STROM ABSTRAKTNÍ SYNTAXE A DEKOMPILACE MSIL

Pavel Fiala, Miroslav Virius
FJFI ČVUT v Praze, fialapa5@fjfi.cvut.cz

ABSTRAKT:

Příspěvek se zabývá popisem abstraktní syntaxe, možnostmi jejího formálního zápisu, využitím při konstrukci stromu abstraktní syntaxe a jeho následné možné oblasti použití. Dále je popsán jazyk MSIL a naznačen postup analýzy a převodu MSIL kódu do stromu abstraktní syntaxe.

ABSTRACT:

This article explains the abstract syntax and abstract syntax tree (AST) concepts and outlines the AST construction for the purpose of the MSIL decompilation.

KLÍČOVÁ SLOVA:

Abstraktní syntaxe, Backusova-Naurova forma, strom abstraktní syntaxe, MSIL

ÚVOD

Abstraktní syntaxe je syntaxe jazyka (např. programovacího) vyjádřená v jednodušším tvaru než konkrétní zápis jazyka. Strom abstraktní syntaxe (Abstract Syntax Tree, AST) je datová struktura reprezentující zdrojový kód programu (nebo jeho části). Listy stromu zastupují hodnoty, uzly vyjadřují operace.

V tomto příspěvku se zabýváme AST z hlediska dekompilace jazyka MSIL do vyšších programovacích jazyků.

Cílem tohoto příspěvku je shrnout popis abstraktní syntaxe a představit návrh algoritmu pro dekompilaci v prostředí .NET.

SOUČASNÝ STAV

V současné době je d dispozici několik programů, které umožňují dekompilaci kódu v jazyce MSIL; nejznámější z nich je Reflektor, jehož autorem je L. Roeder [5] (v současné době ho distribuuje firma Red Gate). Algoritmy, které se při tom používají, však u žádného z nich nebyly zveřejněny.

POPIS ABSTRAKTNÍ SYNTAXE

Popis abstraktní syntaxe je možný pomocí metajazyka Backusovy-Naurovy formy (BNF). Popis syntaxe jazyka se provádí pomocí sady pravidel, která se zapisují ve tvaru:
`<pravidlo> ::= <výraz>`

Každé pravidlo pojmenovává konkrétní část jazyka. Pravá strana (za znaky ‘`::=`‘) může obsahovat více výrazů oddělených mezerou. Samotný `<výraz>` se skládá z posloupnosti symbolů. Má-li pravidlo několik disjunktních možností, jsou na pravé straně odděleny znakem ‘`|`‘:

`<pravidlo> ::= <výraz1> | <výraz2>`

Výrazy, které se nevyskytují na levé straně pravidel, se nazývají terminální symboly y jazyka.

Symboly na levých stranách pravidel se nazývají neterminální symboly jazyka.

BNF byla původně navržena pro popis jazyka ALGOL 60, později byla rozšiřována dle potřeb ostatních jazyků a potřeb aktuálního používání. Zajímavé rozšíření zápisu o znaky ‘[‘ ‘]‘ uvedla firma IBM při popisu svého programovacího jazyka PL/I. Použití je následující

```
<pravidlo> ::= <výraz1> [<výraz2>]
```

Uzavření prvku <výraz2> do hranatých závorek znamená, že jeho přítomnost není povinná. Následně vznikly další možnosti vyjádření opakování:

```
<pravidlo> ::= <výraz1> {<výraz2>} <výraz3>+
```

Uzavření prvku mezi znaky ‘{‘ ‘}‘ udává, že se vyskytovat nemusí nebo se může opakovat vícekrát. Znak ‘+‘ upravuje předchozí možnost na 1 nebo několika opakování výrazu.

Postupné rozšiřování zápisu BNF vedlo k vytvoření mnoha různých variant, a proto byla v roce 1996 standardizována rozšířená BNF (Extended BNF, EBNF) ve standardu ISO/IEC 14977:1996. Ke standardizování vedl také fakt, že při zápisu syntaxe popisovaného jazyka notací BNF se objevoval problém, když se v popisovaném jazyce vyskytovaly znaky ‘<‘ ‘>‘ ‘|‘ ‘::=‘.

Standard EBNF sjednocuje zápis a přidává k BNF další užitečné možnosti, mezi které patří například možnost vkládání komentářů uvedených znaky ‘(*‘ a ukončených znaky ‘*)‘.

STROM ABSTRAKTNÍ SYNTAXE

Strom abstraktní syntaxe (AST) má mnoho oblastí použití; například

- překlad programu,
- převod zdrojového kódu mezi jazyky,
- refaktorování kódu,
- formální popis syntaxe jazyka (např. referenční příručky).

Při využití AST při překladu zdrojového kódu jsou zdrojové soubory transformovány do stromu abstraktní syntaxe, následuje ohodnocení atributů stromu a poté se strom prochází a generuje se kód pro cílovou platformu. To nemusí být strojový kód, může jít o bajtový kód Javy nebo o MSIL (Microsoft Intermediate Language) v případě programu pro platformu .NET. Podobné využití je při převodu mezi různými programovacími jazyky, v poslední fázi se však generuje kód v cílovém programovacím jazyce.

Další možné použití je pro programování refaktorování, kde se zdrojový text převede na strom abstraktní syntaxe, provedou se požadované operace a nová podoba stromu se převede zpět do zdrojového textu.

Skutečnost, že čtení pravidel zapsaných pomocí EBNF je mnohdy jednodušší než čtení zdrojového kódu v konkrétním programovacím jazyce, lze využít k tomu, aby v této formě byly publikovány specifikace jazyka. Touto formou mohou být vyjádřeny všechny možné povolené konstrukce, které jsou v jazyce dostupné.

TYPY UZLŮ

Konstrukce psané ve vyšších programovacích jazycích, které patří mezi tzv. imperativní jazyky, lze většinou zařadit do některé z následujících čtyř skupin:

- popisovač výrazu (expression descriptor),
- popisovač příkazu (statement descriptor),
- popisovač typu (type descriptor),
- popisovač identifikátoru (identifier descriptor),

Při implementaci stromu abstraktní syntaxe pro konkrétní jazyk se tyto skupiny dále dělí na další podskupiny a jsou dispozici jejich konkrétní reprezentace.

Popisovač výrazu může být v extrémním případě upřesněn pro všechny operátory dostupné v daném jazyce. Často se ale vytvoří skupina společná pro několik navzájem podobných operátorů. Příkladem jsou například binární aritmetické operátory, pro které může být vytvořena jedna skupina obsahující popis operátoru, o který se jedná a který se poté vyhodnotí, a dva operandy. Podobné je to s unárními operátory. Podívejme se na příklad zápisu pravidla pro binární aritmetické operátory počítající pouze s čísly

```
<BinVýraz> ::= <operátor> <levýOperand> <pravýOperand>
<operátor> ::= + | - | * | %
<levýOperand> ::= <číslo>
<pravýOperand> ::= <číslo>
<číslo> ::= (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9) {<číslo>}
```

V abstraktní syntaxi jsou výrazy vyjádřené v programovacím jazyce zastoupeny popisovači příkazu. Základní typ popisovače je určen pro jednoduché jednořádkové příkazy. Takovýto příkaz může být například přiřazení hodnoty do proměnné. Pro blok několika příkazů se v abstraktní syntaxi navrhuje vlastní deskriptor, který v sobě zahrnuje posloupnost příkazů.

Popisovače typu se také dělí. První podtyp je určen pro základní datové typy jazyka (celé číslo, reálné číslo atd.). Dále se vytvářejí popisovače pro další typy, jako jsou pole, výčtové typy, struktury, třídy a další dle specifik jednotlivých jazyků.

Popisovač identifikátoru slouží k uchování informací o všech objektech, které se v programovacím jazyce dají pojmenovat – tj. proměnné, konstanty, funkce a další.

MSIL

Platforma .NET je reakce na úspěch jazyka Java. Podobně jako tento jazyk je i .NET navržen jako multiplatformní pro snadnou přenositelnost programu v rámci různých architektur. Pro tuto platformu vytvořila firma Microsoft speciální jazyk MSIL. Pro běh programu lze využít dva módy – řízený a neřízený. V případě použití řízeného módu je program přeložen do bajtového kódu v jazyce MSIL. Tento kód je poté spouštěn virtuálním strojem, který zajišťuje překlad MSIL do strojového kódu cílové platformy. Překlad je proveden tzv. JIT překladačem a jsou tři možnosti překladu:

1. v době instalace programu – během instalace programu je MSIL kód celý přeložen do strojového kódu cílové architektury

2. v době spuštění programu – překlad programu probíhá při každém spuštění
3. tzv. ekonomický mód – obdoba bodu 2, ale při spuštění se přeloží pouze část celku bez jakýchkoli optimalizací a zbytek se překládá za běhu programu při výskytu požadavku na dosud nepřeloženou část

MSIL je sada instrukcí snadno převeditelná do strojového kódu. Instrukce jsou podobné asembleru, ale na rozdíl od něj podporují objektové programování a ošetřování výjimek. Při překladu zdrojového kódu do MSIL jsou připojena metadata popisující přeložený kód. Tento celek je v platformě .NET nazýván sestavení (assembly). Z metadat lze zjistit, z jakých modulů se program skládá, jaké typy jsou definované a na jaké program odkazuje do knihoven.

Každá instrukce je reprezentována hexadecimální hodnotou, a to buď jednobajtovou nebo dvoubajtovou. V případě, že jde o dvoubajtovou, je hodnota prvního bajtu rovna 0xFE. Pro rozeznatelnost jedno- a dvoubajtových instrukcí nereprezentuje tato hodnota žádnou jednobajtovou instrukci.

PŘEVOD MSIL DO STROMU ABSTRAKTNÍ SYNTAXE

Základem dekompilace MSIL je vytvoření AST na základě kódu obsaženého v sestavení.

Pro zjednodušení dekompilace se může použít prostředků obsažených v .NET Frameworku. Jedná se o třídy obsažené ve jmenném prostoru System.Reflection a System.Reflection.Emit. Pomocí třídy System.Reflection.Assembly lze získat seznam modulů obsažených v sestavení (metoda getModules). Pro jednotlivé moduly lze získat seznam globálních metod definovaných v modulu (metoda getMethods) a definovaných typů a tříd (metoda getTypes).

Postupným čtením informací a vytvářením navržených struktur abstraktní syntaxe se sestaví základ stromu abstraktní syntaxe pro zkoumané sestavení. Ten v této fázi obsahuje informace o modulech, jmenných prostorech, metodách, třídách a metodách tříd.

Následuje krok pro rozšíření uzlů metod budovaného stromu. Jazyk MSIL je podobný jazyku asembler, předávání parametrů funkcím a vracení výsledků dochází pomocí zásobníku. Při dekompilaci kódu je tedy nutné simulovat práci instrukcí se zásobníkem – přidávat a odebírat položky a pracovat s nimi. Postupně se prochází strom a hledají se dosud neanalyzované metody, pro které se bude následně získávat jejich tělo. Pro takovou metodu se získá seznam lokálních proměnných, informace, zda metoda obsahuje generické parametry a maximální velikost zásobníku. Je-li velikost zásobníku větší než 0, alokuje se datová struktura s požadovanou velikostí.

Z navržené abstraktní syntaxe se vytvoří struktura pro blok příkazů, do kterého se budou postupně vkládat přečtené instrukce metody. Pro získání těla metody jako posloupnost bajtů lze v .NET Frameworku použít metodu GetILAsByteArray, která vrací pole bajtů. Nyní se toto pole prochází a rozeznávají se jednotlivé instrukce. Pokud má přečtený bajt hodnotu 0xFE, čte se hodnota dalšího bajtu a dochází k identifikaci přečtené instrukce. Je-li to nutné, po identifikaci instrukce dochází k úpravám zásobníku.

Po vykonání pomocných operací je instrukce zařazena do správné skupiny dle navržené abstraktní syntaxe, je vytvořena datová struktura pro danou skupinu a zařazena na konec bloku reprezentujícího tělo metody. Po analyzování všech instrukcí je struktura obsahující tělo metody napojena na odpovídající místo stromu abstraktní syntaxe. Takto se postupně procházejí všechny metody, pro které chceme získat jednotlivé instrukce pro další analyzování nebo prosté zobrazení.

ZÁVĚR

V tomto příspěvku jsme ukázali základní problémy, s nimiž se setkáváme při dekompilaci kódu v jazyce MSIL, a představili jsme návrh algoritmu, který bude základem vyvíjeného nekompilačního programu.

PODĚKOVÁNÍ

Práce na tomto příspěvku byla podporována z grantů MŠMT LA08015 a SGS 10/094.

LITERATURA

1. ISO/IEC 14977:1996. Information technology – Syntactic metalanguage – Extended BNF, First edition. 2006.
2. Gough, J.: Compiling for the .NET Common Language Runtime. Prentice Hall, 2001. 416 s. ISBN 0-13-062296-6.
3. Kuhn, T.; Thomann, O. *Abstract Syntax Tree*. Eclipse Corner Articles [online]. 20. 11. 2006, [cit. 2010–03–17]. Dostupné na http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html.
4. .NET Framework Developer's Guide – Inside the .NET Framework [online]. [cit. 2010-03-21]. Dostupné na http://msdn.microsoft.com/en-us/library/a4t23ktk(VS.71).aspx
5. *.NET Reflektor.* Dostupné na http://www.red-gate.com/products/reflector/. [cit. 2010-04-25]