smyčka.

Pseudokód výpočtu operace *widening* je uveden v Příloze 1 pod názvem Pseudokód LOOP-WIDENING-1.

Pseudokód transformace v uzlu smyčky je uveden v Příloze 1 pod názvem Pseudokód LOOP-2.

3.5 Analýza - provedení analýzy

V Podkapitolách 3.3 a 3.4 byly popsány algoritmy převedení vstupního programu do vnitřní reprezentace pomocí orientovaného grafu a také transformace, které se provádí ve všech jeho uzlech. Byly tak vysvětleny všechny výpočty nutné pro provedení automatického nalezení lineárních vztahů mezi proměnnými v programu. Nyní zbývá jen popsat logiku programu provádějící samotnou analýzu.

Program provádějící analýzu začíná ve vstupním bodě programu, tedy funkci `main`. Nejprve se zpracují a ověří argumenty příkazové řádky. V druhém kroku se provede samotná analýza vstupního programu. Nakonec se provede prezentace výsledků buď jednoduchým výpisem výsledků na standardní výstup, nebo podrobně do XML souboru.

Pseudokód zpracování programu je uveden v Příloze I pod názvem Pseudokód MAIN-1.

Hlavní algoritmus analýzy je implementován ve třídě `Analyzer`, konkrétně v metodě `Analyzer::run()`. Konstruktoru třídy `Analyzer` se předá jméno souboru, ve kterém je uložen vstupní program. Metoda provádí následující posloupnost akcí:

1. Test zda vstupní soubor s programem existuje. Pokud ne, program skončí.
2. Ověření syntaxe a sémantiky vstupního programu a načtení informací o proměnných ve třídě `SyntaxAnalyzer`.
    a. Pokud vstupní program neobsahuje proměnné, program skončí.
3. Převedení programu do formy abstraktního syntaktického stromu ve třídě `ProgramParser`
4. Vytvoření orientovaného grafu z AST ve třídě `FlowchartCreator`.
5. Provedení analýzy nad grafem vstupního programu v metodě `Flowchart::process()`.

Pseudokód zpracování je uveden v Příloze I pod názvem Pseudokód ANALYZE-RUN-1.

Analýza v metodě `Flowchart::process()` prochází seznam uzlů v iteracích a pro každý uzel provede metodu `Vertex::process()`. Návratová hodnota z této metody je objekt typu `bool`. Návratová hodnota:

- `true` znamená, že transformace v uzlu v aktuální iteraci vypočetla lineární omezení shodná s omezeními vypočtenými v předchozí iteraci. Tedy transformace v uzlu je stabilní.
- `false` znamená, že nově vypočtená lineární omezení v aktuální iteraci se liší od omezení vypočítaných v předchozí iteraci. Transformace v uzlu není stabilní.

Pokud v aktuální iteraci byla návratová hodnota funkcí `Vertex::process()` pro každý uzel `true`, bylo dosaženo stabilizace transformací. Pokud se dlouho nedaří dosáhnout stabilizace transformací, je implementována zarážka, která po určitém počtu kroků analýzu zastaví. V prezentaci výsledků bude potom uvedeno, v kolika iteracích byla celá analýza provedena.

Pseudokód analýzy je uveden v Příloze I pod názvem Pseudokód ANALYZE-GRAPH-1.

3.6 Prezentace výsledků

Součástí návrhu programu je také vhodná prezentace výsledných lineárních omezení, která jsou asociovaná s každou hranou programu. Program by měl vydat nejen informace o lineárních omezeních ale také o struktuře grafu. Byly proto zvoleny dva druhy výstupu:

- zjednodušený výpis na standardní výstup
- komplexní výstup do textového souboru ve formátu XML

3.6.1 XML výstup

Tato reprezentace je z obou možných typů výstupu nejpodrobnější. Obsahuje všechny informace o vstupních proměnných, o struktuře objektu flowchart a popis všech omezení asociovaných s každou hranou v objektu flowchart.

První část XML dokumentu je tvořena popisem proměnných. Každá proměnná je definována svým indexem a jménem. Např.:

```
<variables size="2">
	<variable index="0">variable1</variable>
	<variable index="1">variable2</variable>
</variables>
```

Označuje, že v programu jsou dvě proměnné *variable1* a *variable2*. Kde *variable1* má index 0 a *variable2* má index 1.

Druhá část výpisu je věnována popisu objektu flowchart. Nejprve je vypsán seznam uzlů spolu s odkazy na vstupní a výstupní hrany. Například:

```
<flowchart>
	<vertices size=”10”>
		…
		<vertex type="assign" id="5">
			<nonlinear>0</nonlinear>
			<invertible>1</invertible>
			<lhs>0</lhs>
			<rhs><vector dimension="3">
				<item id="0">1</item>
				<item id="1">0</item>
				<item id="2">4</item>
			</vector></rhs>
			<in-arc id="4"/>
			<out-arc id="7"/>
		</vertex>
		…
</vertices>
```

Přiklad výše ukazuje, že objekt flowchart obsahuje 10 uzlů. Uzel s číslem 5 je typu přiřazení. Toto přiřazení je lineární a invertibilní. Je ve tvaru $variable1 = variable1 + 4$. Element `lhs` označuje index proměnné na levé straně přiřazení, tedy *variable1*. Prvky vektoru elementu `rhs` jsou označeny indexem proměnné `id` a hodnotou tohoto prvku v přiřazení ve vstupním programu. Poslední prvek vektoru, zde s `id = 2`, označuje vždy konstantní složku lineárního přiřazení, která není asociovaná s žádnou proměnnou. Elementy `in-arc` resp. `out-arc` označují identifikátor vstupní resp. výstupní hrany.

Po výpisu všech uzlů následuje výpis všech hran spolu s lineárními omezeními mezi hodnotami proměnných ve vstupním programu. Například:

```
<arcs size="11">
       … <arc id="7">
              <restraints size="2">
                     <row id="0" operator="<=">
                            <lhs id="0">0</lhs>
                            <lhs id="1">-1</lhs>
                            <rhs id="2">0</rhs>
                     </row>
                     <row id="1" operator="<=">
                            <lhs id="0">-1</lhs>
                            <lhs id="1">2</lhs>
                            <rhs id="2">-6</rhs>
                     </row>
              </restraints>
       </arc> …
</arcs>
```

Přiklad výše ukazuje, že objekt flowchart obsahuje 11 hran. Hrana číslo 7 obsahuje dva řádky lineárních omezení typu menší nebo rovno v tomto tvaru:

$$-variable2 \leq 0;\ -variable1 + 2 * variable2 \leq -6$$

Element typu `lhs` představuje jeden řádek matice koeficientů lineárních omezení. V atributu `id` je uložen index proměnné, ke které koeficient, uvedený jako hodnota elementu, patří. Element `rhs` představuje pravou stranu omezení.

3.6.2 Zjednodušený výstup

Program umožňuje zobrazit výsledky analýzy ve zjednodušeném textovém výpisu na standardní výstup aplikace v příkazové řádce operačního systému Linux. Pro větší programy může být vhodné přesměrovat tento výstup do souboru. Příklad tohoto výpisu následuje:

```
Graph representation after 4/100 iterations
[vertex id=0 type=start]
start
[no input arcs]
[arc id=0 type=output]
 +a = 0
 +b = 0

[vertex id=1 type=assign]
a = +3
[arc id=0 type=input]
 +a = 0
 +b = 0
[arc id=1 type=output]
 +b = 0
 +a = 3
…
[vertex id=3 type=test]
 +a <= 10
[arc id=2 type=input]
 -a <= -3
-3a +b <= -9
[arc id=3 type=output-true]
```

```
-3a +b <= -9
 -a <= -3
 +a <= 10
[arc id=6 type=output-false]
-3a +b <= -9
 -a <= -10
…
```

Na začátku výpisu je informace, že analýza skončila po čtyřech iteracích ze sta. Došlo tedy ke stabilizaci analýzy. Dále vždy následuje odstavec popisující jeden uzel. Odstavce (uzly) jsou ve výpisu odděleny prázdným řádkem. Každý uzel `vertex` obsahuje výpis lineárních omezení pro každou hranu `arc`. Přitom nejprve se vypisují hrany vstupní `input` a pak výstupní `output`, `output-true` nebo `output-false`.

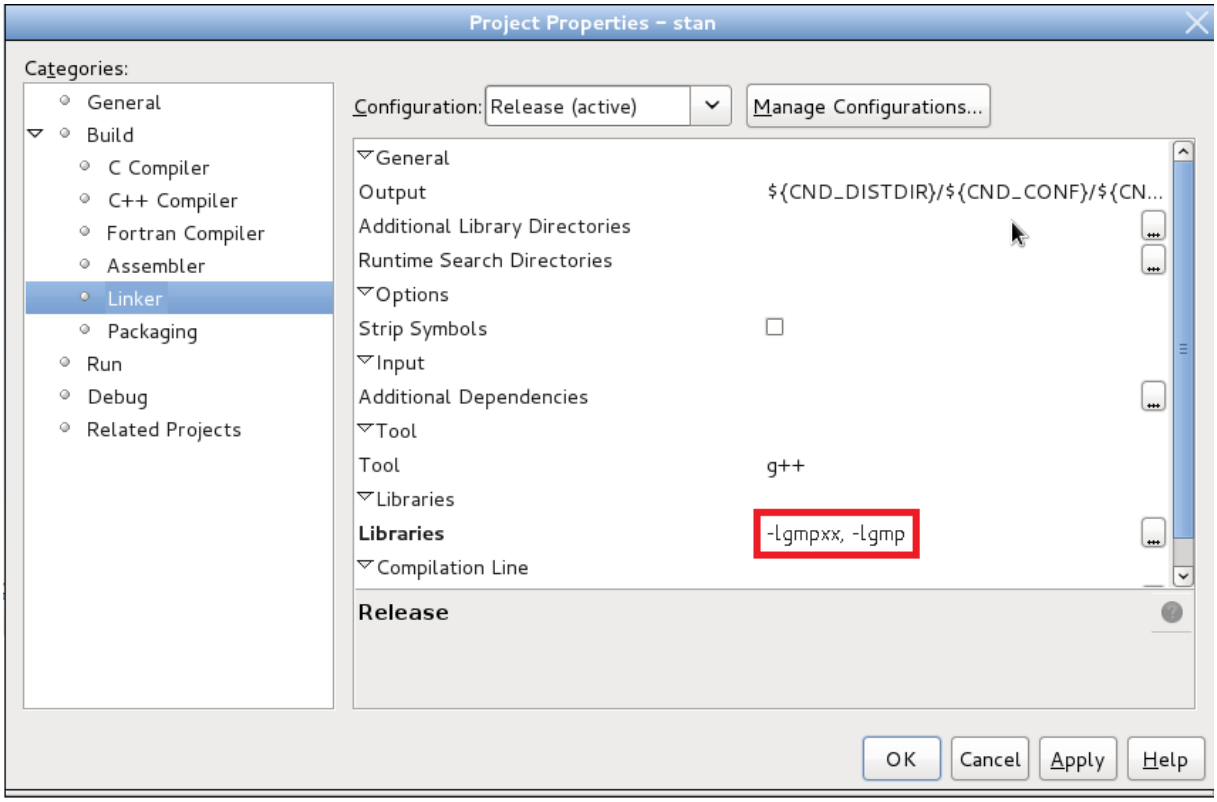
Zjednodušený výstup je určen pro rychlou verifikaci výsledků analýzy. Obsahuje všechny důležité informace pro jeho pochopení. Oproti tomu XML výstup je připraven zejména pro další strojové zpracování.

4 Implementace

Program je konzolová aplikace pro operační systém Linux. Byl vytvářen v operačním sytému Fedora verze 17 na 64 bitovém počítači.

Programovací jazyk byl zvolen C++. Pro programování bylo použito integrované vývojové prostředí *NetBeans*. Pro překlad využívají *NetBeans* program *gcc* verze 4.8.1. Pro kompilaci kódu si prostředí *NetBeans* vytváří vlastní `makefile` soubory.

Aby bylo možno program v prostředí *NetBeans* přeložit, je nutné mít na počítači nainstalovánu knihovnu GMP. *NetBeans* pak si pak samy knihovnu nalinkují. Nastavení lze najít ve vlastnostech projektu v menu `Properties->Build->Linker->Libraries`. Viz následující obrázek.



Obrázek 20: Nastavení knihovny GMP

5 Popis programu

Program implementující algoritmus pro automatické vyhledávání lineárních vztahů mezi proměnnými programu se jmenuje `stan` (zkratka pro *STatic ANalysis*). Jde o konzolovou aplikaci pro operační systém Linux, která dostane jako vstup zdrojový kód programu a provede nad ním výpočet popsaný v návrhu.

Argumenty příkazového řádku:

```
cmd> stan –i input.txt [-h] [-o] [-x xml file name] [-e number]
```

Přepínače mají následující význam:

`-h` zobrazí nápovědu.
`-i` následován jménem vstupního souboru s programem
`-x` následován jménem XML souboru, do kterého se na konci analýzy uloží výsledky
`-o` zajistí vypsání zjednodušených výsledků na standardní výstup
`-e` následovaný hodnotou celého čísla udávající počet iterací čekání předtím než se provede transformace v uzlu smyčky pomocí operátoru widening. Pokud program neobsahuje smyčky je tato hodnota zbytečná. Pokud nebude tento parametr zadán bude se aplikace operátoru provádět tehdy až budou objeveny obě vstupní hrany uzlu smyčky.

6 Testování

Pro účely testování programu byly vytvořeny testovací případy, které jsou součástí datové části této práce. Jsou uloženy ve složce `test` na přiloženém CD a jsou rozděleny do několika kategorií, podle toho co testují.

Jde o textové soubory se vstupním programem, které se předají jako argument programu pro automatické nalezení lineárních vztahů mezi proměnnými v programu.

6.1 Testování zpracování startovního uzlu

Složka: `test/tc1`

Testují správnou inicializaci proměnných ve startovním uzlu grafu. Složka obsahuje testovací případy s různými počty proměnných, kde:

- nejsou inicializovány žádné proměnné
- jsou inicializovány některé proměnné
- jsou inicializovány všechny proměnné

6.2 Testování zpracování uzlu přiřazení

Složka: `test/tc2`

Tetovací případy pro přiřazení s různými počty proměnných. Testují se:

- jednoduchá přiřazení – číslo nebo proměnná
- složitá přiřazení – na pravé straně je složitý výraz přiřazení
- invertibilní přiřazení
- neinvertibilní přiřazení
- nelineární přiřazení

6.3 Testování zpracování testovacího a slučovacího uzlu

Složka: `test/tc3`

Testovací případy pro testování podmínek testu a sloučení obou větví z testovacího uzlu do jedné větvě ve slučovacím uzlu. Testují se:

- jednoduché podmínky – srovnání s číslem nebo proměnnou
- složitější podmínky – na obou stranách přiřazení jsou složité výrazy
- podmínka typu rovnost
- podmínka typu nerovnost
- nelineární podmínka

6.4 Testování zpracování uzlu smyčky

Složka: `test/tc4`

Testují se smyčky v programu (`for`, `while`, `goto label`). Ověřuje se správná funkce operátoru *widening* v uzlu smyčky.

6.5 Testování návratu z programu

Složka: `test/tc5`

Testuje se správnost výpočtů při použití klíčového slova return v různých místech programu.

6.6 Testování složitějších programů

Složka: `test/tc6`

Zde jsou uvedeny zdrojové kódy složitějších programů a programů z Výpisů 1 a 6.

6.7 Poznámky k testovacím případům

6.7.1 Operátor widening

V Podkapitolách 2.5.1.4 a 2.5.1.6 byly popsány transformace v testovacím resp. slučovacím uzlu. Každá smyčka typu `while` a `for` obsahuje podmínku ukončení uzlu. Dle popsaných transformací probíhá analýza smyček grafu následujícím způsobem s určitou nepřesností. Např. smyčka:

```
a = 0;
while(a < 10){
      a++;
      // [point1]
}
// [point2]
```

Na konci analýzy pude platit pro první bod `[point1]` omezení $0 \leq a \leq 11$ a pro druhý bod `[point2]` omezení $a \geq 10$.

V prvním případě je nutné si uvědomit, že v testovacím uzlu se mění jakýkoli typ nerovnosti na nerovnost typu $\leq$ (viz Podkapitola 2.4.1.4). Podmínka ve smyčce `while` bude vnitřně reprezentovaná jako $a \leq 10$. Což bude mít za následek provedení jedenácté inkrementace proměnné *a* uvnitř smyčky.

V druhém případě je to vlastnost operátoru *widening* (viz Podkapitoly 2.5.1.6 a 3.4.5.1), která jednoduše zvětší interval omezení proměnné na nejvyšší možnou hodnotu, tedy $a \geq 10$. Analýza smyčky je tedy v tomto případě dle [1] správná.

U programů, které obsahují složitější smyčky je doporučeno pro zpřesnění odhadu používat zpožděné provádění operátoru *widening* pomocí parametru `-e pocet_iteraci`. Dle [1] Kapitola 4.5 je to jedna z možností, jak se přiblížit k přesnějším odhadům v případě této implementace operátoru.

6.7.2 Reprezentace mnohostěnu

V Kapitole 2.3.2 byly vysvětleny zvláštní případy mnohostěnu, tedy prázdný mnohostěn a mnohostěn přes celý *n*-rozměrný prostor. V textovém výpisu programu na standardní výstup jsou tyto dva případy značeny následovně:

- Prázdný mnohostěn jako `empty space`. V prostoru v daném místě programu není definován žádný mnohostěn. Znamená to že, výpočet se do tohoto místa programu nikdy nedostane.
- Mnohostěn pokrývající celý prostor jako `infinite space`. Všechny proměnné v tomto místě programu můžou nabývat jakékoli hodnoty. Jde například o výstupní hranu startovního uzlu, když nejsou do proměnných při deklaraci přiřazeny žádné hodnoty.

6.7.3 Eliminace proměnné ze soustavy

Operace eliminace proměnné ze soustavy nerovnic byla popsána v Kapitole 3.2.1.1. Její princip spočívá ve vytvoření nové nerovnice z každých dvou původních nerovnic, které mají součin koeficientů záporný, což může vést ke generování velkého počtu výstupních nerovnic. Nejhorší případ nastane, pokud je počet nerovnic sudé číslo *n* a koeficienty sloupce určeného k eliminaci jsou rozděleny na dvě poloviny, z nichž jedna obsahuje kladné koeficienty a druhá záporné. Tehdy je počet nově vzniklých nerovnic:

$$m = \left(n/2\right)^2$$

Je-li například při zpracovávání složitějšího programu počet nerovnic $n = 60$ bude výsledný počet nerovnic $m = 900$. Pokud by se takováto soustava znovu použila pro eliminaci sloupce a opět by nastal nejhorší případ, počet řádků by byl $m = 810000$ což je již vysoké číslo. Zde je nutno upozornit, že výslednou soustavu je nutno zjednodušit protože obsahuje redundantní nerovnice a další typy nepotřebných nerovnic.

Z testování vyplynulo, že pro větší počet proměnných (pět a více) a časté používání smyček a slučovacích a testovacích uzlů, kde se eliminace používá, nastává problém generování velké soustavy nerovnic velmi často. Přitom více než paměť je zatížen procesor počítače, protože hledá nové dvojice nerovnic a vypočítává z nich novou nerovnici a následně celou soustavu zjednodušuje.

Tato vlastnost programu je nepříjemná, protože program při výpočtu dlouhou dobu nereaguje. Ovšem je zde možnost, jak se tohoto problému zbavit. Jelikož výsledná soustava nerovnic obsahuje mnoho redundantních nerovnic, je možno s určitou ztrátou přesnosti některé nerovnice odstranit. V programu je implementována zarážka v podobě limitu počtu nerovnic. Pokud je tento limit překročen, automaticky se odstraní určité procento nerovnic z výsledné soustavy. Toto procento lze v kódu nastavit. Do textového i XML výstupu se potom vloží k uzlu, ve kterém se odstranění provedlo informace, že došlo ke ztrátě přesnosti ve výpočtu jeho transformace díky odstranění některých nerovnic.

7 Závěr

Cílem této práce byl návrh a implementace algoritmu pro automatické vyhledávání lineárních vztahů mezi hodnotami proměnných v programu, dle pokynů a postupů uvedených v [1]. Tohoto cíle bylo také dosaženo.

Implementovaný program je schopen zpracovat textový soubor se vstupním programem podle gramatiky navržené v Kapitole 3. Struktura souboru je následující: Nejprve jsou deklarovány a popřípadě inicializovány všechny proměnné programu. Pak následuje funkce nazvaná *main* obsahující celý výpočet, který je analyzován. Jiné funkce nejsou uvažovány. Všechny proměnné jsou pouze číselného typu, konkrétně jsou to racionální čísla. Proměnné jsou navíc pouze jednoduché, což znamená, že je nelze skládat do větších celků, jako jsou např. pole nebo objekty. Program je schopný vydat stejný výstup jako testovací implementace v [1] v Kapitole 6.

Analýza, popsaná v [1], a jakákoli jiná statická analýza je pouze přibližná analýza programu. Znamená to, že výsledky analýzy nemusí být v některých případech dostatečně přesné. Například se může stát, že program vyhodnotí omezení proměnné od nuly do plus nekonečna, ale reálné hodnoty, které můžou opravdu nastat, budou v menším intervalu. Toto chování je dáno zvolenou abstrakcí a konkrétními algoritmy. Někdy je totiž nutné ulevit od přesnosti výpočtu nad časovou a paměťovou složitostí algoritmu.

Nyní se otevírají další možnosti jak v oblasti automatického vyhledávání lineárních vztahů mezi proměnnými programu pokračovat. Prvním krokem by mohlo být přesnější zpracování smyček programu, použitím některé novější implementace operátoru *widening* (viz [6]) a následně použitím operátoru *narrowning* (viz [3]). Jelikož tato implementace algoritmu z [1] používá pro výpočty pouze racionální čísla, bylo by dále vhodné program rozšířit o zpracování více typů čísel, zejména pak celých čísel. Dalším krokem by mohlo být upravení programu pro zpracovávání vestavěných typů jazyka, polí a složitějších objektů. Poté by se místo jednoduchého, zde navrženého vstupního jazyka mohl použít některý z dnes používaných jazyků, což vyžaduje použití normy definující tento jazyk v přední části vstupní jednotky programu.

8 Použítá literatura

[1] Patric Cousot, Nicolas Halbwachs*: Automatic Discovery of linear Restraints among Variables of Program*. Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 84-97, 1978. ACM Press, New York

[2] Patrick Cousot, Radhia Cousot: *Static Determination of Dynamic Properties of Programs*. In Proceedings of the second international symposium on Programming, B. Robinet (Ed), Paris, France, pages 106—130, 13—15 April 1976, Dunod, Paris.

[3] Michael I. Schwartzbach: *Lecture Notes on Static Analysis*. BRICS, Department of Computer Science, University of Aarhus, Denmark

[4] Patrick Cousot: *Abstract Interpretation in a Nutshell*. Webová prezentace http://www.di.ens.fr/~cousot/AI/IntroAbsInt.html

[5] Thomas S. Ferguson: *Linear Programming, A Concise Introduction*

[6] Axel Simon, Liqian Chen: *Simple and Precise Widenings for H-Polyhedra*. Programming Languages and Systems. 8th Asian Symposium, APLAS 2010, Shanghai, China, pages 139-155, November 28 - December 1, 2010, Springer-Verlag Berlin

9 Příloha I

V této příloze jsou uvedeny pseudokódy algoritmů z kapitoly Analýza a návrh. U každého pseudokódu je uvedena jeho identifikace, pomocí které se na ni odkazuje text.

Pseudokód SIMPLIFY-RESTRAINTS-IRRELEVANT-1

```
function FtRSimplification::findIrrelevantInequalities(LinearSystem restraints){
        for(all inequalities r in restraints){
                if(vertexSaturations[r].saturations.allClear())
                        restraints.removeRow(r);
        }
}
```

Pseudokód SIMPLIFY-RESTRAINTS-EQUALITIES-2

```
function FtRSimplification::findEqualities(LinearSystem restraints){
        for(all inequalities r in restraints){
                if(vertexSaturations[r].saturations.allSet() and raySaturations[r].saturations.allSet())
                        restraints.convertRowToEquality(r);
        }
}
```

Pseudokód SIMPLIFY-RESTRAINTS-REDUNDANT-3

```
function FtRSimplification::findNeedlessInequalities(LinearSystem restraints){
        for(all pairs r1,r2 of restraints.inequalities){

                BitSet vXor = vertexSaturations[r1].saturations
                                XOR vertexSaturations[r2].saturations;
                BitSet rXor = raySaturations[r1].saturations XOR raySaturations[r2].saturations;

                //Both rows are saturated by same Frame items
                if(vXor.allClear() AND rXor.allClear()){ restraints.removeRow(r1); }
                BitSet v1And = vertexSaturations[r1].saturations AND vXor;
                BitSet r1And = raySaturations[r1].saturations AND rXor;
                if(v1And.allClear() AND r1And.allClear()){
                        //First row r1 is saturated by subset
                        //of items of second row r2
                        restraints.removeRow(r1);
                }

                BitSet v2And = vertexSaturations[r2].saturations AND vXor;
                BitSet r2And = raySaturations[r2].saturations AND rXor;
                //Second row r2 is saturated by subset of items of First row r1
                if(v2And.allClear() AND r2And.allClear()){
                        restraints.removeRow(r2);
                }
        }
}
```

Pseudokód SIMPLIFY-RESTRAINTS-4

```
function FtRSimplification::simplifyRestraints(LinearSystem restraints, Frame frame){
        // Fills arrays vertexSaturations and raySaturations
        computeSaturations(restraints,frame);
        findIrrelevantInequalities(restraints);
        findEqualities(restraints);
        findNeedlessInequalities(restraints);
        return restraints;
}
```

Pseudokód SIMPLIFY-FRAME-IRRELEVANT-1

```
function RtFSimplification::findIrrelevantItems(Frame frame){
        for(all vertices s of frame){
                if(rowSaturationsVerticesIneq[s].saturations.allClear())
                        frame.removeVertex(s);

        }

        for(all rays r of frame){
                if(rowSaturationsRaysIneq[r].saturations.allClear())
                        frame.removeRay(r);
        }
}
```

Pseudokód SIMPLIFY-FRAME-LINES-2

```
function RtFSimplification::findLines(Frame frame){
        for(all rays r of frame){
                if(rowSaturationsRaysAll[r].saturations.allSet()){
                        frame.addLine(r);
                        frame.removeRay(r);
                }
        }
}
```

Pseudokód SIMPLIFY-FRAME-REDUNDANT-3

```
function RtFSimplification::findNeedlessVertices(Frame frame){
        for(all pairs s1,s2 of vertices){
                BitSet vXor = rowSaturationsVerticesIneq[s1].saturations
                        XOR rowSaturationsVerticesIneq[s2].saturations;

                //Both vertices saturate same lines
                if(vXor.allClear()){ frame.removeVertex(s1);}

                BitSet vAnd = rowSaturationsVerticesIneq [r1].saturations AND vXor
                if(vAnd.allClear())
                        //First vertex saturates subset of rows of second vertex
                        frame.removeVertex(s1);
                }
        }
}
```

Pseudokód SIMPLIFY-FRAME-4

```
function RtFSimplification::simplifyFrame(LinearSystem restraint, Frame frame){
        // Fills arrays rowSaturationsVerticesIneq,
        // rowSaturationsRaysIneq and rowSaturationsRaysAll
        computeSatutations(frame,restraints);
        findIrrelevantItems(frane);
        findLines(frame);
        findNeedlessVertices(frame);
        findNeedlessVertices(frame);
        return frame;
}
```

Pseudokód MAIN-1

```
function main(Arguments args){
        string inputFile;
        bool toXml, toOut;

        (inputFile, toXml, toOut) = parseArguments(args);

        Analyzer analyzer = new Analyzer(inputFile);
        // Does the analysis
        analyzer.run();

        if(toOut) analyzer.printToStandardOutput();
        if(toXml) analyzer.printToXml();
}
```

Pseudokód ANALYZE-RUN-1

```
function Analyzer::run(){
        // Run syntax analyzer
        SyntaxAnalyzer sa = new SyntaxAnalyzer(fileName);
        sa.analyze();

        if(variables.getCount() == 0) return;

        // Run program parser – parse program
        ProgramParser pa = new ProgramParser (fileName);
        pa.parse();

        // Create flowchart
        FlowchartCreator fc = new FlowchartCreator();
        Flowchart flowchart = fc.create();

        // Do analysis
        flowchart.process();
}
```

Pseudokód ANALYZE-GRAPH-1

```
function Flowchart::process(){
        bool iterResult, notComplete = false;
        int iterations = 0;

        while(iterations < MAX_ITERATIONS){
                iterResult = true;
                iterations++;
                for(vertexs in all vertices of flowchart){
                        if(false = vertex.process()) iterResult = false;
                }

                if(iterResult = true) break;
        }
        flowchartIterations = interations;
}
```

Pseudokód START-1

```
function StartTransform::transform(Vector initVector, Matrix initMatrix)
        vector lines;
        Vertex vertex;
        bool initialized = false;

        for(i = 0; i < variables.count(); i++){
                // If initVector[i] is false, variable with index i was not initialized. Create line
                if(initVector[i] = false){
                        lines.add(create line with value 1 on position i, otherwise 0);
                        continue;
                }

                // initVector[i] is true. Variable with index i was initialized.Find its value in IM
                value = initMatrix[i][ lastColumn];
                vertex.setItem[i]= value;
                initialized = true;
        }

        Frame outputFrame;
        outputFrame.addLines(lines);
        outputFrame.addVertex(vertex);
        outputPolyhedron.setFrame(outputFrame);

        if(initialized) outputPolyhedron.frameToRestraints();
        else outputPolyhedron.setRestraints(empty restraints);
}
```

Pseudokód ASSIGN-NONLINEAR-1

```
function AssignTransform::processNonLinear(Polyhedron inputPolyhedron, int leftSideIndex){
        LinearSystem restraints = inputPolyhedron.getRestraints();
        Frame frame = inputPolyhedron.getFrame();

        // Eliminate variable on left side of assignment
        restraints.eliminateVariable(leftSideIndex);
        // Add line with 1 on index of left side variable
        frame.addLine(create line with value 1 on position leftSideIndex, otherwise 0);
        // Simplify frame
        frame.simplifyByRestraints(restraints);

        Polyhedron outputPolyhedron = new Polyhedron(restraints,frame);
        return outputPolyhedron;
}
```

Pseudokód ASSIGN-FRAME-2

```
function AssignTransform::computeFrame(Polyhedron inputPolyhedron,
                Vector rightSideVector, int leftSideIndex){
        Frame frame = inputPolyhedron.getFrame();
        for(each s ∈ frame.getVertices()){
                // Does the dot operation only on first size(s) items
                // of rightSideVector
                dot = s.dot(rightSideVector,size(s)) + rightSideVector.lastItem;
                s.setItem(leftSideIndex,dot);
        }

        for(each r ∈ frame.getRays()){
                dot = r.dot(rightSideVector,size(r));
                r.setItem(leftSideIndex,dot);
        }

        for(each d ∈ frame.getLines()){
                dot = d.dot(rightSideVector,size(d));
                d.setItem(leftSideIndex,dot);
        }

        return frame;
}
```

Pseudokód ASSIGN-INVERTIBLE-3

```
function AssignTransform::processInvertible(Polyhedron inputPolyhedron,
                Vector rightSideVector, int leftSideIndex){
        LinearSystem restraints = inputPolyhedron.getRestraints();
        // Does the substitution of rightSideVector accrding to item at leftSideIndex. Point 2) above
        Vector subst = (substitute rightSideVector at leftSideIndex position);
        // Substitute vector to restraints at column defined by leftSideIndex. Point 3) above
        restraints.substitute(leftSideIndex,vector);
        return restraints;
}
```

Pseudokód ASSIGN-NONINVERTIBLE-4

```
function AssignTransform::processNonInveritble(Polyhedron inputPolyhedron,
                Vector rightSideVector, int leftSideIndex){
        LinearSystem restraints = inputPolyhedron.getRestraints();
        restraints.eliminateVariable(leftSideIndex);
        restraints.addRestraint(rightSideVector);
        return restraints;
}
```

Pseudokód ASSIGN-5

```
function AssignTransform::transform(Polyhedron inputPolyhedron, int leftSideIndex,
                Vector rightSideVector, bool nonLinear, bool invertible){

        Polyhedron outputPolyhedron;
        if(nonLinear){
                outputPolyhedron  = processNonLinear(inPoly,outPoly,lhs)
                return outputPolyhedron;
        }

        // Computes output frame
        Frame frame = computeFrame(inputPolyhedron, leftSideIndex, rightSideVector);
        LinearSystem restraints;

        if(invertible) restraints = processInvertible(inputPolyhedron, leftSideIndex, rightSideVector);
        else restraints = processNonInveritble(inputPolyhedron, leftSideIndex, rightSideVector);

        outputPolyhedron = new Polyhedron(frame,restraints);
}
```

Pseudokód TEST-INTERSECTION-1

```
function TestTransform::computeIntersectOfPandH(Polyhedron inputPolyhedron,
                Vector condition){
        Frame resultFrame;
        // Condition is of type ax=b
        for(all pairs s1,s2 of vertices){
                if(s1 comply condition){ resultFrame.addVertex(s1); continue; }
                if(s2 comply condition){ resultFrame.addVertex(s2); continue; }
                lambda = (b-a*s2)/(a*s1-a*s2);
                if(0 ≤ lambda ≤ 1) resultFrame.addVertex(lambda*s1+(1- lambda)*s2);
        }

        for(all pairs s1 of vertices and r1 of rays)
                mi = (b-as1)/ar1;
                if(mi ≥ 0) resultFrame.addVertex(s1+mi*r1);

        for(all pairs s1 of vertices and d1 of lines){
                ni = (b-as1)/(ad1);
                resultFrame.addVertex(s1+ni*d1);
        }

        for(all pairs r1,r2 of rays){
                if(r1 comply condition){resultFrame.addRay(r1); continue; }
                if(r2 comply condition){resultFrame.addRay(r2); continue; }
                mi = (a*r1)/(a*r2);
                if(mi ≥ 0) resultFrame.addRay(r1+mi*r2);
        }

        for(all pairs r1 of rays and d1 of lines) resultFrame.addRay(r1-(a*r1/a*d1);
        for(all pairs d1,d2 of lines) resultFrame.addLine(d1-(a*d1/a*d2);

        return resultFrame;
}
```

Pseudokód TEST-INEQUALITY-2

```
function TestTransform::processInequality(Frame inputFrame,
                Frame tempFrame, Vector condition){
        Frame outFrameTrue, outFrameFalse;
        // Condition is of type ax=b
        for(each s ∈ inputFrame.getVertices()){
                if(a*s < b) outFrameTrue.addVertex(s);
                else if(a*s > b) outFrameFalse.addVertex(s);
        }

        for(each r ∈ inFrame.getRays()){
                if(a*r < 0) outFrameTrue.addRay(r);
                else if(a*r > 0) outFrameFalse.addRay(r);
        }

        for(each d ∈ inFrame.getLines()){
                if(a*d < 0) outFrameTrue.addRayUnion(d);
                else if(a*d > b) outFrameFalse.addRayUnion(d);

                if(a*(-d) > 0) outFrameTrue.addRayUnion(-d);
                else if(a*(-d) < b) outFrameFalse.addRayUnion(-d);

                outFrameTrue.addLine(d);
                outFrameFalse.addLine(d);
        }

        // tempFrame is frame of  (P union H)
        outFrameTrue.union(tempFrame);
        outFrameFalse.union(tempFrame);

        return outFrameTrue, outFrameFalse;
}
```

Pseudokód TEST-3

```
function TestTransform::transform(Polyhedron inputPolyhedron, Vector condition,
        bool nonLinear, bool equality){

    Polyhedron outputTrue, outputFalse:
    Frame inputFrame = inputPolyhedron.getFrame();
    cond = conditionIsTrue(inputFrame, condition);
    if(equality && cond){
        outputTrue = inputPolyhedron,
        outputFalse = 0;
        return;
    }

    Frame tempFrame = computeIntersectOfPandH(inputPolyhedron, condition)
    if(equality){
        outputTrue = new Polyhedron(tempFrame,inPolyhedron.getRestraints());
        outputFalse = input;
        return;
    }

    (outputTrue, outputFalse) = processInequality(inputFrame, tempFrame, condition)
    outputTrue = new Polyhedron(outputTrue,0);
    outputFalse = new Polyhedron(outputFalse,0);
    outputTrue.frameToRestraints();
    outputFalse.frameToRestraints();
}
```

Pseudokód JUNCTION-1

```
function JunctionTransform::transform(Polyhedron[] inputPolyhedrons){
    Polyhedron outputPolyhedron;
    if(inPolyhedrons.size() = 1){
        outputPolyhedron = inputPolyhedrons [0];
        reutrn;
    }

    Polyhedron unionFrame = inputPolyhedrons [0].getFrame();
    Polyhedron toUnionFrame = inputPolyhedrons [1].getFrame();

    unionFrame.vertices.union(toUnionFrame.vertices);
    unionFrame.rays.union(toUnionFrame.rays);
    unionFrame.lines.union(toUnionFrame.lines);

    LinearSystem restraints = incorporateFrameToRestraints(
        inputPolyhedrons [0].getRestraints().inputPolyhedrons [1].getFrame());

    unionFrame.simplifyByRestraints(restraints);
    outputPolyhedron = new Polyhedron(unionFrame, restraints);
}
```

Pseudokód LOOP-WIDENING-1

```
function LoopJunctionTransform::widening(Polyhedron lastOutputPolyhedron,
                Polyhedron convexHull){
        LinearSystem resultRestraints;
        LinearSystem restraints = lastOutputPolyhedron.getRestraints();
        Frame frame = convexHull.getFrame();
        for(each i ∈ restraints){
                solveAll = true;
                for(all s in frame.vertices){
                        if(!solveRestraint(restraints[i],s)){ solveAll = false; break; }
                }

                if(solveAll) resultRestraints.addRestraint(restraints[i]);
        }
        return resultRestraints;
}
```

Pseudokód LOOP-2

```
function LoopJunctionTransform::transform(Polyhedron[] inputPolyhedrons,
                Polyhedron latsOutputPolyhedron){
        Polyhedron outputPolyhedron, convexHull;
        if(inputPolyhedrons.size() < 2) convexHull = inputPolyhedrons[0];
        else convexHull = junctionTransform(inputPolyhedrons);

        if(latsOutputPolyhedron == 0){
                outputPolyhedron = convexHull;
                return;
        }

        LinearSystem restraints = widening(latsOutputPolyhedron, convexHull);
        outputPolyhedron.setRestraints(resultRestraints);
        outputPolyhedron.restraintsToFrame();
}
```

Pseudokód LS-ELIMINATE

```
function LinearSystem::project(int c0){
        if(coefficients.isZeroColumn(c0)){
                coefficients.removeColumn(c0);
                return;
        }

        if(this.containsEqualities()) this.convertToInequalities();

        Matrix result;
        for(each pair of rows r1 and r2 in coefficients){
                mpq_class item1 = coefficients.get(r1, c0);
                mpq_class item2 = coefficients.get(r2, c0);

                if(item1 == 0) { result.put(r1); continue; }
                if(item2 == 0) { result.put(r2); continue; }

                if(item1*item2 > 0) continue;

                mpq_class item1Abs = abs(item1);
                mpq_class item2Abs = abs(item2);
                Vector newRow;
                for(all columns c){
                        newRow[c] = item1Abs*r2[c] + item2Abs*r1[c];
                }
                result.addRow(newRow);
        }

        reuslt.removeColumn(c0);
        coefficients = result;
        return;
}
```