Katedra informatiky
Fakulta matematiky, fyziky a informatiky
Univerzita Komenskho

Vlastnosti programov so stratégiami

(Diplomová práca)

Bc. Matej Vince

Vedci: RNDr. Peter Borovanský, PhD.

Bratislava 2009

Vlastnosti programov so stratégiami

Diplomová práca

Bc. Matej Vince

Univerzita Komenského
Fakulta matematiky, fyziky a informatiky
Katedra informatiky

Študijný odbor: 9.2.1 Informatika

Vedúci diplomovej práce: RNDr. Peter Borovanský, PhD.

Bratislava 2009

Čestne prehlasujem, že som túto diplomovú prácu vypracoval samostatne s použitím citovaných zdrojov.

Ďakujem svojmu diplomovému vedúcemu RNDr. Petrovi Borovanskému, PhD. za cenné rady, usmernenia a trpezlivosť. Svojmu kamarátovi Bc. Richardovi Korenčiakovi za ochotu počúvať moje úvahy. A v neposlednom rade chcem poďakovať svojmu bratovi Michalovi Vincemu za zaučenie a pomoc pri vykresľovaní v jazyku Haskell.

Abstrakt

Vince Matej, Vlastnosti programov so stratégiami. Diplomová práca. Katedra informatiky. Fakulta matematiky, fyziky a informatiky. Univerzita Komenského. Vedúci diplomovej práce: RNDr. Peter Borovanský, PhD. Bratislava 2009.

Táto práca sa zaoberá systémami na prepisovanie termov a pojednáva o výpočtoch týchto systémov riadených stratégiami. Zaoberá sa problémom zastavenia pre prepisovacie systémy pri stratégii innermost. Opisuje metódu pre dôkaz zastavenia a jej implementáciu. Popisuje tiež charakteristiku a implementáciu pomocných problémov potrebných pre jej riešenie.

Kľúčové slová: systémy na prepisovanie termov, problém zastavenia, implementácia nerozhodnuteľných problémov

Abstract

This master thesis studies term rewriting systems and their strategy controlled computations. Termination problem of rewriting systems with innermost strategy is discussed, together with a brief overview of termination proof method and its implementation. Moreover, additional problems concerning the proof method are characterized and their implementation is proposed.

Obsah

1 Úvod **3**
1.1 Cieľ 4
1.2 Štruktúra práce 4

2 Základné pojmy a definície **6**
2.1 Termy a prepisovací systém 6
2.2 Prepisovanie termov 7
2.3 Prepisovanie podľa stratégií 8

3 Popis a riešenie problému **12**
3.1 Problém zastavenia pre prepisovací systém 12
3.2 Popis zvolenej metódy 14
3.2.1 Dokazovanie pomocou indukcie 14
3.2.2 Použité operácie a podmienky 15
3.2.3 Popis procedúry 19

4 Návrh a analýza riešenia **21**
4.1 Voľba jazyka a prostredia 21
4.1.1 Jazyk 21
4.1.2 Prostredie 22
4.2 Návrh dátovej štruktúry 23

5 Implementácia **27**

5.1 Popis pomocných funkcií 27

5.2 Hlavné časti programu 29

5.2.1 Abstrahovanie 29

5.2.2 Zužovanie 30

5.2.3 Beh dôkazu 32

6 Súvisiace problémy: popis a implementácia **39**

6.1 Terminácia a použiteľné pravidlá 39

6.1.1 Použiteľné pravidlá 39

6.1.2 Terminácia 41

6.1.3 Usporiadanie 41

6.1.4 Splniteľnosť podmienok s normálnymi formami 43

6.1.5 Disunifikácia 47

6.1.6 Vizualizácia 52

7 Záver **53**

Bibliography **55**

Prílohy **59**

Kapitola 1

Úvod

V dnešnom svete si nedokážeme predstaviť život bez počítačov. Počítače nám uľahčujú každodenný život, a preto sa ich používanie stalo pre väčšinu z nás každodennou súčasťou. To čo robí z počítačov použiteľné nástroje je softvér.

Pri vytvaraní softvéru sa používajú rôzne programovacie jazyky. Tieto jazyky možno rozdeliť do viacerých paradigiem. Najrozšírenejšou paradigmou je imperatívne programovanie. Pri imperatívnom programovaní popisujeme ako sa má výpočet správať na základe postupnosti príkazov a tým meníme stav prostredia. Sem patria všeobecne známe jazyky ako sú Java, C#, .NET, Delphi alebo C++. Protipólom k takémuto prístupu je deklaratívne programovanie. Pri deklaratívnom programovaní nepopisujeme ako sa má výpočet správať, ale čo má program dosiahnuť. Deklaratívne programovanie v sebe zahŕňa viacero prístupov. Najznámejšie sú logické a funkcionálne programovanie. V logickom programovaní program pomocou pravidiel vyhodnocuje vstupný dotaz. Príkladom logického jazyka je Prolog. Pri funkcionálnych jazykoch je celý program popísaný funkciami. Na rozdiel od imperatívnych jazykov, funkcie nie sú závislé na stave výpočtu. K jednému vstupu vracajú rovnaký výstup. Medzi funkcionálne programovacie jazyky patrí napríklad Haskell.

Osobitou skupinou je prepisovanie termov. Pri tomto prístupe máme súbor pravidiel, podľa ktorých prepisujeme vstupný term. Výsledkom je term, na ktorý sa už nedá použiť žiadne pravidlo. Prepisovanie termov je používané najmä v nástrojoch na dokazovanie teorém a pri interpretácii funkcií $\lambda-$kalkulu. Používa sa na výpočty aj v niektorých deklaratívnych jazykoch.

Pri takomto použití prepisovania je dôležité vedieť, či výpočet skončí. Problém zastavenia pre prepisovacie systémy ale nie je rozhodnuteľný. Napriek tejto skutočnosti existuje výskum, ktorý sa zaoberá touto oblasťou. Úlohou je pokryť čo najväčšie množstvo prípadov, v ktorých vieme rozhodovať.

1.1 Cieľ

Cieľom tejto diplomovej práce je preskúmať oblasť prepisovania termov a možnosti riešenia rozhodnuteľnosti zastavenia prepisovacieho systému v špeciálnom prípade, keď je výpočet riadený istou stratégiou. Keďže vo všeobecnosti ide o nerozhodnuteľný problém, tiež preštudovať nástroje, ktoré tento problém riešia. Naimplementujeme poloautomatický nástroj vhodný na ďalšie rozšírenie. Zvolíme vhodný programovací jazyk na implementáciu, v ktorom sa dá jednoducho zachytiť podstata výpočtu prepisovacích systémov.

1.2 Štruktúra práce

Práca sa delí na 7 kapitol, ktoré na seba nadväzujú. V kapitole 2 podávame prehľad o základných pojmoch a definíciách z oblasti prepisovania termov. Kapitola 3 rozoberá možnosti riešenia problému zastavenia prepisovacieho systému. Vysvetľujeme v nej ideu metódy, ktorú sme implementovali. V nasledujúcej kapitole sa zaoberáme návrhom riešenia. Prvá časť zahŕňa voľbu programovacieho jazyka a prostredia pre implelentáciu. Druhá uvádza dátové

štruktúry použité v implementácii. Kľúčovým predmetom kapitoly 5 je implementácia hlavných operácií metódy. Súvisiace podproblémy, ktoré si riešenie halsvného problému vyžadovalo, popisujeme v kapitole 6. Každý podproblém sme vysvetlili pre potreby našej metódy a popisujeme jeho implementácia v našom riešení.

Kapitola 2

Základné pojmy a definície

2.1 Termy a prepisovací systém

Definícia 2.1. Term *Nech* $\mathcal{F}$ *je množina funkčných symbolov a* $\mathcal{X}$ *je množina premenných, potom*

- $\forall x, x \in \mathcal{X}$, *je term.*
- $\forall f|0, f \in \mathcal{F}$, *(každá konštanta, t.j. 0-árny funkčný symbol) je term.*
- *Ak* t_1, t_2, ..., t_n *sú termy a* $f|n \in \mathcal{F}$ *je n-árny funkčný symbol potom výraz* $f(t_1, t_2, ..., t_n)$ *je term.*

Množinu všetkých termov nad premennými $\mathcal{X}$ a funkčnými symbolmi $\mathcal{F}$ označujeme $\mathcal{T}(\mathcal{F}, \mathcal{X})$

Definícia 2.2. Prepisovacie pravidlo

$$l \to r$$

kde:

- $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$.

- $Var(l) \subseteq Var(r)$ - *tzn. každá premenná termu r je premennou aj termu l.*

Definícia 2.3. Prepisovací systém R *(alebo systém na prepisovanie termov R) je množina prepisovacích pravidiel:*

$$R = \{l_1, \rightarrow r_1, l_2 \rightarrow r_2, ..., l_n \rightarrow r_n\}$$

2.2 Prepisovanie termov

Definícia 2.4. Substitúcia

$$t\,(x \leftarrow s)$$

znamená, že v terme t nahradíme každý výskyt premennej x za term s.
Pri substitúcii za viaceré premenné:

$$t\,(x_1 \leftarrow s_1, x_2 \leftarrow s_2, ...x_n \leftarrow s_n)$$

nahradzujeme tieto premenné postupne. Substitúciu budeme označovať ako θ.

Definícia 2.5. Unifikácia termov *t a t' znamená nájsť takú substitúciu θ, aby platilo:*

$$t\theta = t'\theta$$

Hovoríme, že termy t a t' sú unifikovateľné, ak pre ne existuje takáto substitúcia.

Pri prepisovaní, na rozdiel od substitúcie, nahradzujeme celý podterm a nahradzujeme len jeden jeho výskyt. Ak chceme prepísať term t podľa pravidla $l \rightarrow r$, tak musí existovať podterm s termu t a substitúcia θ taká, že:

$$l\theta = s$$

Inými slovami, ak chceme použiť nejaké pravidlo, tak musíme vedieť dosadiť za jeho premenné tak, aby jeho ľavá strana bola podtermom t.

Definícia 2.6. Reláciu krok odvodenia *v prepisovacom systéme R (pomocou pravidla $l \to r$) definujeme:*

$$t \to_R t' \text{ (t sa prepíše, resp. redukuje na } t'\text{)}$$

kde:

- *s je podterm termu t,*
- *exituje substitúcia θ taká, že $l\theta = s$ a*
- *term t' dostaneme nahradením podtermu s v terme t pomocou $r\theta$.*

Term s nazývame redex.

Definícia 2.7. Ireducibilný term *t je taký, že neexistuje term t', pre ktorý by platilo:*

$$t \to_R t'$$

Inak povedané je to taký term, ktorý v danom systéme nevieme prepísať podľa žiadneho pravidla. Takýto term sa nazýva normálnou formou.

2.3 Prepisovanie podľa stratégií

Príklad 2.1. Uvažujme prepisovací systém *R s nasledujúcou množinou pravidiel*

$f(g(x)) \to a$
$g(x) \to f(g(x))$

Prepisovanie v systéme R začínajúce z termu $f(g(x))$ môže vyzerať nasledovne:

- $f(g(x)) \to_R a$
- $f(g(x)) \to_R f(f(g(x))) \to_R f(f(f(g(x)))) \to_R f(f(a))$
- $f(g(x)) \to_R f(f(g(x))) \to_R f(f(f(g(x)))) \to_R f(f(f(f(g(x))))) \ldots$

V prvom prípade sme použili prvé pravidlo a dostali sme term v normálnej forme. V druhom prípade sme najprv použili 2-krát druhé a potom raz prvé pravidlo. V poslednom prípade sme používali len druhé pravidlo. Tu je vidieť, že prepisovanie je nedeterministické a voľba rôznych pravidiel môže viesť k rôznym výsledkom. Tiež vidíme, že používanie niektorých pravidiel môže viesť k nekonečnému výpočtu.

Na kontrolu behu výpočtu sa preto zaviedli stratégie. Stratégie určujú, na ktorých pozíciách sa majú termy redukovať. Medzi najpoužívanejšie stratégie patrí *innermost* stratégia, kedy sa snažíme zredukovať, čo najvnútornejšie podtermy. Term redukujeme, až keď sa všetky jeho podtermy nedajú redukovať. Protipólom k *innermost* stratégii je stratégia *outermost*. Tu sa snažíme zredukovať najvonkajšejší term. Pokiaľ sa redukovať nedá, skúšame redukovať vnútornejšie termy.

Definícia 2.8. Innermost a outermost stratégia: *Keď R je prepisovací systém a pre každý term $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ platí $t \to_R t'$ na pozícii p. Potom definujeme nasledovne prepisovanie pomocou stratégie*

- *innermost: každá pozícia p', na ktorej sa mohol term t prepísať, nemá prefix p.*
- *outermost: každá pozícia p', na ktorej sa mohol term t prepísať, nie je prefixom p.*

V predchádzajúcom prípade nás použitie *innermost* stratégie (posledný prípad) priviedlo k nekonečnému výpočtu, zatiaľ čo pri prepisovaní podľa *outermost* stratégie (prvý prípad) skončilo hneď po prvom kroku. Takže skončenie pri jednej stratégii neznamená skončenie pri inej.

Príklad 2.2. Zoberme prepisovací systém R_2 *s množinou pravidiel*

$$
\begin{aligned}
&f(0,1,x) \to f(x,x,x)\\
&g(x,y) \to x\\
&g(x,y) \to y
\end{aligned}
$$

Začínajúc z termu $f(g(0,1), g(0,1), g(0,1))$, si ukážeme prepisovanie podľa innermost (prepisujeme podčiarknutý redex):

$$
\begin{aligned}
&f\left(\underline{g(0,1)}, g(0,1), g(0,1)\right) \to_{R_2}\\
&f\left(0, \underline{g(0,1)}, g(0,1)\right) \to_{R_2}\\
&f\left(0, 1, \underline{g(0,1)}\right) \to_{R_2}\\
&\underline{f(0,1,0)} \to_{R_2}\\
&f(0,0,0)
\end{aligned}
$$

a podľa outermost:

$$
\begin{aligned}
&f\left(\underline{g(0,1)}, g(0,1), g(0,1)\right) \to_{R_2}\\
&f\left(0, \underline{g(0,1)}, g(0,1)\right) \to_{R_2}\\
&\underline{f(0,1,g(0,1))} \to_{R_2}\\
&f\left(\underline{g(0,1)}, g(0,1), g(0,1)\right) \to_{R_2}\\
&\ldots
\end{aligned}
$$

Posledný riadok pri prepisovaní podľa outermost je náš pôvodný term. Z tohto teda vidíme, že aj prepisovanie podľa outermost stratégie môže viesť k nekonečným výpočtom, zatiaľ čo pri innermost vedie k normálnej forme. Pre nás teda bude zaujímavé zisťovať, či nejaký prepisovací systém skončí výpočet.

Kapitola 3

Popis a riešenie problému

3.1 Problém zastavenia pre prepisovací systém

Definícia 3.1. Zastavenie prepisovacieho systému: *Nech R je prepisovací systém. Hovoríme, že R zastaví, ak pre každý term t, $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, neexistuje nekonečný výpočet vzhľadom na R.*

My potrebujeme vedieť rozhodnúť, či daný prepisovací systém zastaví. Tento problém je však nerozhodnuteľný [17] vo všeobecnosti. Existuje veľa prístupov k jeho riešeniu. Väčšina z nich sa opiera o problém zastavenia na *algebre termov* (definícia v [23]) pre prepisovanie bez stratégií. Založené sú na noetherovských usporiadaniach, či už syntaktických alebo sémantických, ktoré sú indukované na základe pravidiel prepisovacieho systému [20, 18, 10, 4, 11, 7].

Iné metódy zase redukujú problém zastavenia prepisovacieho systému na nejaký problém klesajúcej postupnosti na dobre usporiadanej množine. Napríklad *semantic labelling* [25] alebo *dependency pair method* [3]. Pre väčšinu prístupov je kľúčovým problémom nájdenie vhodného usporiadania. Jeho nájdenie často zahŕňa vyriešenie množiny podmienok.

Pre programovacie jazyky s pravidlami, ako sú `Maude` [8], `TOM` [22] alebo

`ELAN` [5], je programom prepisovací systém a vyhodnocovanie dotazov je dosiahnuté pomocou prepisovania prvorádových termov. Preto je potrebné špecifické dokazovanie zastavenia týchto programov, na základe osobitostí stratégií, použitých na vyhodnocovanie dotazu. Výsledkov pre automatizované dokazovanie pre prepisovacie systémy so stratégiami je len málo. Známe sú metódy pre kontextové prepisovanie [21, 13, 1], ktoré zahŕňajú čiastočne aj lokálne stratégie. Pre innermost stratégiu [2, 14, 12, 16]. Práve metóda popísaná v [12] a [16] sa dá prispôsobiť na dokazovanie pre outermost stratégiu, lokálne stratégie a tiež na prioritné innermost prepisovanie [15].

Práve vďaka tomu, že posledne menovaná metóda je prispôsobená na používanie pri viacerých typoch stratégií, zdá sa byť vhodným kandidátom na dalšie rozšírenie o *definované stratégie*[6]. Dôvodom je tiež, že metóda bola prototypovaná priamo v jazyku `ELAN`, ktorý podporuje takýto typ stratégií. Viaceré pokusy o spojazdnenie frameworku s touto implementáciou neboli úspešné. Framework využíval pri svojom behu viacero pomocných modulov. Keďže framework samotný, ale ani niektoré z modulov, neboli podporované autormi, nepodarilo sa nám napriek snahe dosiahnuť fungovanie celého systému.

Keďže metóda bola navrhnutá pre viacero strátegií, rozhodli sme sa implementovať ju v existujúcom prostredí jazyka Haskell. V rámci diplomovej práce sme implementovali samotnú metódu a aj všetky moduly, ktoré boli potrebné pre jej fungovanie.

Za zmienku ešte stojí, že problém zatavenia pri prepisovaní pomocou innermost stratégie je ekvivalentný štandardnému problému zastavenie pre systém nepoužívajúce stratégie [19]. A aj preto implementácia metódy pre innermost nie je rovnako dôležitá ako metóda pre štandardné prepisovanie.

3.2 Popis zvolenej metódy

3.2.1 Dokazovanie pomocou indukcie

V tejto sekcii najprv vysvetlíme princíp zvolenej metódy, ktorá je podrobne popísana v [12, 16].

Základnou myšlienkou je uvažovanie o probléme zastavenia pre termy, namiesto celého prepisovacieho systému R. Ak máme prepisovací systém R a stratégiu S. Hovoríme, že term $t \in \mathcal{T}(\mathcal{F})$ zastaví pri stratégii S alebo S-zastaví práve vtedy, ak každé odvodenie začínajúce z t je konečné. Zastavenie prepisovacieho systému potom môžeme vyjadriť pomocou zastavenia termov: prepisovací systém zastaví pri stratégii S práve vtedy, ak každý term $t \in \mathcal{T}(\mathcal{F})$ S-zastaví.

Pri dôkazoch budeme používať indukciu vzhľadom na termy pri nejakom noetherovskom usporiadaní $\succ$. Pri dôkaze tiež použijeme stromy odvodenia. Ak pre term t existuje v každom dovodení (podľa stratégie S) term t' taký, že $t \succ t'$, tak t S-zastaví. Na takomto terme t' môžeme použiť indukčný predpoklad, čo znamená t' S-zastaví a teda term t tiež. Tieto stromy odvodenia však budeme kvôli dôkazom schematizovať do dôkazových stromov.

Definícia 3.2. Dôkazový strom *je $n-$árny strom, ktorého vrcholy sú trojice. Prvou zložkou je term. Tento term budeme nazývať aktuálnym termom. Ďalšie dve zložky sú dva typy podmienok. Prvé nazývame podmienky pre abstrahovanie a druhé podmienky pre usporiadanie. Aktuálny term reprezentuje všetky základné inštancie, ktoré sú s ním unifikovateľné. Takisto tento term musí spĺňať obidva druhy podmienok.*

Pri konštruovaní stromu budeme postupovať nasledovne. Za koreň si zvolíme term t, ktorý je inštanciou termu $g\left(x_1, x_2, ..., x_n\right)$, pre nejaký funkčný symbol $g \in \mathcal{F}$. Synov vrcholu budeme vytvárať dvoma operáciami, abstrahovanie a zužovanie. Abstrahovanie slúži na normalizáciu podtermov

aktuálneho termu. Zužovanie schematizuje všetky možnosti prepisovacieho kroku. Pri oboch operáciách budeme využívať usporiadanie $\succ$. Toto usporiadanie musí byť rovnaké pre celý dôkazový strom. Pre rôzne stromy si však môžeme zvoliť rôzne usporiadanie. Na základe týchto operácií sa tiež budú vytvárať obidva typy podmienok. Podmienky nie sú nové pre každý vrchol, ale postupne sa hromadia počas výpočtu.

Pri dokazovaní, že prepisovací systém zastaví, budeme vytvárať takéto stromy. Za dôkaz budeme považovať množinu dôkazových stromov takých, že:

- pre každý funkčný symbol f paltí, existuje dôkazový strom s koreňom $f(x_1, ..., x_n)$,
- všetky strom majú len konečné vetvy.

3.2.2 Použité operácie a podmienky

Podmienky pre usporiadanie

Usporiadanie $\succ$, ktoré sa využíva pri indukčnom kroku, musí byť definované na $\mathcal{T}(\mathcal{F}, \mathcal{X})$. To znamená, že musíme vedieť porovnať každú dvojicu termov. Navyše toto usporiadanie musí splňať: $t \succ t' \implies \theta t \succ \theta t'$ pre každú substitúciu θ. Usporiadanie, ktoré splňa splňa túto implikáciu, nazývame stabilným na substitúciu.

Definícia 3.3. Podmienka pre usporiadanie *je zoznam dvojíc termov. Každú dvojicu t, t' zo zoznamu označíme ako $t > t'$. Podmienka pre usporiadanie je splniteľná, ak existuje usporiadanie $\succ$ také, že pre každú dvojicu termov $t > t'$ platí $\theta t \succ \theta t'$ pre každú inštanciáciu θ.*

Vo všeobecnosti je splniteľnosť podmienky pre usporiadanie C v takejto forme nerozhodnuteľný problém. Ďalšia podmienka, že usporiadanie musí

byť stabilné na substitúciu, je pre rozhodnuteľnosť problému postačujúca. Najznámejšie príklady usporiadaní, ktoré sú stabilné na substitúciu, sú LPO (Lexicographical Path Ordering) [18] a RPO (Recursive Path Ordering) [10]. Keďže tieto dve usporiadania pokrývajú veľké množstvo prípadov, pri hľadaní riešenia pre C je mnohokrát jednoduchšie a efektívnejšie skúsiť, či je niektoré z nich vhodným riešením.

Abstrahovanie

Abstrahovanie termu t na pozícii j, kde o podterme $t|_j$ predpokladáme, že má normálnu formu $t|_j \downarrow$, znamená nahradenie termu $t|_j$ novou *abstrahovanou* premennou X_j. Abstrahovaná premenná X_j zastupuje normálnu formu termu $t|_j$.

Definícia 3.4. Abstrahovanie: *term* $t[t|_j]_{j\in\{p_1,\ldots,p_k\}}$ *abstrahujeme na term* u *na pozíciách* $\{p_1, \ldots, p_k\}$ *práve vtedy, keď* $u = t[X_j]_{j\in\{p_1,\ldots p_k\}}$ *a* X_j *sú nové abstrahované premenné.*

Pri vykonávaní abstrahovania na pozícii j pridávame medzi podmienky rovnosť $t|_j \downarrow = X_j$. Pozíciu pre abstrahovanie vždy vyberáme tak, aby abstrahovanie pokrylo čo najväčší podterm. Na tento podterm musí byť aplikovateľný indukčný predpoklad. Do podmienok pre usporiadanie pridávame podmienku $t > t|_j$.

Zužovanie (Narrowing)

Zužovanie slúži na simuláciu prepisovacieho kroku na aktuálnom terme. Každý aktuálny term zužujeme na všetkých pozíciách, kde môžeme použiť nejaké prepisovacie pravidlo.

Predstavme si situáciu, keď aktuálny term t je tvaru $f(X_1)$, kde X_1 je abstrahovaná premenná, a prepisovací systém obsahuje pravidlo $f(g(x)) \to g(x)$. Pri klasickom prepisovaní by sme toto pravidlo použiť nemohli. Lenže

v tomto prípade abstrahovaná premenná X_1 predstavuje normálnu formu nejakého termu. O X_1 teda môžeme v jednej vetve dôkazu predpokladať, že má tvar $g(X_2)$, kde X_2 je nová abstrahovaná premenná, a term t teda prepísať na $g(X_2)$.

Zoberme si nejakú substitúciu $\sigma = (x_1 \leftarrow t_1, ..., x_n \leftarrow t_n)$. Takúto substitúciu môžeme tiež chápať ako konečnú množinu rovností a budeme ju zapisovať ako $x_1 = t_1 \wedge ... \wedge x_n = t_n$, kde "="je syntaktická rovnosť. Negáciou $\overline{\sigma}$ substitúcie σ budeme nazývať formulu $\bigvee_i (x_i \neq t_i)$. Ohraničenou substitúciou σ budeme nazývať formulu $\sigma_0 \wedge \bigwedge_j \bigvee_{i_j} \left(x_{i_j} \neq t_{i_j}\right)$, kde σ je substitúcia.

Definícia 3.5. Lepšia pozícia: *o pozícii p' hovoríme že je lepšia ako pozícia p pri prepisovaní podľa stratégie innermost, ak p je (nevlastným) prefixom p'.*

Definícia 3.6. Innermost-zužovanie: *term $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ sa innermost-zužuje do termu t' na pozícii p ($t|_p$ nemôže byť premenná) podľa pravidla $l \rightarrow r$ s ohraničenou substitúciou $\sigma = \sigma_0 \wedge \bigwedge_{j\in\{1,...,k\}} \overline{\sigma_j}$ práve vtedy, ak*

$$\sigma_0(l) = \sigma_0(t|_p) \text{ a } t' = \sigma_0(t[r]_p)$$

Kde σ_0 je najvšeobecnejší unifikátor $t|_p$ a l. σ_j je najvšeobecnejší unifikátor $\sigma_0\left(t|'_p\right)$ a ľavej strany nejakého pravidla z prepisovacieho systému, kde p' je lepšia pozícia ako p.

Dôvod, prečo neprepisujeme premenné, je ten, že sú to abstrahované premenné. Tým pádom predstavujú normálnu formu.

Príklad 3.1. *Uvažujme prepisovací systém R s pravidlami:*

$r_1: \quad f(x, h(g(y, y))) \rightarrow x$
$r_2: \quad g(h(x), y) \rightarrow h(y)$

a aktuálny term $f(g(X_1, X_2), h(X_3))$ *Skúsme tento term innermost-zúžiť podľa pravidla* r_1 *na pozícii* ϵ *(v hlave termu):*
aby sme to mohli urobiť, tak premenná X_3 *musí mať tvar* $g(X_4, X_4)$*. Teda najvšeobecnejším unifikátorom bude* $\sigma_0 = (X_3 = g(X_4, X_4))$*.*
Teraz podľa definície nájdeme lepšie pozície. Jedinou možnosťou je použitie pravidla r_2 *pre term* $g(X_1, X_2)$ *s najšeobecnejším unifikátorom* $\sigma_1 = (X_1 = h(X_5))$*.*
Negáciou dostaneme $\overline{\sigma_1} = (X_1 \neq h(X_5))$*.*
Čiže našim novým aktuálnym termom je $g(X_4, X_4)$*. Navyše dostaneme podmienku:* $X_1 \neq h(X_5)$*.*

Dôvod, prečo sme dostali túto podmienku, je to, že ak by term X_1 mal normálnu formu $h(X_5)$, tak by sme museli prepisovať vnútornejší term a nemohli by sme použiť prvé pravidlo. Ak sme pred zužovaním mali podmienky pre abstrahovanie A, tak po tomto zúžení dostaneme podmienku $A \wedge (X_1 \neq h(X_5))$.

Zužovanie robíme na všetkých pozíciách, na ktorých sa dá aktuálny term redukovať, čo spôsobuje vetvenie stromu.

Podmienky pre abstrahovanie

Definícia 3.7. Podmienky pre abstrahovanie (ACF - abstraction constraint formula) *je formula tvaru* $\bigwedge_i (t_i = t'_i) \wedge \bigwedge_k \bigvee_{l_k} (x_{l_k} \neq u_{l_k})$*, kde* $x_{l_k} \in \mathcal{X}$ *a* $t_i, t'_i, u_{l_k} \in \mathcal{T}(\mathcal{F}, \mathcal{X})$*.*

Treba poznamenať, že ak máme ohraničenú substitúciu $\sigma = \sigma_0 \wedge \bigwedge_j \overline{\sigma_j}$, tak môžeme aplikovať substitúciu σ_0 na aktuálny term, na všetky podmienky vo vrchole a tiež na pomienky $\bigwedge_j \overline{\sigma_j}$. Následne môžeme túto substitúciu z formuly vynechať, a tým zredukujeme počet členov v podmienkach pre abstrahovanie. Vďaka tomuto kroku bude ďalšie upravovanie podmienok oveľa jednoduchšie.

Definícia 3.8. Splniteľnosť podmienok pre abstrahovanie: *formula* $\bigwedge_i (t_i \downarrow= t'_i) \wedge \bigwedge_k \bigvee_{l_k} (x_{l_k} \neq u_{l_k})$ *je splniteľná práve vtedy, keď existuje inštanciácia* θ *taká, že* $\bigwedge_i (\theta t_i \downarrow= \theta t'_i) \wedge \bigwedge_k \bigvee_{l_k} (\theta x_{l_k} \neq \theta u_{l_k})$. *Túto inštanciáciu nazývame riešením formuly.*

Ak v nejakom vrchole zistíme, že podmienky pre abstrahovanie sú nesplniteľné, môžeme pre tento vrchol zastaviť výpočet. Keby boli nesplniteľné, tak žiadna inštanciácia aktuálneho termu nie je riešením podmienok. Potom ale množina, ktorú aktuálny term reprezentuje, je prázdna.

Príklad 3.2. *Zoberme aktuálny term a prepisovací systém* R *z predchádzajúceho príkladu. Zostrojíme prepisovací systém* R' *tak, že k systému* R *pridáme pravidlo* $r_3 : h(x) \to x$. *V takomto systéme* R' *skúsme použiť pravidlo* r_1 *na pozícii* ϵ, *tak ako v predchádzajúcom príklade. Dostali by sme dve lepšie pozície, a teda ešte jednu zložku k ohraničenej substitúcii:* $\sigma_2 = \top$. *Celá ohraničená substitúcia má tvar* $\sigma = \sigma_0 \wedge \overline{\sigma_1} \wedge \overline{\sigma_1} = \sigma_0 \wedge \overline{\sigma_1} \wedge \bot = \bot$. *Tá je automaticky nesplnitľná.*
To znamená, že v aktuálnom terme nemôžeme použiť pravidlo r_1 *na pozícii* ϵ.

Disunifikácia[9] rieši časť podmienok s nerovnosťami. Ich riešenie je vo všeobecnosti nerozhodnuteľné, ale dodatočné podmienky na normálne formy rozširujú prípady, kedy je disunifikácia rozhodnuteľná. Tieto prípady sme popísali v sekcii 6.1.4. Samotná disunifikácia je popísaná v sekcii 6.1.5.

3.2.3 Popis procedúry

V našom dôkazovom strome budú vrcholmi trojice: aktuálny term, podmienky pre abstrahovanie, podmienky pre usporiadanie. Začneme z koreňa ($T = g(x_1, ..., x_n)$, $A = \top$, $C = \top$). Potom spustíme iteratívne kroky v nasledujúcom poradí:

- abstrahovanie: najprv abstrahujeme term na pozíciách $\{p_1, ..., p_k\}$. Do podmienok pre usporiadanie C pridáme $t > t|_i$ pre tie termy, pre

ktoré neplatí ani jedna z dodatočných podmienok pre zastavenie z predchádzajúcej sekcie. Do podmienok pre abstrahovanie A zase pridávame podmienky $t|_i \downarrow = X_i$.

- zužovanie: tu prepíšeme term na všetkých možných pozíciách (kým sa neporuší splniteľnosť podmienok A). Tým sa strom rozvetví na toľko vrcholov, koľko je možných prepísaní.
- ukončenie vetvy: dôkaz skončíme, ak aktuálny term spĺňa jedno z podmienok pre zastavenie.

Pri výpočte musí platiť, že podmienky A sú stále splniteľné.

Kapitola 4

Návrh a analýza riešenia

4.1 Voľba jazyka a prostredia

4.1.1 Jazyk

Pri pôvodnej myšlienke rozšírenia metódy sme chceli pokračovať vo vývoji v jazyku `ELAN`. Na beh implementácie bolo potrebné nainštalovať komilátor tohoto jazyka. Kompilátor však nebol kompatibilný s aktuálnym operačným systémom, využíval staré systémové premenné. Pri problémoch sme skúšali hľadať podporu u autorov, ale ani to nepomohlo tieto problémy odstrániť. Rozhodli sme sa nájsť iný vhodný jazyk, ktorý je všeobecne používaný. Zvolili sme si programovací jazyk Haskell [24].

Haskell je najznámejší funkcionálny programovací jazyk. Práve tento fakt bol jedným z hlavných dôvodov jeho výberu. Haskell má oproti klasickým imperatívny jazykom ako sú `Java`, C, C++ alebo C# niekoľko výhod:

- Haskell je silne typový jazyk. To znamená, že všetko má svoj typ. Takže nemôžeme zle pretypovávať objekty a tým prísť k výnimke,

- všetko, čo sa deje pri výpočte vo funkcionálnych jazykoch, je volanie

funkcií. Všetky priradenia a procedúry sú robené pomocou volania funkcií,

- v jazyku Haskell sa nedajú používať globálne premenné. To znamená, že všetko, čo ovplyvňuje výsledok funkcie, musí prísť na vstup ako argument. To znamená, že pri volaní funkcie s rovnakým vstupom dostaneme stále rovnaký výstup,
- keďže nemáme globálne premenné, používame techniku monád. Namiesto týchto premenných pracujeme so špeciálnymi stavovými monádami,
- predošlé vlastnosti spolu tvoria tzv. referenčnú transparentnosť. To znamená, že nemôžu nastať vedľajšie efekty (side efects). Napríklad, ak počítame s nejakou premennou, tá ma po volaní funkcie rovnakú hodnotu ako pred jej volaním. Nemôže sa stať, že počas behu ju nejaká iná funkcia zmenila,
- Haskell poskytuje nástroje pre prácu so zoznamami. Pri imperatívnych jazykoch musíme pri prechádzaní zoznamov použiť iteátor. Pre Haskell existujú vstavané konštrukcie, ktoré mapujú funkcie na prvky zoznamu,
- kód vo funkcionálnom jazyku je vo všeobecnosti omnoho kratší a úspornejší ako v impepratívnych. Tým sa nielen zmenšuje veľkosť kódu ale aj sa zvyšuje jeho čiateľnosť,
- špecifikom jazyka Haskell je tiež takzvané *lenivé vyhodnocovanie*. To znamená, že funkcie sa vyhodnocujú, len ak sú pre výsledok potrebné.

4.1.2 Prostredie

Nevýhodou jazyka `ELAN` bolo, že bol implementovaný len pre operačný systém Linux. My sme chceli poskytnúť platformovú nezávislosť. Keďže existuje

interpreter jazyka Haskell aj v operačnom systém Linux aj vo Windows, túto podmienku sme naplnili. Pri našej implementácii sme mali možnosť voľby medzi nimi. Vybrali sme si operačný systém Linux a distribúcii GENTOO.

4.2 Návrh dátovej štruktúry

Pri návrhu dátových štruktúr sme postupovali hierarchicky, t.j. zdola nahor. To znamená, že sme začali definovať typy od najnižšej vrstvy a postupovali k zložitejším.

Na najnižšej vrstve typovej hierarchie sa nachádza premenná. Typ premennej potrebujeme ako osobitný typ najmä kvôli hľadaniu substitúcie pri prepisovaní a pri unifikácii termu s pravidlami. Namiesto stáleho premenovania premenných pri unifikácii, budeme rozlišovať premenné systému. Ďalším odlišným typom premennej je abstrahovaná premenná. Keďže implementujeme innermost stratégiu, nepotrebujeme obyčajné premenné, lebo ich hneď v prvom kroku všetky abstrahujujeme. Definícia typu premennej vyzerá nasledovne:

```
data Var = ABS String |
           RW String
```

Teraz môžeme zadefinovať term. Podľa definície 2.1, term je buď premenná alebo funkcia vytvorená nad termami:

```
data Term = VAR Var |
            FUNC String [Term]
```

Na definíciu vrcholu stromu ešte potrebujeme zadefinovať podmienky pre abstrahovanie a podmienky pre usporiadanie. Začneme podmienkami pre abstrahovanie.

- Triviálnym prípadom je, keď je celá podmienka pravdivá $\top$ alebo nepravdivá $\bot$.

- V podmienke sa môže vyskytovať rovnosť vzhľadom na normálnu formu: $t_1 \downarrow= t_2$, o ktorej hovorí tretí riadok tejto definície.

- V ďalšom riadku je popísaná nerovnosť dvoch termov. Aj keď podľa definície 3.7 by sa na ľavej strane nerovnosti mala vyskytovať iba premenná, my pripúšťame nerovnosť dvoch ľubovoľných termov. Toto zovšeobecnenie sme urobili kvôli tomu, že pri úpravách podmienok sa na ľavej strane môže dočasne vyskytnúť aj term. Vo vrcholoch je však normálna forma v súlade s definíciou 3.7.

- Konjunkciu a disjunkciu podmienok sme urobili na zoznamoch podmienok, kvôli lepšiemu narábaniu s nimi.

```
data Acond = TRUE |
             FALSE |
             EQNF Term Term |
             NEQ Term Term |
             AND [Acond] |
             OR [Acond]
```

Príklad 4.1. *Pre lepšiu ilustráciu uvedieme príklad. Uvažujme podmienku*

$$g\,(X2) \wedge ((X1 \neq f\,(X2, X3, X4)) \vee (X5 \neq g\,(X2)))$$

jej vnútorná reprezentácia vyzrá nasledovne:

```
AND [EQNF (FUNC "g" [VAR (ABS "X2")]) (VAR (ABS "X1")),
     OR [NEQ (VAR (ABS "X1"))
             (FUNC "f" [VAR (ABS "X2"),VAR (ABS "X3"),VAR (ABS "X4")]),
         NEQ (VAR (ABS "X5"))
             (FUNC "g" [VAR (ABS "X2")])
        ]
    ]
```

Podmienky pre usporiadanie sú jednoduchšie. Keďže máme stále len jeden term, s ktorým porovnávame, tak podmienka bude vyzerať ako dvojica, kde prvý prvok je náš term, a druhý zoznam prvkov, od ktorých je väčsí. Prázdny zoznam bude reprezentovať podmienku $\top$:

```
type Ccond = (Term, [Term])
```

Uzol stromu bude trojica: term, podmienky pre abstrahovanie a podmienky pre usporiadanie:

```
type Node = (Term, Acond, Ccond)
```

Dôkazový strom bude rekurzívna definícia pozostávajúca z vrcholu a zo zonamu synov:

```
data Tree = TREE Node [Tree]
```

Pri výpočte potrebujeme rostredie, simulujúce globálne premenné. V prevej verzii sme použili počítadlo, usporiadanie, podľa ktorého robíme indukciu, a prepisovací systém. Pre výpočet niekedy potrebujeme nové premenné. Aby sme ich odlíšili, číslujeme ich, a k tomu slúži počítadlo. Usporiadanie je pre celý výpočet rovnaké, ale pre rôzne prepisovacie môže byť rôzne. Preto ho posielame ako argument:

```
type Env = (Int, Ordering0, Rw_system)
```

Z ďalších definícií typov ešte spomenieme definíciu usporiadania. Haskell má v sebe zabudovanú podporu pre lambda kalkul. Vďaka tejto vlastnosti je možné vytvárať anonymné funkcie, to znamená lokálne definované funkcie bez mena. Tie potom možno ďalej posielať ako parametre. Preto argumentami môžu byť aj funkcie. Pri definícii teda definujeme vstupy a výstupy funkcie. Usporiadanie bude mať ako vstupy dva termy, ktoré chceme porovnať. Keďže základné usporiadania, ktoré sme použili, využívajú na porovnanie termov čiatočné usporiadanie na funkčných symboloch, tak ďalším argumentom je množina (zoznam) prefernecií. Množina preferencií musí byť dobré usporiadanie na funkčných symboloch, t.j. neexistuje nekonečná klesajúca postupnosť. Výsledným typom pre porovnanie je $True$ alebo $False$:

```
type Ordering0 = Term -> Term -> [(String, String)] -> Bool
```

Ešte poznamenáme, že Haskell pozná dva druhy typov:

- `type` je definovanie výsledného typu ako synonymum už existujúcich typov. Takýmto štýlom sme zadefinovali vrchol stromu ale aj usporiadanie.

- `data` slúži na zložitejšie definície. Pri definícii premennej nám poslúžil ako identifikátor, o akú premennú ide. Pri definícii termu nám zas umožnil rekurzívnu definíciu. Všeobecne nám slúži na definovanie rôznych typov do jedného celku. Pri rozoberaní prípadov potom v hlavách funkcií používame porovnanie vzorov (pattern matching).

Kapitola 5

Implementácia

5.1 Popis pomocných funkcií

V nasledujúcej sekcii popíšeme pomocné funkcie potrebné na beh celej metódy. Uvedené funkcie sú všeobecné a využívajú sa v rôznych častiach programu. Pri implementácii sme postupovali spravidla zdola nahor, podobne ako pri definíciách.

Substitúcia

Substitúciu aplikujeme podľa nasledovného algoritmu: nech máme substitúciu $\sigma = (x_1 \leftarrow s_1, x_2 \leftarrow s_2, ..., x_n \leftarrow s_n)$, potom

- $x\sigma = t_1\,(x_2 \rightarrow s_2, ..., x_n \rightarrow s_n)$, ak $x = x_1$,
- $x\sigma = x\,(x_2 \rightarrow s_2, ..., x_n \rightarrow s_n)$, ak $x \neq x_1$,
- $(f\,(t_1, ..., t_n))\,\sigma = f\,(t_1\sigma, ..., t_n\sigma)$.

Unifikácia

K unifikácii budeme potrebovať funkciu occur check. Táto funkcia je pravdivá pre premennú x a term t, ak sa x nachádza v t.

Unifikáciu sme v implementácii vyriešili podľa naivného algoritmu. Na začiatku si zo vstupných termov vytvoríme zoznam dvojíc. Výsledkom funkcie `unify` na zozname $((t_1, t_2) : ts)$ bude substitúcia:

- $(t_1 \leftarrow t_2) \circ (\texttt{unify}\, ts\, (t_1 \leftarrow t_2))$
  ak $t_1 \in \mathcal{X}$ a neplatí occur check pre t_1 a t_2,
- $(t_2 \leftarrow t_1) \circ (\texttt{unify}\, ts\, (t_2 \leftarrow t_1))$
  ak $t_2 \in \mathcal{X}$ a neplatí occur check pre t_2 a t_1,
- $\texttt{unify}\, ([(s_1, s'_1), ..., (s_m, s'_m)] \oplus ts)$
  ak $t_1 = f\,(s_1, ..., s_m)$ a $t_2 = f\,(s'_1, ..., s'_m)$,
- v ostatných prípadoch termy nie sú unifikovateľné.

Použitie funkcie occur check je dôležité. Ak by sme netestovali výskyt premennej v terme, algoritmus by mohol chybne o niektorých termoch prehlásiť, že sú unifikovateľné. Napríklad, unifikujme termy: $f\,(X, X)$ a $f\,(X, g\,(X))$ bez použitia occur check. Dostaneme substitúciu $X \leftarrow g\,(X)$. Po dosadení dostaneme z prvého termu $f\,(g\,(X)\,, g\,(X))$ a z druhého $f\,(g\,(X)\,, g\,(g\,(X)))$. Tieto sa nerovnajú a teda máme zlú substitúciu, t.j. substitúcia nie je unifikátorom.

Zisťovanie pozícií pre zužovanie

Zisťovanie pozícií pre term t znamená postupné prechádzanie termu. V každom podterme s, ktorý nie je premenná, zisťujeme, či je s unifikovateľný s niektorou ľavou stranou pravidla. Ak áno, medzi pozície pridáme pozíciu s v terme t. Premenné sme vylúčili preto, lebo abstrahované premenné reprezentujú normálnu formu, a teda ich nemôžeme prepísať.

5.2 Hlavné časti programu

5.2.1 Abstrahovanie

Abstrahovanie je funkcia, ktorá dostane na vstup vrchol stromu a prostredie a vráti nový (pravdepodobne) zmenený vrchol a aktualizované prostredie. Aktualizácia spočíva v zmene počítadla, bude vyššie o toľko, koľko nových premenných sa nachádza v terme a podmienkach. Predtým, ako popíšeme abstrahovanie, potrebujeme ešte dve pomocné funkcie. Prvá je spojená s termináciou a je popísaná ďalej v 6.1.2. Druhou funkciou je funkcia, ktorá zisťuje splniteľnosť podmienok pre usporiadanie. Táto funkcia dostane na vstup term t, zoznam termov ts a usporiadanie $\succ$ na ich porovnanie. Zisťuje, či pre každý prvok s zo zoznamu ts platí:

$$t \succ s.$$

Takto definovaná funkcia je v súlade s naším predpokladom, že ak je zoznam ts prázdny, tak reprezentuje splnenú podmienku $\top$. Druhým cieľom je tiež abstrahovať splniteľnosť podmienok od konkrétneho usporiadania. Preto je usporiadanie vstupným parameterom.

Teraz sa môžeme vrátiť k funkcii pre abstrahovanie. Ako sme už povedali, nahrádza podtermy abstrahovanými premennými. Pre podtermy, ktoré abstrahujeme platí, že výpočet pre ne zastaví. Funkcia rozoberá viac prípadov:

- vstupný term je premenná. V tomto prípade nemeníme uzol ani prostredie. Keďže implementujeme innermost stratégiu, tak iné ako abstrahované premenné a premenné prepisovacieho systému nepoužívame. Samozrejme, premenné systému nevstupujú do výpočtu. Inak by celkom stratili zmysel, lebo sme chceli mať rôzne premenné v termoch a v pravidlách. To znamná, že vstupná premenná musí byť abstrahovanou premennou. A táto už je v normálnej forme.

- ak pre term t platí podmienka terminácie. Táto podmienka sa vyhodnotí na základe funkcie podrobne popísanej v 6.1.2. Ak je splnená, pre daný term výpočet skončí a môžeme ho abstrahovať do premennej X_i. Vtedy pridáme do podmienok pre abstrakciu $t \downarrow= X_i$. Taktiež zvýšime počítadlo o jedna.

- ak nie je splnená podmienka pre termináciu, ale je splnená podmienka na usporiadanie. Toto vyhodnotíme na základe hore popísanej pomocnej funkcie tak, že k zoznamu pridáme term zo vstupu. Do výsledku pôjde podmienka ako v predošlom prípade. Navyše vstupný term pridáme do zoznamu podmienok pre usporiadnaie.

- ak nenastal niektorý z predchádzajúcich prípadov, tak abstrahujeme vnútorné termy. Treba dáť pozor, aby argumenty funkcie boli abstrahované postupne, pretože potrebujeme aktualizovať prostredie a počítadlo. Ak by sme to nerobili, mohlo by nastať, že máme viac podmienok typu $t_1 \downarrow= X_i \wedge t_2 \downarrow= X_i$, čím by sme stotožnili normálne formy niektorých termov.

5.2.2 Zužovanie

Zužovanie je zložitejšie ako abstrahovanie. Potrebujeme viac pomocných funkcií. Medzi pomocnými funkcimi máme funkciu na hľadanie všetkých pozícií pre zužovanie. Pri zužovaní na konkrétnej pozícii tiež potrebujeme zistiť aj možnosť prepísania na lepších pozíciách. Pri ich hľadaní predpokladáme, že zužovanie robíme v hlave termu. Keďže pre innermost stratégiu je pozícia p' lepšia ako pozícia p, ak p je prefixom p'. Takže zo všetkých pozícií vylúčime len pozíciu ϵ.

Na vytváranie podmienok potrebných pre zužovanie používame funkciu, ktorá na vstupe dostane term t a dvojicu (pozícia p, pravidlo r). Znamená to,

že v terme t môžeme na pozícii p použiť pravidlo r. Pri hľadaní podmienok potrebujeme zistiť substitúciu potrebnú pre zužovanie. Podľa danej pozície rozoberieme term. Unifikáciou s ľavou stranou pravidla a následnou reštrikciou premenných na premenné termu získame hľadanú substitúciu. Následne túto substitúciu už len transformujeme na podmienky.

Keď vieme vytvárať podmienky zo substitúcie a hľadať lepšie pozície, vytvoríme funkciu, ktorá pre dané pravidlo a pozíciu vytvorí podmienky pre abstrahovanie. Inak povedané vytvoríme ohraničenú substitúciu pre daný vrchol a pozíciu. Funkcia najskôr pre vstupný term zistí všetky lepšie pozície. Pomocou predchádzajúcej funkcie vytvoríme pre každú takúto pozíciu podmienky pre použitie a následne ich znegujeme. Ich konjunkciou získame hľadané podmienky. Ak by sme pri skúmaní niektorej lepšej pozície zistili, že existuje pravidlo, ktoré je stále použiteľné na danej pozícii, tak výslednou podmienkou pre ňu by (pred negáciou) bolo $TRUE$. To by znamenalo, že celá ohraničená substitúcia je $FALSE$, teda tento term nemôžeme prepisovať v hlave, viď príklad 3.2.

Dôležitým prvkom pri prepisovaní je premenovanie premenných. Zoberme si príklad, keď chceme zúžiť term $f(X_1, X_2)$ podľa pravidla $f(g(u), v) \to v$, kde X_1, X_2 sú abstrahované premenné a u, v sú premenné prepisovacieho systému. Výsledkom unifikácie je substitúcia, ktorú keď zúžime na premenné termu, tak dostaneme: $X_1 \leftarrow g(u)$. Takýmto spôsobom sa nám do termu môže dostať premenná prepisovacieho systému. Tieto premenné musíme premenovať. Postupným prechádzaním substitúcie budeme nahradzovať každú premennú prepisovacieho systému za novú abstrahovanú premennú. Tieto abstrahované premenné číslujeme podľa stavu počítadla. Dôležité je však nahradiť rovnaké premenné prepisovacieho systému rovnakými abstrahovanými premennými. Technicky, každé premenovanie berieme ako substitúciu $u \leftarrow X_i$ a skladáme ju s pôvodnou substitúciou. Dôvodom je, že premenné sa už mohli ocitnúť v terme alebo v podmienkach.

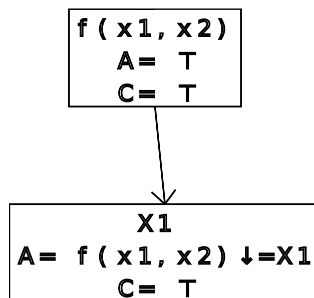
Poslednou funkciou pred zužovaním je funkcia pre vytvorenie vrcholu stromu. Keď pre nejaký vrchol $nd@\,(t, a, c)$ poznáme pozíciu a pravidlo, podľa ktorých ho zužujeme, táto funkcia vytvára nový vrchol, ktorý bude v dôkazovom strome synom vrchola nd. Najprv zistíme ohraničenú substitúciu $(s, a1)$ pre danú pozíciu. Získané podmienky dáme do konjunkcie s už existujúcimi podmienkami pre abstrahovanie. Potom môžeme na danej pozícii prepísať term t na nový term t'. Nakoniec pre vrchol $(t', a \wedge a1, c)$ a substitúciu s použijeme premenovanie premenných, čím upravíme vrchol do konečného tvaru.

Samotné zužovanie je už len jednoduchým zložením predošlých funkcií. Pre term v danom vrchole nd si vytvoríme zoznam všetkých pozícií. Potom si pomocou predchádzajúcej funkcie vytvoríme pre každú pozíciu nový vrchol. Tieto vrcholy predstavujú synov vrchola nd. V závere ešte musíme vylúčiť zo zoznamu synov všetky tie vrcholy, ktorých podmienky pre abstrahovanie sú $FALSE$.

5.2.3 Beh dôkazu

Pri dokazovaní zastavenia pre jeden funkčný symbol si najskôr vytvoríme počiatočný vrchol $(f\,(x_1, ..., x_n)\,, \top, \top)$. Na jednotlivých vrcholoch opakujeme proces, ktorý je nasledovný: najskôr použijeme funkcie abstrahovania a zužovania. Po každom zúžení skúšame, či podmienky pre abstrahovanie sú konzistentné. Ak nie, vetva dôkazu končí. Ak áno, skúsime zistiť, či výpočet na aktuálnom terme končí pomocou podmienok popísaných v časti 6.1.2. Ak term nie je terminujúcim, iterujeme celý proces. Ak proces zastaví, a teda všetky vetvy budú konečné, zastaví aj výpočet pre prepisovací systém, ktorý skúmame.

Pre prepisovací systém popísaný proces vykonávame pre každý funkčný symbol, ktorý sa nachádza v hlave aspoň jedného pravidla. Vidíme, že ak



Obr. 5.1: Dôkazový strom pre funkciu f

sa symbol f nenachádza v hlave žiadneho pravidla, tak výpočet na terme $f(s_1, ..., s_n)$ zastaví práve vtedy, keď sa zastaví na všetkých jeho podtermoch.

Príklad 5.1. *Pre ilustráciu uvedieme príklad. Uvažujme prepisovací systém:*

$r_1 : \quad f(h(u), v) \to h(f(u, v))$

$r_2 : \quad f(a, u) \to u$

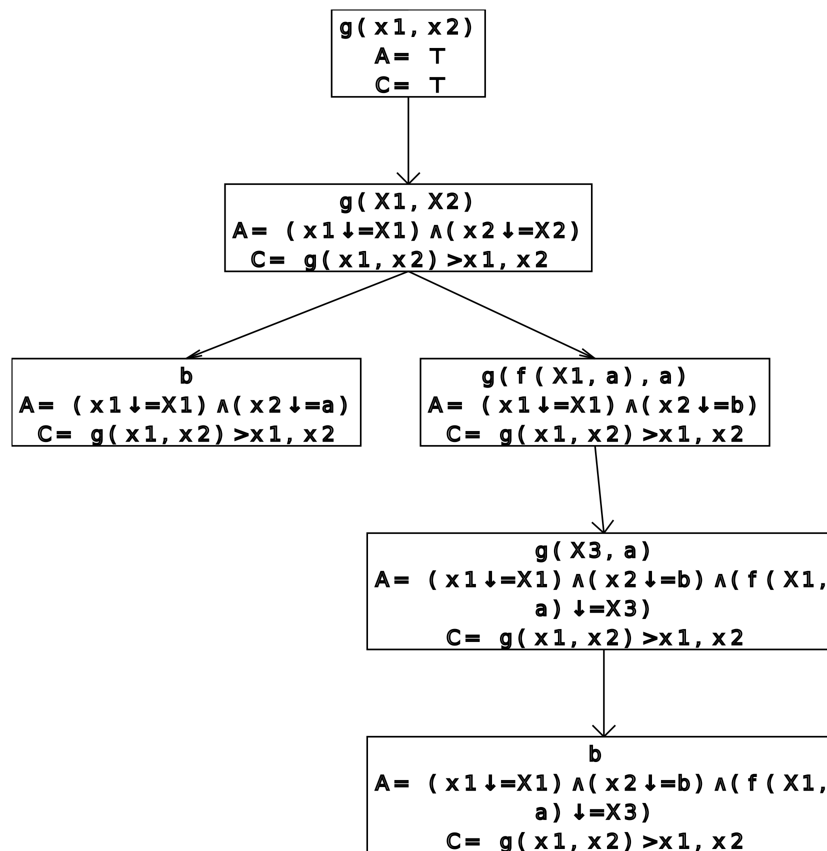
$r_3 : \quad g(u, b) \to g(f(u, a), a)$

$r_4 : \quad g(u, a) \to b$

a nech máme čiastočné usporiadanie funkčných symbolov: $f > h$. Na výpočet použijeme usporiadanie RPO. Predpokladáme, že výpočet skončí pre konštanty a, b a tiež pre funkciu h, lebo nie je v hlave žiadnej ľavej strany pravidla.

Dôkaz začneme s funkciou f a dôkazový strom pre ňu zachytáva obrázok 5.1. Začíname z uzla $f(x_1, x_2)$. V prvom kroku abstrahujeme obidve premenné, keďže premenné x_1 a x_2 sú menšie ako pôvodný term v usporiadaní *RPO*. Preto sa nám zmenia obidva typy podmienok. V druhom kroku vypočítame použiteľné pravidlá. Zistíme, že sú dobre usporiadané a môžeme skončiť.

Pre funkciu g (obrázok 5.2) je prvý krok rovnaký. V druhom kroku môžeme zužovať podľa dvoch pravidiel. V jednej vetve predpokladáme, že



Obr. 5.2: Dôkazový strom pre funkciu g

X_2 má tvar b. Term b je základný term v normálnej forme a preto tu výpočet skončí. V druhej vetve sme uvažovali, že X_2 má tvar a a aktuálny term sa prepísal na term $g(f(X_1, a), a)$. O takomto terme zatiaľ nevieme povedať, že je v normálnom tvare, preto ideme proces opakovať. Najväčší podterm, ktorý vieme abstrahovať, je $f(X_1, a)$, lebo jeho použiteľné pravidlá sú usporiadané v RPO. Tento podterm sme abstrahovali do premennej $X3$ a do podmienok pre abstrahovanie pribudla podmienka $f(X_1, a) \downarrow = X_3$. V ďalšom kroku nový aktuálny term zúžime do termu b, ktorý je v normálnej forme. Výpočet teda skončí.

Príklad 5.2. *V tomto príklade máme prepisovací systém, ktorý vytvára reverz zoznamu:*

$r_1: \quad rev1(0, nil) \rightarrow 0$

$r_2: \quad rev1(s(X), nil) \rightarrow s(X)$

$r_3: \quad rev1(X, cons(Y, L)) \rightarrow rev1(Y, L)$

$r_4: \quad rev(nil) \rightarrow nil$

$r_5: \quad rev(cons(X, L)) \rightarrow cons(rev1(X, L), rev2(X, L))$

$r_6: \quad rev2(X, nil) \rightarrow nil$

$r_7: \quad rev2(X, cons(Y, L)) \rightarrow rev(cons(X, rev(rev2(Y, L))))$

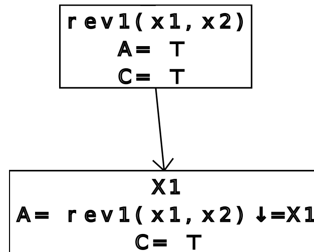
a čiastočné usporiadanie funkčných symbolov:

$$rev > cons, rev > rev1, rev > rev2, rev1 > cons, rev2 > cons$$

s usporiadaním RPO.

V tomto príklade začneme s funkciou $rev1$, obrázok 5.3. Podobne ako v predošlom príklade pre funkciu f, pravidlá sú v usporiadaní. To znamená termináciu tejto vetvy.

Druhou funkciou je funkcia $rev2$ (na obrázku 5.5). V druhom vrchole sa strom rozvetvuje podľa toho, či je X_2 prázdny zoznam (nil), alebo či



Obr. 5.3: Príklad 2, funkcia rev1

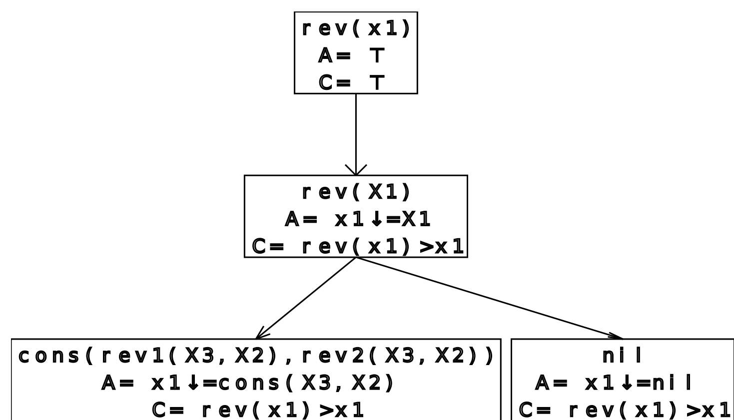
obsahuje aspoň jeden prvok ($cons(X_4, X_3)$). V prípade prázdneho zoznamu výpočet končí. V druhom prípade dostaneme aktuálny term $rev\,(cons\,(X_1, rev\,(rev2\,(X_4, X_3))))$. Abstrahovaním dostaneme jednoduchší term $rev\,(cons\,(X_1, rev\,(X_5)))$. Z tohto termu existujú tri rôzne cesty:

1. ak $X_5 \neq nil \wedge X_5 \neq cons\,(X, L)$. V tomto prípade môžeme použiť iba pravidlo r_5 na pozícii ϵ. Výsledný term je ale v usporiadaní RPO menší a teda výpočet skončí.

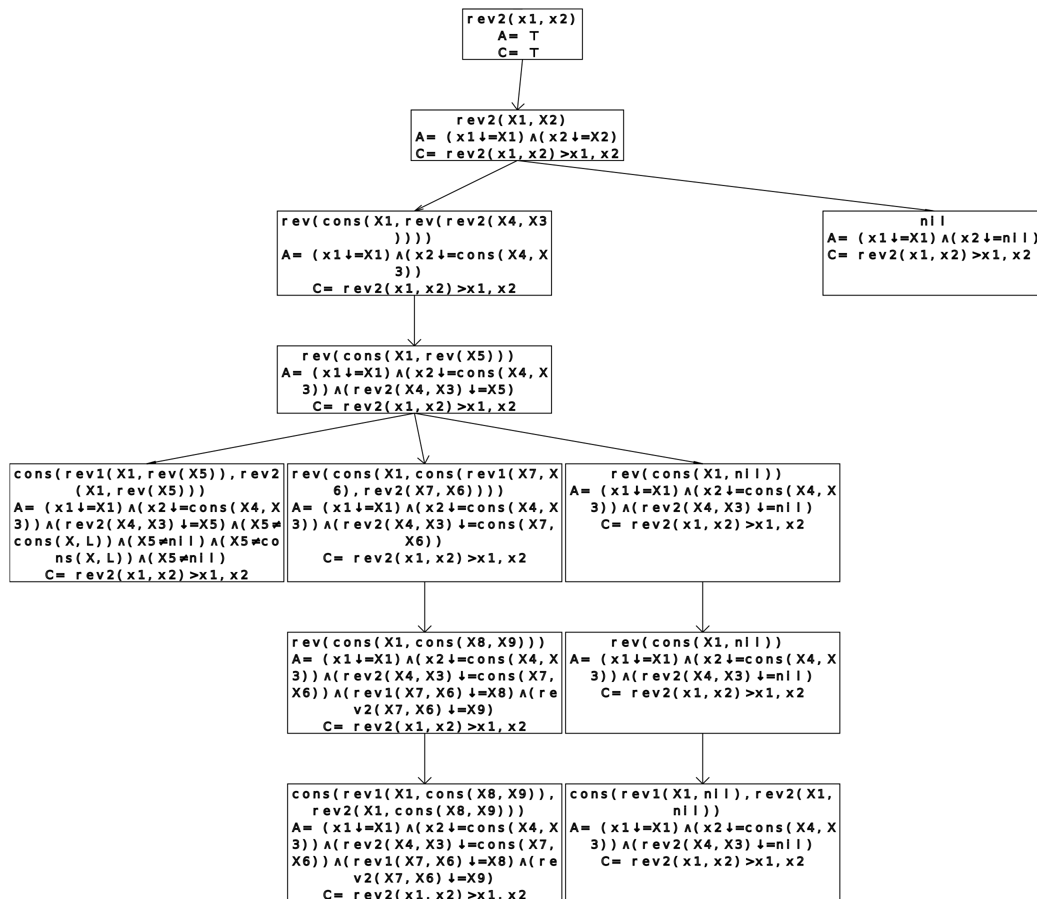
2. ak $X_5 = cons\,(X, L)$. V tomto prípade použijeme tiež pravidlo r_5, ale vo vnútri termu. Táto pozícia je lepšia. Vo výslednom terme abstrahujeme do termu $rev\,(cons\,(X_1, cons\,(X_8, X_9)))$, ktorý vieme znova zúžiť podľa pravidla r_5. Ale tento term bude v danom usporiadaní menší ako predchádzajúci, preto môžeme vetvu ukončiť.

3. v poslednom prípade $X_5 = nil$ máme po zúžení term $rev\,(cons\,(X_1, nil))$. Použitím pravidla r_5, tak ako v predošlej vetve, dostávame menší term a ukončujeme výpočet.

Samotná funkcia pre reverz je zobrazená na obrázku 5.4. Základný term môžeme zužovať na dvoch miestach. V obidvoch prípadoch dostávame menší term a preto dôkaz končí.



Obr. 5.4: Príklad 2, funkcia rev



Obr. 5.5: Príklad 2, funkcia rev2

Kapitola 6

Súvisiace problémy: popis a implementácia

V tejto kapitole popíšeme problémy, na ktoré sme narazili počas implementácie metódy. Medzi nimi je mnoho takých, ktoré sú predmetom samostatného výskumu. My ukazujeme, ako sme tieto problémy prispôsobili a aplikovali pre naše riešenie.

6.1 Terminácia a použiteľné pravidlá

6.1.1 Použiteľné pravidlá

Použiteľné pravidlá [2] nejakého termu sú všetky pravidlá, ktoré sa môžu vyskytnúť pri výpočte na tomto terme. Teraz si ich zadefinujeme formálnejšie.

Definícia 6.1. Použiteľné pravidlá pre stratégiu innermost: *majme prepisovací systém R s množinou funkčných symbolov $\mathcal{F}$. Ďalej nech $Rls\,(f) = \{l \rightarrow r \in R | hlava\,(l) = f\}$. Potom množina použiteľných pravidiel U termu t, je:*
$U\,(t) = \emptyset$, *ak* $t \in \mathcal{X} \cup Var\,(R)$,

$U(f(t_1,...,t_n)) = Rls(f) \cup \bigcup_{i=1}^{n} U(t_i) \cup \bigcup_{l \to r \in Rls(f)} U(r)$.
Poznámka: term t obsahuje len abstrahované premenné a premenné prepisovacieho systému.

Uvažovanie o zastavení výpočtu robíme na terme t a prepisovacom systéme $R' = U(t)$. Pri implementácii je s rekurzívnym prehľadávaním problém.

Príklad 6.1. *Uvažujme term $t = f(u)$, kde u je systémová premenná, a prepisovací systém R pozostávajúci z pravidla $r_1 = f(f(u)) \to f(u)$. Ak by sme rátali tieto pravidlá rekurzívne, dostaneme:*

$$U(f(u)) = Rls(f) \cup U(u) \cup U(f(u)) = ...$$

Príklade ilustruje, že na výpočet $U(f(u))$ potrebujeme vypočítať $U(f(u))$, z čoho by sme sa dostali do nekonečnej rekurzie. Pri implementácii sme teda postupovali výpočtom zhora nadol. Algoritmus pracuje nasledovne:
- výpočet začneme s prázdnou množinou pravidiel,
- ak do výpočtu pre term $f(x_1,...,x_n)$ dostaneme množinu už nájdených pravidiel S, tak

- vypočítame $S_0 = Rls(f)$.
- pre každú pravú stranu pravidla z množiny $S_0 \setminus S$ počítame použiteľné pravidlá s pravidlami $S \cup S_0$.
- nakoniec vypočítame použiteľné pravidlá subtermov na množine $S \cup S_0$.

Takýmto spôsobom nehľadáme v pravidlách, ktoré sú už nájdené. Takýto výpočet zastane, ak napríklad uvážime mieru klesania dvojicu $(|R \setminus S|, \succ)$, kde $\succ$ je noetherovské usporiadanie na termoch.

6.1.2 Terminácia

Pri dokazovaní používame predpoklad, že pre menšie termy ako je aktuálny, výpočet zastaví. Zatiaľ sme ale len na základe terminácie podtermov upravovali aktuálny term. Preto potrebujeme ešte nejaké kritériá, podľa ktorých by sme vedeli povedať, že výpočet zastaví aj pre aktuálny term. Okrem toho, ak podmienka na termináciu aktuálneho termu nie je splnená, oplatí sa nám skúmať aj termináciu podtermov. Dôvodom je, že podterm, pre ktorý je terminácia splnená, môžeme abstrahovať do premennej. Poznáme tri takéto kritériá:

1. ak vstupný term nie je zúžiteľný. To znamená, že neexistuje žiadne pravidlo, ktoré by sme na danú množinu mohli použiť. To znamená, že term je v normálnej forme.

2. ak vstupný term síce zúžiťeľný je, ale len s takou substitúciou, ktorá porušuje splniteľnosť podmienok pre abstrahovanie. Táto skutočnosť znamená, že výpočet skončí, keďže prepísaný term by predstavoval prázdnu množinu inštancií termov.

3. Ak nenastane žiadny z predošlých prípadov, tak vypočítame použiteľné pravidlá. Ak existuje noetherovské usporiadanie $\succ$ také, že pre každé pravidlo $l \rightarrow r$ platí: $l \succ r$, môžeme prehlásiť, že výpočet na tomto terme skončí. Toto usporiadanie môže byť úplne nezávislé od usporiadania, ktoré používame pri indukcii v dôkaze. Vďaka tomuto môžeme skúsiť viacero usporiadaní. V implementácii sme použili dve, ktoré sú rozobraté v ďalšej sekcii.

6.1.3 Usporiadanie

Ako sme už spomínali, usporiadanie je pri celej metóde dôležité. My však nepracujeme s pevne daným usporiadaním, ale metóda ho dostane na vstupe

ako parameter. V práci sme implementovali dve najpoužívanejšie usporiadania *LPO* a *RPO*, ktoré sú najčastejším nástrojom pri dokazovaní pre väčšinu systémov. Sofistikovanejšie usporiadanie je možné modulárne doplniť do implementácie

Pri obidvoch usporiadaniach sa využíva usporiadanie na funkčných symboloch:

Definícia 6.2. Usporiadanie na funkčných symboloch: *je také čiastočné usporiadanie $>$ definované na $\mathcal{F}$, že v ňom neexistuje nekonečná postupnosť taká, že:*

$$f_1 > f_2 > ... > f_n > ...$$

LPO

Definícia 6.3. LPO: *nech je daná množina funkčných symbolov $\mathcal{F}$ a dvojica termov s, t. Binárna operácia $\succ_{LPO}$ je definovaná na termoch nasledovne:*

$$s \succ_{LPO} t$$

ak $s = f(s_1, ..., s_n)$ a platí aspoň jedna z nasledujúcich podmienok:

1. $\exists i, s_i \succ_{LPO} t$ *alebo* $s_i = t$
2. $t = f(t_1, ..., t_n)$ *a*
   $\exists i(\forall j(j < i \wedge s_j = t_j) \wedge s_i \succ_{LPO} t_i \wedge \forall j(j > i \wedge s \succ_{LPO} t_j)$
3. $t = g(t_1, ..., t_m)$ *a* $f > g \wedge \forall j(s \succ_{LPO} t_j)$

RPO

Definícia 6.4. RPO:*pre danú množinu funkčných symbolov $\mathcal{F}$ a dvojicu termov s, t definujeme $\succ_{RPO}$ reláciu nasledovne:*

$$s \succ_{RPO} t$$

ak $s = f(s_1, ..., s_n)$ *a platí aspoň jedna z nasledujúcich podmienok:*

1. $\exists i, s_i \succ_{RPO} t$ *alebo* $s_i = t$

2. $t = f(t_1, ..., t_n)$ *a* $< s_1, ..., s_n > \succ_{RPO}^{Mul} < t_1, ..., t_n >$

3. $t = g(t_1, ..., t_m)$ *a* $f > g \wedge \forall j (s \succ_{RPO} t_j)$

Kde $\succ_{RPO}^{Mul}$ *je rozšírenie usporiadania* $\succ_{RPO}$ *definovanvé na zoznamoch.*

Pri výpočte teda hľadáme také usporiadanie funkčných symbolov, pri ktorom by bol dôkazový strom konečný. Toto uporiadanie musí byť ale pre všetky vrcholy rovnaké. To znamená, že ak chceme takéto usporiadanie nájsť museli by sme strom prehľadávať do šírky, čo by už pri malom počte funkčných symbolov bolo neefektívne. Preto usporiadanie funkčných symbolov berieme ako súčasť charakteristiky prepisovacieho systému a je vstupom pre *LPO* a *RPO*.

6.1.4 Splniteľnosť podmienok s normálnymi formami

Ako sme už povedali predtým v sekcii 3.2.2 podmienký pre abstrahovanie sú zložené z dvoch typov. Teraz si rozoberieme podmienky pre normálne formy a ukážeme prípady, kedy vieme rozhodnúť o ich nesplniteľnosti. Oproti pôvodnému riešeniu uvedenému v [12] sme tieto prípady rozšírili. Navyše, pri väčšine prípadov sa nepokúšame vyvrátiť splniteľnosť, ale upravujeme celé vrcholy stromu.

Prípad 1: Uvažujme o podmienke $t \downarrow = s$, kde t je základný term, tzn. že neobsahuje žiadne, ani abstrahované, premenné. Zoberme si, že táto podmienka vznikla postupným substituovaním z nejakej pôvodnej podmienky $t'_i \downarrow = X_i$. To znamená, že o terme t'_i vieme, že zastaví. Z toho tiež vieme, že aj každá jeho inštancia zastaví, a teda aj t. Takže môžeme spustiť výpočet na

tomto terme a výsledný term v normálnej forme dostaneme v konečnom čase. Označme tento term t_{nf}. Potom ale dostaneme obyčajnú rovnosť $t_{nf} = s$. Podľa tvaru termu s upravíme podmienky nasledovne:
nech $t_{nf} = f(t_1, ..., t_n)$ a

- $s = X_i$, v tomto prípade vytvoríme substitúciu $\sigma = (X_i \leftarrow t_{nf})$ v celom strome, na aktuálnom terme a aj na podmienkach.
- $s = g(s_1, ..., s_m)$, ak $f \neq g$ potom je táto podmienka vždy nesplniteľná a môžeme celý vrchol označiť za nedosiahnuteľný.
- $s = f(s_1, ..., s_n)$, v tomto prípade najskôr zistíme, či sú t_{nf} a s unifikovateľné. Ak nie, podmienka je vždy nesplniteľná, vrchol je nedosiahnuteľný. Ak unifikovateľné sú, tak z unifikácie dostaneme substitúciu, ktorú aplikujeme na celý vrchol.

Ak by sme dostali pri výpočte na t viac normálnych foriem, strom rozvetvíme a každý vrchol upravíme podľa predchádzajúceho popisu.

Prípad 2: $t \downarrow= s$, kde $hlava(t)$ je funkčný symbol c pričom neexistuje pravidlo $l \rightarrow r$ také, že $hlava(l) = c$. Ak term $s = f(s_1, ..., s_n)$ a $c \neq f$, tak podmienka je vždy nesplniteľná. Ak $c = f$ a $t = c(t_1, ..., t_n)$, tak prvotnú podmienku môžeme nahradiť konjunkciou:

$$t_1 \downarrow= s_1 \wedge ... \wedge t_n \downarrow= s_n$$

Uvažujme term $t = c(t_1, ..., t_n)$. Žiadna inštancia t ale nie je unifikovateľná s ľavou stranou niektorého pravidla. To znamená, že všetky odvodenia z t majú tvar $c(t'_1, ..., t'_n)$, kde $t_i \rightarrow^*_R t'_i$. A teda, ak $c(s_1, ..., s_n)$ je normálnou formou t, tak t_i musí mať formu s_i.

Prípad 3: $t \downarrow= s$, kde term s je redukovateľný. Vtedy existuje pravidlo $l \rightarrow t$ a substitúcia σ taká, že $\sigma l = s$. Potom, s nie je v normálnej forme, čo

je spor. V tomto prípade môžeme prehlásiť podmienku za nesplniteľnú.

Prípad 4: $t \downarrow= s \wedge s \downarrow= u$, v tomto prípade musí platiť, že s a u sú unifikovateľné. Keďže pre každú inštanciácia θ spĺňajúcu podmienku platí, že $\theta t \downarrow= \theta s$ a $\theta s \downarrow= \theta u$. Term s je ale v normálnej forme, z čoho vyplýva, že $\theta s = \theta u$. Teda s a u musia byť unifikovateľné. V skutočnosti môžeme zobrať najvšeobecnejší unifikátor σ pre s a u a aplikovať ho na aktuálny term a tiež na podmienky (pretože pre každú inštanciáciu θ existuje σ_0 taká, že $\theta = \sigma \circ \sigma_0$). Týmto krokom obmedzíme také výpočty, ktoré mohli viesť k nesplniteľnosti danej podmienky.

Prípad 5: $t \downarrow= s$, kde s a t nie sú unifikovateľné. V tomto prípade sa pokúsime zúžiť term t. Podmienka bude nesplniteľná, ak pre všetky termy t_i, do ktorých sme sa takýmto spôsobom dostali, nie je splniteľné $t_i \downarrow= s$. Pre každú z podmienok skúsime úpravu pomocou predchádzajúcich prípadov.

Implementácia

Implementácia prípadov 2 a 3 je priamočiara a nevyžadovala si žiadne špeciálne pomocné funkcie. Pri implementácii prvého prípadu sme museli realizovať interpreter prepisovacieho systému. Tento interpreter nám zo vstupného termu a prepisovacieho systému simuluje celý výpočet pri stratégii innermost.

V prípade 4 sme postupovali mierne odlišne. Vo všeobecnosti sa môže stať, že v podmienkach máme $t_1 \downarrow= t_2 \wedge t_2 \downarrow= t_3 \wedge ... \wedge t_{n-1} \downarrow= t_n$. V tomto prípade však môžeme unifikovať celú takúto "reťaz"termov, to znamená množinu $t_2, t_3, ..., t_n$. Dôvod, ktorý platí pre dvojice podmienok, môžeme takýmto spôsobom zovšeobecniť a rozšíriť ho na množinu termov. Iný prípad je, keď si napríklad zoberieme podmienku $t_1 \downarrow= t_2 \wedge t_2 \downarrow= t_3 \wedge t_2 \downarrow= t_4$. V tomto prípade musia byť unifikovateľné aj termy t_3, t_4. Zoberme si ľubovoľnú inštanciáciu θ spĺňajúcu tieto podmienky. Po jej aplikácii dosta-

neme: $\theta t_1 \downarrow= \ \theta t_2 \wedge \ \theta t_2 \downarrow = \ \theta t_3 \wedge \ \theta t_2 \downarrow= \ \theta t_4$. Avšak t_2 je normálna forma, takže to môžeme zapísať $\theta t_1 \downarrow= \theta t_2 \wedge \theta t_2 = \theta t_3 \wedge \theta t_2 = \theta t_4$. Z tranzitívnosti rovnosti dostaneme $\theta t_2 = \theta t_4$ a teda musia byť unifikovateľné. Na záver ešte jedna drobná úvaha. Zoberme si napríklad takéto podmienky: $t_1 \downarrow= t_2 \wedge t_2 \downarrow= X_i$, kde X_i je abstrahovaná premenná. V takomto prípade nebudeme unifikovať X_i a t_2 z dôvodu, že $t_2 \downarrow= X_i$ sme dostali abstrahovaním termu t'_2, z ktorého postupnou substitúciou vznikol term t_2.
Celkový postup pri spracovávaní všetkých týchto možností bude vyzerať nasledovne: budeme používať množinu termov S a pre danú podmienku $t_1 \downarrow= t_2$

1. nájdeme všetky podmienky, ktoré majú na ľavej strane term t_2 a pravé strany pridáme do množiny S,

2. postup zopakujeme pre každý term t_i z množiny S, až kým nedostaneme tranzitívny uzáver množiny S,

3. na konci vyradíme z množiny S všetky termy, ktoré sú premenné. Tieto premenné môžeme nahradiť jednou a vykonať zodpovedajúce premenovanie (substitúciu),

4. unifikujeme všetky termu z S. Získame unifikátor, ktorý aplikujeme na aktuálny term aj na podmienky,

5. z podmienok pre abstrahovanie vylúčime všetky podmienky tvaru $t_i \downarrow= t_i$.

Takýto postup nám umožňuje postupné aplikovanie úprav popísané v prípade 4 vykonať v jednom kroku. Navyše, zachováva abstrahované premenné a nenahrádza ich znova termami.

Zoberme si nakoniec prípad 5. V tomto prípade pre $t \downarrow= s$ považujeme podmienku za splniteľnú, ak existuje aspoň jedna vetva pri zužovaní termu t, v ktorej dostaneme term unifikovateľný s termom s. Problémom je, že

môže existovať obrovské množstvo možností. Z implementačného hľadiska nemôžeme teda zúžovať term t do šírky, ale musíme to robiť do hĺbky. V tomto prípade sa môže stať, že sa program zacyklí aj napriek tomu, že unifikovateľný term sa v jednej vetve vyskytne. Zoberme si príklad, keď prepisovací systém obsahuje pravidlá:

$f(h(u), v) \rightarrow f(u, v)$
$f(g(u), v) \rightarrow g(a)$

a máme podmienku $f(X_1, X_2) \downarrow= g(X_3)$. Ako vidíme, ak by sme začali zužovať term podľa prvého pravidla, tak by sme sa ocitli v nekonečnej vetve a nastalo by zacyklenie. Vidíme, že podľa druhého pravidla, by sme doslatli term unifikovateľný s termom $g(X_3)$. Preto budeme term zužovať len do vopred určenej hĺbky. V každej vetve a na každej úrovni budeme skúšať upravovať podmienky podľa ostatných pravidiel. Ak sa počas týchto krokov neukáže, že daná podmienka je nesplniteľná, pokladáme ju stále za splniteľnú a pokračujeme vo výpočte.

6.1.5 Disunifikácia

Ako sme už spomenuli, práve podmienky s normálnymi formami nám poskytujú kritériá, vďaka ktorým máme väčšiu šancu rozhodnúť o nerozhodnuteľnosti podmienok pre abstrahovanie. Takisto však potrebujeme nástroje na úpravu a zisťovanie splniteľnosti pre druhú časť podmienok. Druhú časť tvoria podmienky s nerovnosťami. Ich riešenie vo všeobecnosti nie je rozhodnuteľné a zaoberá sa nimi disunifikácia.

Definícia 6.5. Ekvacionálna formula bez kvantifikátorov *(quantifier free equational formula) je formula skladajúca sa z logických spojok* $\wedge, \vee, \neg$, *kde atómami sú buď rovnosti termov alebo symbol* $\top$. *Pre skrátenie výraz* $\neg(s = t)$ *zapíšeme ako* $s \neq t$ *a výraz* $\neg\top$ *ako* $\bot$.

Definícia 6.6. Elementárna formula *pre riešenie disunfikácie bude formula*

$$\bigvee_j \exists \vec{z_j} \forall \vec{y_j} \phi_j$$

kde ϕ_i *je ekvacionálna formula bez kvantifikátorov a* $\vec{z_j}, \vec{y_j}$ *sú množiny premenných.*

Definícia 6.7. Vyriešená forma *(solved form) je vo všeobecnosti formula tvaru:*

$$\bigvee_j \exists \vec{w} : x_1 = t_1 \wedge ... \wedge x_n = t_n \wedge z_1 \neq u_1 \wedge ... \wedge z_m \neq u_m$$

Úlohou disunifikácie je previesť elementárnu formulu na vyriešenú formu. Na riešenie existujú pravidlá, ktorými formulu zjednodušujeme. Ich aplikácia ale nemusí byť konečná. Formuly, ktoré chceme zjednodušovať, sú v tvare $\bigwedge_k \bigvee_{l_k} (t_{l_k} \neq s_{l_k})$, kde t_{l_k} a s_{l_k} sú termy, a tiež výslednú formu budeme očakávať v tvare $\bigwedge_i \bigvee_{j_i} (x_{j_i} \neq u_{j_i})$, kde x_{j_i} sú premenné a u_{j_i} sú termy. Samozrejme, táto formula sa dá previesť do disjunktívnej normálnej formy, aby spĺňala definíciu. Ale keďže my podmienky konštruujeme v konjunktívnej forme, tak výsledná formula by bola mnohonásobne väčšia ako vstupná. Takisto pravidlá, ktoré budeme používať, pracujú s disjunkciou a nie konjunkciou. V našej formule sa nevyskytujú kvantifikátory a ani rovnosť, a preto uvedieme len pravidlá, ktoré sa dajú v našej formule použiť.

Nahradenie:

$$z \neq t \vee P[z] \mapsto z \neq t \vee P[z]\,(z \leftarrow t)$$

Ak $z \notin Var\,(t)$ a ak t je premenná, tak sa musí vyskytovať v P.

Zlučovanie:

$$v \neq s \vee v \neq u \mapsto v \neq s \vee s \neq u$$

Ak s nie je premenná.

Konflikt:

$$f(t_1, ..., t_m) \neq g(s_1, ..., s_n) \mapsto \top$$

Dekompozícia:

$$f(t_1, ..., t_m) \neq f(s_1, ..., s_m) \mapsto t_1 \neq s_1 \vee ... \vee t_m \neq s_m$$

Occur check:

$$x \neq u[x] \mapsto \top$$

Okrem týchto pravidiel z kalkulu, môžeme zaviesť ešte pravidlo:

$$t \neq t \mapsto \bot$$

Toto pravidlo je prirodzené, keďže relácia nerovnosti nemôže byť reflexívna. Navyše, implicitne počas výpočtu používame ešte jedno pravidlo. Ako sme už hovorili, ak máme substitúciu, nebudeme ju dávať do konjunkcie so zvyškom podmienok, ale aplikujeme ju na jednotlivé termy v nich.

$$z = t \wedge P[z] \mapsto P[z]\,(z \leftarrow t)$$

Implementácia

Pri implementácii podmienok sme začali s úpravou nerovností. Pravidlá pre nereflexívnosť, konflikt a occur check boli jednoduché. Navyše occur check sme mali už naimplementovaný kvôli unifikácii. Pre dekompozíciu poznamenáme, že ju nestačilo urobiť len na vstupných termoch, ale bolo by dobré ju urobiť aj na výsledku. Zoberme si príklad, keď
$t_1 = f(g(x_1, x_2), x_3, g(x_4, x_5))$ a
$t_2 = f(g(y_1, y_2), h(y_3), y_4)$
Po aplikovaní pravidla dekompozície na $t_1 \neq t_2$ dostaneme disjunkciu:

$g(x_1, x_2) \neq g(y_1, y_2) \vee x_3 \neq h(y_3) \vee y_4 \neq g(x_4, x_5)$
Ako vidíme, na niektoré podmienky budeme môcť znova použiť dekompozíciu (prípadne aj pravidlá konflikt, occur check a pravidlo pre nereflexívnosť), a preto je pri implementácii vhodné redukovať podmienky pokiaľ sa už nedá použiť žiadne zo spomínaných pravidiel. Na konci teda dostaneme ako výsledok disjunkciu nerovností, $\top$ alebo $\bot$.

Pravidlo nahradenia bolo jednoduché a priamočiare. Z nášho hľadiska bola najzaujímavejšia situácia pri implementácii zlučovania. Zoberme si prvý príklad, keď
$\underline{x_1 \neq t_1} \vee \underline{x_1 \neq x_2} \vee x_2 \neq x_3 \vee x_3 \neq x_4$
a postupne aplikujme pravidlo zlučovania na podčiarknuté podmienky:
$\underline{x_1 \neq t_1} \vee \underline{x_1 \neq x_2} \vee x_2 \neq x_3 \vee x_3 \neq x_4 \mapsto$
$x_1 \neq t_1 \vee \underline{x_2 \neq t_1} \vee \underline{x_2 \neq x_3} \vee x_3 \neq x_4 \mapsto$
$x_1 \neq t_1 \vee x_2 \neq t_1 \vee \underline{x_3 \neq t_1} \vee \underline{x_3 \neq x_4} \mapsto$
$x_1 \neq t_1 \vee x_2 \neq t_1 \vee x_3 \neq t_1 \vee x_4 \neq t_1$

Ako vidíme, podmienka $A_1 = x_1 \neq t_1 \vee x_1 \neq x_2 \vee ... \vee x_{n-1} \neq x_n$ je nesplniteľná práve vtedy, keď je nesplniteľná pomienka $A_2 = x_1 \neq t_1 \vee x_2 \neq t_1 \vee ... \vee x_n \neq t_1$.
1) ak A_1 je nesplniteľná, to znamená, že musí platiť $x_1 = t_1 \wedge x_1 = x_2 \wedge ... \wedge x_{n-1} = x_n$. Vďaka komutativite konjunkcie a nerovnosti môžeme túto podmienku prepísať na $x_n = x_{n-1} \wedge ... \wedge x_2 = x_1 \wedge x_1 = t_1$. Keď si túto podmienku zoberieme ako substitúciu a aplikujeme ju na A_2, tak dostaneme $t_1 \neq t_1 \vee ... \vee t_1 \neq t_1$, čo je tiež nesplniteľné.
2) ak je nesplniteľná A_2, tak musí platiť $x_1 = t_1 \wedge ... \wedge x_n = t_1$. Po takejto substitúcii dostaneme z A_1 nesplniteľnú formulu $t_1 \neq t_1 \vee ... \vee t_1 \neq t_1$.

V inom prípade si zoberme podmienku $x_1 \neq t_1 \vee x_1 \neq t_2 \vee x_1 \neq t_3$
a použime pravidlo zlúčovania:

$x_1 \neq t_1 \vee \underline{x_1 \neq t_2} \vee \underline{x_1 \neq t_3} \mapsto$
$\underline{x_1 \neq t_1} \vee \underline{x_1 \neq t_2} \vee t_2 \neq t_3 \mapsto$
$x_1 \neq t_1 \vee t_2 \neq t_1 \vee t_2 \neq t_3$
Vo všeobecnosti teda môžeme napísať, že podmienka $x \neq t_1 \vee x \neq t_2 \vee ... \vee x \neq t_n$ je splniteľná práve vtedy, keď je nesplniteľná podmienka $x \neq t_1 \vee t_1 \neq t_2 \vee t_1 \neq t_3 \vee ... \vee t_1 \neq t_n$. Toto sa dá ukázať podobným spôsobom ako v predchádzajúcom prípade.

Spojením týchto dvoch vecí dostaneme jeden zložitejší prípad. Zoberme si podmienku a upravme ju
$\underline{x_1 \neq t_1 \vee x_1 \neq x_2} \vee x_2 \neq t_2 \iff$ podľa prvého prípadu
$x_1 \neq t_1 \vee \underline{x_2 \neq t_1 \vee x_2 \neq t_2} \iff$ podľa druhého prípadu
$x_1 \neq t_1 \vee x_2 \neq t_1 \vee t_1 \neq t_2$
Vo všeobecnosti teda upravíme formulu nasledovne:

1. budeme si udržiavať dve množiny: množinu termov T a množinu premenných X,

2. vyberieme si počiatočnú nerovnosť $x_i \neq t_j$, x_i pridáme do množiny X. Ak je term t_j premenná, tiež ho pridáme do X, inak ho pridáme do množiny T,

3. pre každú premennú z množiny X nájdeme vo zvyšných nerovnostiach také, v ktorých sa nachádza, a druhú stranu nerovnosti pridáme do niektorej z množín podľa predchádzajúceho kroku,

4. bod 3 opakujeme, až kým nedostaneme uzáver oboch množín,

5. ak je množina T prázdna, ponecháme pôvodné podmienky, inak si vyberieme nejaký term $t \in T$ a skonštruujeme nové podmienky nasledovne:
$\forall x \in X$ pridáme podmienku $x \neq t$,
$\forall t' \in T, t' \neq t$ pridáme podmienku $t \neq t'$.

Samozrejme je dôležité ponechať všetky ostatné podmienky, ktoré sme takýmto spôsobom nevybrali. Vidíme, že takáto aplikácia pravidla zlučovania nám môže značne pomôcť urýchliť zjednodušovanie podmienok. Po takejto úprave môžeme znova skúšať predošlé pravidlá, najmä dekompozíciu, a proces iterovať, pokiaľ sa dá niečo upraviť.

6.1.6 Vizualizácia

Aby sme videli, ako dôkaz prebiehal, potrebovali sme vizualizovať výsledný dôkazový strom. Výsledkom vykresľovanie je obrázok vo formáte svg. Tento formát slúži na zachytenie vektorovej grafiky v jazyku XML.

Pri implementácii sme sa zaoberali dvoma problémami. Prvým bolo zisťovanie parametrov dôkazového stromu a pozície jednotlivých vrcholov. Tieto pozície sme chceli určovať dynamicky na základe veľkosti samotného vrchola a veľkostí jeho synov. Pri výpočte sme preto museli rekurzívne prejsť celý strom a na základe veľkostí podstromov vypočítavať aktuálnu pozíciu.

Druhým problémom bolo samotné vykresľovanie termov a podmienok. Vykresľovanie textu bolo nečitateľné, písmená a najmä znaky ako zátvorky mali príliš malé odsadenie. V dôsledku toho sme sa rozhodli vykresľovať každý znak osobitne. Pri takomto riešení trvá síce vykresľovanie o malý kúsok dlhšie, no výrazne zlepší čitateľnosť textu.

Kapitola 7

Záver

Práca nás podrobne oboznámila s problémom zastevenia pre systémy na prepisovanie termov a výsledkami v predmetnej oblasti. Keďže daná oblasť je úzko špecializovaná, metód na riešenie nášho problému je málo. Väčšina metód sa zaoberala riešením problému pri špecifickej stratégii. Metódu, ktorá pokrývala viacero stratégií a bola vhodná na rozšírenie, sme našli jednu. Keďže jej implementácia nie je podporovaná autormi, rozhodli sme sa pre vlastnú implementáciu tejto metódy.

Môžeme skonštatovať, že sme vytvorili funkčný poloautomatický nástroj, pomocou ktorého vieme dokazovať zastavenie prepisovacieho systému pri stratégii innermost. Táto implementácia vznikla v programovacom jazyku Haskell. Postup, akým program vznikal je zachytený v kapitolách 4, 5.

Súčasťou nášho riešenia bolo aj preskúmanie a implementácia ďalších problémov súvisiacich s realizáciou danej metódy. Medzi týmito problémami sa nachádzajú aj nerozhodnuteľné problémy, napríklad disunifikácia alebo nájdenie vhodného usporiadania na termoch. Mnohé z nich sme neriešili všeobecne, ale prispôsobili sme ich pre naše potreby. Tým sme vedeli zväčšiť pokrytie prípadov pre ktoré je možné rozhodnúť. Príkladom je disunifikácia, kde sme použili dodatočné podmienky o normálnych formách. Iný prístup

sme zvolili pri usporiadaní. Usporiadanie, ktoré sa má použiť, užívateľ zadá ako parameter alebo využije jedno z predvolených.

Môžeme povedať, že ciele, ktoré sme si stanovili na začiatku, sme naplnili. Ostalo ale množstvo priestoru na rozšírenie. V prvom rade je to implementovanie ďalších stratégií, najmä outermost, lokálne stratégie a užívateľsky definované stratégie. V samotnej implementácii je možnosť vylepšiť doteraz použité riešenia a zefektívniť použité algoritmy. V kategórii súvisiacich problémov sa otvárajú možnosti preskúmať niektoré z nich do väčšej hĺbky. To by mohlo znamenať väčšie pokrytie prípadov, pre ktoré sa dá dokázať, že výpočet na nich zastaví. Tetnto výskum by sa mal týkať hlavne podmienok s normálnymi formami, kde sa dá predpokladať existencia lepšieho riešenia. Druhou možnosťou je zmenšenie interakcie užívateľa pri práci so sytémom, napríklad pre riešenie usporadania nájsť vhodné usporiadanie na funkčných symboloch.

Literatúra

[1] Alarcón, B., Lucas, S. *Termination of innermost contex-sensitive rewriting using dependency pairs.* In *Proceedings of the 6th International Symposium on Frontiers of Combining Systems*, Inai, Springer-Verlag, 2007.

[2] Arts, T., Giesl, J. *Proving innermost normalisation automatically.* In *Proceedings 8th Conference on Rewriting Techniques and Applications, Sitges (Spain),* volume 1232 of *Lecture Notes in Computer Science,* pages 157-171, Springer-Verlag, 1997.

[3] Arts, T., Giesl, J. *Termination of term rewriting using dependency pairs.* In *Theoretical Computer Science 236*, pages 133-178, 2000.

[4] Ben Cherifa, A., Lescanne, P. *Termination of rewriting systems by polynomial interpretations and its implementation.* In *Science of Computer Programming 9*, 2 (Oct.), pages 137-160, 1987.

[5] Borovanský, P., Cirstea, H., Deplange, E., Dubois, H., Kirchner, C., Kirchner, H., Moreau, P.-E., Nguyen, Q.-H., Ringeissen, C., Vittek, M. *ELAN: User Manual.* http://elan.loria.fr/manual-elan/index-manual.html, 2006.

[6] Borovanský, P., Kirchner, C., Kirchner, H., Ringeissen, C. *Rewriting with Strategies in ELAN: a Functional Semantics.* In *International Journal

of Foundations of Computer Science, volume 12, pages 69-95, World Scientific Publishing Company, 2001.

[7] Borralleras, C., Ferreira, M., Rubio, A. *Complete monotonic semantic path orderings.* In *Proceedings of the 17th International Conference on Automated Deduction*, Lecture notes in Computer Science, volume 1831, pages 346-364, Pittsburgh, PA, USA, Springer-Verlag, 2000.

[8] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C. *The Maude 2.0 system.* In *Proceedings of the 14th International Conference on Rewriting Techniques and Applications*, R. Nieuwenhuis, Ed. Lecture Notes in Computer Science, volume 2706, pages 76-87, Springer, 2003.

[9] Comon, H., *Disunification: a Survey.* In J.-L. Lassez and G. Plotkin, editors, *Computational Logic. Essays in honor of Alan Robinson,* chapter 9, pages 322-359. The MIT press, Cambridge (MA, USA), 1991.

[10] Dershowitz, N. *Orderings for term rewriting systems.* In *Theoretical Computer Science 17*, pages 279-301, 1982.

[11] Dershowitz, N., Hoot, C. *Natural termination.* In *Theoretical Computer Science 142(2)*, pages 179-207, 1995.

[12] Fissore, O., Gnaedig, I., Kirchner, H. *Induction for innermost and outermost ground termination.* Technical Report A01-R-178, LORIA, Nancy, France, 2001.

[13] Giesl, J., Middeldorp, A. *Innermost termination of context-sensitive rewriting.* In *Proceedings of the 6th International Conference on Developments in Language Theory (DLT2002)*, Lecture Notes in Computer Science, volume 2450, pages 231-244, Kyoto, Japan, Springer-Verlag, 2003.

[14] Giesl, J., Thiemann, R., Schneider-Kamp, P., Falke, S. *Improving dependency pairs.* In *Proceedings of the 10th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, Lecture Notes in Artificial Intelligence, volume 2850, pages 165-179, Almaty, Kazakhstan, Springer-Verlag, 2003.

[15] Gnaedig, I. *Termination of Priority Rewriting.* In *Proceedings of the 3rd International Conference on Language and Automata Theory and Applications, Lecture Notes in Computer Science*, Springer-Verlag, 2009.

[16] Gnaedig, I., Kirchner, H. *Termination of Rewriting under Strategies.* ACM Transactions on Computational Logic, Volume 10 (2), 2009.

[17] Huet, G., Lankford, D. *On the uniform halting problem for term rewriting systems*, Technical Report 283, INRIA, Le Chesnay, France, 1978.

[18] Kamin, S., Lévy, J.-J. *Attempts for generalizing the recursive path oredering.* INRIA, Rocquencourt, 1982.

[19] Krishna Rao, M. *Some characteristics of strong normalisation.* In *Theoretical Computer Science 239*, pages 141-164, 2000.

[20] Lankford, D. *On proving term rewriting systems are noetherian.* Tehnical Report, Louisiana Tech. University, Mathematics Dept., Ruston LA, 1979.

[21] Lucas, S. *Termination of context-sensitive rewriting by rewriting.* In *Proceedings of the 23rd International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science, volume 1099, pages 122-133, Springer-Verlag, 1996.

[22] Moreau, P.-E., Ringeissen, C., Vittek, M. *A Pattern Matching Compiler for Multiple Target Languages.* In *Proceedings of the 12th Conference on*

Compiler Construction, Warsaw (Poland), G. Hedin, Ed. Lecture Notes in Computer Science, volume 2622, pages 61-76, Springer, 2003.

[23] Prívara, I. *Formálne špecifikácie programov.* Vysokoškolské skriptá, Inštitút Informatiky a Štatistiky, 2008.

[24] Wiki homepage of Haskell *http://www.haskell.org/*

[25] Zantema, H. *Termination of term rewriting by semantic labelling.* In *Fundamenta Informaticae 24*, pages 89-105, 1995.

Prílohy

Prílohy k diplomovej práci sa nachádzajú na priloženom CD.

- zdrojové kódy programu v jazyku Haskell,
- používateľská príručka.