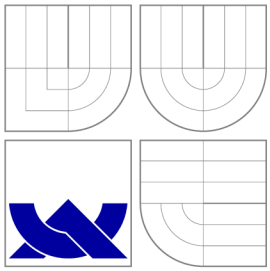
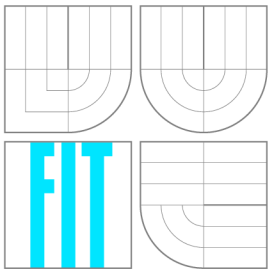


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

OPTIMALIZACE PŘEKLADU AGENTNÍCH JAZYKŮ RŮZNÉ ÚROVNĚ ABSTRAKCE

OPTIMALISATION OF AGENT LANGUAGES COMPILER

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. RÓBERT KALMÁR
AUTHOR

VEDOUCÍ PRÁCE — Ing. FRANTIŠEK ZBOŘIL, Ph.D.
SUPERVISOR

BRNO 2012

Abstrakt

Cílem této práce je optimalizace překladu agentního jazyka AHLL. Jsou představeny různé používané interní formy reprezentace kódu pro překladač jako i optimalizace kódu v těchto reprezentacích. Hlavní část práce je zaměřená implementaci zvolených optimalizací a způsobu generování cílového kódu v jazyce ALLL. Dále se čtenář dozví o přínosu zvolených optimalizací taktéž o dalších možnostech při vývoji jazyka AHLL a jeho optimalizace.

Abstract

The aim of this work is an optimization of AHLL language compiler. Several intermediate representations of compiled code along with code optimization techniques are introduced. The main part of the work is focused on implementing these optimization techniques and generation of the target code in ALLL language. At the end of the work, the results achieved by new version of AHLL compiler are presented. In addition, there are also presented some ideas for the future work on AHLL and the compiler.

Klíčová slova

ALLL, AHLL, překladač, optimalizace, propagace konstant, eliminace mrtvého kódu, propagace kopií, eliminace nedosažitelného kódu , AST, abstraktní syntaktický strom, 3AK, trojadresný kód, agent

Keywords

ALLL, AHLL, compiler, optimization, constant propagation, dead code elimination, copy propagation, unreachable code elimination, AST, abstract syntax tree, 3AK, three-address code, agent

Citace

Róbert Kalmár: Optimalizace překladu agentních jazyků různé úrovně abstrakce, diplomová práce, Brno, FIT VUT v Brně, 2012

Optimalizace překladu agentních jazyků různé úrovně abstrakce

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Františka Zbořila, PhD. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Róbert Kalmár
22. mája 2012

Poděkování

Chcel by som poďakovať vedúcemu diplomovej práce Františkovi Zbořilovi za poskytnutú pomoc pri vyracovávaní tejto diplomovej práce.

Obsah

Kapitola 1

Úvod

Problematike distribuovanej umelej inteligencie na báze umelých agentov a simulácii takýchto systémov je v súčastnosti venované značné výskumné úsilie. Dôkazom je aj to, že na Fakulte Informačních Technológií je táto tematika jedným z hlavných predmetov skúmania výskumnej skupiny inteligentných systémov. S touto problematikou súvisí jednak samotné programovanie a simulácia multiagentných systémov, dôvera a reputácia v týchto distribuovaných a multiagentných systémov a v neposlednom rade možnosti nasadenia umelých agentov v prostredí bezdrôtových senzorových sietí.

V rámci týchto výskumných zámerov bol navrhnutý a implementovaný nízkoúrovňový agentný jazyk ALLL, simulátor distribuovaných multiagentných systémov T-Mass, ako aj platforma pre mobilných agentov v bezdrôtových senzorových sietiach s názvom WSageNt.

V rámci skupiny vznikla potreba návrhu agentného jazyka vyššej úrovne abstrakcie a tak vznikol jazyk AHLL a prekladač tohto jazyka, ktorého cieľovým kódom je jazyk ALLL. Cieľom tejto práce je ďalší rozvoj jazyka AHLL a hlavne návrh a implementácia optimalizačných techník do prekladača tohto jazyka vedúcich k zníženiu veľkosti generovaného kódu.

V úvodných častiach tejto práce si predstavíme súčasný stav problematiky, jazyky ALLL a AHLL ako aj pôvodnú verziu prekladača. Za touto časťou bude nasledovať teoreticky ladená kapitola 4, v ktorej budú predstavené poznatky z optimalizácie prekladačov, ako rôzne formy internej reprezentácie, analýzy a samotné optimalizačné techniky.

V rámci spomínaného rozvoja jazyka AHLL bola v tejto práci mierne zmenená jeho syntax a pribudli v ňom ďalšie vysokoúrovňové konštrukcie. Tomuto bude venovaná kapitola 5. Rovnako v tejto kapitole čitateľ nájde novú štruktúru prekladača jazyka AHLL, ktorá je vhodnejšia na implementáciu optimalizácií.

Kapitola 6 bude venovaná návrhu konkrétnych interných reprezentácií pre prekladač jazyka AHLL. Na na nadviaže kapitola 7 venovaná implementácii vybraných optimalizačných techník.

V kapitole 8 bude predstavený spôsob akým sa generuje cieľový kód v jazyku ALLL z optimalizovanej internej reprezentácie algoritmu. V záverečnej kapitole 9 predstavíme dosiahnuté výsledky novej verzie prekladača a porovnáme ju s pôvodnou verziou, prípadne s ručne napísaným kódom priamo v ALLL. Predstavíme aj možnosti ďalšieho vývoja prekladača vedúceho ku generovaniu efektívnejších kódov.

Kapitola 2

Súčasný stav problematiky

V rámci mojej bakalárskej práce bol navrhnutý jazyk AHLL [5], Agent High Level Language. Tento jazyk by mal umožniť pohodlné vytváranie modelov distribuovaných systémov, prevažne agentných. Tieto modely by sa písali v jazyku AHLL a následne by sa preložili do nízko-úrovňového jazyka ALLL.

V rámci práce bol vytvorený aj prekladač tohto jazyka, kde cieľovým jazykom je jazyk ALLL. Jazyk ALLL (Agent Low Level Language) je nízko-úrovňový jazyk na programovanie mobilných inteligentných agentov[11]. Prvotný návrh tohto jazyka bol predstavený v [10].

V súčastnosti je tento jazyk v 2 mierne odlišných verziách. Jedna verzia je implementovaná priamo v senzorových uzloch, čiže funguje na fyzických uzloch senzorovej siete. Interpret tohto jazyka je umiestnený práve na týchto uzloch senzorovej siete. Tento jazyk ako aj interpret bol navrhnutý v práci [12].

Druhá verzia jazyka ALLL je implementovaná v simulátore T-Mass. T-Mass je nástroj na vytváranie a simuláciu distribuovaných systémov, prevažne vo forme umelých agentov.[11]

V nasledujúcich podkapitolách bude podrobnejšie predstavený súčasný stav jazyka AHLL, prekladača tohto jazyka, simulátor T-Mass a nízko-úrovňový jazyk ALLL.

2.1 Pôvodná verzia AHLL

V tejto časti kapitoly si stručne predstavíme jazyk AHLL v jeho podobe, ktorá bola uvedená v práci [5].

Súčasný jazyk AHLL je imperatívny, štruktúrovaný jazyk, syntaxou podobnou jazyku C[5]. Jazyk podporuje asynchrónne zasielanie správ. Syntax jazyka je nasledovná:

```
plan <meno_plánu>(<zoznam_parametrov>) {
    <zoznam_príkazov>
}*

main(){
    <zoznam_príkazov>
}
```

Kľúčové slovo `plan` slúži na deklaráciu plánov, plán je obdoba funkcií v programovacom jazyku C[5]. Definovaných plánov môže byť ľubovoľné množstvo. `main` určuje vstupný

bod programu. Premenné sa deklarujú kľúčovým slovom `var`, podporované komentáre sú riadkové komentáre z jazyka C: `// komentar`.

V skratke si predstavíme jednotlivé príkazy a štruktúry jazyka AHLL. Väčšina príkazov je ukončená bodkočiarkou, výnimku tvoria podmienené príkazy a cykly. Príkazy patriace do cyklu, resp. podmienky sú uzavreté v bloku. Blok je skupina príkazov v zložených zátvorkách ({ }).

Samozrejmosťou je prázdny príkaz, v tvare `;`. Pokračujeme deklaráciou a inicializáciou premennej. Na deklaráciu premennej slúži kľúčové slovo `var`, nasledované identifikátormi jednotlivých premenných oddelených čiarkou. Inicializácia premennej sa vykonáva priradením konštanty, hodnoty inej premennej alebo výsledku nejakého výrazu do inicializovanej premennej. AHLL rovnako podporuje prácu s výrazmi. Výraz vo forme príkazu je ukončený bodkočiarkou.

Podmienený príkaz je v tvare `if (<podmienka>) <blok> else <blok>`. Vetva `else` je voliteľná. Podľa hodnoty podmienky, ktorý je vlastne výrazom, sa vykonajú inštrukcie vo vetve `if` alebo `else`, tak ako je to známe z iných programovacích jazykov.

Z cyklov je implementovaný cyklus s podmienkou na začiatku, ktorý má tvar `while (podmienka) <blok>`. Sémantický význam tejto operácia je, že kým je podmienka splnená sa vykonávajú príkazy z bloku `<blok>`. Podmienkou je rovnako ako v predchádzajúcom prípade výraz.

Na asynchrónnu komunikáciu slúžia príkazy `send(adresa, správa)` na odoslanie správy a `receive(adresa?) => <identifikátor>` na jej prijatie. V prípade prijímania správy je adresa nepovinná, ak nie je uvedená, vyberie sa prvá správa z príslušnej štruktúry simulátora alebo senzorového uzlu. Oba tieto príkazy sú ukončené bodkočiarkou. Na uloženie prijatej správy slúži operátor `=>`, nasledovaný premennou. Táto konštrukcia nie je povinná.

Ako sme už spomínali v úvode, AHLL podporuje definíciu dodatočných plánov. Tieto plány sa volajú rovnako ako napr. funkcie v jazyku C: `<plan> ( <parametre> );` , kde `<plan>` je meno plánu a `<parametre>` je čiarkou oddelený zoznam premenných alebo konštánt.

Posledným príkazom je príkaz na volanie služby platformy v tvare `platform( <parametre> ) => <identifikátor>`. Spôsobí vyvolanie služby platformy definovanými v parametroch. Ak služba poskytuje nejaký výsledok zo svojej činnosti ten je možné uchovať už spomínaným operátorom `=>`, ktorý je opäť nepovinný.

Vrátime sa spať k výrazom. Operátory, podporované v jazyku AHLL sú uvedené v tabuľke 2.1 a 2.2. Ich priorita je uvedená v tabuľke 2.3.

Viac informácií ohľadne pôvodnej verzie jazyka AHLL ako aj jej gramatiku vo forme EBNF je možné nájsť v práci [5]

2.2 Pôvodná verzia prekladača

Pôvodný prekladač vytvorený v rámci bakalárskej práce [5] a prekladá do verzie jazyka ALLL kompatibilnej s verziou v senzorových uzloch. Je písaný v jazyku C++ a na lexikálnu a syntaktickú analýzu využíva nástroj ANTLR. Sémantická analýza sa vykonáva pri priechode abstraktným syntaktickým stromom a generovaní cieľového kódu.

Štruktúra tohto prekladača je zobrazená na obrázku 2.1. Ako je zo štruktúry prekladača jasné preklad prebieha v 2 krokoch.

Prvým krokom je lexikálna a syntaktická analýza zdrojového kódu. Ako už bolo spomenuté vyššie, tieto analyzátory boli generované pomocou nástroja ANTLR, prekladom

Operátor	Operácia
$*$	násobenie
$/$	delenie
%	modulo
$+$	súčet
$-$	rozdiel
$>$	väčší než
$>=$	väčšie rovno
$<$	menší než
$<=$	menšie rovno
$==$	rovno
$!=$	nerovno
&&	logický súčin
$\|\|$	logický súčet
$=$	operátor priradenia

Tabuľka 2.1: Binárne operátory

Operátor	Operácia
$+$	unárne plus
$-$	unárne mínus
!	negácia

Tabuľka 2.2: Unárne operátory

Unárne operátory:

$+$ $-$!

Unárne operátory majú asociativitu zľava-doprava.

Binárne operátory

(matematické a relačné):

$*$ $/$ %

$+$ $-$

$<$ $>$ $<=$ $>=$

$==$ $!=$

&&

$||$

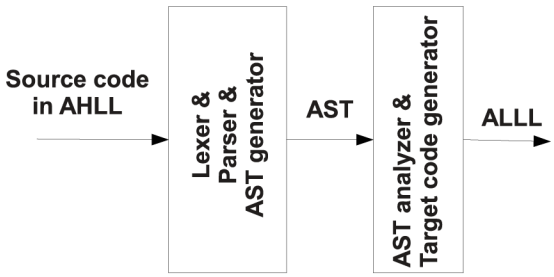
Uvedené binárne operátory majú asociativitu zľava-doprava.

Operátor priradenia:

$=$

Operátor priradenia má najnižšiu prioritu, asociativita tohto operátora je sprava-doľava.

Tabuľka 2.3: Priorita operátorov AHLL



Obr. 2.1: Pôvodná štruktúra prekladača jazyka AHLL

súboru s gramatikou jazyka AHLL. Výstupom z tejto analýzy je abstraktný syntaktický strom.

Tento abstraktný syntaktický strom vstupuje do druhej fázy prekladu. V tejto fázy sa AST analyzuje, hlavne z hľadiska sémantickej správnosti a rovno sa generuje cieľový kód v jazyku ALLL. Sémantická analýza na svoju prácu využíva dve pomocné štruktúry. Sú nimi tabuľka symbolov, lepšie povedané zásobník symbolov. Na rozdiel od tabuľky symbolov totiž neobsahuje v sebe všetky symboly definované v programe, ale len tie, ktoré sú aktuálne prístupné.

Druhou štruktúrou je tabuľka plánov, ktorá nesie informácie o všetkých dovtedy definovaných plánoch v programe.

Z predchádzajúceho textu je jasné, že pôvodný prekladač negeneruje žiadnu vnútornú reprezentáciu programu, vo forme medzikódu. Priamo generuje cieľový kód. To je aj jedným z dôvodov prečo prekladač nevykonáva takmer žiadnu optimalizáciu výsledného kódu.

Táto situácia je značne nepríjemná v systémoch typu senzorové siete, kde samotné uzly majú relatívne malú pamäť na prijatie kódu agenta. Z tohto dôvodu je potrebné zaviesť do existujúceho prekladača modul na optimalizáciu výsledného kódu.

Kapitola 3

Jazyk ALLL

V tejto kapitole budú predstavené základy jazyka ALLL, ktorý je okrem iného cieľovým jazykom pri preklade jazyka AHLL. Jedná sa o nízko-úrovňový jazyk na programovanie a modelovanie diskrétnych systémov s asynchrónnym preposielaním správ. Bol vyvinutý na FITe a ako už bolo spomínané, v súčasnosti existuje v 2 verziách. [11]

V tejto kapitole si predstavíme verziu jazyka dostupnú v simulačnom nástroji T-Mass. Dôvodom je, že prekladač jazyka AHLL má podporovať práve túto verziu jazyka. V tejto verzii jazyka sa už neplánujú nejaké väčšie zmeny, narozdiel od verzie druhej.

Kapitola bola spracovaná za pomoci zdrojov [11, 10] a zdrojových kódov simulátora T-Mass.

3.1 Základné prvky jazyka

V tejto časti predstavíme základné prvky jazyka ALLL, ktorými sú atómy a zoznamy.

Atóm v ALLL je neprázdna postupnosť písmen, číslic a znakov: podtržník, plus, mínus a bodka. Čiže formálne:

```
atóm = ( (['a'-'z'] | ['A'-'Z'] | ['0'-'9'] | '_' | '+' | '-' | '.') )+
```

Zoznamy sa delia na tzv. dotazovací zoznam a bežný zoznam. Bežný zoznam je postupnosť prvkov uzavretá v okrúhlych ('(' ')') zátvorkách. Prvkom zoznamu môže byť atóm, číslo, register, prípadne iný bežný zoznam.

Dotazovací zoznam je postupnosť prvkov uzavretá v hranatých zátvorkách ('[' ']'). Prvkom tohto zoznamu môže byť atóm, číslo, register alebo ďalší dotazovací zoznam. Navyše v tomto zozname môže byť tzv. anonymná premenná, ktorá je reprezentovaná znakom podtržník ('_').

3.2 Základná štruktúra jazyka

ALLL umožňuje definovať zámer agenta, obsah jeho báze znalostí ako aj obsah báze plánov. V bázy plánov sú umiestnené ďalšie plány agenta, ktoré sa v rámci zámeru agenta môžu vykonať.

Zámer agenta sa definuje kľúčovým slovom `intention`, nasledovaný bodkočiarkou a plánom. Zámer agenta je plán (postupnosť akcií), ktorý sa vykoná po spustení agenta. Pri definovaní agenta je definovanie jeho zámeru povinné. Formálne sa zámer agenta definuje:

```
intention = ">intention" ':' plan ';'
```

Konštrukcia `plan`, je plán, ktorý je definovaný ako čiarkou oddelený zoznam akcií:

```
plan = '<' akcia ( ',' akcia )* '>'
```

Definícia obsahu báze znalostí agenta je nepovinná. Do tejto báze znalostí sa ukladajú zoznamy alebo spustiteľné plány. Zoznamy tvoria znalosti agenta, plány sú pomenované spustiteľné postupnosti akcií. Tieto plány sa dajú vkladať do báze znalostí priamo pri definícii agenta, alebo ich agent môže vložiť sám, o tom bude reč neskôr. Spustiť plán je možné pomocou akcie nepriameho spustenia plánu, o ktorej bude reč takisto neskôr.

Formálne, definícia bázy znalostí agenta:

```
beliefBase = ">database" ':' (generalList | ('{' atom ';' plan '}') )*
```

Pre prehľadnosť uvedieme príklad definície agenta v jazyku ALLL:

```
> intention : < +(a,2),-(a,2)> ;

> database : (b,3);
             {vymaz ; <-(b,3)>};
```

3.3 Akcie

Ako sme už spomínali plány, sú postupnosťami akcií. V tejto časti si priblížime jednotlivé akcie jazyka ALLL, vysvetlíme si ich funkčnosť a uvedieme ich syntax.

Základné akcie v jazyku ALLL sú nasledovné:

spustenie plánu Spustenie plánu sa delí na priame a nepriame spustenie. Priame spustenie má tvar `@(plan)`, kde `plan` je konštrukcia uvedená v podkapitole 3.2. Pri nepriamom spustení sa volá plán, ktorý je uložený v báze znalostí. Nepriame volanie má tvar `@^(atom)`, kde atom je identifikátor, tj. meno plánu. V prípade, že plán s takýmto menom neexistuje, akcia zlyhá.

dotaz Je to akcia, ktorá vyhľadáva zoznamy v báze znalostí alebo volá služby platformy. Je v tvare `#zoznam`, kde zoznam môže byť bežný zoznam s okrúhlymi zátvorkami alebo dotazovací zoznam. V prípade dotazovacieho zoznamu je možné použiť anonymnú premennú. Nájdené n-tice sa uložia do aktívneho registru. Ak žiaden zoznam z bázy znalostí nezodpovedá dotazu, akcia zlyhá.
V prípade služieb platformy, prvým prvkom zoznamu je požadovaná služba platformy, ďalším je zoznam reprezentujúci parametre danej služby.

pridanie do bázy znalostí Do bázy znalostí je možné pridať zoznam alebo spustiteľný plán. Pridanie zoznamu má tvar `+bežný_zoznam`. Pridanie plánu má tvar `+{item,plan}`, kde `item` je atóm, číslo alebo registrová konštrukcia, `plan` je plán. Po pridaní do bázy znalostí je plán možno spustiť pomocou nepriameho spustenia plánu.

odstránenie z bázy znalostí Aj v tomto prípade je možné odstrániť zoznam alebo plán. Odstránenie zoznamu sa deje pomocou `-zoznam`, kde zoznam je bežný zoznam alebo dotazovací zoznam. Odstránenie plánu má tvar `-^(item_ap)`, kde `item_ap` je atóm, číslo, podtržník alebo register.

odoslanie správy Odoslanie správy má tvar `!(item,bezny_zoznam)`, kde item je atóm, číslo alebo register. Táto akcia spôsobí odoslanie správy na agenta `item` a obsahom správy je `bezny_zoznam`.

prijatie správy Táto akcia slúži na prijatie správy od iného agenta alebo senzoru. Správa sa ukladá do aktívneho registra. Má tvar `?(item_ap)`.

zmena aktívneho registra Táto akcia spôsobí nastavenie daného registra na aktívny. Má tvar `$cislo`, kde `cislo` je číslo registra, ktorý nastavujeme na aktívny.

3.4 Plány

Definovanie plánov už bolo predstavené v predchádzajúcich častiach, tuto to dopĺňame len pre úplnosť. Plán sa definuje nasledovne:

```
plan = '<' akcia ( ',' akcia)* '>'
```

Plán skončí úspešne ak skončia úspešne všetky jeho akcie. Ak niektorá zlyhá, zlyháva celý plán. Zvyšné akcie tohto plánu sa odstránia z aktuálneho zámeru agenta a pokračuje ďalšou akciou nadradeného plánu.

3.5 Registre a práca s nimi

Jazyk ALLL poskytuje 3 registre všeobecného použitia. Tieto sú označené znakom `&` nasledované číslom v rozsahu 1–3, to znamená, že v ALLL sú nasledovné registre: `&1`, `&2`, `&3`.

Registre slúžia na uloženie dočasných hodnôt. Vždy jeden z registrov je tzv. aktívny. Do aktívneho registra za zapisujú výsledky jednotlivých akcií prípadne výsledky služieb platformy. Register môže byť parametrom akcie alebo služby platformy. Pri vykonávaní akcie sa identifikácia registra nahradí jeho obsahom. Do týchto registrov je možný zápis, ale len pomocou príslušných akcií, ktoré budú predstavené ďalej.

Zmena aktívneho registra sa robí akciou `$`, nasledovanou číslom registru, ktorý sa má stať aktívnym[1], napríklad akcia `$1` nastaví register číslo 1 na aktívny.

Ďalšie 2 registre sú nastavované automaticky interpretom jazyka ALLL a je možné ich len čítať. Jedná sa o registre `&N` a `&L`. V registri `&N` je uložený názov agenta. V registri `&L` je uložená aktuálna úroveň zanorenia plánov. Napríklad, ak plán `rekurzia` v rámci svojich akcií spustí ďalší plán (napr. aj sám seba), tak akcia na pridanie do báze znalostí v tvare `+(a,&L)` spôsobí uloženie n-tice `(a,1)`. Pri ďalšom zanorení `(a,2)`, a tak podobne.

S registrami súvisí aj funkcia znaku apostrof ('), ktorého správanie si predstavíme. Vyššie sme spomenuli, že pri vykonaní akcie sa identifikátor registru nahradí jeho obsahom. Spustenie plánu, či už priame alebo nepriame je takisto akciou v ALLL. To pravdaže znamená, že všetky výskyty identifikátorov registrov sa nahradia ich obsahom v čase spustenia danej akcie. Istým spôsobom sa môže jednať o predávanie parametrov hodnotou.

Ak chceme vo vnorenom pláne použiť registre, nie ich hodnoty v čase spustenia plánu musíme nejakým spôsobom potlačiť spomínané nahradenie identifikátoru registrov. Práve na toto slúži znak apostrofu, ktorý sa zapisuje pred identifikátor registru.

Pri každom spustení plánu sa jeden apostrof odstráni. Pokúsime sa sumarizovať správanie sa registrov v prípade akcie spustenia plánu:

[1] nie je chybou ak v danom okamihu je register aktívnym registrom

- V prípade, že identifikátor registru pred sebou nemá žiaden apostrof nahradí sa aktuálnym obsahom daného registru.
- V prípade, že identifikátor registru má pred sebou práve jeden apostrof, ten sa odstráni a v pláne ostane len jeho identifikátor, napríklad `&1`. V tomto prípade je register voľne použiteľný v akciách daného plánu.
- V prípade, že identifikátor registru má pred sebou viac apostrofov, opäť sa jeden odstráni. Tento register je však zablokovaný a pri spustení akcie, v ktorej je tento identifikátor, plán končí neúspechom.

Pre názornosť si ukážeme túto funkcionalitu na jednoduchom príklade:

Príklad 3.1. Nech báza znalostí obsahuje n-tice `(1),(2),(3)`. Predpokladajme nasledovný kód v ALLL:

```
>intention: <
     #[1],#(lst,(hh,&1)),$2,#[2],#(lst,(hh,&2)),$3,#[3],#(lst,(hh,&3)),
     @<#(tml,(p,'&L)),#(ari,(k,&2,&3)),
       #(ari,(k,&2,'&3)),#(ari,(k,''&3,&2)),#(tml,(p,&L))
      >
     >
```

Po vykonaní prvého riadku sa naplnia registre hodnotami $\$1 = 1$, $\$2 = 2$, $\$3 = 3$. Aktívnym registrom je 3. register. Po spustení vnoreného pod-plánu (druhý a tretí riadok) sa zámer agenta zmení na:

```
<(tml,(p,$L)),(ari,(k,2,3)),(ari,(k,2,$3)),(ari,(k,'$3,2)),(tml,(p,0)),;>
```

Je vidieť, že všetky registre bez apostrofu boli nahradené hodnotami. V 3. akcii pri identifíkátore registra č. 3 bol apostrof, ten sa pri spustení podplánu odstránil, rovnako ako aj pri 1. akcii pri `&L`. 4. akcia spôsobí pád plánu, lebo obsahuje identifikátor registra s apostrofom.

Kapitola 4

Teoretické základy optimalizácie

V tejto kapitole predstavíme teoretické základy optimalizácie prekladaných programov. V sekcii 4.1 budú predstavené rôzne interné reprezentácie algoritmov. Následne bude predstavená analýza toku riadenia prekladaných programov aj so základnými definíciami z teórie grafov. Posledná a najrozsiahlejšia časť tejto kapitoly bude venovaná rôznym optimalizáciám. Budú predstavené a podrobne vysvetlené pseudokódy týchto optimalizačných techník.

4.1 Medzikód

Pod pojmom medzikód sa chápe istá interná forma reprezentácie algoritmu popísaného v zdrojovom jazyku. Existuje viacero rôznych druhov medzikódov.

Návrh alebo voľba vhodného medzikódu je ovplyvnená formou zdrojového jazyka, cieľového jazyka, ako aj vhodnosťou danej internej reprezentácie na jednotlivé optimalizácie.

Podľa úrovne abstrakcie ich môžeme rozdeliť na High Level Representation (HIR), Medium Level Representation (MIR) a Low Level Representation (LIR) [6, s. 67], ktoré si predstavíme v nasledujúcich častiach tejto sekcie. Najvyššia úroveň (HIR) je najbližšie zdrojovému jazyku. Každá nižšia úroveň je syntakticky a sémanticky bližšie k cieľovému jazyku, prípadne k cieľovej architektúre.

Program je pri preklade transformovaný do týchto medzikódov, postupne analyzovaný a optimalizovaný. Záverečnou fázou prekladu je generovanie cieľového kódu. Ten sa väčšinou generuje z LIR prípadne MIR reprezentácie.

4.1.1 High Level Representation

Tento medzikód sa pri prekladačoch používa najčastejšie len v úvodných fázach prekladu. Najčastejšou používanou formou HIR kódu je abstraktný syntaktický strom (AST[1]). Táto forma väčšinou explicitne vyjadruje reprezentáciu zdrojového kódu daného programu a je z nej možné pôvodný zdrojový kód takmer úplne zrekonštruovať. [6, s. 69].

Formálne sa jedná o stromovú štruktúru, matematicky neorientovaný acyklický graf [9]. Tento strom vyjadruje syntaktickú štruktúru zdrojového kódu. Na rozdiel od parsovacieho stromu, ktorý reprezentuje non-terminály a terminály, sú z AST vynechané tzv. pomocné non-terminály a terminály [1, s. 69-77], ako napríklad komentáre, zátvorky a pod. Komentáre sú v prípade prekladu a analýzy zdrojového kódu nepodstatné. Na druhej strane

[1] Abstract Syntax Tree

L1:	$i \leftarrow i + 1$	(1)	$i + 1$
		(2)	$i\ sto\ (1)$
	$t1 \leftarrow i + 1$	(3)	$i + 1$
	$t2 \leftarrow p + 4$	(4)	$p + 4$
	$t3 \leftarrow *t2$	(5)	$*(4)$
	$p \leftarrow t2$	(6)	$p\ sto\ (4)$
	$t4 \leftarrow t1 < 10$	(7)	$(3) < 10$
	$*r \leftarrow t3$	(8)	$r\ *\ sto\ (5)$
	$if\ t4\ goto\ L1$	(9)	$if\ (7), (1)$

Obr. 4.1: Porovnanie 3AK (vľavo) a trojíc (vpravo)

zátvorky nie sú potrebné, lebo štruktúra výrazov je zrejmá už zo samotnej stromovej reprezentácie programu.

4.1.2 Medium Level Representation

Ako už názov tejto skupiny medzikódov napovedá, jedná sa o úroveň, ktorá je sémanticky a syntakticky niekde na pomedzí medzi cieľovým a zdrojovým jazykom. Tieto reprezentácie sú vo všeobecnosti navrhované tak, aby reflektovali možnosti zdrojového jazyka, buď vo forme, ktorá je nezávislá na cieľovom jazyku alebo naopak vo forme relatívne blízkej cieľovej architektúre. Predstavíme si zopár príkladov týchto jazykov.

Najznámejšou formou je troj-adresný kód (3AK), v literatúre nazývaný aj názvom štvorica [6, s. 96]. Je to lineárna forma reprezentácie, ktorá je tvorená operátorom, adresou výsledku a operandmi. Vo väčšine prípadov je počet operandov 2 (binárne aritmetické operátory), z toho aj vychádza názov tohto kódu (obsahuje 4 údaje). Troj-adresný kód tvorí lineárnu postupnosť príkazov, ktorých vykonaním sa vykoná samotný algoritmus.

Pravdaže binárne aritmetické operátory tvoria len jednu časť príkazov. Ďalšími sú napríklad unárne operátory, príkaz volania procedúry, pri ktorej je počet údajov väčšinou premenný (meno procedúry, adresa výsledku, premenný počet parametrov), príkazy vetvenia programu, cyklov a skoky.

Ďalšou formou reprezentácie sú trojice. Sú veľmi podobné predchádzajúcej forme, ktorou bol 3AK. Rozdielom medzi nimi je, že kým v 3AK je výsledok (jeho adresa) explicitne uvedený, v prípade trojíc to tak nie je. Výsledky majú implicitné mená, ktoré môžu byť odvodené od poradia danej trojice. Toto výrazne sťažuje pridávanie, prípadne odoberanie trojíc do už existujúceho kódu. Porovnanie 3AK a trojíc je uvedené na obrázku 4.1 Používanie trojíc ako medzikódu nemá takmer žiadne výhody pri optimalizácii prekladu jazykov. [6, s. 96].

Ďalšími formami internej reprezentácie na tejto úrovni sú stromy, DAG, poľská (prefixová) notácia a iné.

4.1.3 Low Level Representation

Táto forma reprezentácie je zo všetkých najbližšie k cieľovému jazyku. Korešponduje takmer 1:1 k inštrukciám (prípadne akciám, príkazom alebo primitívam) cieľovej architektúry. V prípade MIR, adresy operandov sú často ukazateľmi na symboly v tabuľke symbolov. V prípade LIR sú to už reálne adresy do pamäte, registre a pod.

Abstrakcia LIRu oproti cieľovej architektúre môže spočívať napríklad v tom, že LIR môže obsahovať operáciu celočíselného násobenia aj v prípade, že cieľová architektúra túto operáciu priamo neimplementuje. Prípadne opačný prípad, kde LIR obsahuje len základný adresový mód (register + register, register + konštanta) a cieľová architektúra môže obsahovať pokročilejšie adresovacie módy. [6, s. 71]

Výber konkrétnych inštrukcií je poslednou fázou prekladu programu až na prípadne optimalizácie priamo nad inštrukciami cieľovej architektúry. [6, s. 71]

4.2 Analýza toku riadenia

V kapitole 4.1 sme predstavili najčastejšie používané medzikódy. Väčšina optimalizácií sa vykonáva na MIR a LIR reprezentáciách [6, s. 67]. Tieto kódy sa vyznačujú tým, že program reprezentujú ako lineárnu postupnosť inštrukcií alebo akcií. Na rozdiel od zdrojového kódu, prípadne AST, sa v týchto reprezentáciách stráca explicitná informácia o toku kódu, ako napríklad podmienené vykonávanie niektorých častí programu, cykly a pod.

Práve z tohto dôvodu reprezentácia vo formách MIR-u/LIR-u prechádza rôznymi analýzami. Cieľom týchto analýz je "aby si prekladač vytvoril celkový pohľad na to, ako program používa dostupné zdroje" [7, s. 169].

Jednou z najdôležitejších analýz je analýza toku riadenia. Neformálne, je to statická analýza [2], ktorá zisťuje ktorými všetkými vetvami môže vykonávanie programu viesť. Jej výstupom je tok riadenia programu, ktorý je reprezentovaný grafom toku riadenia [2].

Keďže sa jedná o analýzu, ktorej výstupom je graf, uvedieme nevyhnutné matematické definície z oblasti teórie grafov.

Definícia 4.1. (*Orientovaný graf – oriented graph.*) Orientovaný graf G je dvojica (N,E), kde N je neprázdna množina uzlov a $E \subseteq N \times N$ je konečná množina orientovaných hrán medzi uzlami. [7, s. 273]

Hrany zapisujeme buď vo forme dvojice (a, b) alebo častejšie $a \to b$. Ďalej definujeme funkcie nasledovníka a predchodcu uzlu [6, s. 175]:

$$Succ(b) = \{n \in N | \exists e \in E \text{ také, že } e = b \to n\}$$

$$Pred(b) = \{n \in N | \exists e \in E \text{ také, že } e = n \to b\}$$

Jak sme si uviedli základné definície pokročíme k formálnej definícii grafu toku a graf toku riadenia.

Definícia 4.2. (*Graf toku – flow graph.*) Graf toku je štvorica $(N, E, Start, Stop)$, kde (N, E) je orientovaný graf, $Start \in N$ a $Stop \in N$ sú špeciálne uzly, pre ktoré platí, že $Pred(Start) = \emptyset$ a $Succ(Stop) = \emptyset$. V tomto grafe existuje cesta z uzlu $Start$ ku každému inému uzlu grafu. Rovnako, z každého uzlu grafu existuje cesta do uzlu $Stop$. [7, s. 273]

Definícia 4.3. (*graf toku riadenia – control flow graph.*) Graf toku riadenia je graf toku, kde každý uzol je základným blokom a každá hrana grafu reprezentuje cestu, ktorou môže vykonávanie programu pokračovať. [2]

V definícii 4.3 bol uvedený ďalší pojem, ktorým je základný blok:

Definícia 4.4. *(Základný blok – basic block.)* Základný blok je lineárna postupnosť príkazov (napríklad v 3AK), ktorá má jeden vstupný bod a jeden výstupný bod [2].

Inštrukcie v základnom bloku sa vždy vykonajú všetky, v poradí v akom sú uvedené v základnom bloku. Základný blok môže mať niekoľko predchodcov ako aj následníkov [2].

Rozdelenie vstupného programu na základné bloky je jednou z hlavných častí analýzy toku riadenia. Druhou je určenie následníkov a predchodcov (funkcie *Succ* a *Pred*) každého základného bloku. Týmto spôsobom je vytvorený kompletný graf toku riadenia.

Najprv si predstavíme algoritmus identifikovania základných blokov. Prvým krokom je identifikácia tzv. *leader* inštrukcií v programe. Sú to inštrukcie, ktoré sú prvými inštrukciami v základných blokoch [1, s. 526]. Tieto vymedzujú jednotlivé základné bloky. Takže základný blok je postupnosť inštrukcií od jednej *leader* inštrukcie k nasledujúcej, pričom táto je už súčasťou ďalšieho základného bloku.

Leader inštrukcie sú [1, s. 526][6, s. 173]:

1. prvá inštrukcia v programe, prípadne procedúre (vstupný bod)
2. návestie, alebo inštrukcia, ktorá je cieľom nejakej inštrukcie skoku
3. inštrukcia, ktorá nasleduje za inštrukciou skoku

V základných blokoch identifikujeme 2 špeciálne bloky alebo uzly. *Uzol vetvenia*, tzv. *branch*, ktorý má viac ako jedného následníka. Jeho opakom je *uzol spojenia*, tzv. *join*, ktorý má viac ako jedného predchodcu. [6, s. 175]

Po rozdelení programu na základné bloky ostáva už len určiť hrany grafu toku riadenia (funkcie *Succ* a *Pred*). Z bloku B vedie hrana do bloku C, vtedy a len vtedy, ak je možné v nejakom behu programu vykonať za poslednou inštrukciou bloku B okamžite prvú inštrukciu z bloku C. Toto je splnené ak je posledná inštrukcia bloku B skokovou inštrukciou na prvú inštrukciu bloku C. Alebo ak prvá inštrukcia bloku C nasleduje za poslednou inštrukciou v bloku B.

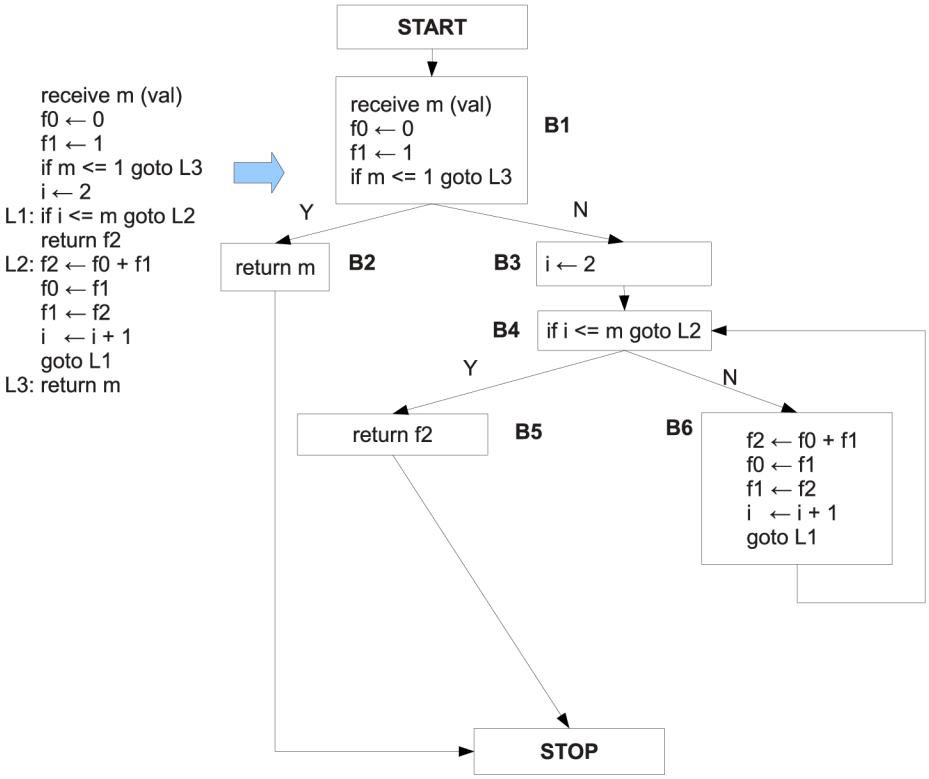
Na obrázku 4.2 je uvedený príklad rozkladu programu na základné bloky a vytvorenie grafu toku riadenia z týchto základných blokov. Bloky B1, B4 sú uzly vetvenia. Uzly B4 a STOP sú uzly spojenia. Ako je vidieť jeden uzol môže byť zároveň uzlom spojenia aj vetvenia, príkladom je uzol B4.

Ďalej si definujeme tzv. *rozšírený základný blok*. Rozšírený základný blok je maximálna postupnosť inštrukcií začínajúca *leader*-om, ktorá neobsahuje žiaden spájajúci (join) uzol. Čiže rozšírený základný blok má jeden vstup a môže mať viacero výstupov. Jeho obdobou je *reverzný základný blok*, ktorý je maximálnou postupnosťou inštrukcií, končiacich uzlom vetvenia (*branch*), neobsahujúci žiaden iný vetviaci uzol. [6, s, 175]

4.3 Optimalizačné algoritmy

Túto sekciu začneme objasnením pojmu optimalizácia. Optimalizácia v doméne prekladačov je technika, lepšie povedané súhrn rôznych techník, ktoré sa snažia zvýšiť výkon prekladaného programu. „Vo všeobecnosti optimalizácia zlepšuje výkon, niekedy podstatne, aj keď existuje reálna možnosť, že výkon zníži alebo v lepšom prípade nebude mať žiadny vplyv na niektoré (prípadne všetky) vstupy daného programu“[6, s. 319]. Problém, či daná optimalizačná technika zvýši alebo zníži výkon programu, je formálne nerozhodnuteľný [6, s. 319].

Pri optimalizácii prekladaného programu je snaha byť agresívny ako sa len dá, ale len do takej miery, aby sme dostali korektný program.



Obr. 4.2: Príklad vytvorenia grafu toku riadenia z 3AK [6, s. 170]

Vo všeobecnosti existujú 2 kritériá rozhodujúce o použití danej optimalizačnej techniky. Sú nimi čas a priestor[6, s. 320]. Podstatnosť týchto kritérií silne závisí od cieľovej architektúry a jej vlastností. Mnoho optimalizačných techník zvyšuje rýchlosť a zároveň znižuje priestor[6, s. 320]. Takže nejedná sa o 2 protichodné kritériá.

V našom prípade, ktorým je optimalizácia cieľového kódu v jazyku ALLL, je podstatnejšia optimalizácia na priestor používaný výsledným programom. Dôvodom je obmedzená pamäť na agenta v prípade agentov v senzorových uzloch. Ďalším dôvodom je to, že optimalizácie zvyšujúce rýchlosť na úkor priestoru, sú často založené napr. na rozbalení slučiek[6, s. 320]. Tým sa síce ušetria inštrukcie na skok, ale hlavne sa zníži tzv. *cache missrate*. V prípade jazyka ALLL, ktorý ako taký nemá inštrukcie skoku ani cache, by tieto optimalizácie nepriniesli žiaden efekt. V prípadoch, ako napríklad rozbalenie slučiek, by to mohlo naopak veľkosť výsledného kódu zvýšiť.

V nasledujúcom texte si predstavíme vybrané optimalizačné techniky.

4.3.1 Vyhodnotenie konštantných výrazov

Táto optimalizácia má za cieľ vyhodnotiť výrazy zložené z konštánt už počas prekladu a celý výpočet nahradiť výsledkom. Táto procedúra sa musí správať rovnako, ako keby daný výpočet prebiehal na cieľovej architektúre. Táto požiadavka sa týka hlavne rôznych neštandardných stavov vo výpočte, ako napríklad pretečenie, delenie nulou a pod.

Predstavíme si preudokód algoritmu aj s príslušným komentárom[6, s. 330]:

```
1  procedure Const_Eval(inst) returns MIRInst
2      inst: inout MIRInst
```

Procedúra má jeden vstupno-výstupný parameter (`inst`), ktorým je aktuálne spracovávaná inštrukcia.

```
3  begin
4      result: Operand
5      case Exp_Kind(inst.kind) of
```

Do premennej `result`, ktorá je novým symbolom v programe sa bude ukladať výsledok. Budeme uvažovať len binárne a unárne matematické operácie, ktoré sa algoritmus pokúsi vyhodnotiť:

```
6          binexp: if Constant(inst.opd1) & Constant(inst.opd2) then
7                              result := Perform_Bin(inst.opr, inst.opd1,
                                                     inst.opd2)
8                      if isnt.kind = binasgn then
9                          return <kind:valasgn, left:inst.left, opd:result>
10                     elif inst.kind = binif then
11                         return <kind:valif, opd:result, lbl:inst.lbl>
12                     fi
13                 fi
14         unexp:  if Constant(inst.opd) then
15                     result := Perform_un(inst.opr, inst.opd)
16                     if isnt.kind = unasgn then
17                         return <kind:valasgn, left:inst.left, opd:result>
18                     elif inst.kind = unif then
19                         return <kind:valif, opd:result, lbl:inst.lbl>
20                     fi
21                 fi
22         default:return inst
23     esac
24 end
```

V prípade binárnej operácie sa najprv zistí, či sú oba operandy konštantami (6. riadok). Ak sú, do premennej `result` sa vypočíta výsledok operácie funkciou `Perform_Bin`, ktorá simuluje výpočet na cieľovej architektúre. Ďalej sa rozlišujú 2 typy inštrukcií. Ak inštrukcia priraďuje svoj výsledok do nejakého symbolu, táto inštrukcia sa zmení na obyčajnú inštrukciu kopírovania hodnoty a priradí sa do nej výsledok ako konštanta (8. a 9. riadok). V prípade inštrukcie podmieneného skoku sa nahradí inštrukciou, ktorá priamo vykonáva skok na dané návestie.

V prípade unárnych inštrukcií je postupnosť krokov rovnaká, akurát sa zmení funkcia simulujúca výpočet na cieľovej architektúre. Ostatné inštrukcie sa nespracovávajú.

Tento algoritmus je výhodné implementovať ako podprogram a ten volať vždy, keď je to vhodné.

4.3.2 Eliminácia nedosiahnuteľného kódu

Eliminácia nedosiahnuteľného kódu je typická optimalizácia, ktorá výrazne znižuje veľkosť výsledného kódu. Je založená na tom, že v programe môžu existovať základné bloky, pri ktorých neexistuje cesta od vstupného bodu k danému bloku inštrukcií.

Takéto bloky sa v programe nevyskytujú preto, že by programátor písal kód, ktorý sa nikdy nespustí, ale niektoré optimalizácie takýto kód produkujú. Rovnako aj používanie rôznych štandartných knižníc takéto bloky produkuje. V týchto knižniciach sú definované mnohé procedúry, ktoré sa v dano programe nepoužívajú.

Postup tohto algoritmu je nasledovný[6, s. 580]:

```
1  procedure Elim_Unreach_Code(en,nblocks, ninsts,Block,Pred,Succ)
2      en: in integer
3      nblocks: inout integer
4      ninsts: inout array [1 .. nblocks] of integer
5      Block: inout array [1..nblocks] of array [..] of Instruction
6      Pred, Succ : inout integer -> set of integer
8  begin
9      again: boolean
10     i: integer
```

Algoritmus pracuje s nasledovnými parametrami, `en` je vstupný bod optimalizovaného kódu (`Start` blok), počet základných blokov je `nblocks` a každý základný blok `i` obsahuje `ninsts[i]` inštrukcií. Zoznam základných blokov je v premennej `Block` a funkcie `Pred` a `Succ` vracajú predchodcu resp. nasledovníka základného bloku. Lokálna premenná `again` nadobudne `false`, keď algoritmus ukončí svoju činnosť. Algoritmus pokračuje nasledovne:

```
11     repeat
12         again := false
13         i := Succ(en)
14         while i<= nblocks do
15             if No_Path(en,i) then
16                 ninsts[i] := 0
17                 Block[i] := nil
18                 again := true
19                 delete_block(i,nblocks,ninsts,Block,Succ,Pred)
20             fi
21             i++
22         od
23     until !again
24 end
```

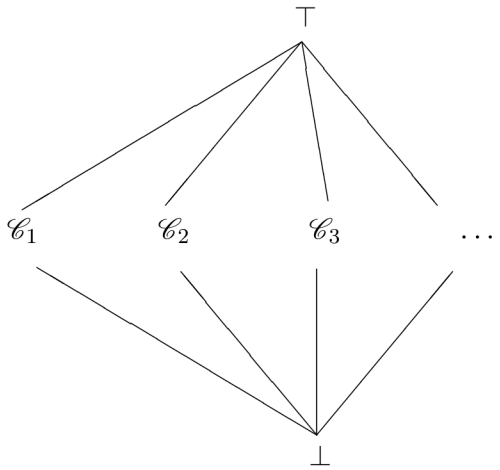
Predchádzajúci blok kódu má nasledovný význam:

1. Nastavíme premennú `again` na `false` a do premennej i nastavíme nasledovníka `Start` bloku.

2. Iterovaním cez pole `Block[]`, hľadáme bloky, pri ktorých neexistuje neprázdna cesta z bloku `Start` do aktuálneho bloku `i`. Ak sa takýto blok nájdeme, vymažeme ho a upravíme funkcie `Succ` a `Pred`, aby reflektovali zmenu. Nastavíme `again` na true.

3. Pokračujeme bodom 1, kým `again` nadobúda hodnotu `true`.

Pre úplnosť uvedieme, že funkcia `No_Path(i,j)` vracia `true` v prípade, že existuje cesta z bloku `i` do bloku `j`, inak vracia `false`.



Obr. 4.3: Hasseov diagram množiny premenných

4.3.3 Propagácia konštánt

Propagácia konštánt je transformácia, ktorá pre priradenie $x \leftarrow c$, pre premennú x a konštantu c, nahradí neskoršie použitie premennej x konštantou c až kým premenná x nenadobudne inú hodnotu [6, s. 362]. Inak povedané propagácia konštánt sa snaží nájsť premenné a výrazy, ktoré sú v niektorých častiach programu konštantné a namiesto premenných použiť práve ich hodnotu.

Ďalej predstavené algoritmy používajú na reprezentáciu premenných štruktúru, ktorej prvky sú usporiadané tak, ako je vidieť na obrázku 4.3. Jedná sa o Hasseov diagram tejto štruktúry. Nazvime túto štruktúru *Lattice*. Prvok tejto štruktúry môže byť jedným z troch typov. Najvyšším elementom je tzv. top, $\top$, reprezentujúci, že daná premenná zatiaľ nebola detekovaná ako konštanta. Najnižším je tzv. bottom, $\bot$, ktorý znamená, že sa nedá garantovať konštantnosť danej premennej (t.j. bude považovaná za premennú). Medzi nimi sú konštanty, označené ako $\mathscr{C}$. Z diagramu je jasné, že konštánt je nekonečne mnoho, a sú navzájom neporovnateľné. [8]

Nad touto štruktúrou si definujeme operátor zjednotenia dvoch prvkov, $\sqcap$, nasledovne [8]:

$$\begin{aligned} any \sqcap \top &= any \\ any \sqcap \bot &= \bot \\ \mathscr{C}_i \sqcap \mathscr{C}_j &= \mathscr{C}_i \,\text{if}\, i = j \text{ , tj. konštanty sú rovnaké} \\ \mathscr{C}_i \sqcap \mathscr{C}_j &= \bot \,\text{if}\, i \neq j \text{ , konštanty sú rôzne} \end{aligned}$$

Predstavené algoritmy z článku [8] inicializujú každú premennú optimisticky na hodnotu $\top$. Algoritmy pokračujú postupným znižovaním (v zmysle štruktúry *Lattice*) hodnoty každej premennej v každej inštrukcii, až kým sa nedosiahne pevného bodu.

V článku [8] boli predstavené 4 algoritmy na vyhľadávanie konštánt. Sú to algoritmy *Simple Constant (SC)*, *Sparse Simple Constant (SSC)*, *Conditional Constant (CC)* a *Sparse Conditional Constant*.

Algoritmus *Simple Constant* používa graf toku riadenia pre svoju prácu. Predpokladáme, že každý uzol grafu obsahuje práve jednu inštrukciu. S každým uzlom sú stotožnené

2 štruktúry *Lattice*, pre každú premennú v programe (prípadne procedúre), jedna s hodnotou na vstupe do bloku a jedna s hodnotou na jeho výstupe.

Algoritmus *Sparse Single Constant* nachádza rovnaké množstvo konštánt ako algoritmus *Simple Constant* avšak miesto grafu toku riadenia používa tzv. *Static Single Assigment* graf (SSA). V tomto grafe je hodnota každej premennej zmenená práve raz. . Tento algoritmus je rýchlejší ako algoritmus *Simple Constant*.

Algoritmus *Conditional Constant* používa graf toku riadenia. Navyše oproti predchádzajúcim algoritmom vyhodnocuje aj bloky vetvenia a ďalšie bloky spracováva podľa výsledku tohto bloku. V praxi to znamená, že ak algoritmus zistí, napríklad, že blok vetvenia nadobúda vždy konštantnú hodnotu (napríklad `true`), pokračuje len cestou pre túto vetvu. To znamená, že algoritmus ignoruje definície premenných, ktoré sú nedosiahnuteľné. Okrem toho označuje hrany grafu toku riadenia za spustiteľné a nespustiteľné.

Tento algoritmus spája propagovanie konštánt s elimináciou nedosiahnuteľného kódu.

Pre algoritmus *Sparse Conditional Constant* platí všetko ako pre algoritmus *Conditional Constant*, avšak opäť používa SSA graf, takže je rýchlejší.

Tretí predstavený algoritmus, *Conditional Constant* si predstavíme podrobnejšie priamo na pseudokóde tohto algoritmu [7, s. 114]:

```
//Graf = (N,E), 𝒱  je množina premenných používaných v grafe
begin
  Pile = (Start → n)| (Start → n) ∈ E;
 Označ všetky hrany (Start → n)| (Start → n) ∈ E za spustitelne
  a ostatné hrany za nespustitelne;
```

V rámci inicializácie do nej dáme hrany vedúce zo štartovacieho uzlu. Všetky tieto hrany označíme za spustiteľné, ostatné hrany za nespustiteľné. Ďalej inicializujeme jednotlivé bloky:

```
  //y.oldval a y.newval uchovávajú hodnotu (štruktúra Lattice) výrazu v uzle y
  for all y ∈ N do
    y.oldval = ⊤;
    y.newval = ⊤;
    for all v ∈ 𝒱 do
      y.v.incell = ⊤;
      y.v.outcel = ⊤;
    end for
  end for
```

zmysle štruktúry *Lattice*) v uzle `y`. V rámci inicializácie ich nastavíme na vrchol tejto štruktúry ($\top$). Rovnako nastavíme aj hodnotu každej premennej v každom uzle (`y.u.incell` pre vstupnú hodnotu a `y.u.outcell` pre výstupnú). Treba podotknúť, že v algoritme každý uzol grafu toku riadenia obsahuje hodnotu všetkých premenných používaných v programe. Tieto môžu nadobúdať v rôznych uzloch rôzne hodnoty. Totiž, v niektorých uzloch sa daná premenná môže správať ako konštanta, tj. nadobúdať stále rovnaké hodnoty, ale ďalej sa už môže jej hodnota meniť, takže v ostatných uzloch môže byť označená ako premenná.

Po inicializácii sa pustíme priamo do hlavnej slučky algoritmu:

```
  while Pile ≠ ∅
  begin
    (x,y) = remove(Pile);
```

Algoritmus pokračuje, až kým množina `Pile` nie je prázdna. Ak množina nie je prázdna, vyberieme z nej nejaký prvok a odstránime ho.

```
// y.i.incell a y.i.outcell sú hodnoty premennej i v bloku y
for all i ∈ 𝒱 do
  y.i.incell = ⊓_{p ∈ Pred(y)} p.i.outcell;
end for
```

Pre každú premennú prepočítame jej hodnotu na vstupe do bloku, ako zjednotenie definované nad štruktúrou zo všetkých predchádzajúcich uzlov.

```
y.newval = evaluate(y);
switch(y)
```

Novú hodnotu výrazu v bloku `y` vypočítame pomocou funkcie `evaluate`, ktorá funguje nasledovne:

$$\begin{aligned} val(a\,op\,b) &= \bot, \text{ ak a alebo b je } \bot \\ &= \mathscr{C}_i\, op\, \mathscr{C} \text{ ak val(a) a val(b) sú konštanty, resp. } \mathscr{C}_i \text{ a } \mathscr{C}_j \\ &= \top, \text{ inak} \end{aligned}$$

Podľa typu uzlu (priradenie, uzol vetvenia) postupujeme nasledovne:

```
case y je priraďenie:
  // y.instruction.output.outcell je hodnota outcell pre premennú,
  // ktorá je cieľom priradenia v uzle y
  if(y.newval < y.instruction.output.outcell) then
  begin
    Označ všetky uzly (y → n)|(y → n ) ∈ E ako spustitelne
            a pridaj ich do Pile;
    for all i ∈ 𝒱 do
      y.i.outcell = y.i.incell;
    end for
    y.instruction.output.outcell = y.newval;
    y.oldval = y.newval;
  end
end case
```

Ak sa jedná o priradenie, zistíme či sa hodnota uzlu zmenila od predchádzajúcej návštevy. Ak áno, označíme všetky východzie uzly za spustiteľné a pridáme ich do `Pile`. Tieto uzly sú ovplyvnené uzlom `y`, takže ich treba prepočítať. Prekopírujeme vstupné hodnoty premenných do výstupných hodnôt. Do premennej, ktorá bola cieľom priradenia dáme vypočítanú hodnotu pre uzol `y`.

```
case y je uzol vetvenia
  //skopírujeme hodnoty
  for all i ∈ 𝒱 do
    y.i.outcell = y.i.incell;
  end for
```

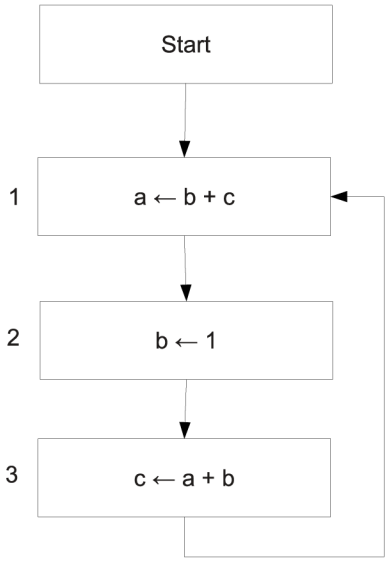
```
        if(y.newval < y.oldval) then
        begin
          switch(y.newval)
            case ⊥:
              //je to premenná, obe vetvy sú vykonateľné
              Označ všetky hrany (y → n)|(y → n ) ∈ E ako spustitelne
                a pridaj ich do Pile;
              y.oldval = y.newval;
            end case
            case true:
              Označ vetvu true z uzlu y ako spustitelnu
                a pridaj ju do Pile;
              y.oldval = y.newval;
            end case
            case false:
              Označ vetvu false z uzlu y ako spustitelnu
                a pridaj ju do Pile;
              y.oldval = y.newval;
            end case
          end switch
        end  //if
      end case
    end switch
  end //while
end
```

V prípade, že sa jedná o uzol vetvenia, prekopírujeme vstupné hodnoty do výstupných. Ďalej, ak sa zmenila (znížila) hodnota uzlu vyhodnotíme túto zmenu nasledovne. Ak je hodnota uzlu $\perp$, obe vetvy v grafe sú spustiteľné. V prípade, že je hodnota konštantná, tak je v danom momente spustiteľná len jedna z vetiev a tú vložíme do `Pile`. Hodnota $\top$ by znamenala používanie neinicializovanej premennej v programe, čo môže byť vyhodnotené ako chyba.

Po skončení algoritmu sú označené vetvy v grafe, ktoré sú spustiteľné. Rovnako každý uzol obsahuje hodnoty každej premennej v grafe v mieste tohto uzlu. Takže v prípade, že parametre daného bloku sú konštanty môže sa daná premenná nahradiť konštantou. V prípade, že celý uzol má konštantný výsledok, je možné miesto výpočtu priamo priradiť výsledok do premennej.

Pri implementácii tejto procedúry sme narazili na problém. Podľa nášho názoru tento preudokód algoritmu obsahuje chybu. Totiž do `Pile` sa pridávajú len priamy následníci uzlov, ktoré od predchádzajúcej návštevy zmenili svoju hodnotu. Avšak predstavme si graf toku riadenia z obrázku 4.4. Všetky premenné sú inicializované konštantou. Pri inicializácii sa označí hrana $START \rightarrow 1$ za spustiteľnú a pridá sa do `Pile`. Postupne sa takto vykonajú všetky bloky cyklu. Po spracovaní hrany $2 \rightarrow 3$, tj. po vyhodnotení hodnoty bloku 3, majú všetky bloky `y.oldval` nastavenú na konštantu. V `Pile` je hrana $3 \rightarrow 1$.

Táto hrana sa spracuje, zistí sa, že blok 1 zmenil svoju hodnotu na $\perp$. Do `Pile` sa pridajú následníci tohto bloku, čiže hrana $1 \rightarrow 2$. Po jej spracovaní sa zistí, že blok 2 nezmenil svoju hodnotu (do premennej `b` sa priraďuje vždy konštanta). V tomto okamihu algoritmus končí, lebo nedá následníka uzlu 2 do množiny `Pile`. V tomto okamžiku je blok č. 3 nesprávne



Obr. 4.4: Zlyhanie algoritmu CCP

nastavený na konštantu. Je evidentné, že hodnota premennej `a` a tým pádom aj výsledku sa bude meniť.

Blok musí označiť na prepočítanie, tj. pridať do `Pile`, svojich následníkov vždy, keď sa mu zmenila hodnota čo i len jednej premennej. Preto bol algoritmus opravený nasledovne. Blok po aktualizácii hodnôt svojich premenných, zistí či sa mu niektorá premenná zmenila:

```
// y.i.incell a y.i.outcell sú hodnoty premennej i v bloku y
// change je premenná typu bool, ktorá detekuje
//        zmenu hodnoty niektorej premennej
change = false;
for all i ∈ 𝒱 do
  y.i.incell = ⊓_{p ∈ Pred(y)} p.i.outcell;
  if( y.i.incell ≠ y.i.outcell) then
     change = true;
  end
end for
```

Následne každá podmienka, kde sa testuje, či hodnota uzlu sa znížila, v zmysle štruktúry *Lattice*, musí testovať aj či sa zmenila hodnota niektorej premennej:

```
if(y.newval < y.instruction.output.outcell OR change) then
...
if(y.newval < y.oldval OR change) then
...
```

4.3.4 Propagácia kópií

Propagácia kópií je transformácia kódu, ktorá ak narazí na priradenie $x \leftarrow y$, pre premenné x a y, nahradzuje neskoršie použitie premennej x premennou y, až pokiaľ nejaká iná inštrukcia nezmení hodnotu niektorej z premenných x alebo y [6, s. 356].

Propagácia kópií sa môže rozdeliť na 2 fázy, lokálnu a globálnu. Lokálna propaguje kópie v rámci základného bloku, globálna v rámci v rámci celého grafu toku riadenia.

Predstavíme si pseudokód algoritmu lokálnej propagácie kópií [6, s. 356]:

```
procedure Local_Copy_Prop(m,n,Block)
  m, n: in integer
  Block: inout array [1..n] of array [..] of MIRInstr
```

Algoritmus pracuje nad grafom toku riadenia, parameter `Block` je zoznam základných blokov v grafe. `m` značí blok, ktorý sa má spracovať a `n` je počet inštrukcií v bloku `m`.

```
begin
  ACP := ∅: set of (Var × Var)
  i: integer
```

Premenná `ACP` je množina dvojíc (Premenná, Premenná), v ktorej prvá položka je premenná, do ktorej sa kopírovalo, druhá položka je premenná, ktorá sa kopírovala. Napríklad inštrukcia $b \leftarrow a$ vytvorí dvojicu (b,a).

```
for i := 1 to n do
    case Exp_King(Block[m][i].kind) of   # podľa typu inštrukcie
              # opd.val je meno premennej
       binexp:   # binárny výraz
         Block[m][i].opd1.val := Copy_Value(Block[m][i].opd1, ACP)
         Block[m][i].opd2.val := Copy_Value(Block[m][i].opd2, ACP)
       unexp:    # unárny výraz
         Block[m][i].opd.val := Copy_Value(Block[m][i].opd, ACP)
       listexp:  # volanie funkcie, zoznam argumentov
         for j := 1 to |Block[m][i].args| do
           # ↓j@1 výber j-tého argumentu
           Block[m][i].args↓j@1.val :=
               Copy_Value(Block[m][i].args↓j@1, ACP)
         od
        default:
    esac
```

Iterovaním cez všetky inštrukcie bloku postupujeme nasledovne. Podľa typu inštrukcie (z hľadiska je počtu operandov), každý operand upravíme podľa návratovej funkcie `Copy_Value`, ktorú si predstavíme neskôr. Zatiaľ nám stačí, že táto funkcia vráti novú premennú podľa informácií v množine `ACP`.

```
# odstránime dvojice z ACP, ktoré boli zneplatnené aktuálnou inštrukciou
    if Has_Left(Block[m][i].kind) then
      Remove_ACP(ACP, Block[m][i].left
    fi
    # vložíme nové dvojice do ACP
    if Block[m][i].kind = valasgn &       # ak sme inštrukcia typu a ← b
           Block[m][i].opd.kind = var &   # ak b je premenná
           Block[m][i].left ≠ Block[m][i].opd.val then  # ak a ≠ b
```

```
      ACP ∪= { <Block[m][i].left, Block[m][i].opd.val> }
    fi
end
```

Cyklus pokračuje ďalej. Ak inštrukcia mala tzv. ľavý operand (premennú do ktorej sa priraďuje), odstránime z množiny `ACP` všetky dvojice, zneplatnené aktuálnou inštrukciou. Ďalej, ak bola inštrukcia kopírovaním hodnoty premennej do inej premennej, aktualizujeme množinu `ACP`, tj. pridáme do nej aktuálnu dvojicu premenných.

Predstavíme si ešte pseudokódy pomocných funkcií používaných algoritmom.

```
procedure Remove_ACP(ACP, v)
  ACP: inout set of (Var × Var)
  v: in Var
begin
  T:= ACP: set of (Var × Var)
  acp: Var × Var
  for each acp ∈ T do
    if acp@1 = v OR acp@2 = v then
      ACP -={acp}
    fi
  od
end
```

Funkcia `Remove_ACP` odstráni z množiny `ACP` všetky dvojice, v ktorých jeden z prvkov je premennou `v`. A druhá funkcia:

```
procedure Copy_Value(opnd, ACP) returns Var
  opnd: in Operand
  ACP: in set of (Var × Var)
begin
  acp: Var × Var
  for each acp ∈ ACP do
    if opnd.kind = var AND opnd.val = acp@1 then
      return acp@2
    fi
  od
  return opnd.val
end
```

Funkcia `Copy_Value` vráti novú premennú (operand), podľa množiny `ACP`. Ak sa operand nachádza ako prvý prvok v niektorej z dvojíc v `ACP`, vráti sa druhý prvok z tejto dvojice. Ako už bolo spomínané táto dvojica vznikla inštrukciou typu $b \leftarrow a$. To znamená, že premenná b má v danom okamihu rovnakú hodnotu ako premenná a. Premenná a bola inicializovaná skôr, takže sa použije ona (b je kópiou a a jedná sa o algoritmus propagácie kópií).

Rozšírenie lokálnej propagácie konštánt na globálnu úroveň vykonáme nasledovne. Definujeme si množinu $COPY(i)$. Táto množina obsahuje n-tice typu$< u, v, i, pos >$také, že $u \leftarrow v$ je inštrukcia priradenia, pos je pozícia tejto inštrukcie v bloku i. Hodnota u ani v nie je prepísaná neskoršie v bloku i. Ďalej si definujeme množinu $KILL(i)$, obsahujúcu n-tice typu $< u, v, blk, pos >$, kde u, v a pos sú ako v predchádzajúcej množine, ktoré sa

nachádzajú v bloku blk, $blk \neq i$. Inak povedané sú to definície, ktoré sú v tomto bloku zabité. [6, s. 358]

Definujeme si $CPin(i)$ a $CPout(i)$ [6, s. 360], reprezentujúce priradenia, ktoré sú k dispozícii pri vstupe, prípadne pri výstupe z bloku i. Výpočet robíme iteratívne, kým sa nedosiahneme pevný bod:

$$CPin(i) = \bigcap_{j \in Pred(i)} CPout(j)$$

$$CPout(i) = COPY(i) \cap (CPin(i) - KILL(i))$$

a inicializujeme $CPin(entry) = \emptyset$ a $CPin(i) = U \; \forall i \neq entry$. U je univerzum, tj. zjednotenie všetkých n-tíc z $COPY(i)$:

$$U = \bigcup_i COPY(i)$$

Algoritmus globálnej propagácie kópií je v tomto prípade nasledovný [6, s. 360]:

1. Spustíme algoritmus `Local_Copy_Prop` tak, aby vrátil výsledný ACP pre daný blok. Tento pretransformujeme do množiny $COPY(blok)$.
2. Vypočítame univerzum U a množiny $KILL(i)$, pre každý blok.
3. Iteratívne vypočítame $CPin(i)$a $CPout(i)$ podľa vzťahov uvedených predtým.
4. Pre každý základný blok, nastavíme množinu

$$ACP = \{a \in Var \times Var, \text{ kde } \exists w :< a@1, a@2, B, w >\in CPin(B)\}$$

5. Spustíme opäť algoritmus `Local_Copy_Prop`, s vypočítaným `ACP` pre daný blok.

4.3.5 Eliminácia mŕtveho kódu

Táto optimalizácia odstraňuje inštrukcie, ktoré sú mŕtve. Najprv si definujeme mŕtvu premennú. Mŕtva premenná je premenná, ktorá nie je použitá v žiadnej ceste od miesta jej definície až ku koncu procedúry, ako parameter inej inštrukcie.

Inštrukcia je mŕtva, keď jej výsledok nie je použitý v žiadnej spustiteľnej vetve vedúcej od danej inštrukcie ku koncu procedúry. Inak povedané, mŕtva inštrukcia je inštrukcia, ktorej výsledok je mŕtva premenná a zároveň nemá žiadne postranné efekty[2], ktoré by nasledujúce inštrukcie v spustiteľných vetvách programu mohli využiť.

Opakom mŕtvej inštrukcie je živá alebo užitočná inštrukcia.

V programe sa mŕtve inštrukcie vyskytujú opäť kvôli tomu, že mnohé iné optimalizácie takéto inštrukcie generujú. Predtým, ako uvedieme algoritmus odstránenia mŕtvych inštrukcií, definujeme 2 štruktúry[6, s. 251]:

$DU-chain$ (Definition-Use) spája definíciu danej premennej so všetkými jej použitiami, ku ktorým vykonávanie kódu môže viesť.

$UD-chain$ (Use-Definition) spája použitie premennej ku všetkým definíciám, z ktorých je daná premenná dosiahnuteľná. Čiže sa jedná o štruktúry, ktoré sú k sebe zrkadlové.

Algoritmus začína tak, že všetky inštrukcie v procedúre prehlási za mŕtve, okrem tých, ktoré buď vracajú nejakú hodnotu z procedúry alebo ovplyvňujú vstupne-výstupné zariadenia, prístupné aj z vonka procedúry. V prípade AHLL by to boli aj volania služieb platformy.

[2] napríklad nastavenie príznakov

V algoritme je každá inštrukcia reprezentovaná ako dvojica <`blok,index`>, kde `blok` je základný blok, v ktorom je daná inštrukcia a `index` je index (poradie) tejto inštrukcie v danom základnom bloku. `Worklist` je množina dvojíc <`blok,index`>. Funkcia `Vars_Used` vracia množinu premenných používaných danou inštrukciou.

Teraz si predstavíme samotný pseudokód algoritmu [6, s. 592]:

```
UdDu =  integer x integer
UdDuChain = (Symbol x UdDu) -> set of UdDu
procedure Dead_Code_Elim(nblocks, ninsts,Block, Mark, UD,DU, Succ,Pred)
    nblocks: inout integer
    ninsts: inout array [1..nblocks] of integer
    Block: inout array [1..nblocks] of array [..] of MIRInstr
    Mark: in array [1..nblocks] of array [..] of boolean
    UD, DU: in UdDuChain
    Succ, Pred : inout integer -> set of integer
```

V zozname `Mark` sú označené užitočné inštrukcie (tj. tie, ktoré nie sú mŕtve). `UD` a `DU` sú spomínané štruktúry $UD-chain$ a $DU-chain$.

```
begin
    i, j : integer
    x, y: integer x integer
    v: Var
    Worklist: set of (integer x integer)
    # mnozina pozicii "uzitocnych" instrukcii
    Worklist :=  (i,j) in integer x integer where Mark[i][j]
```

V množine `Worklist` si uchovávame užitočné inštrukcie (ich pozície). Ako už bolo spomenuté tento zoznam inicializujeme tak, že do neho vložíme inštrukcie, ktoré hneď na začiatku boli označené ako užitočné (zoznam `Mark`).

```
while Worklist <> Empty do
        x := Worklist.head
        Worklist -= x
```

Cyklíme, kým nespracujeme všetky inštrukcie z množiny `Worklist`. V každej iterácii vyberieme z množiny jeden prvok (jednu inštrukciu) a uchováme si ho v premennej `x`.

```
        # oznacime instrukcie, ktore definuju hodnotu pouzivanu v 'x'
        for each v in Vars_Used(Block,x) do
            for each y in UD(v,x) do
                if !Mark[y@1][y@2] then
                    Mark[y@1][y@2] := true
                    Worklist ∪= y
                fi
            od
        od
```

V tomto kroku si označíme za užitočné všetky inštrukcie, ktoré definujú hodnotu operandu používaného v aktuálnej inštrukcii `x` a neboli označené za užitočné. Tieto inštrukcie vložíme do množiny `Worklist`. Na tento krok využijeme $UD-chain$.

```
        # oznacime podmienkove instrukcie, pouzivajuce 'x'
        if Has_Left(Block[x@1][x@2].kind) then
            for each y in DU(Block[x@1][x@2].left, x) do
                if !Mark[y@1][y@2] & Block[y@1][y@2].kind in {binif, unif}
                    Mark[y@1][y@2] := true
                    Worklist ∪= y
                fi
            od
        fi
```

V ďalšom kroku, ak inštrukcia `x` definuje nejakú premennú, tak označíme všetky inštrukcie vetvenia, ktoré túto premennú používajú za užitočné. Ak predtým boli označené za mŕtve, dáme ich do množiny `Worklist`.

```
    od
    Delete_Unmarked_Insts(nblocks,ninsts,Block, Succ, Pred,Mark)
end
```

Po ukončení cyklu vymažeme všetky inštrukcie, ktoré neboli označené za užitočné, tj. sú mŕtve.

4.4 Poradie jednotlivých optimalizácií

Pri optimalizačných technikách je potrebné určiť poradie jednotlivých techník, ako aj ich opakovanie. Niektoré techniky ťažia práve z toho, že sú aplikované na zdrojový program v danom poradí.

Ako príklad môžme uviesť algebraické zjednodušovanie výrazov, ktoré výrazne ťaží z propagovania konštánt[6, s. 325]. Niektoré, ako napríklad odstraňovanie tzv. mŕtveho kódu je výhodné robiť opakovane, v rôznych miestach kódu.

Výsledkom je, že určenie poradia jednotlivých optimalizácií je netriviálny problém. Jedno z možných poradí veľkého množstva optimalizácií, podľa [6] je možné nájsť v prílohe B. V našom prípade sme zvolili nasledovné poradie predstavených optimalizačných techník:

1. Vyhodnotenie konštantných výrazov.
2. Propagácia konštánt a odstránenie nedosiahnuteľného kódu.
3. Propagácia kópií.
4. Eliminácia mŕtveho kódu.

Kapitola 5

Zmeny v AHLL a nová štruktúra prekladača

5.1 Zmeny v AHLL

V rámci tejto diplomovej práce sme pristúpili k úprave syntaxe jazyka AHLL. Úprava vychádza prevažne z konzultácií v rámci výskumného tímu zaoberajúceho sa problematikou multiagentných systémov a WSN na FIT VUT Brno. EBNF novej verzie jazyka AHLL je uvedené v prílohe A.

Z dôvodu úpravy syntaxe sme sa rozhodli reimplementovať prekladač jazyka AHLL z jazyka C++ do programovacieho jazyka Java. Mierne sa tým zjednoduší prípadná integrácia tohto prekladača do simulačného nástroja T-Mass.

Keďže gramatika jazyka AHLL je uvedená v externom súbore a lexikálny, syntaktický analyzátor ako aj konštruktor AST je z tejto gramatiky generovaný nástrojom ANTLT [5] bola úprava gramatiky pomerne jednoduchá a nenáročná. Drvivá väčšina pravidiel gramatiky bola zachovaná v pôvodnej forme, rovnako ako aj forma AST generovaného pôvodnou gramatikou, resp. prekladačom.

V gramatike jazyka AHLL boli vykonané nasledovné úpravy:

- Kľúčové slovo `platform` bolo nahradené deklaráciou platformy ako pomenovanej funkcie s parametrami.
- Volania služieb platformy a príkaz `receive` je možné používať vo výrazoch.
- Bol odstránený operátor `=>` na ukladanie výsledkov služieb platformy a príkazu `receive`.
- Bol doplnený cyklus `foreach` pre priechod zoznamom.
- Pribudli unárne operátory inkrementu a dekrementu v prefixovej aj postfixovej forme.
- Pribudla možnosť vkladania hlavičkových súborov.
- Pribudla možnosť definovania konštánt.

V nasledujúcej časti kapitoly si predstavíme jednotlivé zmeny v AHLL ako aj ich implementáciu.

Z jazyka bolo odstránené kľúčové slovo `platform`, slúžiace na volanie služieb platformy [5]. Bolo nahradené deklaráciou služby platformy v nasledujúcej forme:

```
service meno(parametre){
call n-tica;
return null?;
}
```

`service` je kľúčové slovo, označujúce deklaráciu novej služby platformy, *`meno`* je zvolené meno tejto služby a *`parametre`* sú vyžadované parametre služby vo forme premenných oddelených čiarkou. `call` je kľúčové slovo označujúce n-ticu akú má prekladač zavolať na spustenie tejto služby. V tejto n-tici sa môžu vyskytnúť premenné, ktoré sú uvedené v položke *`parametre`*. Pri generovaní cieľového kódu v ALLL sa tieto parametre nahradia hodnotami daných premenných. `return` označuje že daná služba má nejakú návratová hodnotu, čo v praxi znamená, že je ju možné použiť vo výraze. `return null` označuje opak, čiže služba žiadnu návratovú hodnotu nemá.

Príklad 5.1. V tomto príklade si ukážeme definíciu konkrétnej služby platformy. Jedná sa o službu, ktorá vráti prvý prvok zoznamu, čiže jeho hlavičku. Službu si pomenujeme ako `head` s jedným parametrom `list` a služba má návratovú hodnotu:

```
service head(list){
    call ["lst",["h",list]];
    return;
}
```

Pri volaní tejto služby, napríklad `head([1,2,3])` sa vygeneruje kód v jazyku ALLL v tvare `#(lst,(h,(1,2,3)))`. Služba vráti prvý prvok zoznamu, ktorým je `1`.

Pôvodne v AHLL nebolo možné používať príkaz `receive` a volania služieb platformy vo výrazoch. V novej verzii jazyka to už možné je. Je to spôsobené tým, že služby platformy je nutné už deklarovať a pri deklarácii sa uvádza aj to či služba platformy má alebo nemá návratovú hodnotu. Ak túto skutočnosť prekladač v čase prekladu pozná, nič nebráni tomu aby kontroloval používanie služieb platformy a umožnil ich použitie aj vo výrazoch. Z pochopiteľných dôvodov, vo výrazoch je možné použiť len služby platformy, ktoré majú návratovú hodnotu.

Ak je už možné používať služby platformy a príkaz `receive` vo výrazoch, operátor `=>`[1] stráca opodstatnenie. Preto bol operátor z jazyka odstránený. Výsledok služby platformy je možné do nejakej premennej uložiť štandardne priradzovacím príkazom, napríklad:

```
a = head([1,2,3]);
```

Doteraz jazyk AHLL podporoval iba cyklus s podmienkou na začiatku, tzv. `while` cyklus. Keďže jazyk ALLL je do značnej miery založený na používaní n-tíc, boli možnosti jazyka AHLL rozšírené o podporu cyklu na priechod zoznamom. Jedná sa o tzv. `foreach` cyklus. Jeho tvar je nasledovný:

`foreach` *`a`* `in` *`zoznam|premenná`* `do{`
<block>
}
`a` je premenná, v ktorej bude v každej iterácii uložená aktuálna hodnota zo zoznamu.

[1] slúžil na uloženie návratovej hodnoty do premennej

zoznam je zoznam, *premenná* je premenná, ktorá by mala obsahovať v čase spustenia iterácie zoznam. <*block*> je blok kódu v AHLL. Podotýkame, že túto premennú, ktorá sa používa netreba deklarovať pomocou kľúčového slova `var` a musí byť unikátna.

Príklad 5.2. Príklad použitia cyklu `foreach`:

```
var i,list;
list = [1,2,3];
i = 1;
foreach a in list do {
    i = i+a ;
}
```

Jazyk AHLL bol rozšírený o podporu unárnych operátorov inkrementu a dekrementu v prefixovej a postfixovej forme. Operátory sa zapisujú pred premennú, v prípade prefixového tvaru. Alebo za premennú v prípade postfixovej varianty. Inkrement má tvar `++` a dekrement `--`. V prípade prefixovej formy sa hodnota premennej inkrementuje pred použitím premennej, v prípade postfixovej po jej použití, presnejšie, po vyhodnotení celého výrazu.

Pre pohodlnosť v programovaní a možnosť znovupoužitia už vytvorených kódov bola zavedená možnosť vkladania tzv. hlavičkových súborov. Hlavičkové súbory sa vkladajú pomocou konštrukcie `#include<subor>`, kde `subor` je meno vkladaného súboru. Viac informácií o hlavičkových súboroch a preprocessingu jazyka AHLL je uvedených v kapitole 6.1.

Zvýšenie programátorského komfortu priniesla aj možnosť definovania a pomenovania konštánt. Konštanty sa definujú kľúčovým slovom `const`. Tvar definície je nasledovný:
`const` *name* `=` *value* `;`
kde *name* je meno konštanty a *value* je jej hodnota.

Príklad 5.3. Definovanie konštanty s názvom `JEDNA` a hodnotou 1.

```
const  JEDNA = 1;
```

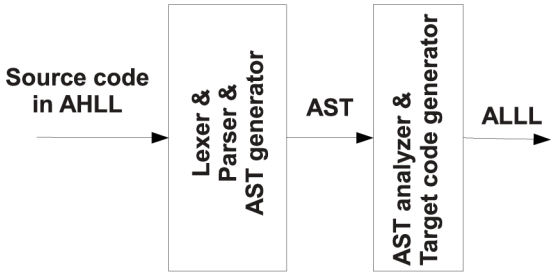
Z dôvodu, že ALLL obsahuje len 3 registre sme zaviedli obmedzenie, že služby a plány môžu mať najviac 3 parametre. v oboch prípadoch by povolenie viacerých služieb výrazne zvyšovalo veľkosť vygenerovaného kódu v ALLL. Toto obmedzenie je možné v budúcnosti odstrániť.

5.2 Nová štruktúra prekladača

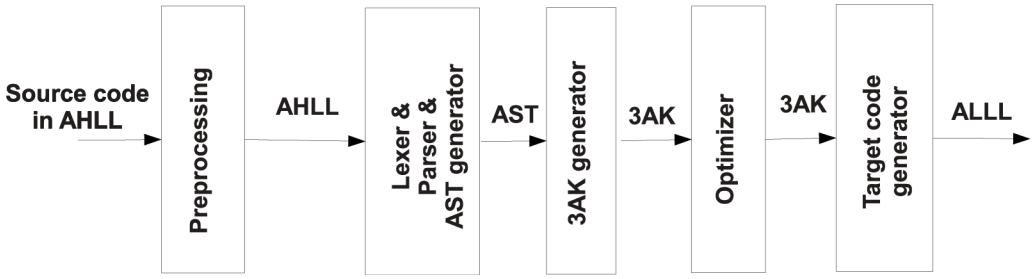
Pri vypracovaní diplomovej práce bola, okrem jazyka AHLL, zmenená aj logická štruktúra samotného prekladača.

Pôvodný prekladač bol koncipovaný do 2 základných blokov ako je vidieť na obrázku 5.1. Prvým bola analýza a transformácia zdrojového kódu z jazyka AHLL do AST. Na to nadväzujúci blok robil sémantickú analýzu nad týmto stromom a priamo generoval cieľový kód v jazyku ALLL.

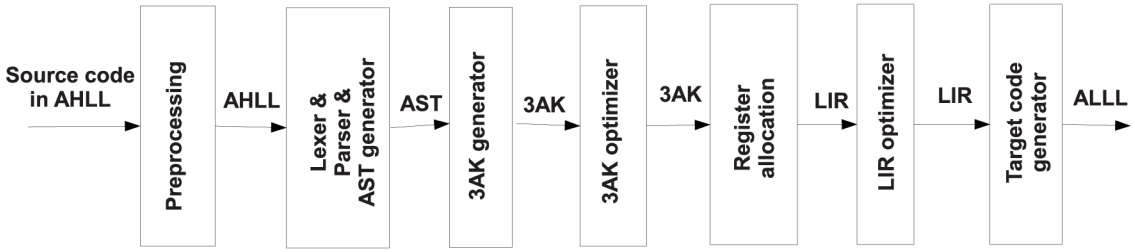
Keďže pôvodný prekladač nerobil takmer žiadne optimalizácie zamerané na zmenšenie veľkosti výsledného kódu bola táto štruktúra postačujúca. Nebola potrebná žiadna komplikovanejšia štruktúra ani iná forma medzikódu. Postačoval abstraktný syntaktický strom.



Obr. 5.1: Pôvodná štruktúra prekladača



Obr. 5.2: Nová štruktúra prekladača



Obr. 5.3: Plánovaná štruktúra prekladača

V novej verzii prekladača sa v budúcnosti predpokladá výrazná orientácia na optimalizovanie výsledného kódu. Rovnako aj fakt, že jazyk AHLL je stále vo vývoji a prepokladáme ďalší rozvoj tohto jazyka a z toho vyplývajúce zmeny v jeho syntaxy. Tomuto je potrebné prispôsobiť štruktúru prekladača tak, aby prípadná mierna zmena syntaxe nevyžadovala reimplementáciu celého prekladača vrátane všetkých dovtedy implementovaných optimalizačných techník. Ich funkčnosť by mala byť zachovaná aj po takomto zásahu.

Aby sme vyhoveli týmto požiadavkám, je potrebné navrhnúť relatívne stabilné jadro prekladača. Jadrom prekladača je v našom prípade vhodný medzikód, ktorý je štrukturálne podobný cieľovému kódu. To čiastočne zaručuje, že aj po zmene syntaxe AHLL bude potrebné reimplementovať len časti, ktoré sa starajú o vygenerovanie tohto medzikódu. Tento medzikód by mal byť takisto vhodný na rozsiahlu množinu optimalizačných metód a algoritmov. V neposlednom rade tento medzikód musí byť pomerne jednoducho prevoditeľný do cieľového kódu, tj. jazyka ALLL.

Tieto požiadavky veľmi dobre spĺňa troj-adresný kód (3AK), ktorý sme teoreticky predstavili v kapitole 4.1.2. Je možné navrhnúť formu 3AK, ktorá je štrukturálne podobná jazyku ALLL. To zaručí aj pomerne jednoduchý preklad z 3AK do ALLL. Rovnako je tento kód široko používaný v iných prekladačoch, čo značí obrovské možnosti jeho optimalizácie.

V tejto kapitole sa nebudeme priamo venovať forme 3AK pre náš prekladač. Konkrétnu formu 3AK, ktorý používame si predstavíme ďalej, v kapitole 6.3.

Novo vytvorená štruktúra prekladača je zobrazená na obrázku 5.2. Táto štruktúra umožňuje zmeny syntaxe jazyka AHLL. Centrálnou časťou tejto štruktúry je program v 3AK. Tento je ďalej optimalizovaný. Pri takejto štruktúre prekladača sa už naskytuje možnosť implementácie optimalizácií relatívne nezávisle na sebe, viacerými programátormi. Stačí aby každá optimalizačná technika, resp. modul starajúci sa o istú skupinu týchto optimalizácií ako výsledok svojej činnosti predal validnú formu 3AK.

Ako už bolo spomínané v úvodných častiach tejto práce, v súčasnosti jazyk ALLL existuje v dvoch variantách. Jedna implementovaná v simulátore T-Mass, druhá v reálnych uzloch senzorovej siete. To, že sa výsledný kód vytvára až z 3AK kódu umožňuje v budúcnosti prekladať do oboch foriem jazyka ALLL. Stačí len vytvoriť verziu generátoru cieľového kódu pre oba varianty jazyka.

Na záver tejto kapitoly ešte uvedieme plánovanú štruktúru prekladača v budúcnosti. Tá je zobrazená na obrázku 5.3. V tejto štruktúre je zahrnutá optimalizácia, ktorá sa nazýva alokácia registrov. Táto produkuje vo väčšine prípadov LIR kód. Ako už bolo spomínané v kapitole 4.1.3, na rozdiel od 3AK už miesto symbolov pracuje s reálnymi pamäťovými možnosťami cieľovej architektúry. V prípade ALLL sú nimi reálne registre, prípadne akcie na načítavanie hodnôt z bázy znalostí do týchto registrov a ich ukladanie späť.

Na tento blok prípadne môžu nadväzovať ďalšie optimalizácie, pracujúce už na LIR kóde. Je jasné, že v tomto prípade sa cieľový kód v jazyku ALLL bude generovať z tohto LIR kódu. Možnosť nezávislej implementácie optimalizácií ostáva zachovaná ako aj možnosť generovať kód v ALLL v oboch jeho existujúcich verziách.

Kapitola 6

Preprocessing a transformácie jazyka AHLL

V úvode tejto kapitoly bude predstavený preprocessing jazyka AHLL. Hlavná časť kapitoly bude venovaná rôznym interným reprezentáciám, ktorými prechádza AHLL pri preklade. Rovnako budú predstavené aj transformácie medzi nimi.

6.1 Preprocesing jazyka AHLL

K prekladaču AHLL bol doplnený pomerne jednoduchý preprocessor. Tento slúži výhradne na možnosť používať hlavičkové súbory v zdrojových kódoch. V týchto hlavičkových súboroch sa môže vo všeobecnosti nachádzať hocijaký legálny kód jazyka AHLL.

Pravdaže, hlavným účelom hlavičkových súborov je možnosť definovať služby platformy, prípadne pomocné plány (podprogramy). Implementovaný preprocessor má jedinú funkciu, ktorou je nahradenie výskytov konštrukcie `#include<include_file>` obsahom súboru `include_file`. Táto konštrukcia bola inšpirovaná jazykom C .

Preprocessor pracuje rekurzívne, to znamená že v hlavičkových súboroch sa opäť môžu vyskytnúť ďalšie konštrukcie `#include`, ako je to známe z jazyka C.

Preprocessor pracuje na jednoduchom princípe hľadania a nahradenia riadkov s výskytom reťazca, ktorý sa zhoduje s regulárnym výrazom pre konštrukciu `#include<fileName>`, obsahom súboru `fileName`.

Regulárny výraz s takouto funkcionalitou má tvar `(^#include)(<([^>]+)>)\\s*`. Význam regulárneho výrazu je nasledovný:

- `(^#include)` – Výskyt `#include` na začiatku riadku .
- `(<([^>]+)>)` – Ten je nasledovaný reťazcom začínajúcim znakom <. Za ním nasleduje neprázdny reťazec znakov odlišných od znaku >, ukončený znakom >. Toto reprezentuje meno vstupného súboru.
- \\s* – Zvyšok riadku, tvorený tzv. bielymi znakmi.

6.2 Transformácia AHLL do AST

Preklad AHLL do formy AST zaisťuje priamo parser pri preklade. Parser bol generovaný z gramatiky pre jazyk AHLL pomocou nástroja ANTLR .

Typ inštrukcie	Popis
ADD	súčet operandov
SUB	rozdiel operandov
MULT	násobenie operandov
DIV	celočíselné delenie
MODULO	zvyšok po celočíselnom delení
OR	logický súčet
AND	logický súčin
EQUAL	rovnosť
NOT_EQUAL	nerovnosť
LESS	menšie
LESS_EQ	menšie rovno
MORE	väčšie
MORE_EQ	väčšie rovno
IS	kopírovanie (priradenie) hodnoty
UNARY_MINUS	unárne mínus
UNARY_NEG	negácia operandu

Tabuľka 6.1: Matematické inštrukcie

Návratovou hodnotou parsera je priamo AST jazyka AHLL. V tomto strome sú odstránené pomocné nonterminály, ako napríklad komentáre a znak bodkočiarky z konca príkazov. V niektorých prípadoch sú dopĺňané nové tokeny ako napríklad identifikátor bloku kódu a pod.

Príklad transformácie programu v jazyku AHLL do AST je uvedený na obrázku 6.1. Kompletná množina transformácií je uvedená v súbore s gramatikou v prílohe A.

6.3 3AK jazyka AHLL

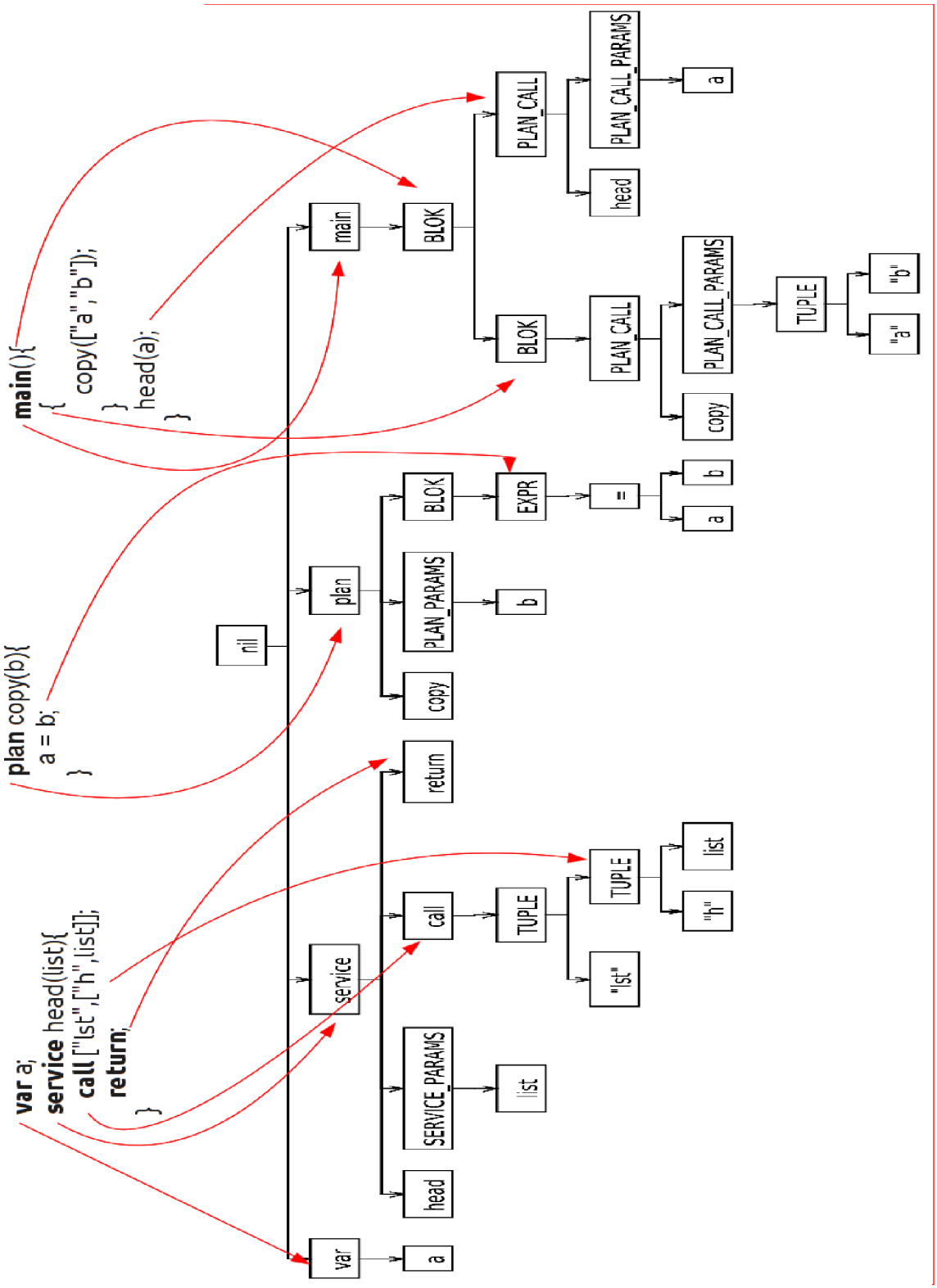
Návrh troj-adresného kódu, ktorý je použitý ako hlavná reprezentácia prekladaných algoritmov, vychádza z podporovaných vysokoúrovňových konštrukcií v jazyku AHLL. Ale na druhej strane sa ich už snažíme reprezentovať vo forme blízkej k nízkoúrovňovému jazyku ALLL.

Jednotlivé inštrukcie 3AK môžme rozdeliť do nasledovných kategórií:

- matematické, logické a relačné inštrukcie
- riadiace inštrukcie
- skokové inštrukcie
- ostatné inštrukcie

Matematické, logické a relačné inštrukcie majú tvar `res` ← `a INSTR b` v prípade binárnych operácií a `res` ← `INSTR a` v prípade unárnych. Položka `INSTR` je typ inštrukcie. Zoznam týchto typov je v tabuľke 6.1.

Riadiace inštrukcie sú inštrukcie ktoré ovplyvňujú tok programu. Jedná sa o inštrukcie, ktorými sú implementované cykly a podmienené výrazy. Zoznam riadiacich inštrukcií je uvedené v tabuľke 6.2. `COND_TYPE` v tabuľke 6.2 označuje typ podmienky (väčšie, menšie,



Obr. 6.1: Príklad transformácie AHLL do AST

Inštrukcia	Popis
`IF a COND_TYPE b ID`	v prípade platnosti výrazu (`a COND_TYPE b`) sa pokračuje nasledujúcou inštrukciou
`IF_TRUE ID`	vždy platná podmienka
`IF_FALSE ID`	nikdy neplatná podmienka
`ELSE ID`	alternatívna vetva
`END_IF ID`	koniec vetvenia
`WHILE ID`	začiatok cyklu
`END_WHILE ID`	koniec cyklu
`TEST a COND_TYPE b ID`	testovanie podmienky `(a COND_TYPE b)`, v prípade neplatnosti sa ide na `END_WHILE ID`
`TEST_TRUE ID`	vždy platná podmienka (nekonečný cyklus)
`TEST_FALSE ID`	nikdy neplatná podmienka (cyklus sa nikdy nevykoná)
`FOREACH ID`	začiatok cyklu na priechod zoznamom
`END_FOREACH ID`	koniec cyklu na priechod zoznamom
`res` ← `HEAD a ID`	posun na ďalší prvok v zozname
`res` ← `TAIL a ID`	odstránenie prvého prvku zoznamu

Tabuľka 6.2: Riadiace inštrukcie

test na rovnosť a pod.), `a`, `b`, `res` sú symboly alebo operandy. `ID` označuje jedinečný číselný identifikátor danej konštrukcie, slúžiaci na to, aby bolo možné priradiť navzájom začiatok a koniec podmienky, cyklu a pod. Tento identifikátor sa rovnako využíva pri generovaní kódu, kde v prípade cyklov sa od neho odvodzuje názov plánu, implementujúceho daný cyklus.

Podmienky sú implementované pomocou inštrukcií `IF`. V prípade, že podmienka platí pokračuje sa ďalšou inštrukciou. Inak sa pokračuje inštrukciou `ELSE` s rovnakým `ID` alebo, v prípade že táto inštrukcia chýba sa pokračuje inštrukciou `END_IF` s rovnakým `ID`. Inštrukcie `IF_TRUE` a `IF_FALSE` označujú podmienku, ktorá je vždy platná alebo naopak. Sú generované v rámci optimalizačnej fázy, ktorá je obsahom kapitoly 7.

Cyklus s podmienkou na začiatku, tzv. *while* cyklus je implementovaný pomocou príkazov typu `WHILE`, `TEST` a `END_IF`. Príkaz `WHILE` označuje začiatok cyklu, `WHILE_END` jeho koniec. Príkaz `TEST` testuje podmienku, ak podmienka platí pokračuje sa nasledujúcou inštrukciou, ak nie pokračuje sa inštrukciou `END_WHILE`. Opäť platí, že inštrukcie, ktoré patria k sebe majú rovnaké `ID`. Inštrukcie `TEST_TRUE` a `TEST_FALSE` sú opäť generované pri optimalizáciách.

Cyklus na priechod zoznamom je generovaný pomocou inštrukcií typu `FOREACH`, `HEAD`, `TAIL` a `END_FOREACH`. `FOREACH` označuje začiatok cyklu, `END_FOREACH` jeho koniec. Inštrukcia `HEAD` spôsobí posun na ďalší prvok v zozname a inštrukcia `TAIL` odstraňuje prvý prvok v zozname. Ak doiterujeme do situácie, že zostávajúci zoznam je jednoprvkový, inštrukcia `TAIL` pokračuje inštrukciou `END_FOREACH`.

Skokové inštrukcie sú uvedené v tabuľke 6.3. Tieto inštrukcie nemajú svoj ekvivalent v jazyku ALLL. Ten neobsahuje žiadne skoky. Slúžia len ako pomocné značky, ktoré sa využívajú pri analýze toku riadenia programu. Bez týchto inštrukcií je budovanie grafu toku riadenia značne náročné.

Inštrukcia `GOTOIF` označuje skok dopredu, tj. na inštrukciu `END_IF`. Naopak inštrukcia `GOTOWH` označuje skok späť na inštrukciu `WHILE` s rovnakým `ID`. To znamená, že označuje ďalšiu iteráciu cyklu. Podrobnejšie si zmysel týchto inštrukcií objasníme v kapitole 7.1.

Inštrukcia	Popis
GOTOIF ID	pokračovanie inštrukciou END_IF s rovnakým ID
GOTOWH ID	pokračovanie inštrukciou WHILE s rovnakým ID

Tabuľka 6.3: Skokové inštrukcie

Inštrukcia	Popis
FI	zákaz prijímania správ
OI	povolenie prijímania správ
SEND addr message	odoslanie správy
res ←RECEIVE addr	prijatie správy
CALL name params	spustenie plánu s názvom `name` a parametrami `params`
res ←SERVICE name params	volanie služby platformy s názvom `name`, parametrami `params`, výsledok sa uloží do `res`
SERVICE name params	volanie služvy platformy s názvom `name`, parametrami `params`

Tabuľka 6.4: Ostatné inštrukcie

Ostatné inštrukcie sú uvedené v tabuľke 6.4. Jedná sa o doplnkové inštrukcie, ktoré neboli zaradené do žiadnej z predchádzajúcich kategórií. V tabuľke 6.4 je význam skratiek nasledovný: `res`, `addr`, `message` sú operandy (symboly) `name` je meno služby platformy alebo plánu definovaného v AHLL, `params` je zoznam operandov (symbolov).

6.4 Transformácia AST do 3AK

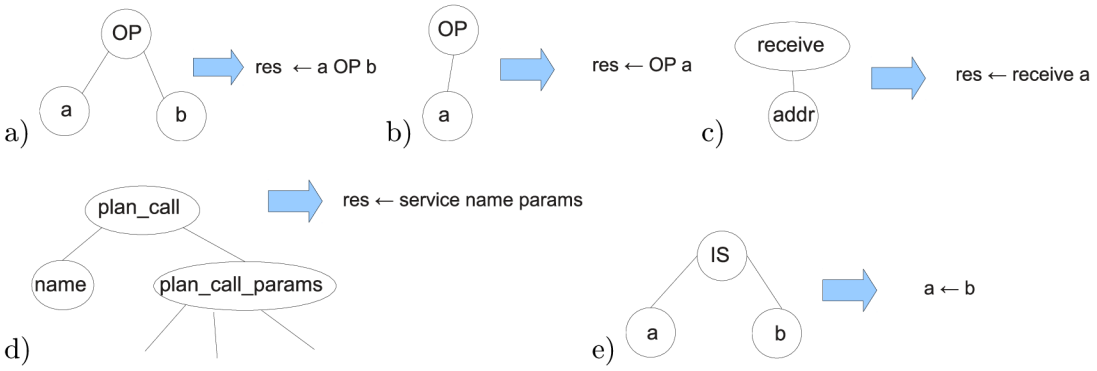
V tejto časti si predstavíme transformáciu jednotlivých častí AST do 3AK, ktorý bol predstavený v kapitole 6.3. Transformácia je robená spôsobom rekurzívneho zostupu na AST a postupnom generovaní 3AK inštrukcií. Ako už vyplýva z použitia rekurzívneho zostupu, najprv sa vyhodnotia potomkovia uzlov, následne uzol samotný. Jedná sa o priechod stromu typu postorder.

Najprv si ukážeme transformácie výrazov. Transformácia výrazov je zobrazená na obrázku 6.2. `OP` znamená operácia, `a`, `b`, `addr`, sú už vyhodnotené symboly. `res` je novo vytvorený symbol, v ktorom bude výsledok operácie. V AST sú služby platformy rovnako ako plány označené tým istým tokenom (`plat_call`). Je to z toho dôvodu, že gramatika pre obe typy volaní je rovnaká. V prípade, že sa spracúva tento token, prekladač vyhľadá podľa mena (name) v tabuľke plánov a tabuľke služieb príslušnú funkciu.

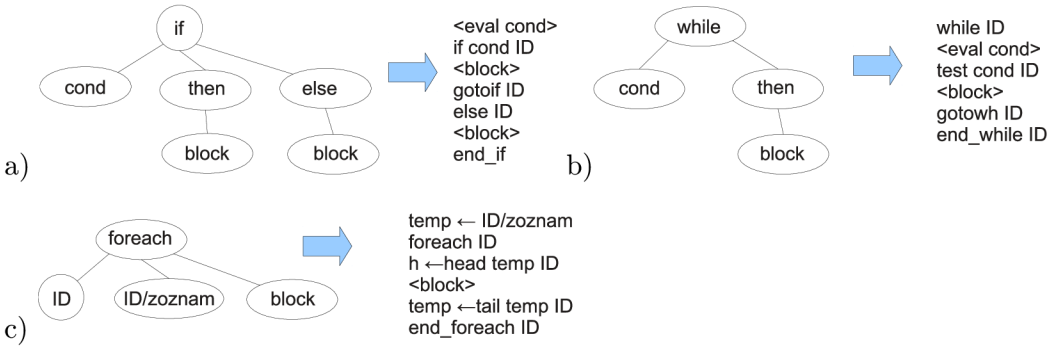
Transformácia blokov spočíva v tom, že sa vyhodnotí každý podstrom uzlu `block`. Pri vstupe do nového bloku sa vytvorí nová tabuľka v zásobníku symbolov, ktorá spravuje premenné deklarované v tomto bloku. Je to z dôvodu prekrývania premenných v AHLL. V prípade vyhľadávania premenných sa postupuje od aktuálneho bloku smerom k nadradeným blokom.

Transformácia konštrukcií (`if`, `while`, `foreach`) do značnej miery vychádza z ich implementácie v jazyku ALLL uverejnených v [10] a [5]. Transformácia konštrukcií je znázornená na obrázku 6.3. Ako už bolo spomínané, inštrukcie `GOTOIF` a `GOTOWH` nemajú svoj ekvivalent v jazyku ALLL. Slúžia len na zjednodušenie analýzy toku riadenia.

Transformácia podmieneného príkazu je uvedená na obrázku 6.3 a). Podmienený príkaz



Obr. 6.2: Transformácia výrazov: a) binárny výraz b) unárny výraz c) prijatie správy d) volanie služby platformy e) kopírovanie symbolu do iného symbolu



Obr. 6.3: Transformácia konštrukcií a) podmienený príkaz b) cyklus s podmienkou na začiatku (while) c) cyklus na priechod zoznamom (foreach)

je v ALLL implementovaný pomocou priameho spustenia plánov pre vetvu splnenej a nesplnenej podmienky [5]. Vždy jeden z plánov zlyhá. Plán pre vetvu, kde je splnená podmienka je vymedzený inštrukciami `IF` a `ELSE`, prípadne `IF` a `END_IF`, ak je vetva *else* vynechaná. Plán pre vetvu, kde je podmienka nesplnená je vymedzený inštrukciami `ELSE` a `END_IF`.

Synom uzlu `cond` je výraz, ktorý reprezentuje vyhodnotenie podmienky. Tento sa vyhodnotí pomocou transformácií pre výrazy. V prípade, že priamy následník uzlu `cond` je relačný operátor[1], tento je priamo súčasťou inštrukcie `IF`. V opačnom prípade, napríklad keď je podmienkou výraz typu $a + b$ sa vygenerujú inštrukcie na jeho výpočet. V inštrukcii `IF` je potom testovanie výsledku na rozdielnosť od 0.

Transformácia cyklu s podmienkou na začiatku, tj. *while* cyklu, je zobrazená na obrázku 6.3 b). Transformácia je obdobná ako transformácia podmieneného príkazu. Rozdiel je len v tom, že pri cykle treba zaistiť opakovanie daného bloku inštrukcií. Namiesto priameho spustenia plánu sa vytvorí nový plán v báze plánov s týmito inštrukciami. Tento plán sa v mieste cyklu zavolá. Jeho poslednou inštrukciou je volanie samého seba [5]. Tento plán je vymedzený inštrukciami `WHILE` a `END_WHILE`. V pláne sa vyhodnotí výraz s podmienkou (podstrom uzlu `cond`) a obdobne ako v prípade podmieneného príkazu sa nastaví inštrukcia `TEST`. Tá v prípade neúspechu ukončí opakovanie cyklu.

Transformácia cyklu na priechod zoznamom je zobrazená na obrázku 6.3 c). Keďže sa jedná opäť o cyklus, použije sa rovnaká technika ako pri predchádzajúcom cykle *while*, tj.

[1] v ALLL relačné operátory v prípade nesplnenia podmienky spôsobujú pád podplánu

cyklus bude implementovaný ako pomenovaný plán v báze plánov. Tento plán je vymedzený inštrukciami `FOREACH` a `END_FOREACH`. Pred jeho spustením je potrebné zoznam, cez ktorý sa iteruje prekopírovať do pomocnej premennej (`temp`). následne sa spustí plán, kde každá iterácia začína uložením hlavičky zoznamu v premennej `temp` do ďalšej pomocnej premennej, `h`. Následne sa spustia inštrukcie vnútri cyklu. Ako poslednou inštrukciou je uloženie tela zoznamu do premennej `temp`, pomocou inštrukcie `TAIL`. Táto inštrukcie (a jej ekvivalent v ALLL) v prípade jednoprvkového zoznamu spôsobuje zlyhanie podplánu.

Transformácia zvyšku stromu je pomerne priamočiara a takmer vždy existuje jej ekvivalent v podobe inštrukcie v tabuľke 6.4, prípadne sa jedná len o prácu s tabuľkou symbolov, prípadne tabuľkou plánov, napríklad pri uzloch `PLAN` a pod.

Kapitola 7

Implementácia optimalizácií

V tejto kapitole bude predstavená implementácia jednotlivých optimalizácií prekladača, vychádzajúca z kapitoly 4.

7.1 Implementácia analýzy toku riadenia

Táto sekcia ukazuje implementáciu analýzy toku riadenia v prekladači. Analýza toku riadenia bola teoreticky predstavená v kapitole 4.2. Najprv bude predstavená vlastná reprezentácia grafu toku riadenia, tak ako je implementovaná v prekladači.

7.1.1 Reprezentácia grafu toku riadenia

Graf toku riadenia je reprezentovaný samostatnou triedou. Táto trieda obsahuje 2 verejné statické vnorené triedy.

Prvá reprezentuje základný blok programu. Obsahuje jednoznačný číselný identifikátor bloku, ďalej obsahuje zoznam inštrukcií, ktoré sú v danom bloku. Každý základný blok obsahuje referencie na nasledovníkov a predchodcov tohto bloku.

Druhá trieda reprezentuje plán. Obsahuje meno plánu a zoznam základných blokov (predchádzajúca trieda).

Samotný graf toku riadenia celého prekladaného programu je potom reprezentovaný ako zoznam týchto plánov. Každý plán obsahuje k nemu prislúchajúci lokálny graf toku riadenia.

7.1.2 Vytvorenie grafu toku riadenia

Graf toku riadenia sa vytvára v troch na seba nadväzujúcich krokoch.

Prvým krokom je rozdelenie inštrukcií do základných blokov. To sa deje identifikáciou *leader* inštrukcií, ako už bolo spomínané v kapitole 4.2. Leader inštrukciami sú nasledovné inštrukcie z používaného 3AK:

- Prvá inštrukcia v pláne.
- Inštrukcie spôsobujúce vetvenie programu. Jedná sa o istú formu inštrukcií skoku. Sú to posledné inštrukcie v základnom bloku:
  - `IF`, `IF_TRUE`, `IF_FALSE`
  - `TEST`, `TEST_TRUE`, `TEST_FALSE`

  - `GOTOIF`, `GOTOWH`, `TAIL`

- Inštrukcie, ktoré sú cieľom nejakej skokovej inštrukcie. V tomto prípade sa jedná o istú formu návestia. Tieto inštrukcie sú prvými inštrukciami základného bloku:

  - `ELSE`, `WHILE`, `FOREACH`
  - `END_IF`, `END_WHILE`, `END_FOREACH`

Druhým krokom je vyhľadanie následníkov základných blokov. Následníci základných blokov sú vypočítaný podľa poslednej inštrukcie daného bloku nasledovne: Ak je poslednou inštrukciou bloku

- `IF`, `IF_TRUE`, `IF_FALSE` – prvý následník je ďalší blok v poradí (vetva, kde je splnená podmienka), druhým je blok začínajúci inštrukciou `ELSE` alebo `END_IF` s rovnakým identifikátorom.
- `GOTOIF` – následník je blok, začínajúci inštrukciou `END_IF` s rovnakým ID.
- `GOTOWH` – následník je blok, začínajúci inštrukciou `WHILE` s rovnakým ID.
- `TEST`, `TEST_TRUE`, `TEST_FALSE` – prvý následník je ďalší blok v poradí (splnená podmienka), druhým je blok začínajúci inštrukciou `END_WHILE` s rovnakým ID.
- `TAIL` – prvým nasledovníkom je blok začínajúci inštrukciou `FOREACH` s rovnakým identifikátorom, druhým je nasledujúci blok.

V tomto kroku sa do grafu pridajú aj uzly START a STOP.

Posledným krokom vytvárania grafu je vyhľadanie predchodcov základných blokov. Toto sa deje za pomoci už vypočítaných následníkov. Algoritmus na výpočet je pomerne jednoduchý:

```
foreach b ∈ BasicBlocks
   foreach bs ∈ BasicBlocks
      if Succ(bs) contains b then
         Pred(b) += bs
```

V skratke, ak nejaký iný uzol obsahuje práve spracovávaný uzol ako svojho následníka, tak práve spracovávaný uzol je jeho predchodcom.

Nakoniec dodáme, že v rámci analýzy bola implementovaná aj funkcia, ktorá už vytvorený graf toku riadenia pretransformuje do podoby, v ktorej každý blok[1] obsahuje práve jednu inštrukciu. Niektoré optimalizačné algoritmy sú popisované a implementované s využitím práve takejto formy grafu toku riadenia.

7.2 Implementácia algoritmu na vyhodnotenie konštantných výrazov

Tento algoritmus bol teoreticky popísaný v kapitole 4.3.1. Algoritmus využíva funkciu `Exp_Kind(instr)`, ktorá vracia triedu inštrukcie. Táto funkcia delí 3AK inštrukcie na nasledujúce triedy:

[1] matematicky sa už nejedná o základné bloky

- `BINEXP` (binárne): `ADD`, `SUB`, `MULT`, `DIV`, `MODULO`, `AND`, `OR`, `LESS`, `LESS_EQ`, `MORE`, `MORE_EQ`, `EQUAL`, `NOT_EQUAL`
- `UNEXP` (unárne): `UNARY_MINUS`, `UNARY_NEG`
- `TESTEXP` (testovacie): `TEST`
- `IFEXP` (inštrukcie podmieneného výrazu): `IF`
- `CALL`
- `SERVICE`

Implementovaný algoritmus pracuje podľa algoritmu uvedeného v kapitole 4.3.1. V prípade triedy `UNEXP`, resp. `BINEXP` je výraz vyhodnotený a inštrukcia zmenená na inštrukciu priradenia v prípade, že operand, resp. oba operandy sú konštantné.

V prípade triedy `TESTEXP` sa postupuje nasledovne. Ak sú oba operandy konštantné výraz určujúci podmienku sa vyhodnotí a inštrukcia sa zmení na inštrukciu `TEST_TRUE` resp. `TEST_FALSE`. Inštrukcie triedy `IFEXP` sa obdobne akurát inštrukcia sa mení na `IF_TRUE`, resp. `IF_FALSE`.

7.3 Implementácia propagácie konštánt

Propagácia konštánt je implementovaná podľa algoritmu a jeho opravy predstaveného v kapitole 4.3.3. Optimalizácia je implementovaná na lokálnej úrovni, tj. na úrovni plánov. Propagácia konštánt v rámci celého kódu zatiaľ nie je implementovaná.

Podotýkame, že v každom bloku sa nachádza práve jedna inštrukcia, okrem blokov START a STOP. V našom prípade bola zvolená nasledovná inicializácia jednotlivých symbolov používaných v pláne. Symboly reprezentujúce konštanty, či už zapísané priamo vo vstupnom kóde alebo symboly definované kľúčovým slovom `const` sú priamo inicializované ako konštanty.

Na druhej strane globálne premenné sú hneď inicializované na hodnotu $\bot$. Inicializácia globálnych premenných na $\bot$ je zvolená z dôvodu, že hodnotu globálnych premenných môže zmeniť každý plán. Skutočnosť, že ktoré plány menia ktoré premenné sa v súčastnosti neanalyzuje. Preto postupujeme konzervatívne a globálne premenné rovno prehlásime za premenné, ktoré môžu meniť svoju hodnotu v každej inštrukcii.

Ostatné, lokálne premenné inicializujeme na $\top$, tak ako je to v algoritme.

Inštrukcie, prípadne bloky delíme do troch rôznych kategórií:

- bloky, v ktorom je inštrukcia priradenia
  - `ADD`, `SUB`, `MULT`, `DIV`, `MODULO`, `OR`, `AND`, `EQUAL`, `NOT_EQUAL`, `LESS`, `LESS_EQ`, `MORE`, `MORE_EQ`, `IS`, `UNARY_MINUS`, `UNARY_NEG`
  - `RECEIVE` a `SERVICE`, len v prípade, že svoj výsledok priraďujú do nejakej premennej
- bloky vetvenia
  - `IF`, `IF_TRUE`, `IF_FALSE`, `TEST`, `TEST_TRUE`, `TEST_FALSE`, `TAIL`

- ostatné bloky, vrátane blokov priradenia, ak je cieľom globálna premenná. Ako sme už spomínali v takomto prípade sa správame konzervatívne a chceme potlačiť optimalizáciu globálnych premenných.

Ďalším krokom, ktorý bolo potreba prispôsobiť potrebám nášho prekladača bolo vyhodnotenie výrazov v jednotlivých blokoch (inštrukciách). Bloky, obsahujúce inštrukciu:

- `ADD, SUB, MULT, DIV, MODULO, EQUAL, NOT_EQUAL, MORE, MORE_EQUAL, LESS, LESS_EQ, AND, OR, UNARY_MINUS a UNARY_NEG` sa vyhodnotia štandardne podľa pravidiel uvedených v kapitole 4.3.3 a podľa bežného matematického chápania.
- `IF` a `TEST` sa podmienka vyhodnotí, rovnako ako v predchádzajúcom prípade, podľa pravidiel uvedených v kapitole 4.3.3.
- `IF_TRUE, IF_FALSE, TEST_TRUE, TEST_FALSE` sa vyhodnotí ako konštanta s príslušnou hodnotou.
- `TAIL, HEAD, SERVICE, RECEIVE` sa vyhodnotí ako $\perp$, kvôli postranným efektom týchto inštrukcií.

Ako sme už spomínali v kapitole 4.3.3, po skončení algoritmu máme k dispozícii informácie o stave premenných v každom bloku, ako aj o výsledkoch výrazov v jednotlivých blokoch programu. Rovnako máme k dispozícii aj informácie o tom, ktoré vetvy programu sú spustiteľné a ktoré nie. Tieto informácie využijeme pri transformácii kódu. Rovnako v tomto algoritme rekonštruujeme kód tak, aby sa odstránili všetky `*_FALSE`[2] a `*_TRUE`[3] inštrukcie. Tieto transformácie si popíšeme v nasledujúcich odsekoch.

Transformáciu robíme v 4 krokoch:

1. Transformácia inštrukcií podľa hodnôt v blokoch.
2. Vytvorenie programu tak, aby v ňom boli len bloky, ktoré sú dosiahnuteľné.
3. Odstránenie `IF_TRUE` a `IF_FALSE` inštrukcií.
4. Odstránenie `TEST_TRUE` a `TEST_FALSE` inštrukcií.

Inštrukcie transformujeme podľa hodnôt, na ktorých sa ustálila štruktúra *Lattice* z kapitoly 4.3.3 v jednotlivých blokoch nasledovne. Ak sa jednalo o inštrukciu priradenia a výsledok inštrukcie je konštanta, tak sa táto inštrukcia nahradí priradením konštanty do premennej. Ak výsledok nebol konštantou, skúsime nahradiť aspoň operandy inštrukcie konštantami.

Ak sa jedná o inštrukciu vetvenia a výsledok tejto inštrukcie je konštantný nahradíme inštrukciu príslušnou `*_TRUE` alebo `*_FALSE` inštrukciou. Ak výsledok nebol konštantný, opäť skúsime nahradiť operandy konštantami. V prípade inštrukcie `CALL` a `SERVICE`, resp. `SEND` a `RECEIVE` skúsime nahradiť parametre, resp. operandy konštantami, ak to je možné.

Ďalším krokom je transformácia grafu toku riadenia späť do lineárnej formy 3AK. Do výsledného optimalizovaného kódu, vo forme 3AK sa dostanú len inštrukcie z uzlov grafu, ktoré sú dosiahnuteľné. Detekcia týchto uzlov je jednoduchá. Ak existuje hrana do bloku b, ktorá je spustiteľná, uzol b je dosiahnuteľný. Matematicky

$$\exists (n \to b) \in E \wedge n \in N \wedge (n \to b)\, je\, spustiteľná \Rightarrow b\, je\, dosiahnuteľný$$

[2] IF_FALSE, TEST_FALSE
[3] `IF_TRUE, TEST_TRUE`

Výnimku tvoria len inštrukcie `END_IF`, `END_WHILE` a `END_FOREACH`, ktoré sa do výsledného kódu vkladajú, ak existuje ich zrkadlová varianta (`IF`, `WHILE`, `FOREACH`) s rovnakým `ID`.

Predposledným krokom transformácie kódu je odstránenie `IF_TRUE` a `IF_FALSE` inštrukcií. Tieto inštrukcie boli generované optimalizáciou vyhodnotenia konštantných výrazov a propagáciou konštánt. Transformáciu vykonávame nasledovne. Prechádzame kódom a ak narazíme na inštrukciu `IF_TRUE` alebo `IF_FALSE`, tak vymažeme všetky inštrukcie `ELSE` a `END_IF` s rovnakým `ID`. Treba podotknúť, že túto transformáciu môžeme vykonať takto jednoducho preto, lebo využívame fakt, že inštrukcie asociované s nedosiahnuteľnou vetvou už boli vymazané v predchádzajúcom kroku transformácie. Takto prechádzame kódom až kým neodstránime všetky `IF_TRUE` a `IF_FALSE` inštrukcie.

Poslednou transformáciou je odstránenie inštrukcií `TEST_TRUE` a `TEST_FALSE`. Ak sa v kóde nachádza inštrukcia `TEST_TRUE`, tak sa sémanticky jedná o nekonečný cyklus. V takomto prípade sa inštrukcia `TEST_TRUE` vymazáva. Inštrukcie `WHILE` a `END_WHILE` ostávajú.

Iná situácia nastáva pri inštrukcii `TEST_FALSE`, pretože v tomto prípade sa naopak jedná o cyklus, ktorý sa nikdy neprevedie. Preto vymažeme okrem inštrukcie `TEST_FALSE` aj inštrukcie `WHILE` a `END_WHILE` s rovnakým `ID`. Opäť využívame fakt, že nedosiahnuteľný kód bol už eliminovaný.

Po týchto krokoch máme kód opäť vo forme 3AK, kde inštrukcie sú umiestnené lineárne za sebou. Nenachádzajú sa v ňom inštrukcie, ktoré boli (konzervatívnym spôsobom) vyhodnotené ako nedosiahnuteľné. Rovnako sa tam nenachádzajú inštrukcie `IF_TRUE`, `IF_FALSE`, `TEST_TRUE` a `TEST_FALSE`.

7.4 Implementácia propagácie kópií

Táto optimalizácia je teoreticky popísaná v kapitole 4.3.4. Optimalizácia je implementovaná podľa algoritmu uvedeného v zmienenej kapitole. Na svoju činnosť využíva graf toku riadenia.

V rámci tejto optimalizácie je spustená najprv propagácia kópií na lokálnej úrovni, tj. na úrovni základných blokov. V rámci tohto kroku sa okrem samotnej propagácie vypočíta aj množina symbolov, ktoré sú v danom bloku definované. Jedná sa o množinu ACP z algoritmu.

Následne sa vypočítajú množiny $COPY$, $KILL$, $CPin$ a $CPout$ pre každý základný blok, tak ako to bolo uvedené v kapitole 4.3.4. Následne sa opäť spustí metóda lokálnej propagácie kópií, avšak s už prednastavenou množinou ACP podľa údajov v množine $CPin$ daného bloku.

Výstupom tejto metódy je graf toku riadenia s transformovanými inštrukciami.

7.5 Implementácia eliminovania mŕtvych inštrukcií

Implementácia tejto optimalizácie sa odlišuje od implementácie spomínanej v kapitole 4.3.5, ale je založená na podobnom princípe.

Eliminácia mŕtveho kódu používa na svoju činnosť graf toku riadenia a postupuje nasledovne. Najprv pre každý blok sa vypočítajú všetci jeho následníci, formou tranzitívneho uzáveru.

Následne sa pre každú inštrukciu v každom bloku vypočíta, či je určite mŕtva alebo pravdepodobne nie. Tento výpočet sa robí jednoduchým spôsobom tak, že keď premenná, do

ktorej inštrukcia priradzuje svoj výsledok nie je použitá ako parameter v žiadnej inštrukcii, ktorá za ňou nasleduje v danom bloku alebo v nasledujúcich blokoch je prehlásená za mŕtvu.

Výnimku tvoria opäť inštrukcie, ktoré priraďujú svoju hodnotu do globálnej premennej. Tieto inštrukcie sú automaticky prehlásené za živé.

Ak je premenná detekovaná ako mŕtva, upraví sa nasledovným spôsobom, podľa inštrukcií:

- `ADD`, `SUB`, `MULT`, `DIV`, `MODULO`, `OR`, `AND`, `EQUAL`, `NOT_EQUAL`, `LESS`, `LESS_EQ`, `MORE`, `MORE_EQ`, `IS`, `UNARY_MINUS`, `UNARY_NEG` – inštrukcia sa vymaže
- `RECEIVE` a `SERVICE` – inštrukcia sa nevymaže, kvôli postranným efektom. Avšak vymaže sa z tejto inštrukcie premenná, do ktorej inštrukcia priraďovala svoj výsledok.

Tieto kroky sa opakujú, kým sa nedosiahne pevného bodu, tj. žiadna inštrukcia nebola zmenená/vymazaná.

Tento algoritmus je pomalší ako algoritmus predstavený v kapitole 4.3.5, rovnako aj menej účinný. Jeho výhodou je ale jednoduchšia implementácia.

Kapitola 8

Generovanie cieľového kódu v ALLL

V tejto kapitole si predstavíme záverečnú časť prekladu, ktorou je generovanie cieľového kódu v jazyku ALLL. Cieľový kód sa generuje z internej reprezentácie prekladaného programu, ktorou ako sme už spomínali, je troj-adresný kód.

Predtým, ako si bližšie prestavíme generovanie jednotlivých inštrukcií z 3AK, si priblížime prácu s registrami a to, ako tieto registre využívame. Následne si formou makier predstavíme generovanie primitív jazyka ALLL, ktoré využijeme pri generovaní jednotlivých inštrukcií 3AK a tým vlastne generovanie cieľového kódu v ALLL z 3AK.

8.1 Generovanie primitív v ALLL

Pod pojmom primitíva budeme rozumieť jednoduché vysokoúrovňové konštrukcie, ako napríklad kód na načítanie hodnoty premennej z bázy znalostí, spustenie plánu a pod. Generovanie jednotlivých primitív si prestavíme formou makier. Na tieto primitíva sa budeme odvolávať v ďalších častiach tejto kapitoly.

Premenné používané v AHLL sú v ALLL uložené v báze znalostí. Sú uložené vo forme `(meno,hodnota)`. Ako prvé si predstavíme pomenovávanie premenných používaných v AHLL v nízkoúrovňovom ALLL:

```
alllName(symbol):
  if symbol is global variable  return symbol.name
  elif symbol is in main        return #block + symbol.name
  else                          return (#block+symbol.name,plan.name)
```

V prípade, že meno premennej potrebujeme použiť pri dotaze na bázu znalostí použijeme nasledovné pomenovanie:

```
alllQueryName(symbol):
  if symbol is global variable  return symbol.name
  elif symbol is in main        return #block + symbol.name
  else                          return [#block+symbol.name,plan.name]
```

Aby sme to zhrnuli premenné sú pomenované nasledovne:

- globálne premenné – ich názvom v AHLL

- lokálne premenné plánu `main` – reťazcom zloženým z konkatenácie čísla bloku (`#block`) a mena premennej
- lokálne premenné ostatných plánov – n-ticou tvorenou 2 položkami:
  - reťazcom zloženým z konkatenácie čísla bloku a mena premennej
  - menom plánu

Lokálne premenné sú uložené vo forme n-tice z dôvodu, aby sa dali odstrániť všetky lokálne premenné daného plánu z bázy znalostí, po jeho skončení, jedinou akciou jazyka ALLL.

Teraz si predstavíme generovanie identifikátorov pre registre. Parametrom tejto operácie je úroveň zanorenia plánu a číslo registra. O hierarchii plánov a ich úrovni zanorenia si povieme viac v nasledujúcej sekcii tejto kapitoly. Nateraz je len podstatné, že toto makro vráti identifikáciu registra, predchádzané počtom apostrofov podľa premennej `level`:

```
genRegister(level,register):
  return (level*')&register
```

Ďalším predstaveným primitívom je načítanie premennej do registra. Využívame 2 akcie ALLL. Prvou je dotaz na bázu znalostí a druhou je volanie služby platformy. Dotaz na bázu znalostí spôsobí, že v aktívnom registri bude zoznam n-tíc, ktoré vyhovovali dotazu. V našom prípade to bude jednoprvkový zoznam. Následne zavoláme službu platformy na prácu so zoznamami, s parametrom, ktorý vráti postupne prvý prvok zoznamu, tým sa dostaneme k n-tici reprezentujúcej premennú. Ďalej, z tejto n-tice vrátime zvyšok zoznamu, tým sa dostaneme k zoznamu hodnôt danej premennej (jednoprvkovému). Z tohto zoznamu vrátime prvý prvok, čo je hodnota danej premennej. Jedná sa o postupnosť operácií *head tail head* nad zoznamom.

```
readVariable(symbol,register):
   return  #[alllQueryName(symbol),_],#(lst,(hth,&register))
```

Ďalším primitívom je uloženie premennej z registra do bázy znalostí. Keďže prekladač zatiaľ nezbiera informácie o tom, kedy sa premenná prvýkrát použila, tj. či sa už nachádza v báze znalostí, treba predpokladať, že premenná v báze znalostí môže aj nemusí byť. Totiž pokus o vymazanie n-tice, ktorá v báze znalostí nie je spôsobuje zlyhanie podplánu. Preto v tomto kroku využijeme priame spustenie plánu, v ktorom sa túto premennú pokúsime vymazať. Aby sme si v prípade zlyhania tohto plánu nepremazali hodnotu premennej v registri (predpokladáme, že môže byť aktívnym registrom) nastavíme na aktívny register č. 3:

```
storeVariable(symbol, register):
   return @<$3,-[alllQueryName(symbol),_]>,+(alllName(symbol),&register)
```

V prípade, že potrebujeme uložiť do premennej konštantu používame nasledovné makro. Opäť musíme predpokladať, že premenná môže ale nemusí byť v báze znalostí, preto sa ju pokúsime vymazať rovnako ako v predchádzajúcom prípade. Akcia priradenia konštanty do premennej má tvar:

```
addVarToBB(symbol, value):
  return @<$3,-[alllQueryName(symbol),_]>,+((alllName(symbol),value))
```

Ďalšie primitíva sú pridanie, resp. vymazanie hodnoty z bázy znalostí a priame, resp. nepriame spustenie plánu:

```
addToBB(value):
  return +(value)

removeFromBB(value)
  return -(value)

imRun(code):
  return @<code>

planRun(planName):
  return @^(planName)
```

Nastavenie registra na aktívny, odoslanie správy, resp. jej prijatie:

```
setReg(register):
  return $register

genSend(address, message):
  return !(address,message)

genRecv(address):
  if address ≠ ∅    return ?(address)
  else              return ?(_)
```

Záverom si predstavíme primitíva na generovanie aritmetiky a relačných operácií. Aritmetika, tj. operácie sčítania, odčítania, násobenia, celočíselného delenia generované nasledovným makrom:

```
arithmetic(type, op1,op2):
  switch(type):
    case ADD:   return #(ari,(p,op1,op2))
    case SUB:   return #(ari,(m,op1,op2))
    case MULT:  return #(ari,(k,op1,op2))
    case DIV:   return #(ari,(d,op1,op2))
```

Relačné operácie sú generované nasledovným makrom:

```
testRel(type, op1, op2):
  switch(type):
    case LESS:     return #(rel,(l,op1,op2))
    case LESS_EQ:  return #(rel,(le,op1,op2))
    case MORE:     return #(rel,(g,op1,op2))
    case MORE_EQ:  return #(rel,(ge,op1,op2))
    case EQUAL:    return #(rel,(e,op1,op2))
    case NOT_EQUAL:return +(p),@<#(rel,(e,op1,op2)),-(p)>,-(p)
```

8.2 Hierarchia plánov a registre

Táto časť kapitoly bude prevažne zameraná na prácu s registrami pri generovaní kódu v spojitosti s hierarchiou plánov v ALLL. Najväčší dôraz bude kladený na to, akým spôsobom sa využíva potlačenie nahradenia identifikácie registrov ich hodnotami v prípade volania podplánov. Jedná sa o akciu operátora apostrof, predstaveného v kapitole 3.5.

Ako už bolo spomínané v kapitole 3.5, ALLL obsahuje 3 registre na všeobecné použitie. Tieto registre využívame na dočasné uloženie parametrov operácií a ich výsledkov. Avšak v podplánoch potrebujeme tieto registre takisto používať ako úložisko dočasných dát, preto si v nasledujúcich riadkoch objasníme spôsob dosiahnutia tejto funkcionality.

Spôsob, akým pracujeme s registrami je potrebné objasniť z nasledujúcich dôvodov. Pri transformácii jazyka AHLL do 3AK sme vytvorili lineárnu reprezentáciu prekladaného programu. Táto linearizácia nám výrazne uľahčila optimalizovanie 3AK. Avšak jazyk ALLL je založený na hierarchii plánov, preto bol 3AK navrhnutý tak, aby sa táto hierarchia bola do istej miery zachovaná. O zachovanie tejto hierarchie sa starajú inštrukcie `END_IF`, `END_WHILE` a `END_FOREACH`.

V práci [5] bol predstavený spôsob prekladu jednotlivých vysokoúrovňových konštrukcií do jazyka ALLL. Cykly sa implementujú formou dodatočných plánov, uložených v báze plánov. Podmienky naopak priamym spustením plánov. Toto spôsobuje dodatočné zanorovanie sa plánov, s ktorým pri generovaní musíme počítať.

Pri generovaní cieľového kódu postupujeme sekvenčne, inštrukciu za inštrukciou. Na uchovávanie informácií o úrovni zanorenia plánov používame 2 zásobníkové štruktúry. Do prvej sa generuje zámer aktuálne generovaného plánu, či už sa jedná o plán definovaný v prekladanom zdrojovom kóde, alebo plán implementujúci nejaký cyklus. Takže vždy na vrchole zásobníku je uložený zámer práve generovaného plánu. Po skončení generovania tohto plánu sa jeho kód zo zásobníka vyberie a vloží sa do príslušnej štruktúry.

Druhý zásobník určuje aktuálne zanorenie plánov pre generovanie registrov. Jedná sa o zásobník kladných celých čísel.

Práca s týmito zásobníkmi prebieha nasledovne. V prípade, že začíname generovať akcie nejakého cyklu alebo plánu iného od plánu `main` do zásobníku zámerov sa vloží nová položka. Ostatné inštrukcie generujú svoje akcie v ALLL do zámeru na vrchole tohto zásobníku.

Do zásobníku zanorenia pre registre sa v tomto prípade vloží nová položka s číslom 1. Dôvodom je, že v čase spustenia plánu sa z každej identifikácie registra odmaže jeden apostrof, čiže registre budú použiteľné na zápis po spustení tohto plánu.

V prípade plánu `main` sa na vrchol zásobníka zanorenia pre registre vloží číslo 0. Tento plán je prvým zámerom prekladaného agenta, to znamená registre musia byť hneď použiteľné na zápis a čítanie. Do zásobníku zámerov sa tak ako v predchádzajúcom prípade vloží nový zámer.

Po ukončení generovania plánov sa z oboch zásobníkov vyberie ich vrchol. Vrchol zo zásobníka zámerov sa spracuje nasledovne:

- bol generovaný cyklus alebo plán rôzny od `main`-u – výsledný zámer sa vloží ako nový plán, s príslušným menom do bázy plánov
- bol generovaný plán `main` – zámer sa vloží do zámeru agenta

V prípade podmienok sa postupuje mierne odlišne. Do zásobníku zámerov vložíme opäť nový zámer. Rozdiel nastáva pri zásobníku zanorenia pre registre. V tomto prípade sa nevkladá nová položka ale položka na vrchole zásobníku sa inkrementuje, v prípade ukončenia

generovania dekrementuje. Dôvodom je to, že pre tieto akcie bude generované priame spustenie plánu takže musíme zvýšiť ich zanorenie tak, aby do momentu, keď sa tento plán spustí ostal jediný apostrof. Tento sa spustením plánu odstráni a register je použiteľný na potrebné účely.

Predchádzajúce odstavce si predvedieme na jednoduchom príklade. Predpokladajme nasledovný 3AK kód v pláne `main`:

```
WHILE ID=1
IF a > 5 ID=2
a <- a + 1
END_IF ID =2
a <- a - 2
END_WHILE ID=1
a <- a * 3
```

Tento kód sa preloží nasledovne. Upozorňujeme čitateľa, že pre väčšiu prehľadnosť boli z kódu odstránené pre náš príklad nepodstatné časti ako načítavanie premenných a pod.

```
>intention: <@^(w1),...,#(ari,(k,&1,3))>

>database:  { w1: < @<...#(rel,(g,''&1,5)),..., #(ari,(p,''&1,1))>, ...
                    #(ari,(m,'&1,2)),..., ^@(w1)
                  >
            }
```

Najprv sa generuje plán `main`, čiže na zásobníku zámerov je vložený nový zámer a do zásobníku zanorenia číslo 0. Generovaním inštrukcie `WHILE` sa na zásobník zámerov vloží nový zámer a na zásobník zanorenia pre registre sa vloží 1. Nasleduje inštrukcia `IF`. Opäť sa vkladá nový zámer na zásobník zámerov, na zásobníku zanorenia sa inkrementuje číslo na vrchole, čiže na vrchole zásobníku je 2. Generuje sa inštrukcia súčtu, identifikátor registra má pred sebou 2 apostrofy, čo zodpovedá 2 spusteniam plánov (plán w1, implementujúci cyklus a priame spustenie pre podmienku). Takže v čase vykonávania tejto akcie už identifikátor bude bez apostrofom a register je použiteľný.

Nasleduje inštrukcia `END_IF`, zo zásobníka sa vyberie zámer z vrcholu zásobníku. Spraví sa z neho priame spustenie plánu, a pridá sa ku kódu, ktorý je teraz na vrchole zásobníku a tým je kód pre cyklus. Hodnota na vrchole zásobníku zanorenia sa dekrementuje, na vrchole je teraz 1. Generuje sa rozdiel, identifikátoru registru predchádza 1 apostrof. Ten sa odstráni pri spustení plánu w1. Inštrukcia `END_WHILE` vyberie opäť vrchol zásobníku plánov a jeho obsah vloží do bázy plánov, ako plán w1. Zo zásobníka zanorení sa odstráni vrchol. Pri generovaní poslednej inštrukcie je na vrcholu zásobníku zámerov zámer plánu `main`, čiže zámer agenta. Register neobsahuje pred sebou žiaden apostrof.

8.3 Reprezentácia ALLL kódu

Predtým, ako prejdeme na samotný preklad jednotlivých inštrukcií, predstavíme akým spôsobom je výsledný kód v ALLL reprezentovaný. Výsledný kód v jazyku ALLL je reprezentovaný štruktúrou, ktorá má 3 položky:

- zámer agenta

- báza plánov
- báza znalostí

Tieto štruktúry sú postupne napĺňané výsledným kódom v jazyku ALLL vo fáze generovania cieľového kódu, presnejšie pri preklade jednotlivých inštrukcií 3AK. V nasledujúcej časti tejto kapitoly budú predstavené techniky, akým spôsobom sú jednotlivé inštrukcie implementované v ALLL. Rovnako ako aj spôsob akým vybrané inštrukcie plnia túto štruktúru.

8.4 Preklad inštrukcií 3AK do ALLL

V tejto sekcii predstavíme akým spôsobom sa prekladajú jednotlivé inštrukcie používaného 3AK do jazyka AHLL. Ak nebude spomenuté inak, všetky inštrukcie generujú svoj kód do zámeru, ktorý je na vrchole zásobníku zámerov. Rovnako aj premenná `level` odkazuje na vrchol zásobníku zanorenia pre registre.

Začneme inštrukciami `ADD`, `SUB`, `MULT` a `DIV`. Tieto sa prekladajú, tak, že sa načítajú oba operandy. V prípade, že niektorý z operandov je konštanta alebo je uložený v registri[1] tak sa akcie na načítanie operandu negenerujú, priamo sa vkladá hodnota alebo príslušný register. Následne sa generuje akcia danej matematickej operácie a výsledok sa uloží do premennej:

```
res <- a ADD b:    setRegister(1),op1=readVariable(a,1,level),
                   setRegister(2),op2=readVariable(a,2,level),
                   aritmethic(ADD,op1,op2),
                   storeVariable(res,1,level)
```

Rutina `readVariable` načíta premennú do registra, ak je to nutné (tj. ak nebola konštanta alebo parameter plánu). Tento kód na načítanie generuje do zámeru na vrchole zásobníku. Ako návratovú hodnotu vracia reťazec, ktorým môže byť buď identifikácia registru alebo konkrétna konštanta.

Ukážeme si to názorne aj na príklade prekladu inštrukcie $res \leftarrow a\, ADD\, 5$, a a res sú globálne premenné:

```
$1,#[a,_],#(lst,(hth,&1)),
#(ari,(p,&1,5)),
@<$3,-[res,_]>,+(res,&1)
```

Pri inštrukciách `ADD`, `SUB`, `MULT` a `DIV` bola k dispozícii príslušná služba platformy. V prípade inštrukcií `AND` a `OR` to tak nie je. Preto sme sa tieto inštrukcie rozhodli implementovať ako násobenie (`MULT`) resp. súčet (`ADD`). V prípade násobenia nám to zaistí chovanie ako logický súčin (`AND`). V prípade logického súčtu môže nastať problém pri sčítavaní záporného a kladného čísla s rovnakou absolútnou hodnotou. Ošetrenie tohto nedostatku by zabralo ďalšie inštrukcie naviac, preto sme sa rozhodli tento nedostatok ignorovať. Koniec koncov dodatočné služby platformy, napríklad pre logický súčet a súčin je možné doimplementovať.

Relačné inštrukcie, ktorými sú `MORE`, `MORE_EQ`, `LESS`, `LESS_EQ`, `EQUAL` a `NOT_EQUAL` sa implementujú obdobným spôsobom ako aritmetické inštrukcie. Oba operandy sa načítajú do registrov, ak to nie sú konštanty alebo parametre plánu. Avšak služby platformy na relačné operátory v prípade, že daná relácie neplatí spôsobujú pád podplánu.

[1] Bol parametrom daného plánu

Preto sú implementované tak, že pred volaním služby platformy na daný relačný vzťah sa do báze znalostí vloží značka. Následne sa metódou priameho spustenia plánu spustí plán, kde prvá je akcia služby platformy pre danú reláciu. Ak uspeje, odstráni značku a do báze znalostí vloží ako hodnotu výsledku 1 (True). Ak neuspeje značka v báze znalostí ostáva. Za ňou je spustený druhý plán, ktorý sa pokúsi túto značku odstrániť, ak uspeje, znamená to, že predchádzajúci plán neuspel a výsledok operácie je 0 (False).

```
res <- a EQUAL b :  setReg(1),op1=readVariable(a,1,level)
                    setReg(2),op2=readVariable(b,2,level),
                    addToBB(p),
                    imRun( testRel(EQUAL,op1,op2), removeFromBB(p),
                           addVarToBB(res,1) ),
                    imRun( removeFromBB(p), addVarToBB(res,0) )
```

Inštrukcia `NOT_EQUAL` nemá svoju obdobu medzi službami platformy. Jej implementácia je ale pomerne jednoduchá. Prvý plán zavolá službu platformy na porovnanie na zhodu (`EQUAL`) a ak uspeje ako výsledok vloží 0. Druhý plán naopak 1.

Inštrukcia `IS` je implementovaná postupnosťou načítania hodnoty premennej z bázy znalostí a jej následného uloženia. V prípade, že operandom je konštanta, tak sa priamo uloží jej hodnota, nie je potrebné nič načítavať:

```
a <- b :    setReg(1), op=readVariable(b,1,level),
            storeVariable(a,op,level)
```

Inštrukcia `UNARY_MINUS` sa implementuje rovnako ako aritmetická inštrukcia odčítania, kde prvým operandom je 0 a druhým je operand inštrukcie `UNARY_MINUS`.

Inštrukcia `UNARY_NEG` na logickú negáciu sa implementuje obdobne ako relačné inštrukcie:

```
a <- UNARY_NEG b:   setReg(1), op=readVariable(b,1,level),
                    addToBB(p),
                    imRun( testRel(EQUAL,op,0,level), removeFromBB(p),
                           storeVarToBB(a,1,level) ),
                    imRun( removeFromBB(p), storeVarToBB(a,0,level) )
```

Inštrukcia `TEST` má za úlohu overiť platnosť podmienky, ktorú obsahuje. V prípade, že táto podmienka nie je splnená vygenerované akcie majú za úlohu zlyhať, tým sa vynúti zlyhanie plánu, v ktorom sa táto inštrukcia vyskytuje. Implementácia inštrukcie zahrňuje, rovnako ako v predchádzajúcich inštrukciách s 2 operandmi ich načítanie do registrov a následné volanie služby platformy pre danú podmienku. Svoj výsledok inštrukcia nikam neukladá, jej úlohou je len uspieť alebo zlyhať:

```
TEST a type b : setReg(1), op1 = readVariable(a,1,level),
                setReg(2), op2 = readVariable(b,2,level),
                testRel(type,op1,op2)
```

Implementácia inštrukcie `SEND`:

```
SEND addr messg: setReg(1), op1 = readVariable(addr,1,level),
                 setReg(2), op2 = readVariable(messg,2,level),
                 genSend(op1,op2)
```

Inštrukcia `RECEIVE`:

```
res <- RECV addr: setReg(1), op = readVarialbe(addr,1,level),
                  genRecv(op),
                  if res ≠ ∅: storeVariable(res,1,level)
```

Implementácia inštrukcií `FI` a `OI` je najjednoduchšia. Nahradí sa len príslušnou akciou v ALLL:

```
FI : $FI
OI : $OI
```

Inštrukcie `HEAD` a `TAIL` sú implementovaná nasledovne. Prvým krokom je získanie n-tice v tvare (premenná, hodnota) z bázy znalostí. V rámci zníženia počtu akcií, získanie hodnoty a následné získanie hlavičky alebo tela zoznamu sme spojili do jednej akcie. Poslednou akciou je uloženie výsledku späť do bázy znalostí:

```
res <- HEAD a/ TAIL a: setReg(+1), #[alllQueryName(a),_],
                       if HEAD: #(lst,(hthh,genReg(1,level)) ),
                       if TAIL: #(lst,(htht,getReg(1,level)) ),
                       storeVariable(res,1,level)
```

Teraz prejdeme k inštrukciám, ktoré reprezentujú podmienky a cykly. Začneme inštrukciou `IF`. Táto inštrukcia zvýši číslo na vrchole zásobníku zanorenia pre registre, ako aj pridá nový zámer do zásobníku zámerov. Do tohto zámeru vloží akciu na test podmienky. Ak táto akcia zlyhá plán obsahujúci akcie podmienky zlyhá tiež.

```
IF a type b :  levelStack.top++
               intentionStack.push(new intention)
               setReg(1), op1 = readVariable(a,1,level) ,
               setReg(2), op2 = readVariable(b,2,level) ,
               testRel(type,op1,op2)
```

Inštrukcia `ELSE` vyberie vrchol zo zásobníku zámerov. Tento kód obsahuje akcie vetvy keď bola podmienka splnená. Na koniec tohto zámeru pridá odstránenie prvku z bázy znalostí, ktorým je značka, obsahujúca `ID` tejto inštrukcie. Značkou detekujeme či vetva *if* uspela alebo zlyhala. Ak zlyhala, vykonáva sa vetva *else*.

V tomto okamihu je na vrchole zásobníku zámerov zámer nadradeného plánu. K nemu sa pridá vloženie značky a priame spustenie plánu s akciami pre *true* vetvu.

Nakoniec sa pridá do zásobníku zámerov nový zámer, implementujúci akcie vetvy *else*. Prvou akciou tohto plánu je pokus o odstránenie značky z bázy znalostí:

```
ELSE ID :
    code = intentionStack.pop()
    code += removeFromBB(ID)
    intentionStack.top.extend(  addToBB(ID) + imRun(code) )
    intentionStack.push(removeFromBB(ID) )
```

Inštrukcia `END_IF` vyberie zámer z vrcholu zásobníku zámerov. Tento zámer implementuje akcie vetvy *if* alebo *else*. Nie je podstatné, ktorá z nich to je. V tomto prípade je na vrchole zásobníka zámerov kód nadradeného plánu. K týmto akciám sa pridajú akcie zámeru *if/else* formou priameho spustenia plánu. Nakoniec sa dekrementuje číslo na vrchole zásobníku zanorenia:

```
END_IF ID:
    code = intentionStack.pop()
    intentionStack.top.extend( imRun(code) )
    levelStack.top--
```

Inštrukcie `WHILE` a `FOREACH` pridajú k akciám na vrchole zásobníku plánov volanie plánu, ktorý vytvorí, čiže plán pre tento cyklus. Meno tohto plánu je vytvorené ako konkatenácia písmena w s `ID` plánu, v prípade inštrukcie `WHILE`. V prípade inštrukcie `FOREACH` ide o konkatenáciu písmena f s `ID` inštrukcie. Do tohto zásobníka následne pridá nový zámer, rovnako pridá novú položku na vrchol zásobníka zanorenia:

```
WHILE ID/ FOREACH ID:
    planName = w+ID / f+ID
    intentionStack.top.extend( planRun(planName) )
    intentionStack.push( new intention )
    levelStack.push(1)
```

Inštrukcie `END_WHILE` a `END_FOREACH` detekujú koniec inštrukcií pre cyklus s rovnakým `ID`. Inštrukcie vyberú zámer z vrcholu zásobníku zámerov. Ten sa rozšíri o volanie plánu pre tento cyklus a pridá tieto akcie do bázy plánov. Zo zásobníku zanorenia odstráni jeho vrchol.

```
END_WHILE ID / END_FOREACH ID:
    planName = "w"+ID/ "f"+ID
    code = intentionStack.pop()
    code += planRun(planName)
    planBase.add( new plan(planName, code) )
    levelStack.pop()
```

Nakoniec si ukážeme pseudokódy pre implementáciu volaní služieb platformy a spustení plánu. Pri volaní služieb platformy postupujeme tak, že najprv si načítame postupne jednotlivé operandy do registrov a následne vygenerujeme kód pre volanie služby platformy s týmito parametrami. V prípade, že `res`[2] nie je prázdny, uložíme výsledok služby platformy do tejto premennej:

```
res <- SERVICE name params:
        i = 0
        foreach p in params:
             i++; setReg(i); op[p] = readVariable(p,i,level)
        end
        generateService(name,op)
        if res ≠ ∅
             storeVariable(result,i,level)
```

[2]Operand na uloženie výsledku

Kód pre volanie plánu je obdobný:

```
CALL name params:
     i = 0
     foreach p in params:
        i++; setReg(i); op[p] = readVariable(p,i,level)
     end
     generateCall(name)
```

V tejto sekcii sme ukázali akým spôsobom sú prekladané jednotlivé inštrukcie 3AK do výsledného kódu v jazyku ALLL. Niektoré inštrukcie generujú priamo kód iné, prevažne riadiace, pracujú hlavne so spomínaným zásobníkom zámerov a zásobníkom zanorenia.

Problémom je generovanie plánov tak, aby boli rekurzívne spustiteľné. Síce ALLL poskytuje konštrukciu `$L`, ktorou by sme mohli rozlíšiť jednotlivé premenné rôznych volaní toho istého plánu ale predstavme si situáciu, že máme plán `rekurzia`, v jeho tele je cyklus obsahujúci ďalší cyklus:

```
plan rekurzia(){
  var a;
  while(...){
    while(...){
       a++;
    }
  }
}
```

Plán by pomenovával premenné v tvare `(planName,'$L,varName)`. Apostrof je potrebný z dôvodu, že ak by tam nebol, pri spustení by sa tento identifikátor nahradil úrovňou plánu, z ktorého bol plán `rekurzia` spustený. A práve tento problém nastáva pri cykloch, ktoré sa v ALLL takisto implementujú ako plány.

V prípade prvého (hlavného) cyklu by sme vedeli použiť `$L` bez apostrofov. Tým pádom by sa pri spustení nahradil úrovňou nadradeného plánu, ktorým je plán rekurzia. Premenná `a` by bola prístupná, mala by správne označenie zanorenia. To je síce správne, ale v prípade vnoreného cyklu by sa nahradilo `$L` rovnako identifikáciou nadradeného plánu, ktorým je v tomto prípade nadradený cyklus. Premenné deklarované priamo na úrovni plánu rekurzia by neboli prístupné (ich úroveň by sa nezhodovala), čo je v rozpore s sémantikou AHLL.

Ošetriť by sa to dalo dodatočnou informáciou v báze znalostí, ktorá by každému plánu určila aké má zanorenie. V prípade spustenia plánu by sa toto zanorenie inkrementovalo, resp. dekrementovalo v prípade ukončenia plánu. Toto riešenie by však znamenalo dodatočné akcie a s tým súvisiace zvýšenie veľkosti generovaného kódu.

Kapitola 9

Dosiahnuté výsledky

V tejto kapitole si ukážeme príklady kódov agentov, na ktorých zhodnotíme prínos optimalizácií. V práci [5] boli predstavené dva príklady agentov. Prvým bolo blikanie LED diódami a druhým výpočet faktoriálu. Rozhodli sme sa ich preložiť novou verziou prekladača a zhodnotiť prínos zvolených optimalizácií a samotnej novej verzie prekladača oproti verzii, ktorá bola predstavená v spomínanej práci. Snažili sme sa nemeniť zdrojový kód, pokiaľ to bolo možné.

Testovali sme tak, že sme agenta preložili novou verziou prekladača so zapnutými optimalizáciami. Následne sme ho preložili s novou verziou tak, že sme optimalizácie vypli. Agenta sme porovnali s veľkosťou agenta generovaného pôvodnou verziou prekladača.

V poslednom teste sme naprogramovali agenta na priechod sieťou. V tomto teste sme generovaný kód neporovnávali s pôvodnou verziou prekladača ale s ručne napísaným kódom v jazyku ALLL.

Ako metriku na veľkosť kódu sme použili počet akcií.

9.1 Blikanie LED diódami

Tento kód bol mierne upravený. Ako sme už spomínali nová verzia prekladača generuje cieľový kód v jazyku ALLL kompatibilný s verziou implementovanou v simulátore T-Mass. Tento simulátor neobsahuje službu na blikanie LED diódami, preto sme túto službu nahradili službou na výpis do terminálu.

Zdrojový kód je nasledovný:

```
service print(text){
  call ["tml",["pl",text] ];
  return null;
}
plan blik(time, led){
  print(led);
  print(time);
  print(led);
}
main(){
  while(1){  //nekonecny cyklus
    blik(300,"red");  //blik cervenou
    blik(300,"green");   //blik zelenou
```

```
    blik(300,"yellow");   //blik zltou
  }
}
```

Výsledky:

- nová verzia, zapnuté optimalizácie: 21 akcií
- nová verzia, vypnuté optimalizácie: 31 akcií
- pôvodná verzia : 65 akcií

Pri tomto kóde sa nám podarilo znížiť počet akcií takmer o 70% oproti pôvodnej verzii prekladača.

9.2 Výpočet faktoriálu

Tento kód nebolo potrebné nijak upravovať. Kód na výpočet faktoriálu a jeho odoslanie na ďalší uzol je nasledovný:

```
var c;   //globalna premenna kde bude vysledok
main(){
  c = 1;
  var a;
  a = 1;
  while(a <= 5){
   c= c *a;
   a = a+1;
  }
  send(2,c);
}
```

Výsledky:

- nová verzia, zapnuté optimalizácie: 55 akcií
- nová verzia, vypnuté optimalizácie: 55 akcií
- pôvodná verzia: 54 akcií

Kód faktoriálu sa nepodarilo takmer nijak optimalizovať. Avšak tento výsledok nie je vôbec prekvapujúci. Totiž výpočet faktoriálu je typický matematický výpočet, na ktorý nebola implementovaná takmer žiadna optimalizácia. Rovnako matematické výpočty, akou je napríklad výpočet faktoriálu, nie sú práve doménou skúmaných distribuovaných systémov a agentov vo WSN, implementovaných v jazyku AHLL/ALLL.

9.3 Priechod agenta sieťou

V tomto teste sme sa pokúsili implementovať agenta na priechod sieťou v jazyku AHLL, podľa článku [4]. Najprv si predstavíme služby platformy použité v spomínanom článku. Následne ukážeme implementáciu tohto algoritmu v AHLL. Upozorňujeme čitateľa, že tento kód reálne v simulátore T-Mass nefunguje. Dôvodom sú chýbajúce služby platformy pre implementáciu agenta na priechod sieťou, spomínané v článku [4]:

- `moveStop(addr)` spôsobí premiestenie agenta na uzol s adresou `addr`. Činnosť agenta na pôvodnom uzle sa ukončí.
- `neighbours()` umiestni do registra, uzly ktoré sú v dosahu. Uzly sú reprezentované vo forme n-tice v tvare `(NB,id,strength)`, kde `id` je číslo uzlu a `strength` je sila signálu od neho. Verzia tejto služby z článku ukladala tieto n-tice do bázy znalostí, pre naše účely je výhodnejšie, aby to ukladala do aktívneho registru.
- `getNode()` umiestni do aktuálneho registru číslo uzlu, na ktorom sa agent práve nachádza.
- `from()` umiestni do aktívneho registru číslu uzlu, z ktorého agent prišiel.
- `notVisited(addr)` je služba, ktorá zlyhá, keď agent na uzle `addr` už bol.

Ďalšie používané služby sú:

Predstavíme si jednotlivé služby platformy.

- `append(l1,l2)` pridá do zoznamu `l1` prvok `l2`
- `hth(list)` vráti druhú položku zo zoznamu
- `length(list)` vráti dĺžku zoznamu alebo 1 ak parametrom nie je zoznam ale jeden prvok.

Takže výsledný kód agenta pre priechod sieťou je nasledovný:

```
service moveStop(addr){
  call ["m",[addr,"s"]];
  return;
}

service neighbours(){
  call ["n"];
  return;
}

service getNode(){
  call ["t",["i"] ];
  return;
}

service from(){
  call ["t",["b","f"] ];
  return;
}

service notVisited(addr){
  call ["t",[addr,"v"] ];
  return null;  //it will fail, we will use it
}
```

```
service append(l1, l2){
  call ["lst",["a",l1,l2] ];
  return ;
}

service hth(list){
  call ["lst",["hth",list]];
  return;
}

service length(list){
  call ["lst",["l",list]];
  return;
}

var nodes, notOn,notJ ;
nodes = [];
notOn = 1 ; //aby sa spustila vnutorna slucka

plan notVis(i){
  notVisited(i); //ak sme boli plan pada
  notOn = i;    //ak padol nepriradime nic
}

main(){
   var a,i;
   while(1){
      notOn = 1;
      while(length(notOn) >0){  //ideme dopredu na dalsi uzol
      nodes = append(nodes,getNode() );
      a = neighbours();
      notOn = [];
      foreach i in a do {  //vyhladame dalsi volny
         notVis(hth(i) );
      }
      moveStop(notOn);
      }
   moveStop(from() ); //padne ak sme na 1. uzle
   }
}
```

Rozoberieme si jednotlivé plány. Plán `notVis(i)` uspeje, ak uzol `i` ešte nebol navštívený. Tento uzol dá do globálnej premennej `notOn`.

Teraz si rozoberieme slučku `main`. V nej vidíme 2 cykly `while`. Prvý je nekonečný cyklu, ktorý skončí až pri zlyhaní služby `from()`, keď sa agent objaví na východzom uzle. Vnorený cyklus pokračuje kým premenná `notOn` nie je prázdna. Uzol, na ktorom sa agent práve nachádza sa pridá do premennej `nodes`, následne vo cykle `foreach` sa vyhľadá ďalší uzol,

Agent	blik	faktorial	priechod sieťou
pôvodná verzia	65	54	N/A
nová, neoptimalizovaná	31	55	123
nová optimalizovaná	21	55	137
prínos oproti pôvodnej verzii	67%	-2%	N/A
prínos oproti vypnutým optimalizáciám	32%	0%	10%

Tabuľka 9.1: Vyhodnotenie prínosu optimalizácií

na ktorom ešte agent nebol. Ak sa takýto uzol nenájde premenná `notOn` obsahuje prázdny zoznam a tento cyklus končí (zhavaruje na pokuse o presunutie sa). Agent sa v tomto okamžiku vracia na predchádzajúci uzol.

Tento algoritmus sme prekladali len novou verziou prekladača. Porovnávali sme to s kódom uvedeným v článku [4]:

- nová verzia, zapnuté optimalizácie: 123 akcií
- nová verzia, vypnuté optimalizácie: 137 akcií
- ručne napísaný kód v ALLL: 31 akcií

V tomto teste sme porovnávali prekladačom generovaný kód oproti človekom generovanému. Podotýkame, že nová verzia prekladača aj s vypnutými optimalizáciami generuje kratší kód ako pôvodná verzia prekladača, ako sme to videli už pri teste blikaní LED diód. Je to z dôvodu, že časť optimalizácií robí aj samotný generátor cieľového kódu, ktorý z pochopiteľných dôvodov vypnúť nevieme.

Avšak ručne napísaný kód v ALLL je ešte stále výrazne lepší ako generovaný.

9.4 Vyhodnotenie a ďalšie možnosti optimalizácie

Nová verzia prekladača generuje v bežných agentných programoch zhruba o 20–50% kratší kód ako pôvodná verzia z práce [5], v niektorých prípadoch až takmer 70%. Naopak pri numericky zameraných programoch je skrátenie zatiaľ nulové, čo je dané hlavne tým, že na optimalizáciu matematiky sme neimplementovali takmer žiadnu metódu. Oproti vypnutým optimalizáciám je prínos zhruba 10–30%. Výsledky sme zhrnuli do tabuľky 9.1.

V porovnaní s ručne napísaným kódom je prekladač ešte stále pomerne neefektívny. Tento fakt ale nepovažujeme za výrazný neúspech práce vzhľadom aj na implementovaný počet optimalizácií. Iné prekladače programovacích jazykov, napríklad GNU C, obsahujú desiatky až stovky rôznych optimalizačných techník [3]. Implementovanie takéhoto množstva optimalizácií výrazne prevyšuje možnosti tejto práce.

Čo sa týka ďalšieho pokračovania prác, tak veľmi sa osvedčila zmena, ktorá odstránila kľúčové slovo `platform` a nahradila sa definíciou služby platformy a jej vlastnosťami (návrat nejakej hodnoty). Výrazne to napomohlo optimalizátoru a generátoru cieľového kódu generovať efektívny kód. Dôkazom je príklad blikania LED diód, kde sa podarilo vygenerovať veľmi efektívny kód približujúci sa ručne písanému kódu.

Toto ukazuje smer vývoja jazyka, kde by sa táto definícia služieb rozšírila o ďalšie položky, ako napríklad to či služba testuje nejakú reláciu a pri jej nesplnení zlyháva a pod. Takisto následné povolenie týchto služieb ako podmienok cyklov a podmienených príkazov.

Toto by bolo možné rozšíriť aj na operátory, keďže oni sú tiež implementované ako služby platformy. Operátory by bolo možné predefinovať na iné služby podľa potreby danej aplikácie.

Ako príklad priechodu agenta sieťou ukázal, bolo by výhodné okrem doteraz implementovaných cyklov povoliť aj cyklus, ktorý by mal podmienku ukončenia na ľubovoľnom mieste v tele cyklu. Táto zmena by bola sémanticky blízko jazyku ALLL.

Z optimalizácií by v súčasnom stave bolo výhodné doimplementovať alokáciu registrov a odstrániť tak výraznú nevýhodu generovania cieľového kódu z 3AK, ktorý vynucuje neustále načítavanie a ukladanie premenných.

Kapitola 10

Záver

V tejto práci sme sa zamerali na optimalizáciu prekladača jazyka AHLL. Pri riešení práce bola takisto mierne zmenená syntax jazyka AHLL a pridané nové vysokoúrovňové konštrukcie ako cyklus na priechod zoznamom, definovanie služieb platformy a pod.

Okrem syntaxe sa menila aj logická štruktúra prekladača. Novo zvolená štruktúra je podstatne vhodnejšia na dopĺňanie ďalších optimalizáčných techník. Rovnako umožňuje rozdeliť prácu na ich implementáciu medzi viacerých programátorov, čo je z budúceho pokračovania vývoja významným prínosom.

Podarilo sa integrovať do prekladača prvé optimalizačné techniky. Tieto umožnili v niektorých prípadoch prekonať pôvodnú verziu prekladača čo sa týka do veľkosti generovaného kódu. Rovnako bol predstavený príklad, kde zatiaľ nová verzia prekladača nedosiahla významnejšie úspechy. To naznačuje ďalšie smerovanie vývoja a pridávania optimalizácií.

Ďalší vývoj tohto prekladača by mal byť zameraný hlavne na implementáciu ďalšej množiny optimalizačných techník. Tie by mali byť smerované na zjednodušovanie matematických výrazov a hlavne na implementáciu alokácie registrov, ktorá v súčastnosti chýba. Jej implementovaním by sa prekladač mal výraznejšie priblížiť ručne písanému kódu.

Literatura

[1] Aho, A. V.; Sethi, R.; Ullman, J. D.: *Compilers: principles, techniques, and tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986, ISBN 0-201-10088-6.

[2] Allen, F. E.: Control flow analysis. *SIGPLAN Not.*, ročník 5, č. 7, Červenec 1970: s. 1–19, ISSN 0362-1340, doi:10.1145/390013.808479.
URL http://doi.acm.org/10.1145/390013.808479

[3] GNU compilers manual: Options That Control Optimization. [online], [cit. 19.5.2012].
URL <http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>

[4] Horáček, J.; Zbořil, F.: WSageNt: A case study. In *Proceedings of CSE 2010 International Scientific Conference on Computer Science and Engineering*, Volume 1, The University of Technology Košice, 2010, ISBN 978-80-8086-164-3, s. 258–264.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=9343

[5] Kalmár, R.: *Jazyk vyšší úrovně abstrakce pro programování mobilních inteligentních agentů*. bakalářska práce, FIT VUT v Brně, Brno, 2010.

[6] Muchnick, S. S.: *Advanced compiler design and implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997, ISBN 1-55860-320-4.

[7] Srikant, Y. N.; Shankar, P. (editoři): *The Compiler Design Handbook: Optimizations and Machine Code Generation*. CRC Press, 2002, ISBN 0-8493-1240-X.

[8] Wegman, M. N.; Zadeck, F. K.: Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, ročník 13, č. 2, Duben 1991: s. 181–210, ISSN 0164-0925, doi:10.1145/103135.103136.
URL http://doi.acm.org/10.1145/103135.103136

[9] Weisstein, E. W.: Tree. [online].
URL http://mathworld.wolfram.com/Tree.html

[10] Zbořil, F.: *Plánování a komunikace v multiagentních systémech*. dizertačná práca, Brno, FIT VUT v Brně, 2004.

[11] Zbořil, F.; Kočí, R.; Zbořil, V. F.; aj.: T-Mass v.2, State of the art. In *Second UKSIM European Symposium on Computer Modeling and Simulation*, IEEE Computer Society, 2008, ISBN 978-0-7695-3325-4, str. 6.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=8720

[12] Zbořil, F.; Spáčil, P.: Automata for Agent Low Level Language Interpretation. In *Proceedings of UKSim 2009*, IEEE Computer Society, 2009, ISBN 978-0-7695-3593-7, str. 6.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=8886

Dodatok A

EBNF jazyka AHLL

```
//========================================================================
//************************************************************************
// SYNTAKTICKE PRAVIDLA
//************************************************************************

// vstupny bod analyzy
prog:   (var_definition | var_init | plan
         | service_def | const_definition)*  main ;

service_def
: SERVICE ID '(' plan_params? ')' '{' call ret '}'
           -> ^(SERVICE ID ^(SERVICE_PARAMS plan_params?) call ret)
;

fragment call
: CALL zoznamService SC
                 -> ^(CALL zoznamService)
;

fragment ret
: RETURN SC          -> RETURN
|       RETURN NULL SC    ->  RETNULL
;

//definicia planov
plan :  PLAN ID '(' plan_params?  ')' blok
                 -> ^(PLAN ID ^(PLAN_PARAMS plan_params? ) blok )
;

//parametre planov
fragment plan_params
: ID (',' ID )* -> ID*
;
```

```
//pociatocny plan na zasobniku
main : MAIN '(' ')'blok -> ^( MAIN blok )
;


//prikaz
statement  : blok
| if_stat
| while_stat
| foreach_stat
| plan_call SC -> plan_call
| var_definition
| var_init
| const_definition
|  expression SC ->  ^( EXPR expression  )
// | plat_call
| send_call
|  receive_call  //odstranit...receive je vo vyraze
| FI SC -> ^(FI)
|  OI SC -> ^(OI)
| SC  ->
;

//blok prikazov
blok : '{' statement* '}' -> ^(BLOK statement*)
;

//podmienka
if_stat : IF '(' expression ')'  blok (ELSE blok)?
               -> ^(IF ^(COND expression) ^(THEN blok) ^(ELSE blok)? )
;

//cyklus while
while_stat
: WHILE '(' expression  ')' blok
               -> ^(WHILE ^(COND expression) ^(THEN blok) )
;


//cyklus foreach
foreach_stat : FOREACH ID 'in'  ID 'do' blok
                         -> ^(FOREACH  ID  ID       blok)
               |       FOREACH ID 'in'  zoznam 'do' blok
                         -> ^(FOREACH  ID  zoznam   blok)
        ;
```

```
var_definition
: VAR ID ( ',' ID)*  SC -> ^(VAR ID+ )
;

const_definition
: CONSTANT ID '=' konstant SC
                -> ^(CONSTANT ID konstant);


var_init : ID IS konstant SC
                          -> ^(INITIALIZATION ID konstant )
;

plat_call
: PLATFORM '(' STR  (',' expression)*  ')' ( '=>' ID  )?  SC
        ->  ^( PLATFORM ^(PLAT_PARAM STR expression*)
                        ^(PLAT_SAVE ID)?  )
;

send_call : SEND '(' expression ',' expression ')' SC
                        -> ^(SEND expression expression )
;

//contains to expression
receive_call : RECEIVE '(' expression?  ')'
                        ->  ^(RECEIVE  expression? )
;

plan_call   //or platform call
: ID '('  plan_call_params  ')'
                -> ^(PLAN_CALL ID  plan_call_params  )
;

fragment plan_call_params
: (expression  ( ',' expression )*)?
                -> ^(PLAN_CALL_PARAMS expression* )
;
//=====================================================================
//syntakticke pravidla pre vyrazy
expression
:  ID IS expression  ->  ^(IS ID expression)
| or_op ( OR ^ or_op  )*
;


or_op   //operacia logickeho suctu (or ... || )
```

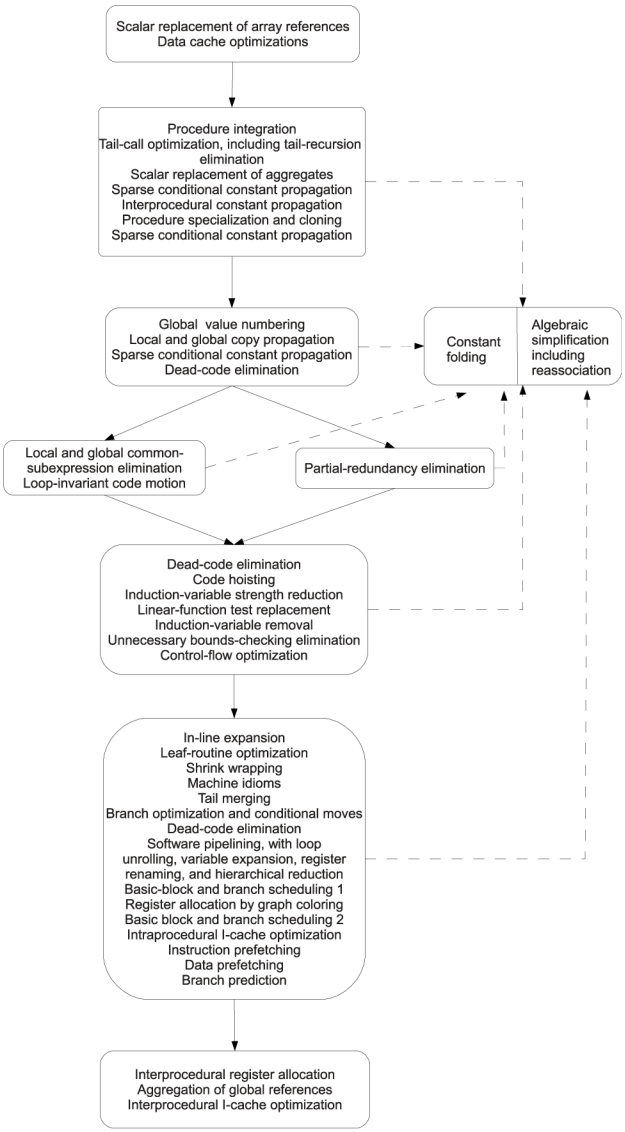
```
: and_op ( AND^ and_op)*
;


and_op   //operacia logickeho sucinu  &&
: equal_op ( (EQUAL| NOT_EQUAL) ^ equal_op)*
;

equal_op //operacia je_rovno je_nerovno   == | !=
: more_less_op ( (LESS | MORE | LESS_EQ | MORE_EQ )
                ^ more_less_op )*
;

more_less_op //operacia vacsie, mensie : < | > | <= | >=
: aditive_op ( ( PLUS | MINUS )^ aditive_op )*
;

aditive_op  //operacia suctu a rozdielu: + | -
: multiplicative_op ( ( MULT | DIV | MODULO )^
                multiplicative_op )*
;


multiplicative_op   //operacia nasobenia
:  ( (MINUS|PLUS|NEG)^ )?   ( atom|bracket)
|  plan_call
| receive_call
;


bracket   //vyraz v zatvorkach
: '(' expression ')' -> expression
;

atom      //tu by sa mal najst lexem (bud ID alebo INT)
: INC ID -> ^(PRE_INC ID)
| DEC ID -> ^(PRE_DEC ID)
| ID INC -> ^(POST_INC ID)
| ID DEC -> ^(POST_DEC ID)
| ID
| konstant
;

konstant : INT|STR|zoznam
;

konstant2 :  INT|STR
;
```

```
konstant3
: INT
|       STR
|       zoznamService
|       ID
;
zoznam :             '[' ']'           //prazdny zoznam
|             '[' (konstant)  (',' (konstant) )*  ']'
                       ->  ^(TUPLE  (konstant)*  )
;
zoznamService
: '[' ']'
| '[' (konstant3) (',' (konstant3))* ']'
              -> ^(TUPLE (konstant3)*)
;

//=========================================================================
//*************************************************************************
//                          LEXIKALNE PRAVIDLA
//*************************************************************************

//operatory:
IS : '='  ;
OR  : '||' ;
AND : '&&' ;
EQUAL : '==' ;
NOT_EQUAL
: '!=' ;
LESS : '<'  ;
LESS_EQ : '<=' ;
MORE : '>'  ;
MORE_EQ : '>=' ;
PLUS : '+'  ;
MINUS : '-'  ;
MULT : '*'  ;
DIV : '/'  ;
MODULO : '%'  ;
NEG : '!'  ;
INC : '++' ;
DEC : '--' ;
//CAR : 'car' ;
//CDR : 'cdr' ;

//KLUCOVE SLOVA
//podmieneny prikaz
IF : 'if';
ELSE : 'else';
```

```
WHILE : 'while' ;
FOR : 'for';
FOREACH : 'foreach';

VAR : 'var';
PLAN : 'plan' ;
MAIN  : 'main' ;
PLATFORM : 'platform';
SEND : 'send';
RECEIVE : 'receive';
SERVICE : 'service';
CALL : 'call';
RETURN : 'return';
NULL : 'null'|'NULL';
CONSTANT: 'const';
FI : 'FI' ;
OI : 'OI';

//THEN
//STAT : ;
//BLOK

// ostatne
ID  :   (('a'..'z'|'A'..'Z') ('a'..'z'|'A'..'Z'|'0'..'9')*) ;
INT :   '0'..'9'+ ;

STR : '"' STR_frag '"'   //->  ^(STR STR_frag)
;
fragment STR_frag
:  ('a'..'z'|'A'..'Z'|'0'..'9')*
;
NEWLINE:'\r'? '\n'
WS  :   (' '|'\t')+
COMMENT : '//'(~('\r'|'\n')) * '\r'? '\n'
SC  :  ';' ;
```

Dodatok B

Poradie jednolivých optimalizácií podľa [6, s. 326]



Dodatok C

Obsah CD

README	README súbor s návodom na inštaláciu
text/src/	zdrojová podoba textu práce
text/projekt.pdf	text diplomovej práce
src/	zdrojové kódy prekladača
ahll_v2.jar	binárna verzia prekladača
examples/	príklady agentov uvedené v texte práce