CODE ANALYSIS AND TRANSFORMATION TO A HIGH-LEVEL LANGUAGE

Jakub Křoustek
Master Degree Programme (2), FIT BUT
E-mail: xkrous00@stud.fit.vutbr.cz

Supervised by: Alexander Meduna
E-mail: meduna@fit.vutbr.cz

ABSTRACT

Primary objective of this thesis is a construction of a generic decompiler, i.e. tool that can recompile from any binary form to a chosen high level language. Output must be functionally equivalent to the input. Process of decompilation is highly dependent on the processor architecture. This problem is solved with description of semantic of each instruction by a special language designed for this use. This proposal is implemented in practice as a part of project Lissom. Generic decompiler is completely new idea. The thesis contains entirely new techniques from theory of compilers and optimizations made by the author.

1. ÚVOD

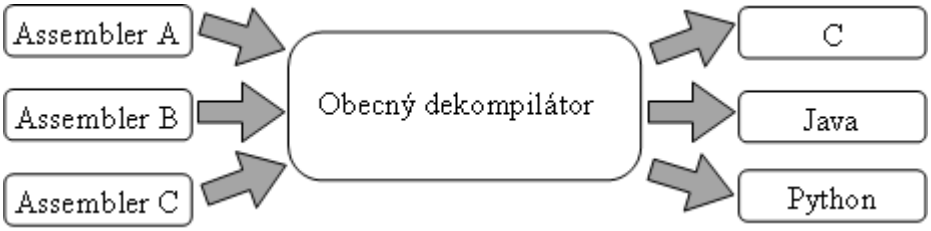
V oblasti překladačů můžeme nalézt dosti opomíjené téma – *zpětný překlad*. Při tomto postupu dochází k ději, kdy se binární spustitelný soubor transformuje zpět do vyššího programovacího jazyku. Tomuto procesu se také často říká *dekompilace*. Právě zpětný překlad může v budoucnu hrát významnou roli v softwarovém inženýrství. Mezi hlavní využití patří verifikace programů, ladění optimalizovaného kódu, migrace programů na jiné architektury a hledání malware. Jde tedy o využití v oblasti teoretické i praktické.

V minulosti již bylo několik zpětných překladačů vytvořeno, až na výjimky však pracovaly pouze s jedinou instrukční sadou a převáděly do jednoho konkrétního programovacího jazyku. Cílem této práce je vytvořit *univerzální zpětný překladač*, který bude schopen zpracovávat jakoukoliv instrukční sadu a programy v ní zapsané převádět do libovolného programovacího jazyku. Praktická implementace tohoto návrhu vzniká jako součást výzkumného projektu *Lissom*, který probíhá na FIT VUT v Brně. Tento projekt se zabývá vytvořením automatizovaných prostředků pro návrh hardwaru i softwaru aplikačně specifických procesorů. Zpětný překladač zde bude sloužit jako jeden z automaticky generovaných nástrojů, jenž mají za cíl usnadnit uživateli návrh a testování nových procesorů.

2. ROZBOR

Dříve používané metody zpětného překladu byly založené na znalosti konkrétní instrukční sady. Každý bajt strojového kódu analyzovaného programu byl pomocí převodních tabulek rozpoznán a odpovídajícím způsobem zpracován. Z těchto informací se následně generoval kód ve vyšším programovacím jazyku (nejčastěji C či Cobol).

Tento přístup však není možné aplikovat při tvorbě obecného zpětného překladače. Ten musí být schopen pracovat s jakoukoliv instrukční sadou. To znamená, že nemůže být vytvořen tak, aby znalost sémantiky jednotlivých instrukcí této sady byla přímo v něm obsažena. Proto je nutné vytvořit prostředek, s jehož pomocí půjde popsat jakoukoliv instrukční sadu. Oním prostředkem je *jazyk pro popis sémantiky instrukcí*. Každá instrukce v instrukční sadě tak bude charakterizovaná pomocí sekvence atomických instrukcí popisného jazyku. Ty instrukci popíší jako celek, tj. její chování, počet a typy operandů a změny vyvolané v prostředí procesoru. [Obrázek 1] zobrazuje obecný zpětný překladač.



Obrázek 1: Princip obecného zpětného překladače

2.1. JAZYK PRO POPIS SÉMANTIKY INSTRUKCÍ

Jednou ze stěžejních částí projektu je návrh jazyku pro popis sémantiky instrukcí. Cílem je sestavit množinu atomických *pseudo-instrukcí*, které umožní uživateli popsat každou dílčí činnost instrukce v jeho nových instrukčních sadách. Tyto pseudo-instrukce by měly být jednoduché pro zápis, ale přitom dostatečně silné pro popis každého druhu chování. Je třeba dodržet, aby množina těchto pseudo-instrukcí měla přiměřenou velikost. [Obrázek 2] je ukázkou zjednodušeného popisu instrukce v rámci projektu Lissom.

```
OPERATION daxjnz {
      ASSEMBLER { "daxjnz" "ax" "," address };
      CODING { 0b0111 address };
      SEMANTIC {
            @ax = sub i8 @ax, 1
            %Condition1 = icmp ne i8 @ax, 0
            br i1 %Condition1, label %address, label %else
            else:
      };
}
```

Obrázek 2: Ukázka popisu sémantiky

2.2. STRUKTURA ZPĚTNÉHO PŘEKLADAČE

Zpětný překladač můžeme rozdělit do tří částí: *přední*, *optimalizační* a *výstupní*[1]. Toto dělení je inspirováno zpětným překladačem z [1]. Přední část je vstupní částí překladače, která načítá binární soubory se strojovým kódem a převádí je na vnitřní reprezentaci. Tato část je parametrizovatelná modelem mikroprocesoru, který je uložen ve vstupním souboru.

[1] Často se též můžeme setkat s anglickou terminologií *front-end*, *middle-end* a *back-end*.

Bližší informace o tomto převodu jsou k nalezení v [2]. Poté následuje část optimalizační, která provádí analýzu, transformace a optimalizace vnitřního kódu. Překladač se zde snaží například v kódu vyhledávat řídící struktury vyšších jazyků, jako jsou konstrukce *if-then-else* apod. Stále je však tato část platformě i jazykově nezávislá. Proces optimalizace je z části založen na poznatcích z [3]. Výstupní část se skládá z několika modulů. Každý modul slouží pro generování kódu do vybraného vyššího programovacího jazyku.

2.3. FUNKCE ZPĚTNÉHO PŘEKLADAČE

Uživateli se po předložení popisu instrukční sady vygeneruje příslušný zpětný překladač. S jeho pomocí je pak schopen převádět binární soubory této sady do vyšších forem reprezentace. Důraz je v současné době kladen především na modul jazyku C, ale do budoucna to není limitací. [Obrázek 3] ukazuje výstupy jednotlivých částí překladu (vlevo strojový kód, uprostřed reprezentace jazykem pro popis sémantiky, vpravo výstup v jazyku C).

```
...      | ...                          | ...
01001100 | %1 = load i32* %a_addr       | #include <stdio.h>
00010011 | %2 = load i32* %b_addr       |
00010000 | %3 = add i32 %1, %2          | int add (int a, int b)
01001100 | store i32 %3, i32* %0        | {
00010011 | %4 = load i32* %0            |   return a+b;
00010000 | store i32 %4, i32* %retv     | }
01001100 | br label %return             |
11010101 | return:                      |
01101100 | %retv1 = load i32* %retv     |
```

Obrázek 3: Etapy zpětného překladu

3. ZÁVĚR

Způsobů využití obecného zpětného překladače je mnoho, ať už při verifikaci překladačů, tak například při hledání sekvencí nebezpečného kódu. Oproti stávajícím nástrojům, jako je například disassembler, získáváme nově možnost studovat programy v libovolném vyšším jazyku. Díky tomu se můžeme oprostit od zkoumání programů v jazyku symbolických instrukcí, který je zvláště u větších programů velice nepřehledný.

Proces návrhu tohoto systému je však poměrně obtížný a zdlouhavý, především kvůli nutnosti zachytit odlišnosti architektur procesorů. V současnosti jsou navržené postupy převáděny do praxe a testovány.

LITERATURA

[1] Cifuentes, C.: Reverse Compilation Techniques, PhD thesis. Queensland University of Technology, Department of Computing Science, 1994.

[2] Přikryl, Z.: Implementace obecného zpětného assembleru, Diplomová práce. Brno, FIT VUT v Brně, 2007.

[3] Van Emmerik, M.: Static Single Assignment for Decompilation, PhD thesis. School of ITEE, University of Queensland, 2007.