«Ленивый» анализ исходного кода на языках C и C++

Савицкий В.О., Сидоров Д.В.
ssavitsky@ispras.ru, sidorov@ispras.ru

Аннотация. В статье описывается метод построения синтаксического анализатора, позволяющий существенно сократить требуемые для анализа ресурсы. Метод основан на том факте, что каждый исходный файл подключает множество заголовков, из которых используется лишь небольшое количество определений. Разбор определений из заголовков можно пропускать до момента непосредственного обращения к ним, таким образом, неиспользуемые определения анализироваться не будут. Отличительной особенностью метода является необходимость внесения лишь небольшого количества изменений в существующий парсер. Метод реализован в статическом анализаторе Klocwork Insight.

Ключевые слова: ленивый анализ, синтаксический анализ, C/C++.

1. Введение

По мере развития любого продукта, объем программного кода растет. Зачастую скорость прироста строк кода в проекте не линейна по времени. Такой рост, например, показывает ядро системы Linux [1]. Как следствие, растет время компиляции проектов. Кроме того, при работе с большими проектами программисты часто используют инструменты анализа и трансформации кода. Эти инструменты включают в себя рефакторинг кода, статический анализ, построение графа зависимостей проекта и интеллектуальное автодополнение. Такие задачи обычно выполняются по дереву абстрактного синтаксиса, которое получается в результате работы синтаксического анализатора (парсера) [2]. Как правило, время работы парсера сравнимо со временем последующего анализа, поэтому разработчики компиляторов прилагают немало усилий для минимизации требуемого количества ресурсов.

Проблема сокращения времени анализа наиболее актуальна для языков C и C++, так как они не обладают свойством инкапсуляции времени компиляции. Например, для языка C++ это означает, что изменение защищенного члена класса потребует повторной компиляции всего кода, в котором этот класс используется. Одним из вариантов сократить время компиляции кода является инкрементальный анализ [3] - кеширование используемых в файле заголовков

для ускорения повторного анализа. Однако, этот метод работает только при многократном анализе одного и того же файла, и требует работы в интегрированной среде разработки. Рассмотрим метод синтаксического анализа, который позволит ускорить первый разбор файла, а также может использоваться при полной сборке проекта.

2. Обзор существующих решений

Обычно структура проектов, написанных на языках C и C++, такова: в начале каждого исходного файла записаны директивы препроцессора для подключения заголовочных файлов, затем следуют определения, относящиеся непосредственно к этому файлу. Ниже показан пример такого файла.

```
#include <windows.h>
#include <iostream>
#include <user_types.h>
typedef unsigned int boxes;
```

В подключаемые файлы выносятся декларации и определения, которые могут использоваться во многих файлах. Каждый файл может использовать лишь небольшое количество определений из заголовочного файла, но вынужден подключать его целиком. Кроме того, одни и те же заголовки анализируются много раз. В результате время компиляции файла и всего проекта неоправданно увеличивается. Одним из возможных решений проблемы является исключение из компиляции неиспользуемых объектов. Такое решение частично реализовано в компиляторе MS Visual Studio [4].

В подключаемых заголовках содержится много шаблонов, многие их которых никогда не используются. Когда парсер обнаруживает заголовок какого-либо шаблона, он пропускает его тело, сохраняя контекст, необходимый для последующего разбора. Если затем парсер обнаружит инстанциацию (инициализацию типом) шаблона, то будет выполнен анализ его тела. Если же такой инстанциации не обнаружится, то тело шаблона проанализировано не будет. В результате такой код не вызовет ошибки компилятора:

```
template <typename T> class error_example {
  >>>> example error  <<<<
};
```

Такое поведение парсера можно назвать "ленивым" - анализ тела шаблона откладывается настолько, насколько возможно, или не выполняется совсем.

Другим примером «ленивого» поведения являются бесконечные списки в языке Haskell. Значение элемента такого списка вычисляется только при

непосредственном обращении к нему, если же элемент не требуется, то и значение его вычислено не будет.

«Ленивый» метод анализа можно распространить и на другие существенные части кода языков C и C++ – тела функций, структур и классов, не обязательно шаблонных.

3. «Ленивый» анализ.

Для оценки количества объектов, которые реально используются в файлах проекта, мы проанализировали несколько проектов с открытым исходным кодом. Множество используемых функций и классов определяется так. Просмотрим все объекты в исходном коде следующих типов. Если это класс, то просматриваем его члены и функции-члены. Если это функция, то просматриваем ее тело. Если встречается класс или функция, которых нет в множестве просмотренных, то добавляем их. Таким образом мы получим множество функций и классов, которые нужны для транслирования данного исходного файла. Остальные функции и классы можно пропустить при анализе. На рис. 1 представлено общее количество классов и функций на проектах в сравнении с количеством используемых.

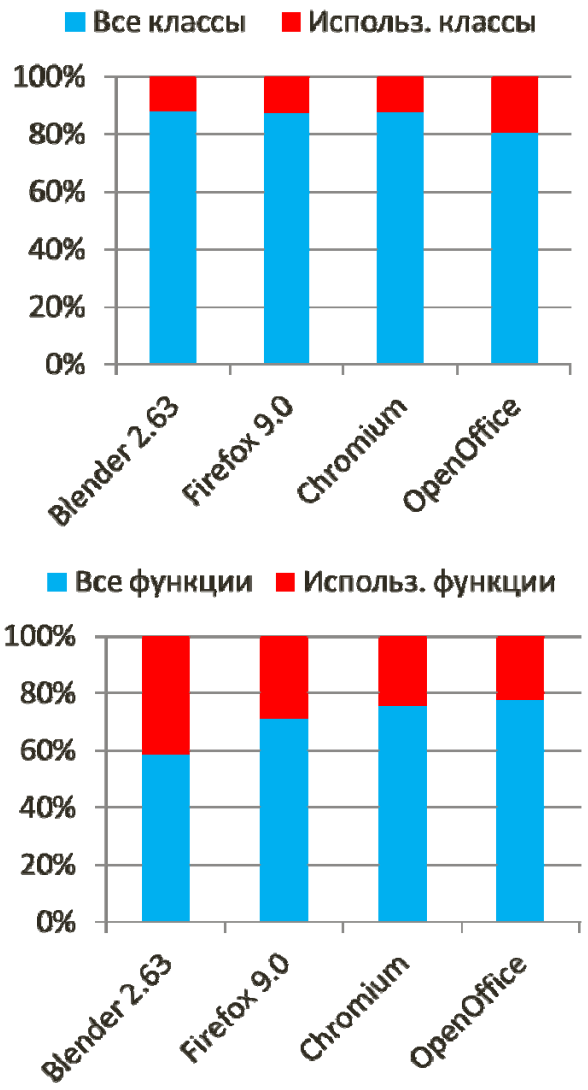

Рис. 1. Соотношение использованных и всех объектов

Получается, что при анализе этих проектов можно пропускать более половины строк, относящихся к классам, и почти 80% строк функций. Рассмотрим подробнее, как будет работать такой анализатор.

4. Описание алгоритма.

Схема работы компилятора выглядит так. Для языков C\C++ вначале исходный текст обрабатывается препроцессором, который заменяет макросы и раскрывает директивы подключения заголовочных файлов. Затем на стадии лексического анализа полученный текст разбивается на распознаваемые единицы текста - лексемы, которые передаются синтаксическому анализу (парсеру). Синтаксический анализ определяет, может ли такая последовательность лексем соответствовать грамматике анализируемого языка, и строит дерево разбора, отражающее синтаксическую структуру исходной программы. После синтаксического анализа или одновременно с ним выполняется семантический анализ. Семантический анализ осуществляет проверку типов, вычисляет значения выражений времени компиляции, разрешает вызовы функций и снабжает узлы дерева разбора атрибутами – семантическими элементами.

Как правило, парсер для языков C/C++ строится по алгоритму восходящего синтаксического разбора LALR(1). Такой парсер просматривает последовательность токенов слева направо, принимая решение о сдвиге или свертке на основании предпросмотра одного токена. При сдвиге, текущий токен или нетерминал переносится на стек; при свертке символы, соответствующие правилу свертки, со стека снимаются. Грамматики языков C и C++ в особенности не являются однозначными, и зачастую выбор продукции для свертки зависит от типа идентификатора. Поэтому при описании грамматики языка нередки конфликты вида сдвиг\свертка и свертка\свертка. Такие конфликты разрешаются по методу обобщенного LR(1) анализа - стек дублируется для каждого из возможных вариантов, и анализ проводится для каждого из стеков отдельно до тех пор, пока на одном из них не будет обнаружена ошибка синтаксиса, или оба стека не придут в одинаковое состояние.

Для добавления функциональности отложенного разбора тел функций и классов потребуется не большое количество изменений. Когда в процессе просмотра списка лексем парсер встречает символ открывающей фигурной скобки, он должен пропустить все лексемы вплоть до символа закрывающей фигурной скобки, с сохранением баланса скобок. Вместо узла дерева, соответствующего телу класса, парсер вставит специальный узел, содержащий необходимую информацию, чтобы обеспечить разбор тела, если он понадобится в дальнейшем. Эта информация будет включать в себя указатели на первый и последний символы тела класса в последовательности лексем, а также указатель на текущую область видимости. В дальнейшем, если будет необходимо создать объект класса или обратиться к объекту внутри области видимости класса, парсер установит сохраненную область видимости в качестве текущей и вызовет действие для анализа тела класса. Как правило, этой действие является процедурой того же парсера, поэтому потребуются изменения для обеспечения реентерабельности процедуры.

Следующий фрагмент демонстрирует отложенный анализ тела функции.

```
int foo(int n) {
  if (n > 0)
    return foo(n – 1);
  return 0;
}
…
int main() { // тело функции main не пропускается
  return foo(3);  // анализ тела функции foo
}
```

В этом примере анализ тела функции **foo** будет отложен до момента обнаружения ее вызова в теле функции **main**. В качестве текущей лексемы в парсере будет установлена открывающая фигурная скобка в первой строке, а в качестве текущей области видимости будет установлена область видимости функции **foo**. В ходе анализа необходимо отслеживать рекурсивный вызов текущей функции и обращение к области видимости текущего класса, чтобы не войти в бесконечный цикл при попытке запустить анализ по той же сохраненной информации. Для этого можно помечать узел, из которого берется контекст для анализа, как «используемый», и в дальнейшем не запускать анализ при обнаружении таких узлов.

При анализе тела класса можно в свою очередь откладывать разбор тел функций-членов и вложенных классов. То же верно и для тел обычных функций.

Естественно, что синтаксическое дерево, построенное при «ленивом» разборе включает не все узлы из дерева, которое будет получено при полном анализе. На момент завершения анализа в таком дереве будут содержаться специальные узлы с сохраненной информацией для тел классов и функций, к которым не было обращений. Это необходимо учитывать при обходе дерева, к примеру, с целью поиска дефектов. Однако, отсутствие дефектов для неиспользуемого кода вполне оправданно, ведь такие дефекты не будут влиять на качество программы. Кроме того они будут обнаружены после изменений, приводящих к использованию пропущенной части кода.

При сохранении контекста для отложенного разбора тела класса - указателей на лексемы и области видимости, необходимо запоминать информацию о моменте, в который была сохранена область видимости. Это нужно для того, чтобы избежать ошибок в ситуациях, когда в область видимости добавляются имена, которые в ней уже есть, но с другими значениями. Так, в следующем примере на момент объявления функции **foo** в текущей области видимости имя **x** обозначает переменную типа int. Если мы отложим разбор тела функции до момента вызова, то при восстановлении области видимости как текущей,

имя x в ней будет обозначать тип. Таким образом вместе со ссылкой на область видимости необходимо сохранять ее состояние. К примеру, можно ввести понятие **ревизии** наподобие систем контроля версий. При добавлении имени в область видимости, ее ревизия будет увеличиваться на единицу, а добавленное имя будет хранить номер ревизии. Затем значение ревизии можно будет использовать при поиске имени в области видимости - в данном примере тело функции было сохранено с областью видимости с ревизией 2, значит, все имена с номером ревизии больше 2 были добавлены позже, и не могут учитываться при поиске имен в процессе анализа тела функции.

```
template <int wheels> class Vehicle {};
-----------------------------------------------
int x;          // x: rev. 1
void foo() {    // foo: rev. 2
 x = 2;
}
typedef int x; // x: rev. 3
foo();
```

В этом примере анализ тела функции будет происходить только, когда парсер определит вызов функции **foo**. Для этого в качестве текущей будет установлена глобальная область видимости с номером ревизии 2 - то есть на момент декларации функции **foo**. В процессе определения, к какой из двух деклараций на верхнем уровне относится имя x в теле функции, будут использованы ревизии. Декларация синонима типа не попадает в список кандидатов: это имя было добавлено в область видимости с номером ревизии 3, то есть уже после объявления тела функции.

Так как шаблон **Vehicle** не используется, то его тело проанализировано не будет. Ленивый анализ имеет смысл проводить только для элементов в заголовках, ведь исходные файлы обычно используют все свои декларации, кроме того их размер не велик в сравнении с размерами заголовков.

5. Результаты.

В таблице 1 представлено суммарное время синтаксического и семантического анализа для всех файлов проектов с использованием ленивого режима и без.

проект	строк кода, тысяч	обычный режим, сек	«ленивый» режим, сек	увеличение скорости анализа
Blender 2.63	1,029	168	132	22%
Firefox 9.0	2,198	1,144	735	36%
Chromium	3,099	13,069	8,816	33%
OpenOffice 3	6,986	27,706	21,693	22%

Таб. 1.

Видно, что данные результаты не показывают такого сокращения времени, как ожидалось по оценкам количества строк кода. Это связано с тем, что зависимость времени анализа от количества строк кода не линейна. Так, например, при декларации переменной, создается всего один семантический элемент, который добавляется в текущую область видимости. При декларации массива, таких элементов будет создано несколько, а при описании шаблона кроме семантических элементов необходимо сохранять и структуру шаблона.

Однако, полученные результаты – ускорение от 22% до 36% - представляют существенное улучшение времени синтаксического анализа.

6. Заключение.

Описанный метод позволяет получить выигрыш более 20% времени синтаксического анализа проектов. Данный метод реализован в инструменте статического анализа Klocwork Insight. Еще одним способом оптимизации времени выполнения анализа может стать совмещение инкрементального и ленивого анализа при работе в средах разработки.

Список литературы.

[1] http://www.easterbrook.ca/steve/?p=694

[2] А. Аветисян, А. Белеванцев, А. Бородин, В. Несов. Использование статического анализа для поиска уязвимостей и критических ошибок в исходном коде программ. Труды ИСП РАН, том 21, с.23-38, 2011

[3] Савицкий В.О., Сидоров Д.В., Инкрементальный анализ исходного кода на языках C/C++. Труды ИСП РАН, том 22, с. 119-129, 2012

[4] http://msdn.microsoft.com/en-us/library/x5w1yety%28v=vs.71%29.aspx
[5] Aho, Sethi, Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986