VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

TRANSFORMACE KÓDŮ APLIKOVANÁ PRO DEKOMPILACI

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE IVAN ŠOMLO
AUTHOR

BRNO 2007

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

TRANSFORMACE KÓDŮ APLIKOVANÁ PRO DEKOMPILACI

CODE TRANSFORMATION - DECOMPILATION

BAKALÁŘSKÁ PRÁCE

BACHELOR´S THESIS

AUTOR PRÁCE IVAN ŠOMLO
AUTHOR

VEDOUCÍ PRÁCE Prof. RNDr. ALEXANDER MEDUNA, CSc.
SUPERVISOR

BRNO 2007

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2006/2007

Zadání bakalářské práce

Řešitel: **Šomlo Ivan**
Obor: Informační technologie
Téma: **Transformace kódů aplikovaná pro dekompilaci**
Kategorie: Překladače

Pokyny:

1. Seznamte se s teorii kódů a metodami jejich analýzy a transformace.
2. Navrhněte vlastní metody analýzy a transformace kódů.
3. Studujte užití metod navržených v předchozím bodě. Zaměřte se na dekompilaci.
4. Zhodnoťte dosažené výsledky a diskutujte další možný vývoj projektu.

Literatura:

1. Tremblay, J. P. and Sorenson, P. G.: The Theory and Practice of Compiler Writing, McGraw-Hill, New York, 1985

Při obhajobě semestrální části projektu je požadováno:

- Body 1 a 2.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním paměťovém médiu (disketa, CD-ROM), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Meduna Alexander, prof. RNDr., CSc.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2006
Datum odevzdání: 15. května 2007

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Ing. Jaroslav Zendulka, CSc.
vedoucí ústavu

LICENČNÍ SMLOUVA
POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO

uzavřená mezi smluvními stranami

1. Pan

Jméno a příjmení: **Ivan Šomlo**
Id studenta: 83958
Bytem: Szokolyiho 3, 936 01 Šahy
Narozen: 27. 09. 1984, Šahy
(dále jen "autor")

a

2. Vysoké učení technické v Brně

Fakulta informačních technologií
se sídlem Božetěchova 2/1, 612 66 Brno, IČO 00216305
jejímž jménem jedná na základě písemného pověření děkanem fakulty:

.....
(dále jen "nabyvatel")

Článek 1
Specifikace školního díla

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):
   bakalářská práce

Název VŠKP: Transformace kódů aplikovaná pro dekompilaci
Vedoucí/školitel VŠKP: Meduna Alexander, prof. RNDr., CSc.
Ústav: Ústav informačních systémů
Datum obhajoby VŠKP: .....

VŠKP odevzdal autor nabyvateli v:

tištěné formě počet exemplářů: 1
elektronické formě počet exemplářů: 2 (1 ve skladu dokumentů, 1 na CD)

2. Autor prohlašuje, že vytvořil samostatnou vlastní tvůrčí činností dílo shora popsané a specifikované. Autor dále prohlašuje, že při zpracovávání díla se sám nedostal do rozporu s autorským zákonem a předpisy souvisejícími a že je dílo dílem původním.
3. Dílo je chráněno jako dílo dle autorského zákona v platném znění.
4. Autor potvrzuje, že listinná a elektronická verze díla je identická.

Článek 2
Udělení licenčního oprávnění

1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizování výpisů, opisů a rozmnoženin.
2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.
3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti:
   - ☒ ihned po uzavření této smlouvy
   - ☐ 1 rok po uzavření této smlouvy
   - ☐ 3 roky po uzavření této smlouvy
   - ☐ 5 let po uzavření této smlouvy
   - ☐ 10 let po uzavření této smlouvy

   (z důvodu utajení v něm obsažených informací)
4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č. 111/ 1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

Článek 3
Závěrečná ustanovení

1. Smlouva je sepsána ve třech vyhotoveních s platností originálu, přičemž po jednom vyhotovení obdrží autor a nabyvatel, další vyhotovení je vloženo do VŠKP.
2. Vztahy mezi smluvními stranami vzniklé a neupravené touto smlouvou se řídí autorským zákonem, občanským zákoníkem, vysokoškolským zákonem, zákonem o archivnictví, v platném znění a popř. dalšími právními předpisy.
3. Licenční smlouva byla uzavřena na základě svobodné a pravé vůle smluvních stran, s plným porozuměním jejímu textu i důsledkům, nikoliv v tísni a za nápadně nevýhodných podmínek.
4. Licenční smlouva nabývá platnosti a účinnosti dnem jejího podpisu oběma smluvními stranami.

V Brně dne: .....

.....	Šomla .....
Nabyvatel	Autor

Abstrakt

Táto práca popisuje proces a jednotlivé časti dekompilácie. Dekompilácia je opačný proces ku kompilácii. Jej úlohou je transformácia vstupného programu, obyčajne v strojovom kóde, do funkčne ekvivalentnej podoby vo vyššom programovacom jazyku. Práca popisuje jednotlivé fázy dekompilácie: disasembler, generátor vnútornej reprezentácie programu, analýza základných blokov, optimalizácia, analýza dátových a riadiacich tokov a generovanie výsledného kódu.

Kľúčové slová

dekompilácia, strojový kód, disasembler, trojadresný kód, analýza dátových tokov, analýza riadiacich tokov

Abstract

This thesis discusses the decompilation process and its phases. Decompilation is the reverse process of compilation. Its goal is to transform input program, usually in machine code, into functionally equivalent form in some high level language. The thesis describes disassembling, intermediate code generation, basic block analysis, optimization, data flow analysis, code flow analysis and output code generation.

Keywords

decompilation, machine code, disassembler, three address code, data flow analysis, control flow analysis

Citácia

Ivan Šomlo: Transformace kódů aplikovaná pro dekompilaci, bakalářská práce, Brno, FIT VUT v Brně, 2007

Transformácia kódov aplikovaná pre dekompiláciu

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením Prof. RNDr. Alexandra Medunu, CSc.
Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

……
Ivan Šomlo
11.5.2007

Poďakovanie

Na tomto mieste by som sa chcel poďakovať Prof. RNDr. Alexandrovi Medunovi, CSc. za podporu pri písaní tejto prace.

© Ivan Šomlo, 2007.
Táto práca vznikla ako školské dielo na Vysokom učení technickom v Brne, Fakulte informačných technológií. Práca je chránená autorským zákonom a jej použitie bez udelenia oprávnenia autorom je nezákonné, s výnimkou zákonom definovaných prípadov.

Obsah

Obsah 1
Úvod 2
1 Dekompilácia 3
1.1 Proces dekompilácie 3
2 Jazyk symbolických inštrukcií 5
3 Medzikód 10
3.1 Trojadresný kód 10
3.2 Pásky 11
3.3 Generovanie trojadresného kódu 12
4 Trasovanie základných blokov 14
4.1 Základné bloky 14
4.2 Návestia 15
4.3 Propagácia konštánt v bloku 16
4.4 Graf riadiacich tokov 18
4.5 Eliminácia pások 20
4.5.1 Statická metóda 20
4.5.2 Dynamická metóda 21
5 Viazanie základných blokov 22
6 Optimalizácia 25
7 Analýza dátových a riadiacich tokov 26
7.1 Analýza dátových tokov 26
7.2 Analýza riadiacich tokov 28
8 Abstraktný syntaktický strom, generovanie cieľového kódu 29
9 Implementácia 30
10 Záver 31
Literatúra 32

Úvod

Procesory v počítačových systémoch vykonávajú programy písané v špeciálnom jazyku čísel – v strojovom kóde. Vo všeobecnosti platí, že čo je jednoduché pre procesor, je zložité pre človeka a naopak. Preto v histórii informatiky vzniklo niekoľko programovacích jazykov, ktoré sú bližšie nášmu spôsobu myslenia. Tieto jazyky sa volajú vyššie programovacie jazyky. Konkrétny program, konkrétny algoritmus je pre nás ľudí jednoduchšie napísať vo vyššom programovacom jazyku. Ten ale nie je možné ihneď vykonať (spustiť). Musí sa transformovať do podoby, ktorej rozumie procesor. Tento proces sa volá kompilácia.

V niektorých situáciách je nutné tento proces obrátiť a pretransformovať program z jazyka, ktorému rozumie procesor do podoby, ktorá je ľahko pochopiteľná pre človeka. Realizáciou tohto procesu sa zaoberá napríklad dekompilácia. Jedným zo základných problémov dekompilácie je strata veľkej časti informácie pri procese kompilácie. Okrem toho má strojový kód úplne inú filozofiu ako vyššie programovacie jazyky, preto nie je všeobecne možné spätne získať originálny zdrojový kód, ktorý slúžil ako vstup kompilácie. V niektorých prípadoch dokonca nie je možné pomocou dekompilácie získať ani funkčne ekvivalentný kód.

Dekompilácia je súčasťou širšieho oboru nazývaného reverzné inžinierstvo. Reverzné inžinierstvo [4] je proces analýzy systému za účelom:

1. identifikácie komponentov systému a ich vzťahov
2. vytvorenia modelu systému v inej forme, alebo na vyššom stupni abstrakcie

Táto práca opisuje možné spôsoby dekompilácie a činnosti jej častí. Nejde o úplne vyčerpávajúci prehľad, ale o základné princípy jednotlivých častí. Primárne sa venuje úvodným fázam dekompilácie.

Kapitola 1 obsahuje stručný úvod do dekompilácie. Kapitola 2 sa venuje strojovému kódu a jazyku symbolických inštrukcií. V kapitole 3 sa popisuje forma reprezentácie programu – medzikód. Niekoľko základných transformácií nad medzikódom je popísaných v kapitole 4. Zjednodušeniu medzikódu sa venuje kapitola 5. V kapitolách 6, 7 a 8 sú principiálne popísané ďalšie fázy dekompilácie: optimalizácia, analýza dátových a riadiacich tokov a generovanie výsledného kódu. V kapitole 9 je popísana praktická časť tejto práce. Kapitola 10 obsahuje záver.

1 Dekompilácia

Dekompilácia je proces transformácie vstupného programu, ktorý je väčšinou v strojovom kóde alebo v jazyku symbolických inštrukcií (ďalej len JSI), na funkčne ekvivalentný program na vyššej forme abstrakcie (obyčajne vyšší programovací jazyk). Ide o opačný proces ku kompilácii. Nie je všeobecne automatizovateľný a v niektorých prípadoch vyžaduje zásah človeka. Počítačový program, ktorý vykonáva dekompiláciu sa nazýva dekompilátor.

Je vždy ľahšie postupovať z vyšších foriem abstrakcie do nižších ako naopak. Je jednoduchšie napísať program, ktorý vykreslí jeden z desiatich druhov lokomotív, ako program, ktorý podľa obrázka zistí, ktorá lokomotíva je na obrázku nakreslená.

Z teoretického hľadiska je ľubovoľný algoritmus zapísaný v strojovom kóde taktiež možné napísať v ľubovoľnom vyššom jazyku. Podmienkou je, aby teoretický model daného jazyka bol minimálne tak silný ako teoretický model daného strojového kódu. Napríklad konkrétny počítačový systém je možné modelovať lineárne ohraničeným automatom. Stačí dokázať, že tento teoretický model je možné simulovať vo vyššom programovacom jazyku; napríklad v jazyku C by sa k tomu použili prostriedky ako pole (model pásky) a premenné (pozícia hlavy, číslo aktuálneho stavu).

Dôvody pre dekompiláciu môžu byť rôzne: strata zdrojových kódov, snaha o získanie použitého algoritmu, portovanie programu na inú architektúru [2], odhaľovanie nebezpečných kódov (vírusy). Zároveň môže znalosť princípov dekompilácie poslúžiť pri písaní bezpečnejšieho softvéru.

Dekompilácia nie je vždy legálna. Niektoré programy a ich licenčné podmienky zakazujú používanie disasemblerov a dekompilátorov. Príkladom takéhoto softvéru je Microsoft Office 2003, v licenčnej zmluve ktorého stojí:

„*OMEZENÍ TÝKAJÍCÍ SE ZPĚTNÉ ANALÝZY, DEKOMPILACE A PŘEVODU ZE STROJOVÉHO KÓDU. Nejste oprávněni provádět zpětnou analýzu, dekompilaci nebo převod ze strojového kódu Softwaru, s výjimkou a v rozsahu takové aktivity, která je výslovně povolena rozhodným právem bez ohledu na toto omezení.*“.

1.1 Proces dekompilácie

Dekompilátor sa skladá (podobne ako kompilátor) z niekoľkých častí, ktoré nie sú rovnaké ako pri kompilátoroch, ale je možné badať určitú podobnosť s obrátenou postupnosťou častí kompilátora. Rozdelenie na časti a ich funkcia závisia na spôsobe dekompilácie. Napríklad dekompilácia popísaná v [3] hľadá niektoré bežne používané vzory vo vstupnom programe, predpokladá štandardné volacie konvencie procedúr a pod. Táto práca popisuje všeobecnejší spôsob, ktorý nie je založený na podobných predpokladoch.

Dekompilácia popísaná v tejto práci obsahuje nasledovné časti:

1. Prevod strojového kódu do jazyka symbolických inštrukcií
2. Prevod na medzikód
3. Rozdelenie medzikódu do základných blokov a trasovanie blokov
4. Viazanie základných blokov
5. Optimalizácia
6. Analýza dátových a riadiacich tokov
7. Prevod na abstraktný syntaktický strom
8. Generovanie cieľového kódu

V nasledujúcich kapitolách sú tieto jednotlivé časti (fázy) popísané. Dôraz sa kladie na prvé 4 časti.

2 Jazyk symbolických inštrukcií

Programy sú obyčajne uložené vo forme strojového kódu. Strojový kód je možné definovať ako binárny jazyk nad abecedou $\sum = \{0,1\}$. Je to jazyk, s ktorým pracuje procesor. Konkrétny program je potom reťazec nad binárnou abecedou Σ. Strojový kód obsahuje inštrukcie a dáta. Inštrukcie je možné chápať ako také podreťazce kódu, ktoré môžu byť (a sú) vykonané procesorom.

Existujú dve základné počítačové koncepcie:

1. Harvardská koncepcia – inštrukcie a dáta sú uložené oddelene
2. von Neumannova koncepcia – inštrukcie a dáta nie sú oddelene (je rozšírenejšia)

Keďže strojový kód je pre človeka nečitateľný, používa sa jeho textová reprezentácia – JSI. V tejto podobe sú jednotlivé inštrukcie uložené v textovej podobe. Dáta môžu byť reprezentované viacerými spôsobmi. Úsek programu v strojovom kóde môže vyzerať napríklad takto:

```
0xEB 0x04 0x19 0x84 0x27
```

a jeho možná reprezentácia v JSI (ide o architektúru x86 [5]):

```
jmp lab        ; 0xEB 0x04
db 0x19
db 0x84
lab:
db 0x27
```

Dva bajty `0xEB 0x04` sa nahradili ich textovou reprezentáciou `jmp lab` (v skutočnosti `0xEB` je kód pre inštrukciu `jmp` a `0x04` je relatívna cieľová adresa). Rovnaký program v strojovom kóde však môže byť v JSI reprezentovaný aj v podobe:

```
db 0xEB
add al, 0x19   ; 0x04 0x19
dw 0x2784
```

Z toho vyplýva že neexistuje jednoznačné priradenie z množiny programov v strojovom kóde do množiny programov v JSI.

Všetky inštrukcie v JSI majú priradený jednoznačný číselný kód. Transformáciou dvoch predchádzajúcich programov v JSI do strojového kódu vznikne identický reťazec v strojovom kóde. Preto existuje jednoznačné priradenie z množiny všetkých programov v JSI do množiny všetkých programov v strojovom kóde, ale nie naopak.

Z toho vyplýva nasledujúca veta. Nech $\mathrm{asm}(x)$ je funkcia, ktorá pre program x v JSI vráti jeho podobu v strojovom kóde a nech $\mathrm{disasm}(x)$ je funkcia ktorá vráti niektorú z nekonečne mnoho reprezentácii v JSI programu x v strojovom kóde. Potom platí:

$$\mathrm{asm}(\mathrm{disasm}(x)) = x$$

a

$$\mathrm{disasm}(\mathrm{asm}(x)) \neq x$$

Teda platí, že nie je možné získať z programu v strojovom kóde originálny zdrojový kód v JSI (ak nejaký bol), ale vždy je možné k nemu nájsť funkčne ekvivalentný kód v JSI (a dokonca automaticky) .

Počítačový program, ktorý vykonáva činnosť funkcie asm sa volá Asembler. Naopak program, ktorý vykonáva činnosť funkcie disasm je Disasembler. Najväčší problém, s ktorým sa disasembler stretáva vyplýva z vlastnosti von Neumannovskej architektúry, v ktorej neexistuje oddelenie inštrukcií od dát. Ako je ukázané vyššie, úsek v strojovom kóde môže v skutočnosti byť inštrukciou, alebo môže ísť o dáta. Je tiež otázne, či úsek programu v strojovom kóde, ktorý bol v skutočnosti určený na vykonávanie, ale nikdy sa nevykoná, je inštrukcia alebo dáta.

Veľkým problémom je samomodifikovateľný kód, ktorý pristupuje k svojim inštrukciám ako k dátam. Z toho vyplýva filozofický problém, čí daný úsek sú inštrukcie alebo dáta. Kompilátory taktiež často vkladajú do výsledného programu úseky bajtov, ktoré nie sú určené na vykonávanie, ani na prístup k nim, ale slúžia na zarovnávanie.

Algoritmom na oddelenie inštrukcií od dát môže byť algoritmus 2.1.

Algoritmus 2.1: Oddelenie inštrukcií od dát

---

1 Vstup: program S v strojovom kóde.

2 Výstup: pre slovo (bajt) na každej pozícii x v S rozhodni či sa $S(x)$ vykoná.

Keby sme chceli dostať úplne správny výsledok, museli by sme činnosť programu simulovať. Jedine simulácia môže rozhodnúť, ktoré časti programu budú vykonané. Stav systému, v ktorom by sa vykonávanie programu nachádzalo na adrese x je možné označiť za koncový stav. Všetky ostatné existujúce stavy by boli nekoncové. Problém oddelenia inštrukcií od dát sa potom redukuje na otázku „zastaví daný program ?“. Ide o známy nerozhodnuteľný problém zastavenia (aj keď Minsky ukázal, že na strojoch s konečným počtom stavov a s konečnou páskou je rozhodnuteľný, ale prakticky len dosť ťažko). Z toho vyplýva nemožnosť všeobecného plne automatického prevodu programu zo strojového kódu do JSI.

Tento problém sa prakticky ukazuje pri inštrukciách skoku. Pri inštrukcii `jump 0x1984` je možné predpokladať, že na adrese `0x1984` sa bude nachádzať inštrukcia. Pri inštrukcii `jump eax`

sa ale inštrukcia bude nachádzať na adrese, ktorá je uložená v registri `eax`, kde môže byť prakticky ľubovoľné číslo.

Podobný je ďalší príklad, ktorý ukazuje situáciu, kde nie je jasné, čo dané číslo vyjadruje:

```
push eax
push 0
push 0x00411091
push 3
call 0x004183B4
```

Inštrukcia `push 0x00411091` umiestni na zásobník číslo `0x00411091`. Je otázne, či dané číslo je obyčajná konštanta, ukazovateľ na dáta alebo adresa inštrukcie. Tento problém nie je možné jednoducho vyriešiť.

Pre úplnosť sú tu uvedené dva základné algoritmy použiteľné v disasembleroch. Algoritmus 2.2 počíta funkciu $\mathrm{DecodeInstr}(x)$. Funkcia sa pokúsi preložiť inštrukciu na adrese x. Keďže strojový kód je prefixový kód [1], žiadna inštrukcia nie je prefixom inej inštrukcie. Preto sa na danej adrese môže nachádzať najviac jedna inštrukcia. Väčšina architektúr má desiatky až stovky inštrukcií. Funkcia ktorá priradí každej inštrukcii v textovej podobe jej binárnu podobu (a naopak) je obyčajne definovaná pomocou tabuľky. Napríklad takáto tabuľka inštrukčnej sady x86 sa rozprestiera na stovkách strán [5].

Ďalej sú uvedené dva algoritmy [7], ktoré počítajú množinu adries $Insset$, na ktorých sa pravdepodobne nachádzajú inštrukcie. Algoritmus 2.3 je jednoduchší, ale dostatočne nevyužíva kontextové informácie v programe. Napríklad po načítaní inštrukcie `jc 0x2709` (skoč ak $C = 1$, inak pokračuj) je pravdepodobné, že na adrese `0x2709` sa bude nachádzať inštrukcia. Túto skutočnosť reflektuje až algoritmus 2.4. V praxi sa používa kombinácia týchto dvoch prístupov.

Dekompilátor by mal predpokladať vstup v JSI, alebo by aspoň mal obsahovať vstavaný disasembler. Je totiž lepšie, v prípade neúspechu alebo nemožnosti dekompilácie, preložiť program aspoň do JSI. V tejto práci sa uvažujú architektúry x86 [5] a HC08 [6]. V nasledujúcich kapitolách sa predpokladá vstup v JSI.

Algoritmus 2.2: Funkcia $\mathrm{DecodeInstr}(x)$

```
Vstup: x je adresa v programe S v strojovom kóde, InsRef je tabuľka, ktorá priraďuje jednej
       inštrukcii v strojovom kóde jej podobu v JSI.
Výstup: inštrukcia, pokiaľ je na adrese x inštrukcia, ináč φ.
Postup:
s := S(x)                    {do s ulož podreťazec S na pozícii x dĺžky najkratšej inštrukcie}
if s = ε then
    DecodeInstr(x) := φ
else
    loop := true
    while loop do
        if y = InsRef(s) then {inštrukcia existuje}
            DecodeInstr(x) := y
            loop := false
        {ak s je prefixom niektorej inštrukcie}
        else if ∃z, y = InsRef(s.z) and S ≠ ε then
            a.S := S        {odstráň prefix dĺžky jeden z S }
            s := s.a        {pridaj ho na koniec s }
        else
            DecodeInstr(x) := φ
            loop := false
        end
    end
end
```

Algoritmus 2.3: Linear Sweep

```
1   Vstup: program S v strojovom kóde.
2   Výstup: množina adries Insset , na ktorých sa nachádzajú inštrukcie.
3   Postup:
4   x := 0
5   while x <| S | do                    {pre každú adresu v S}
6       instr := DecodeInstr(x)          {pokúsi sa dekódovať inštrukciu na adrese x}
7       if instr ≠ ϕ then                {na adrese x sa môže nachádzať inštrukcia}
8               Insset := Insset ∪ {x}
9               x := x+ | instr |        {zvýš aktuálnu adresu o veľkosť identifikovanej inštrukcie}
10      else
11              x := x + 1
12      end
13  end
```

Algoritmus 2.4: Recursive Traversal

```
1   Vstup: program S v strojovom kóde, entry je vstupný bod programu.
2   Výstup: množina adries Insset , na ktorých sa nachádzajú inštrukcie.
3   Postup:
4   Task := {entry}
5   while | Task |> 0 do
6       be x ∈ Task
7       Task := Task − {x}
8       instr := DecodeInstr(x)
9       if instr ≠ ϕ then
10              Insset := Insset ∪ {x}
11              nech A je množina všetkých možných následníkov instr
12              Task := Task ∪ (A − Insset)
13      end
14  end
```

3 Medzikód

V kompilátoroch sa väčšinou vstupný program neprekladá priamo do výstupného programu, ale prekladá sa najprv do medzikódu. Medzikód (intermediárny kód) je vnútorná reprezentácia programu. V kompilátoroch používajúcich túto stratégiu predná časť kompilátora prekladá vstupný program do medzikódu a zadná časť ho prekladá do cieľového kódu. Samotné jadro kompilátora je potom nezávislé na zdrojovom a cieľovom jazyku. Taktiež je možné medzikód optimalizovať nezávisle od cieľovej architektúry.

Medzikód môže mať rôzne formy. Môže sa vyskytovať vo forme stromu, zásobníkového kódu alebo trojadresného kódu. Každá forma má svoje výhody a nevýhody. Zásobníkový kód je napríklad vhodné použiť v prípade prekladača generujúceho kód pre zásobníkové architektúry. Trojadresný kód je použiteľný v prípade nutnosti rozsiahlejších optimalizácii.

V dekompilátoroch je tiež výhodné vstupný program prekladať do medzikódu. Vďaka tomu sú na architektúre závislé len predné časti dekompilátora.

3.1 Trojadresný kód

Trojadresný kód je postupnosť príkazov základného tvaru

```
c := a op b
```

kde `c`, `a`, `b` sú mená, konštanty a dočasné premenné a `op` je operátor. Operandy `a`, `b` sú zdrojové operandy. Operand `c` je cieľový operand. Operácia `op` môže byť napríklad niektorá z aritmetických operácii. Zložitejšie výrazy sa v trojadresnom kóde vyjadrujú pomocou dočasných premenných. Napríklad výraz $y := (m1 + m2) * (m3 - m4)$ kompilátor rozloží do tvaru

```
u := m1 + m2
v := m3 - m4
y := u * v
```

kde `u` a `v` sú dočasné premenné vytvorené kompilátorom.

Systém trojadresných príkazov by mal spĺňať nasledujúce vlastnosti:

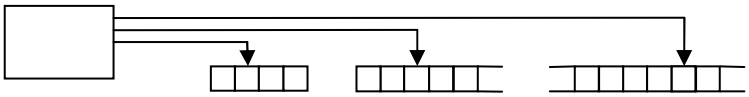
1. Jednotnosť – mal by byť nezávislý na architektúre. V daných kompilátoroch alebo dekompilátoroch by sa mal vždy používať len jeden spoločný medzikód.
2. Jednoduchosť – mal by obsahovať čo najmenej príkazov.
3. Ortogonálnosť – jeden príkaz by nemal byť reprezentovateľný sériou zvyšných príkazov. Napríklad operácia `+=` toto porušuje, pretože sa dá vyjadriť pomocou operácie `+`.

V kompilátoroch môže existovať niekoľko typov príkazov v trojadresnom kóde. Väčšinou sa ako operandy používajú premenné, prípadne dočasné premenné vytvorené kompilátorom. V dekompilátoroch je situácia mierne odlišná. V dekompilátoroch sa ako zdrojový jazyk používa jazyk, ktorý neobsahuje klasický systém premenných. Namiesto toho sa tam vyskytujú registre a pod.

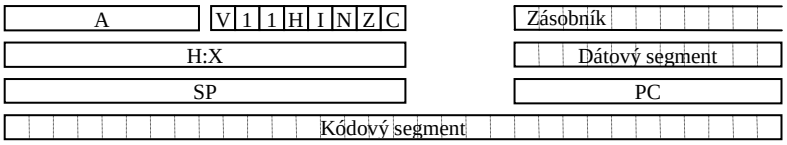
3.2 Pásky

Páska je pojem, ktorý sa používa v teórii automatov. Spolu s aktuálnym stavom konečného stavového riadenia vytvára aktuálny stav celého automatu. Inými slovami, pásky slúžia na uchovávanie informácie. Turingov stroj, ako najsilnejší automat môže mať ľubovoľný počet pások.

Na obrázku 3.1 je znázornený Turingov stroj. Na obrázku 3.2 je zobrazené prostredie programu na architektúre HC08. Je vidieť určitú podobu medzi týmito obrázkami. Programy v strojovom kóde nepoužívajú filozofiu premenných. Pracujú so štruktúrami ako sú registre, registre v zásobníkovej štruktúre, programové zásobníky, rôzne dátové segmenty, segmenty s konštantnými údajmi atď. Tieto štruktúry pripomínajú pásky a preto sú týmto pojmom v tejto práci označované.



Obrázok 3.1: Turingov stroj s páskami



Obrázok 3.2 Programové prostredie HC08

Absencia premenných a prítomnosť pások v prostredí strojového kódu je dôvodom na pridanie špeciálnych príkazov do štandardnej sady trojadresných kódov. Ide o príkazy, vďaka ktorým môže trojadresný kód v dekompilátore pracovať okrem premenných aj s páskami. To „priblíži“ trojadresný kód ku predným častiam dekompilátora, konkrétne ku generátoru trojadresného kódu. Generátor trojadresného kódu bude potom oslobodený od transformovania štruktúr používaných v danej architektúre na premenné. Tým sa jeho činnosť zjednoduší.

Pri generovaní medzikódu sa programové štruktúry nachádzajúce sa v zdrojovej architektúre nahradia páskami. Každá štruktúra sa nahradí jednou z troch možných typov pások: páskou konštantnej dĺžky, páskou neohraničenou z jednej strany, alebo páskou neohraničenou z oboch strán. Napríklad v prípade akumulátora pôjde o ohraničenú pásku, zatiaľ čo v prípade zásobníku pôjde o neohraničenú pásku.

3.3 Generovanie trojadresného kódu

Generovanie trojadresného kódu má na starosti predná časť dekompilátora – Generátor trojadresného kódu. Je to časť silne závislá na vstupnej architektúre. Generátor trojadresného kódu vygeneruje pre každú inštrukciu v JSI sériu príkazov v trojadresnom kóde. Ide o slepé generovanie. Na rozdiel od kompilátorov je možné ho použiť, pretože optimalizácia nasleduje až za touto fázou (v kompilátoroch optimalizácia predchádza fázu generovania medzikódu).

Základné typy príkazov trojadresného kódu použiteľného v dekompilátore sú:

1. Aritmetické príkazy
   - `c := a op b` kde `op` je binárna aritmetická operácia, napr. +, -, *, /
   - `c := op a` kde `op` je unárna aritmetická operácia, napr. -, ~
2. Propagačný príkaz
   - `c := a` uloženie premennej `a` do premennej `c`
3. Ukazovatele
   - `c := * a` dereferencovanie ukazovateľa
   - `c := & a` adresa premennej
4. Páskové príkazy
   - `T [p] := totape a` priradenie premennej `a` do pásky `T` do polička `p`
   - `c := fromtape T [p]` priradenie obsahu pásky `T` na políčku `p` do premennej `c`
   - `c := getrtapesize T` dĺžka pravej strany pásky `T`
   - `c := getltapesize T` dĺžka ľavej strany pásky `T`
   - `T := setrtapesize a` nastavenie dĺžky pravej strany pásky `T`
   - `T := setltapesize a` nastavenie dĺžky ľavej strany pásky `T`
5. Symboly
   - `label S` definovanie návestia `S`, kde `S` je symbol
   - `c := symbol S` uloženie symbolu `S` do premennej `c`
6. Literál
   - `c := literal I` uloženie literálu `I` do premennej `c`
7. Skoky
   - `jump c` skok na pozíciu uloženú v premennej `c`
   - `cjump a c` skok na pozíciu uloženú v premennej `c` ak platí `a`
8. Procedúry
   - `call c` volanie procedúry `c`
   - `endproc` koniec procedúry
9. Špeciálne príkazy

Pri generovaní trojadresného kódu dochádza k rozkladu inštrukcie v JSI do série príkazov v trojadresnom kóde. Tieto príkazy v podstate definujú elementárne kroky, ktoré procesor pri vykonaní danej inštrukcie v JSI vykonáva. Nasleduje príklad generovania príkazov pre inštrukciu `add #opr` na architektúre HC08. Táto inštrukcia pripočíta ku akumulátoru `A` okamžitú hodnotu `opr`. Zároveň ovplyvní aj niektoré bity v príznakovom registri `CCR`. Vygenerované príkazy sú:

`op1 := opr`	{prvý operand je okamžitý operand `#opr` z inštrukcie}
`op2 := fromtape`[1] `A`	{druhý operand je uložený v registri `A`}
`aa := op1 + op2`	{výsledok ich sčítania}
`vv := op1 +`$_V$ `op2`	{príznak pretečenia}
`hh := op1 +`$_H$ `op2`	{príznak prenosu z 3. bitu}
`nn := is_negative aa`	{príznak záporného výsledku}
`zz := is_zero aa`	{príznak nulového výsledku}
`cc := op1 +`$_C$ `op2`	{príznak prenosu z najvyššieho bitu}
`A := totape`[2] `aa`	{všetky medzivýsledky sa uložia späť do registrov}
`V := totape vv`	
`H := totape hh`	
`N := totape nn`	
`Z := totape zz`	
`C := totape cc`	

Význam neštandardných operátorov je:

- `a +`$_V$ `b` $\Leftrightarrow$ (pri sčítaní došlo k pretečeniu)
- `a +`$_C$ `b` $\Leftrightarrow$ (pri sčítaní došlo k prenosu z najvyššieho bitu)
- `a +`$_H$ `b` $\Leftrightarrow$ (pri sčítaní došlo k prenosu z 3. bitu)
- `is_negative a` $\Leftrightarrow$ $(a < 0)$
- `is_zero a` $\Leftrightarrow$ $(a = 0)$

Rovnakým spôsobom je nutné preložiť každú inštrukciu. Výsledkom je neštruktúrovaná sekvencia príkazov – trojadresný kód. Je dôležité aby sa v medzikóde každá premenná vyskytovala vo forme cieľového operandu maximálne raz. Dodržaním tejto podmienky sa zjednoduší analýza dátových tokov.

---

[1] Pre jednoduchosť je príkaz (`c := fromtape T [0]`) zapísaný ako (`c := fromtape T`)

[2] Pre jednoduchosť je príkaz (T [0] `:= totape a`) zapísaný ako (T `:= totape a`)

4 Trasovanie základných blokov

Fáza trasovania základných blokov predpokladá ako vstup lineárny zoznam príkazov v trojadresnom kóde (ďalej len kóde). Táto fáza:

1. Rozdelí kód do základných blokov.
2. V každom základnom bloku vykoná propagáciu konštánt.
3. V každom základnom bloku nájde úvodné návestie a určí následníkov blokov.
4. Eliminuje pásky, nahradí ich dočasnými premennými.

Na konci bude kód rozdelený do základných blokov. Každý blok bude tvorený sekvenciou príkazov, pričom jednotlivé príkazy budú pracovať už len s dočasnými premennými, symbolmi a literálmi. Pre každý blok bude dostupná informácia o jeho predchodcoch a následníkoch. Trasovaním jednotlivých blokov sa určí ich vplyv na systém.

4.1 Základné bloky

Základný blok (ďalej len blok) je sekvencia príkazov alebo kódov, ktorá má jeden vstupný a jeden výstupný bod. Pokiaľ má byť vykonaný ľubovoľný príkaz bloku, sú vykonané vždy všetky príkazy v bloku, a to v poradí, v akom sa v bloku nachádzajú.

Pri dekompilácii je nutné program rozdeliť na bloky. Je to podmienka úspešnej optimalizácie a identifikovania riadiacich štruktúr v programe. Nad blokmi sa potom vytvorí graf riadiacich tokov.

Rozdelenie neštruktúrovaného programu na bloky sa deje prostredníctvom množiny vedúcich príkazov (lídrov). Líder je prvý príkaz bloku. Platí:

1. Prvý príkaz programu je líder.
2. Každé návestie je líder.
3. Príkaz nasledujúci za príkazom skoku (ale aj volania a ukončenia procedúry, atd.) je líder.

Príklad 4.1: Nasledujúci program v trojadresnom kóde

```
1.  a := 0
2.  label Lab01
3.  b := a + m
4.  c := b < n
5.  s := symbol Lab01
6.  cjump c s
7.  d := b
```

má troch lídrov na riadku 1, 2 a 7. Výsledkom sú teda 3 základne bloky.

```
B1 :  a := 0
B2 :  label Lab01
      b := a + m
      c := b < n
      s := symbol Lab01
      cjump c s
B3 :  d := b
```

Pri delení programu na bloky je nutné pre každý blok B nájsť jeho následníka v pôvodnom programe $\text{NextBlock}(B)$. Ak sú jednotlivé bloky číslované vzostupne od 1 po n, kde n je počet blokov, potom

$\text{NextBlock}(b) = b + 1$ pre $b < n$ a

$\text{NextBlock}(b) = \phi$ pre $b = n$.

Pre program v príklade 4.1 je

$$\text{NextBlock}(1) = 2,\ \text{NextBlock}(2) = 3 \text{ a } \text{NextBlock}(3) = \phi.$$

4.2 Návestia

Z definície lídrov, ako prvých príkazov blokov, vyplýva, že prvý príkaz bloku je príkaz návestia, alebo blok tento príkaz neobsahuje vôbec.

Nech $\text{Symbol}(S)$ je funkcia, ktorá symbolu S priradí blok, v ktorom je symbol S použitý v príkaze návestia. Algoritmus 4.1 slúži na výpočet funkcie $\text{Symbol}(S)$.

Algoritmus 4.1: $\text{Symbol}(S)$

1 Vstup: Základné bloky B.

2 Výstup: $\text{Symbol}(S)$ pre každý symbol S.

3 Postup:

4 **for** $\forall$ základné bloky B **do**

5 **if** $B \neq \phi$ **and** $B(0) = \text{label}\, S$ **then**

6 $\text{Symbol}(S) := B$

7 **end**

8 **end**

Pre program v príklade 4.1 je

Symbol(Lab01) = 2

4.3 Propagácia konštánt v bloku

V strojovom kóde sa za normálnych okolností po vykonaní inštrukcie pokračuje inštrukciou, ktorá leží v programe hneď za vykonávanou inštrukciou. Pre zmenu tohto chovania sa používajú inštrukcie skoku. Podľa toho kde je uložená cieľová adresa skoku sa skoky delia na priame a nepriame. Pri priamych skokoch je cieľová adresa zakódovaná priamo v samotnej inštrukcii. Pri nepriamych skokoch je v inštrukcii uložený popis, kde je možné cieľovú adresu nájsť.

Tento stav je nutné v trojadresnom kóde reflektovať. To je dôvod, prečo príkazy skoku z kapitoly 3.3 obsahovali ako cieľ skoku premennú a nie priamo symbol. Tým sú v medzikóde podporované aj nepriame skoky. Potom sa jednoduchý priamy skok na zadaný symbol rozprestiera na dvoch príkazoch, napríklad

```
c := symbol S
jump c
```

V ďalších častiach dekompilátora je však potrebné presne vedieť kam sa môže vykonávanie programu z daného bloku dostať. Preto je potrebné poznať možné ciele príkazov skoku. To je jeden z dôvodov prečo je nutné uskutočniť propagáciu konštánt v bloku.

Propagácia konštánt je optimalizačná technika, ktorá zjednodušuje konštantné výrazy v dobe prekladu. Algoritmus 4.2 sa pre každý príkaz bloku pokúsi identifikovať známe konštantné premenné. Je vhodné ho použiť v kombinácii s nasledujúcimi algoritmami.

Algoritmus 4.2: Propagácia konštánt v bloku B

```
1   Vstup: Základný blok B .
2   Výstup: v je premenná, v.val je v každom kroku jej hodnota, alebo ϕ ak hodnota nie je známa.
3   Postup:
4   for ∀ premenné v do
5       v.val := ϕ
6   end
7   for ∀ príkazy C bloku B od začiatku do konca do
8       if C = (c := literal I) then
9               c.val := I
10      else if C = (c := symbol S) then
11              c.val := S
12      else if C = (c := a op b) then
13              if a.val ≠ ϕ and b.val ≠ ϕ then
14                      c.val := a.val op b.val
15              else
16                      c.val := ϕ
17              end
18      else if C = (c := op a) then
19              if a.val ≠ ϕ then
20                      c.val := op a.val
21              else
22                      c.val := ϕ
23              end
24      else if C = (c := a) then
25              c.val := a.val
26      end
27  end
```

4.4 Graf riadiacich tokov

Graf riadiacich tokov nad základnými blokmi je orientovaný graf $G = (V, E)$, kde V je množina základných blokov a E je množina usporiadaných dvojíc (h,t), $h,t \in V$. Platí $(h,t) \in E \Leftrightarrow$ riadenie programu môže prejsť z bloku h do bloku t.

Pre určenie grafu riadiacich tokov je nutné poznať následníkov každého základného bloku. Na to je možné použiť algoritmus 4.3 množiny $\text{Successors}(B)$. Pre základný blok B množina $\text{Successors}(B)$ obsahuje usporiadané dvojice (d,c), kde $d \in V$ je blok, do ktorého môže prejsť riadenie programu z bloku B ak platí podmienka c. Špeciálne dvojica $(d,\phi) \in \text{Successors}(B)$ má význam „za predpokladu, že neplatia ostatné podmienky v $\text{Successors}(B)$“.

Algoritmus 4.3: $\text{Successors}(B)$

```
Vstup: Základný blok B.
Výstup: Successors(B) pre B.
Postup:
Successors(B) = φ,  fall := true
for ∀ príkazy C bloku B od začiatku do konca do
    if C = (jump x) then
            if x.val = symbol S
                    Successors(B) := Successors(B) ∪ (Symbol(S), φ)
                    fall := false
            else error
            end
    else if C = (cjump p x) then
            if x.val = symbol S
                    Successors(B) := Successors(B) ∪ (Symbol(S), p)
            else error
            end
    else if C = (endproc) then fall := false
end
if fall = true and NextBlock(B) ≠ φ then
    Successors(B) := Successors(B) ∪ (NextBlock(B), φ)
end
```

Graf riadiacich tokov sa potom zostaví podľa algoritmu 4.4.

Algoritmus 4.4: Graf riadiacich tokov

1 Vstup: množina základných blokov V .

2 Výstup: Graf riadiacich tokov $G = (V, E)$.

3 Postup:

4 **for** $\forall$ bloky $B \in V$ **do**

5 **for** $\forall (d, c) \in$ Successors(B) **do**

6 $E := E \cup (B, d)$

7 **end**

8 **end**

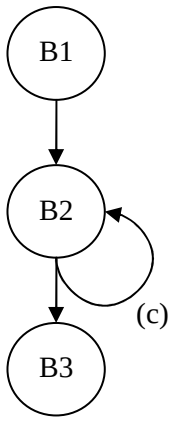
Pre základné bloky z príkladu 4.1 platí:

$$\text{Successors}(1) = \{(2, \phi)\}$$

$$\text{Successors}(2) = \{(2, c), (3, \phi)\}$$

$$\text{Successors}(3) = \phi$$

a výsledný graf riadiacich tokov je $G = (\{B1, B2, B3\}, \{(B1, B2), (B2, B2), (B2, B3)\})$. Jeho grafická reprezentácia je na obrázku 4.1.



Obrázok 4.1: Graf riadiacich tokov

4.5 Eliminácia pások

Pásky (popísané v kapitole 3.2) zjednodušujú činnosť prednej časti dekompilátora. Táto kapitola popisuje možné spôsoby nahradenia pások premennými. Pre jednoduchosť sa táto kapitola venuje len páskam s konštantnou dĺžkou veľkosti jedného políčka. Takéto pásky väčšinou modelujú samostatný register, príznakový bit a pod. Napríklad na architektúre HC08 môže isť o akumulátor `A`. Príkazy medzikódu pracujúce s takýmto typom pásky sú

```
T [p] := totape a
c := fromtape T [p]
```

kde `p = 0`, preto ďalej v texte sa `[p]` vynecháva.

Sú dva základne spôsoby eliminácie pások: statická a dynamická.

4.5.1 Statická metóda

Princíp tejto metódy je jednoduchý. Ku každej páske `T` sa priradí nová premenná `t`. Potom bude páska `T` reprezentovaná premennou `t` po celý čas. Algoritmus statickej metódy je nasledovný:

1. Každý príkaz (`T := totape a`) nahraď príkazom (`t := a`).
2. Každý príkaz (`c := fromtape T`) nahraď príkazom (`c := t`).

Statická metóda je jednoduchá a je použiteľná vo vnútri základného bloku a tiež medzi základnými blokmi navzájom. Jej nevýhodou je, že v určitých prípadoch vytvára falošné závislosti medzi logicky nezávislými informáciami. Tento typ pásky napríklad modeluje akumulátor. Akumulátor je miesto, kde sa vykonávajú aritmetické operácie s údajmi. Dve logicky nezávislé premenné ktoré sa po sebe umiestnia do akumulátora sú statickou metódou spojené do jednej premennej. Výsledkom je potom kód, v ktorom sa medzi niekoľkými premennými opakovane presúvajú dáta. Takýto kód je ťažko čitateľný.

Príklad 4.2: Statická metóda pretransformuje program

```
T := totape a
b := fromtape T
c := literal 89
e := b + c
T := totape e
```

do podoby:

```
t := a
b := t
c := literal 89
e := b + c
t := e
```

4.5.2 Dynamická metóda

Dynamická metóda priraďuje premennú k páske `T` dynamicky. Potom páska `T` slúži len ako úložný priestor pre premenné, ale sama premennou nie je. V každom momente je v páske `T` uložená jedna premenná, alebo je páska prázdna. Tento spôsob je možné použiť len v rámci základného bloku. Výhodou tejto metódy je že rešpektuje pôvodné toky dát vo vstupnom programe. Algoritmus dynamickej metódy je nasledovný:

1. Na začiatku bloku umiestni do pásky `T` novú premennú `t`, `T := {t}`.
2. Postupne prechádzaj príkazy od prvého až po posledný.
3. Príkaz (`T := totape a`) odstráň, do pásky `T` umiestni `a`, `T := {a}`.
4. Príkaz (`c := fromtape T`) nahraď príkazom (`c := x`), kde `x` ∈ `T`.
5. Na konci bloku zostane v páske `T` posledná uložená premenná.

Dynamická metóda pretransformuje program z príkladu 4.2 do podoby:

```
T = {t}          {na začiatku sa priradí páske T nová premenná t}
b := a
c := literal 89
e := b + c
T = {e}          {na konci obsahuje páska T premennú e}
```

5 Viazanie základných blokov

Trasovanie blokov sa vykonávalo samostatne pre každý blok. Trasovaním každého bloku vzniklo niekoľko dočasných premenných. Takýto základný blok má sadu vstupných premenných, ktoré sú v bloku použité a sadu výstupných premenných, ktoré blok poskytuje jeho následníkom. Vo všeobecnosti majú všetky základné bloky rovnaké rozhrania. Napríklad všetky základné bloky používajú vstup uložený v páske (v registri) `T` avšak každý blok predpokladá na vstupe inú premennú uloženú v tejto páske. Úlohou tejto časti dekompilátora je vzájomne naviazať vstupné a výstupné rozhrania základných blokov. Podmienkou je znalosť grafu riadiacich tokov.

Príklad 5.1: Nech je program zložený z dvoch základných blokov, pričom druhý blok je následníkom prvého.

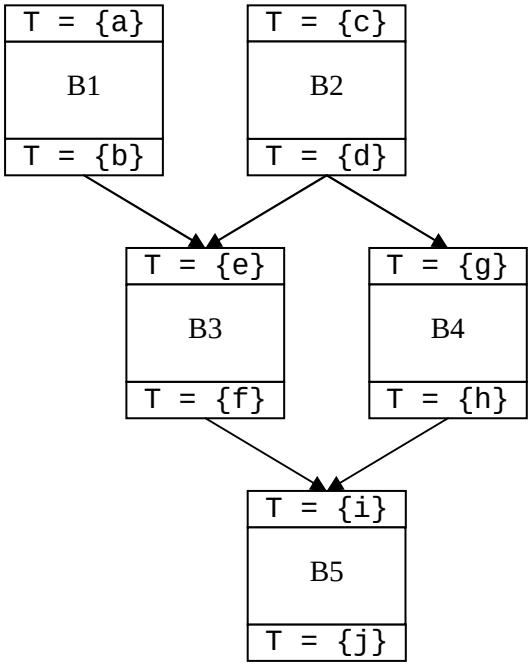
B1:	T = {`s`}	Na vstupe bloku 1 je v páske `T` premenná `s`.
	`i := literal 3`	Blok 1 k tejto premennej pripočíta číslo 3.
	`a := s + i`	Vznikne nová premenná `a`.
	T = {`a`}	Blok 1 ju poskytne na výstupe v páske `T`.
B2:	T = {`x`}	Na vstupe bloku 2 je v páske `T` premenná `x`.
	`j := literal 4`	Blok túto premennú vynásobí číslom 4.
	`y := x * j`	Vznikne nová premenná `y`.
	T = {`y`}	Blok 2 ju poskytne na výstupe v páske `T`.

Tieto dva základné bloky sa jednoducho naviažu umiestnením príkazu (`x := a`) medzi dané dva bloky.

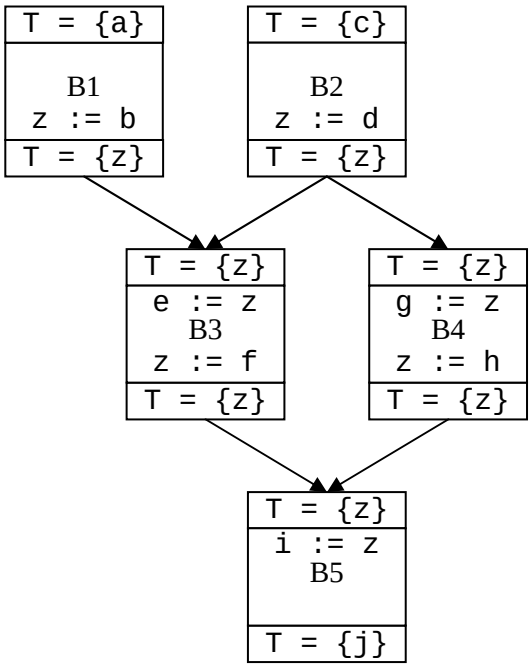
V skutočnosti môže byť graf riadiacich tokov ľubovoľne zložitý. Na obrázku 5.1 je príklad takého grafu. Ako je vidno, naviazanie nie je triviálne, pretože každý blok má na jeho rozhraniach iné premenné.

Tento problém sa dá jednoducho vyriešiť zavedením globálneho priradenia novej dočasnej premennej `t` páske `T`. Na obrázku 5.2 je rovnaký graf s globálnou premennou `z`. Do jednotlivých blokov bolo nutné pridať niekoľko propagačných príkazov, aby na rozhraniach blokov boli v jednotlivých páskach rovnaké premenné.

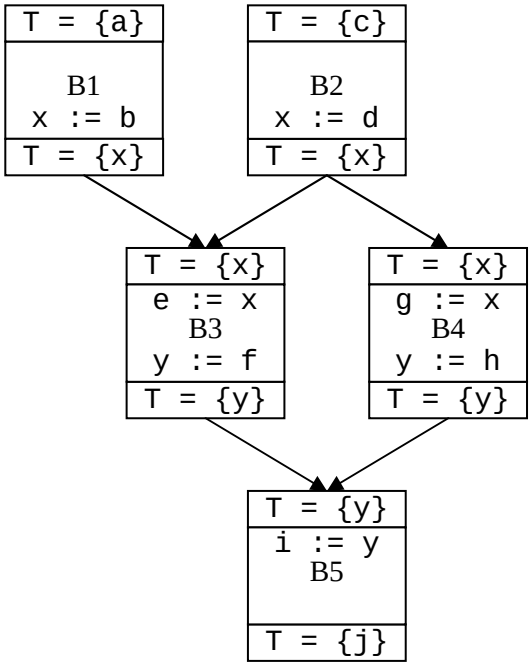
Problémom tohto prístupu je práve vytvorenie globálnych premenných. Je možné tento postup vylepšiť tak, že namiesto vytvorenia jednej globálnej premennej sa vytvorí viac lokálnych premenných a každej hrane grafu bude priradená jedna z týchto premenných. V grafe na obrázku 5.1 je najvhodnejšie priradiť jednu premennú (`x`) hranám (B1,B3), (B2,B3), (B2,B4) a druhú premennú (`y`) hranám (B3,B5), (B4,B5). Situácia je zobrazená na obrázku 5.3.



Obrázok 5.1: Graf riadiacich tokov



Obrázok 5.2: Graf riadiacich tokov s globálnou premennou z



Obrázok 5.3: Graf riadiacich tokov s premennými x, y

Jednotlivé hrany grafu sa takto rozdelia do skupín – kontrolných bodov. Kontrolný bod je množina hrán K_x grafu $G = (V, E)$, kde $K_x \subset E$ a platí že:

$(h,t) \in K_x \wedge (h,a) \in E \Rightarrow (h,a) \in K_x$ a

$(h,t) \in K_x \wedge (a,t) \in E \Rightarrow (a,t) \in K_x$

Z vlastností kontrolných bodov vyplýva, že K_1, K_2, K_3, …, K_n sú disjunktné množiny a zároveň $K_1 \cup K_2 \cup K_3 \cup \ldots \cup K_n = E$.

Použitím kontrolných bodov sa dosiahne lepšia lokalizácia premenných.

6 Optimalizácia

Optimalizácia je také modifikovanie systému, ktoré zlepší niektorú sledovanú vlastnosť systému. V kompilátoroch sa používa optimalizácia na zvýšenie rýchlosti, zmenšenie výsledného kódu a pod. Dochádza k ďalšej strate informácie, ktorá nie je potrebná vo výslednom kóde.

Existuje niekoľko optimalizačných techník. Tak ako bolo napísané v úvode tejto práce, väčšinou platí, že ak sa niektorá z nich osvedčí pri kompilácii, pri dekompilácii je to presne naopak. V kompilátoroch sa používajú zložité optimalizačné techniky. Zároveň ale platí, že v dekompilátoroch nie je potrebné tieto techniky používať, pretože „to za nás urobil kompilátor“. Program v JSI (x86)

```
mov eax, 2          ; u = 2 + 5
add eax, 5
mov ebx, 2          ; v = 2 + 5
add ebx, 5
add eax, ebx        ; u+v
```

síce láka na použitie optimalizátora pri dekompilácii, ale to nie je potrebné. Dôvody, pre ktoré tento kód nebol optimalizovaný sú irelevantné. Dôležité je že z nejakého dôvodu bol tento kód napísaný takto a nie ako 2 * (2 + 5).

Napriek tomu je nutné použiť v dekompilátore niektoré optimalizačné techniky. Program v JSI

```
add #4
add #10
```

vytvorí približne takýto kód:

```
r1 := a + b
z1 := is_zero r1
Z := totape z1        {do pásky Z sa uloží výsledok porovnania (a + b = 0)}
⋮
r2 := c + d
z2 := is_zero r2
Z := totape z2        {do pásky Z sa uloží výsledok porovnania (c + d = 0)}
```

Príkaz (`Z := totape z1`) nemá v konečnom dôsledku žiadny efekt, pretože príznak nulového výsledku po prvom sčítaní sa prepíše druhým sčítaním. Z toho ďalej vyplýva, že (`z1 := is_zero r1`) nebude mať tiež žiadny efekt, pretože `z1` sa už efektívne nikde nepoužije. Tento nepriaznivý efekt je nutné vyriešiť globálnymi optimalizáciami.

Ďalej je nutné globálne optimalizovať niektoré propagačné príkazy.

7 Analýza dátových a riadiacich tokov

Táto kapitola sa zaoberá spôsobom vytvorenia diagramu dátových tokov v blokoch a transformáciou príkazov skoku a podmieneného skoku na štruktúry na vyššej forme abstrakcie.

7.1 Analýza dátových tokov

Vzhľadom na pokročilé optimalizačné techniky používané pri kompilácii nemá veľký zmysel sa snažiť získať dekompiláciou program čo i len trochu podobný vstupnému programu kompilácie. Je skôr lepšie vytvoriť jeho najčitateľnejšiu a najjednoduchšiu možnú podobu.

Preto je základný blok vhodné reprezentovať vo výstupnom programe dekompilácie sériou n priraďovacích príkazov v_1, v_2, …, v_n, kde $\mathrm{v}_i = \mathrm{expr}_i(\mathrm{v}_1, \mathrm{v}_2, \ldots, \mathrm{v}_{i-1}, \mathrm{p}_1, \mathrm{p}_2, \ldots, \mathrm{p}_m);$, kde expr_i je ľubovoľný výraz nad skôr definovanými premennými v_1, v_2, …, v_{n-1} a vstupnými premennými bloku p_1, p_2, …, p_m a pre všetky $i \neq j$ sú premenné v_i a v_j rôzne.

To znamená, že každá premenná sa v rámci bloku definuje maximálne raz. Ak má táto premenná v bloku len jedno použitie (a nie je výstupnou premennou bloku), potom je možné celú pravú stranu priraďovacieho príkazu dosadiť na miesto použitia a tento príkaz odstrániť. To sa dá dosiahnuť vytvorením diagramu dátových tokov pre každý blok, čo je cieľom analýzy dátových tokov.

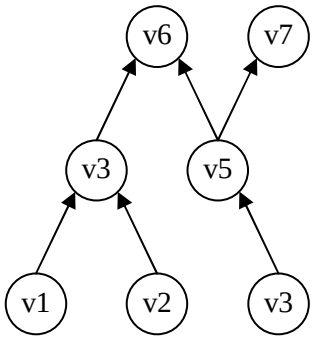
Diagram dátových tokov všeobecne vyjadruje tok dát systémom. V danom prípade je diagram dátových tokov orientovaný graf, ktorého uzly sú premenné medzikódu a orientovaná hrana (h,t) vyjadruje závislosť premennej (uzlu grafu) t na premennej h.

Pokiaľ sa v bloku každá premenná definuje maximálne raz (čo je princíp, ktorý rešpektuje dynamická eliminácia pások a ktorý je vhodné dodržiavať aj pri transformovaní programu z JSI do medzikódu) potom diagram dátových tokov pre daný blok je orientovaným acyklickým grafom. Vhodným rozdelením tohto grafu na podgrafy so stromovou štruktúrou vzniká usporiadaná množina stromov, ktoré sa pri generovaní výsledného programu prepíšu do série priraďovacích príkazov. Na to je možné použiť algoritmus postupného odstraňovania uzlov, ktoré nemajú žiadnych predchodcov. Potom z niektorých premenných v tomto grafe vzniknú vo výstupnom programe skutočné premenné a niektoré bude možné dosadiť na miesto ich použitia, čím vznikne zložitejší výraz.

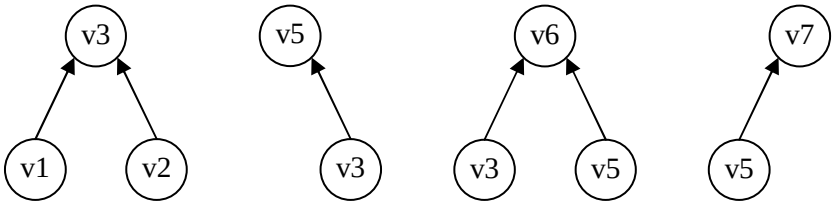
Príklad 7.1: Nech

```
v3 := v1 + v2
v5 := - v4
v7 := ~ v5
v6 := v3 * v5
```

je základný blok v trojadresnom kóde. Jeho graf dátových tokov je znázornený na obrázku 7.1. Rozdelením na podstromy vznikne usporiadaná množina stromov na obrázku 7.2. Z toho vyplýva, že premenné budú definované v poradí `v3`, `v5`, `v6`, `v7`. Ostatné premenné budú súčasťou výrazov definujúcich tieto 4 premenné.



Obrázok 7.1: Diagram dátových tokov



Obrázok 7.2: Usporiadaná množina stromov

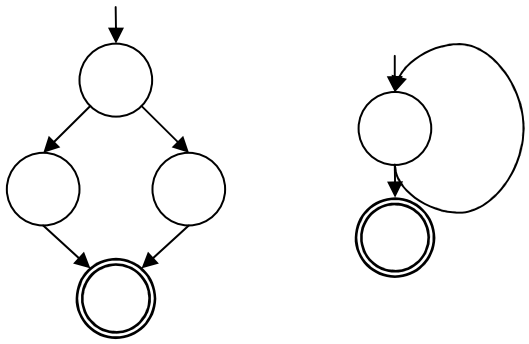
7.2 Analýza riadiacich tokov

Program v strojovom kóde je lineárna štruktúra. Na zmenu aktuálneho miesta vykonávania sa používajú napríklad inštrukcie skoku. V niektorých vyšších jazykoch majú tieto inštrukcie svoj ekvivalent, napríklad príkaz `goto` v jazyku C. Program používajúci príkazy skoku je neprehľadný a porušuje všetky zásady dobrého programovania.

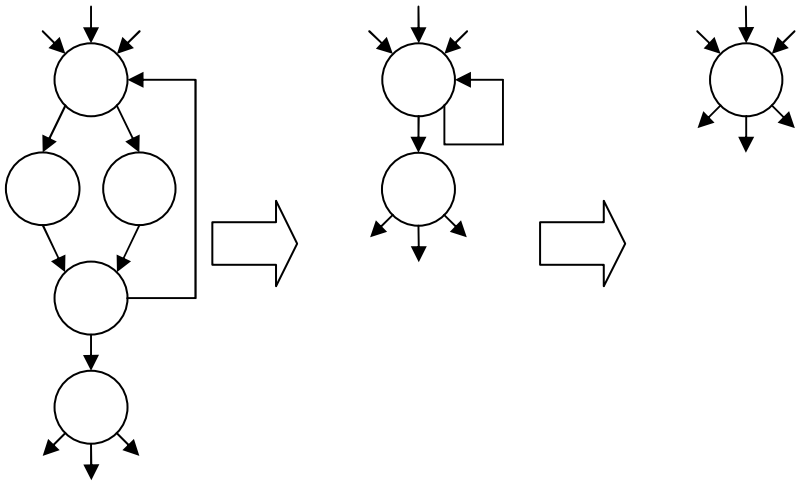
Namiesto príkazov skoku sa preto vo vyšších jazykoch používajú zložitejšie konštrukcie a to bloky, podmienené príkazy, cykly a pod. Pri kompilácii sa všetky takéto konštrukcie nahradzujú inštrukciami skoku.

Úlohou analýzy riadiacich tokov v dekompilátore je pravé identifikovať tieto konštrukcie v grafe riadiacich tokov. V tejto kapitole je názorne ukázaný jednoduchý spôsob, ktorý vyhľadáva v grafe riadiacich tokov známe vzory – šablóny. Každá šablóna je graf. Obsahuje dva významné uzly: vstupný uzol a výstupný uzol. Vyhľadávanie sa deje rekurzívne.

Príklad 7.2: Na obrázku 7.3 sú dve šablóny pre podmienený príkaz s alternatívou a pre cyklus s podmienkou na konci. Na obrázku 7.4 je príklad postupného nahradzovania blokov v grafe riadiacich tokov do zložených blokov.



Obrázok 7.3: Šablóny niektorých riadiacich konštrukcii



Obrázok 7.4: Graf riadiacich tokov

8 Abstraktný syntaktický strom, generovanie cieľového kódu

Pri analýze dátových tokov sa lineárny trojadresný kód v základných blokoch štruktúroval do zoznamu priraďovacích príkazov so zloženými výrazmi na pravých stranách. Pri analýze riadiacich tokov sa štruktúroval graf riadiacich tokov do jednotlivých podgrafov, ktoré sa skladali z ďalších podgrafov alebo základných blokov.

Výsledkom je grafová štruktúra – strom. Stromy je možné prechádzať rekurzívne. Každý uzol stromu má svoju textovú hodnotu vypočítanú rekurzívne z jeho podstromu a z neho samotného. Potom textová hodnota koreňa celého stromu je výsledný program vo vyššom jazyku.

9 Implementácia

Súčasťou tejto práce je aj implementácia pokusného dekompilátora. Ako zdrojová architektúra je použitá 8-bitová architektúra HC08. Implementovaná je väčšina algoritmov zmienených v tejto práci.

Vstupom je zdrojový kód v jazyku symbolických inštrukcií pre túto architektúru. Dekompilátor obsahuje jednoduchý lexikálny a syntaktický analyzátor. Pri syntaktickej analýze používa tabuľku inštrukcií. V tabuľke inštrukcií existuje podrobný záznam o každej inštrukcii, o jej názve, operandoch, a pod. Ďalej sa pre každú inštrukciu v tabuľke nachádza k nej prislúchajúca séria príkazov trojadresného kódu. Táto informácia sa použije pri transformovaní programu do medzikódu.

Zásobník nie je podporovaný. Z registrov sú podporované A, X a príznakové bity.

Je implementovaná optimalizácia redundantných príkazov. Dekompilátor obsahuje analýzu dátových a riadiacich tokov popísaných v tejto práci. Každá šablóna riadiacej konštrukcie obsahuje aj jej výslednú textovú podobu.

Dekompilátor bol implementovaný v prostredí Microsoft .NET v programovacom jazyku C#. Je to konzolová aplikácia. Spustením dekompilátora bez parametrov sa zobrazí jeho manuál. Zdrojové kódy sú riadne okomentované.

10 Záver

Táto práca sa venovala dekompilácii. Proces dekompilácie bol rozdelený do niekoľkých častí a boli predstavené základné postupy a algoritmy, ktoré sú v jednotlivých častiach použiteľné. Použité algoritmy nesúvisia nutne s dekompiláciou, ale sú použiteľné aj v iných prípadoch. Bol predstavený jednoduchý koncept modelujúci štruktúry v počítačových systémoch. Súčasťou práce bola aj konkrétna implementácia dekompilátora na jednoduchej počítačovej architektúre. Z pohľadu ďalšieho vývoja je možné doplniť ešte hlbšie analýzy dátových tokov, pridať podporu pre komplikovanejšie počítačové štruktúry, nepriame adresovanie a pod. V budúcnosti by bolo vhodné portovať tento dekompilátor na architektúru x86.

Literatúra

[1] Adámek, J. *Kódování*. SNTL, Praha, 1989.

[2] Bochs Project members. *Bochs: The Open Source IA-32 Emulation Project.* [online], Posledná modifikácia: 27.08.2006, [citované 11.05.2007]. Dostupné na URL: <http://bochs.sourceforge.net>.

[3] Cifuentes, C. N. *Reverse Compilation Techniques.* Dizertačná práca. Queensland University of Technology, 1994.

[4] Deursen, A. V. *Reverse Engineering*. [online], 2007, [citované 11.05.2007]. Dostupné na URL: <http://www.program-transformation.org/Transform/ReverseEngineering>.

[5] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual - Volume 2: Instruction Set Reference*. 2001

[6] Motorola, Inc. *MC68HC908LJ12 - Technical Data*. Rev. 2. 2002.

[7] Schwarz, B., Debray S., Andrews G. *Disassembly of executable code revisited*. University of Arizona, Tucscon, 2002.