РАЗРАБОТКА ОПТИМИЗИРУЮЩЕГО ГЕНЕРАТОРА КОДА НА ПЛАТФОРМЕ .NET

А. А. Астапов, А. М. Статкевич

ВВЕДЕНИЕ

Принципы и технологии написания компиляторов используются в самых разнообразных областях информационных технологий. Написание компиляторов охватывает языки программирования, архитектуру вычислительных систем, теорию языков, алгоритмы и технологию создания программного обеспечения.

В нашей работе описывается разработка проекта компилятора. Применяются современные технологии спецификации входного языка, порождения сканера и синтаксического анализатора. Определяются алгоритмы генерации объектного CIL-кода (кода Общего Промежуточного Языка платформы .NET). Центральной задачей является разработка оптимизирующих преобразований и процедур фазы генерации объектного кода.

ФАЗЫ КОМПИЛЯТОРА

Концептуально компилятор работает пофазно, причем в процессе каждой фазы происходит преобразование исходной программы из одного представления в другое [1]. На практике некоторые фазы могут быть сгруппированы вместе, и промежуточные представления программы внутри таких групп могут явно не строиться.

Процесс компиляции состоит из следующих этапов:

1. Лексический анализ. На этом этапе последовательность символов исходного файла преобразуется в последовательность лексем.

2. Синтаксический (грамматический) анализ. Последовательность лексем преобразуется в дерево разбора.

3. Семантический анализ. Дерево разбора обрабатывается с целью установления его семантики (смысла). Выполняются привязки идентификаторов к их декларациям, типам; проверки совместимости; определения типов выражений и т. д. Результат обычно называется «промежуточным представлением/кодом», и может быть дополненным деревом разбора, новым деревом, абстрактным набором команд или чем-то ещё, удобным для дальнейшей обработки.

4. Оптимизация. Выполняется удаление излишних конструкций и упрощение кода с сохранением его смысла. Оптимизация может быть на разных уровнях и этапах, например, над промежуточным кодом или над конечным машинным кодом.

5. Генерация кода. Из промежуточного представления порождается код на целевом языке.

АВТОМАТИЧЕСКОЕ ПОРОЖДЕНИЕ ЛЕКСИЧЕСКОГО И СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА

Процесс построения синтаксического анализатора вручную является очень трудоемкой задачей и не всегда гарантирует хороший результат – он, как правило, представляет собой многие килобайты кода, который достаточно сложно отлаживать даже для простого входного языка. Поэтому существует ряд инструментов, помогающих в этом процессе. Одним из таких инструментов, который и был использован в работе, является ANTLR.

ANTLR (ANother Tool for Language Recognition — ещё один инструмент для распознавания языков) — это инструмент, который предоставляет платформу для создания анализаторов исходного текста по описанию грамматики исходного языка на языке близком к форме Бэкуса-Наура [2].

Результатом работы ANTLR являются два класса — «лексер» и «парсер». «Лексер» разбивает поток символов на поток токенов в соответствии с правилами, а парсер обрабатывает поток токенов в соответствии с другими правилами. Правила пишутся в грамматике на специальном языке. На выходе «парсера» получаем абстрактное синтаксическое дерево (АСД).

ДИНАМИЧЕСКАЯ ГЕНЕРАЦИЯ CIL-КОДА

Генератор кода принимает на вход АСД, построенное по тексту программы на исходном языке. Результатом компиляции является создание динамической сборки для платформы .NET Framework, которая выполняется как обычное приложение, т.е. полностью готовая программа.

Динамические сборки создаются непосредственно в ходе выполнения приложения (статической сборки) и существуют в оперативной памяти. По завершении выполнения приложения они исчезают, если, конечно, не были сохранены на диск.

Для работы с динамическими сборками используется пространство имен System.Reflection.Emit [3]. Это множество типов позволяет создавать и выполнять динамические сборки, а также добавлять новые типы и члены в загруженные в оперативную память сборки.

ОПТИМИЗИРУЮЩИЕ ПРЕОБРАЗОВАНИЯ ГЕНЕРАТОРА КОДА

Существует много различных классификаций оптимизирующих преобразований. Рассмотрим классификацию по уровню представления программы.

В зависимости от уровня представления программы различают следующие виды оптимизации:

- Оптимизацию на уровне исходного языка. При этом в результате трансформации получается программа, записанная в том же самом языке.
- Машинно-независимую оптимизацию. В этом случае преобразованию подвергается программа на уровне машинно-независимого промежуточного представления, общего для группы входных или машинных языков.
- Машинно-зависимую оптимизацию, то есть оптимизацию на уровне машинного языка.

В нашем компиляторе реализована как машинно-независимая оптимизация (преобразования синтаксического дерева), так и машинно-зависимая оптимизация (оптимизация на уровне CIL-кода, которую можно условно считать машинно-зависимой).

Возможны следующие оптимизирующие преобразования АСД: предвычисление функций и выражений, содержащих константы; применение алгебраических тождеств; удаление “мертвого” кода; удаление пустого оператора; вынесение инварианта цикла и др. Заметим, что специфика входного языка позволяет применить только ограниченную часть перечисленных преобразований.

Для оптимизации на уровне CIL-кода используется peephole-оптимизация [4]. Её суть заключается в том, что оптимизатор ищет в коде метода сравнительно короткую последовательность инструкций, удовлетворяющую некоторому образцу, и заменяет ее более эффективной последовательностью инструкций.

Алгоритм peephole-оптимизации использует понятие фрейма. Фрейм можно представить как окошко, двигающееся по коду метода. Содержимое фрейма сравнивается с образцом, и в случае совпадения выполняется преобразование.

Peephole-оптимизация линейного участка кода должна выполняться многократно до тех пор, пока на очередном проходе фрейма по этому участку кода не будет найдено ни одного образца. С другой стороны, алгоритм peephole-оптимизации может быть остановлен в любой момент, что позволяет добиться требуемой скорости работы оптимизатора.

ЗАКЛЮЧЕНИЕ

В данной работе сделан обзор проблем спецификации входного языка, изучено представление исполняемого кода независимого от аппаратной платформы (язык CIL платформы .NET),

Проект компилятора реализован с использованием современной технологии спецификации входного языка и порождения сканера и синтаксического анализатора. На этом этапе используется инструмент ANTLR. Алгоритмы генерации и оптимизации кода зависят от входного и объектного языка. Наша реализация, продемонстрированная в работе при создании компилятора специального собственного входного языка, отражает общие черты оптимизирующих преобразований.

Алгоритмы и методы построения оптимизирующих генераторов кода, предлагаемые в настоящей работе, могут быть использованы при написании компиляторов, интерпретаторов входных языков, реализуемых на объектных машинно-независимых языках, а также при создании инструментов облегчающих разработку компиляторов.

Литература

1. *Ахо А., Сети Р., Ульман Д.* Компиляторы: принципы, технологии и инструментарий / М., 2003.
2. *Parr, T.* The Definitive ANTLR Reference / T. Parr – Dallas: The Pragmatic Bookshelf, 2007.
3. System.Reflection.Emit Namespace: [Electronic resource]. http://msdn.microsoft.com/en-us/library/system.reflection.emit.aspx.
4. *Макаров, А. В.* Common Intermediate Language и системное программирование в Microsoft .NET: Учебное пособие / А. В. Макаров, С. Ю. Скоробогатов, А. М. Чеповский. М.: Инт.-Ун. Инф. Тех.; БИНОМ, Лаб. Знаний, 2006.