Инкрементальный анализ исходного кода на языках C/C++

В.О. Савицкий ssavitsky@ispras.ru, Д.В. Сидоров sidorov@ispras.ru [1]

Аннотация. В статье описан метод построения статического анализатора кода, позволяющий существенно сократить время повторного поиска дефектов для языков C\C++. Для этого используется свойство малого отношения количества лексем из исходного файла к количеству лексем из заголовочных файлов. Метод реализован в программном продукте для среды разработки MS Visual Studio.

Ключевые слова: статический анализа кода, повторный поиск дефектов, C\C++

1. Введение.

Одной из наиболее распространенных проблем, возникающих в процессе разработки программного обеспечения, является проблема недостаточной надежности. Поэтому неудивительно, что существует большое количество современных средств, позволяющих проводить анализ программ с целью выявления тех или иных дефектов. Анализ, выполняемый по исходному коду до выполнения программы, называется статическим. Такой анализ подходит для обнаружения ошибок реализации программы. Если программист заметит предупреждение о реальной ошибке при компиляции программы, то исправить ее не составит труда. Сложнее предоставить решение, если ошибку обнаружил пользователь, который приобретает готовый продукт. Еще труднее диагностировать и исправить ошибку, если программа установлена во встраиваемой системе. Исследования на конкретных продуктах показывают, что рост времени, затраченного на устранение дефекта, экспоненциален. То есть чем раньше обнаружен дефект, тем проще его исправить.

Статический анализ довольно сложен и обычно выполняется на мощной серверной машине для всего проекта раз в сутки. Программист просматривает результаты через какой-либо интерфейс, затем вносит исправления и на следующий день снова проверяет результаты. В таком режиме дефекты могут

[1] Работа проводится в рамках реализации ФЦП «Исследования и разработки по приоритетным направлениям развития научно-технологического комплекса России на 2007-2013 годы», ГК № 07.514.11.4040 от 29.09.2011. Работа проводится при поддержке программы «Университетский кластер», http://www.unicluster.ru.

попасть, например, в ночную сборку, которая будет использоваться тестировщикам. Кроме того, на следующий день программист может уже переключиться на другую задачу, и ему придется потратить время на то, чтобы переключаться между вчерашней и сегодняшней задачами. Очевидно, большой интерес представляет возможность сокращения времени этого цикла. Так, если дефекты будут отображаться по мере ввода и редактирования текста программы, то время, затраченное на исправление, будет минимально. Рассмотрим метод инкрементального анализа исходного файла, который позволяет существенно сократить это время.

2. Обзор существующих решений

Рассмотрим общую схему работы статического анализатора. На начальной стадии она совпадает со схемой обычного компилятора.

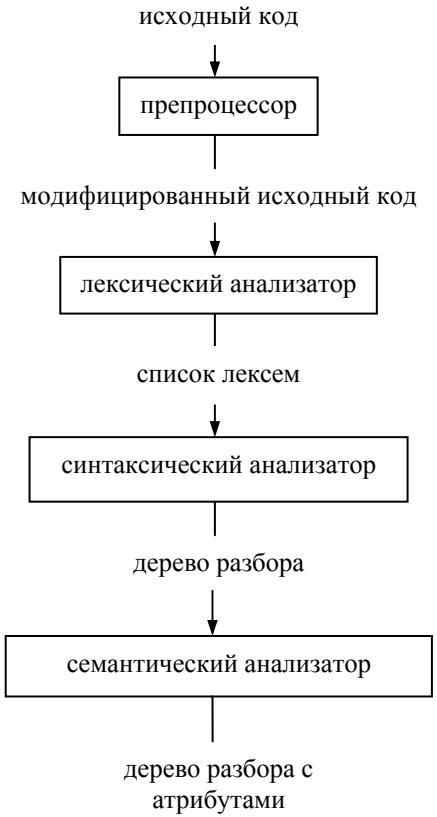


Рис. 1.

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. Простой поиск дефектов может осуществляться уже на этой стадии путем обхода дерева разбора и анализом атрибутов. К примеру, можно обнаружить сравнение переменной целочисленного беззнакового типа с нулем. Для этого в узле дерева, соответствующего бинарному выражению с оператором сравнения, необходимо проверить типы и значения операндов.

```
void foo(unsigned int n_boxes) {
 …
  if (n_boxes < 0)  { // always false
    …
  }
}
```

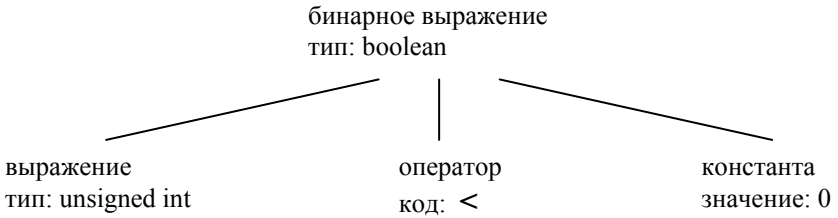


Рис.2.

Например, в продукте Klocwork для описания таких дефектов используется язык KAST [2]. Этот язык позволяет описывать **чекеры** - шаблоны поиска поддеревьев с условиями. Если при выполнении обхода дерева его часть соответствует указанному шаблону и выполнены заданные условия, то поддерево помечается как содержащее дефект. Например, для приведенного дефекта шаблон может выглядеть так:

```
// BinaryExpr
   [ OpCode = OP_LESS | OpCode = OP_GREATER_OR_EQUAL ]
   [ Left.isUnsigned() ]
   [ Right.getValue() = 0]
```


Поиск более сложных дефектов, таких как разыменование нулевого указателя или переполнение буфера, требует более сложного анализа.

```
void foo(user_type *element) {
  *element = getNext(element); // not checked for NULL
}
int main() {
  …
  element = NULL;
  …
  foo(element);
  …
}
```

Для того, чтобы определить, не является ли аргумент при вызове функции foo нулевым указателем, необходим анализ значений объектов в каждой точке программы. Такой анализ называется анализом потоков данных. Для этого дерево разбора преобразуется в промежуточное представление [1], содержащее информацию о базовых блоках (последовательность инструкций без ветвления) и вызовах функций, которые изменяют значения переменных. Анализ потоков данных существенно сложнее поиска дефектов по дереву разбора и, как правило, занимает больше времени.

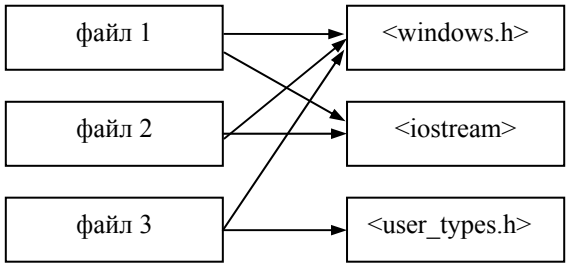
Когда программист изменяет исходный код, необходимо заново построить синтаксическое дерево, выполнить на нем анализ и сравнить новый список дефектов со старым. Для того, чтобы быстро построить новое дерево разбора, можно строить его инкрементально. История исследования возможности обновления только изменившихся частей дерева ведется еще с 70х годов. В работах [7-11] рассматриваются вопросы построения парсеров LALR(1) и GLR грамматик для работы в инкрементальном режиме. От такого парсера требуется сохранение множества состояний в каждой точке входной последовательности и поиск цепочки узлов дерева для замены при построении узлов для новых лексем. Так как возможных состояний парсера очень много, то рассматриваются оптимизации хранения этих данных в памяти. Не менее сложной является задача поиска минимального множества зависимых узлов для замены, поэтому рассматриваются возможности сокращения времени этого поиска. Любой метод построения анализатора, позволяющего инкрементальное построение дерева разбора, предполагает внесение серьезных изменений в существующие генераторы парсеров или же реализацию новых сложных систем. Покажем, что небольшой доработкой существующего анализатора для языков C и C++ можно значительно ускорить построение дерева разбора, если изменения не касаются подключаемых заголовочных файлов.

Разработчики компиляторов всегда искали пути оптимизации времени обработки кода. Для проектов, написанных на языках C и C++, важную роль сыграло следующее наблюдение. Обычно структура каждого файла такова: в начале записаны директивы препроцессора для подключения заголовочных файлов, затем следуют определения, относящиеся непосредственно к этому файлу.

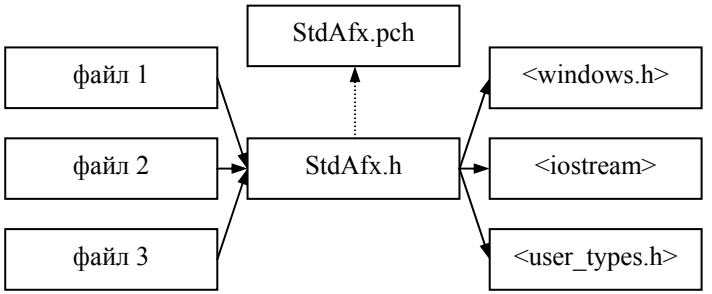
```
#include <windows.h>
#include <iostream>
#include <user_types.h>

typedef unsigned int boxes;
```

В подключаемые файлы обычно 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. Естественное решение в таком случае обрабатывать общие заголовочные файлы лишь однажды. Такая технология получила название "предварительно откомпилированные заголовки". Например, в варианте, реализованном в компиляторе MS Visual Studio [3], для всего проекта или его части создается один общий заголовочный файл, в который подключаются все заголовочные файлы, которые могут использоваться в разных исходных файлах проекта. Затем директивы подключения заголовков в каждом файле заменяются одной директивой подключения этого общего заголовочного файла.



без предварительно
откомпилированного заголовка



с предварительно
откомпилированным заголовком

Компилятору сначала передается команда создания предварительно откомпилированного заголовка. В результате компилятор создает файл, в который записывает таблицу макросов препроцессора, таблицу найденных идентификаторов, и другие данные, необходимые для последующего разбора файлов. При дальнейшей сборке проекта, компилятор тратит время только на загрузку сохраненной информации. Это существенно сокращает время анализа каждого отдельного файла. Несколько иной подход используется в компиляторе gcc [4]. gcc позволяет скомпилировать заголовочный файл как обычный файл проекта, при этом будет создан файл с расширением .gch в директории, в которой находится заголовок. Затем во время обработки директив подключения заголовков компилятор сначала ищет соответствующий .gch файл и использует его, если это возможно. Возможен и другой способ - часто используемые заголовки собираются в один. Этот заголовок компилируется и подключается к файлам проекта при помощи опции **–include**. Если в заголовочных файлах присутствует защита подключения (include guard), то заголовки из общего файла не будут обрабатываться повторно. Метод компилятора gcc предпочтительнее метода, используемого компилятором MS Visual Studio, поскольку не требует вносить изменения в исходный код проекта.

3. Анализ «на лету».

Существенное отличие выполнения статического анализа в среде разработки от анализа полной сборки проекта состоит в том, что пользователь одновременно работает лишь с небольшим количеством исходных файлов. Однако по мере ввода и редактирования файла, один и тот же исходный код с небольшими изменениями должен анализироваться вновь и вновь. Поэтому

исключительное значение приобретает скорость выполнения анализа. Будем пользоваться следующим наблюдением. Анализ нескольких проектов с открытым кодом показывает, что более 90% символов, обрабатываемых компилятором, приходят из заголовочных файлов. В таблице приведены результаты сборки проектов firefox-9.0 и boost 1.47 на платформе Linux с раздельным подсчетом лексем из заголовочных и исходного файлов.

проект	Firefox-9.0	Httpd-2.2.22	Boost 1.47
количество файлов в проекте	3,523	391	569
А) количество лексем из всех исходных файлов	13,932,951	1,037,941	214,114
Б) количество лексем из всех заголовочных файлов	477,861,983	9,294,281	195,378,599
Отношение количества А к Б	0.029	0.112	0.001
Максимальное количество лексем в одном исходном файле	414,422	37,690	6,201
Максимальное количество лексем в заголовочных файлах, подключенных в один исходный	832,767	44,920	977,346

Как показывает практика, на платформе Windows объем системных заголовочных файлов еще больше. Приведем статистику сборки тех же проектов на этой платформе:

проект	Firefox-9.0	Httpd-2.2.22	Boost 1.47
количество файлов в проекте	3,753	425	304
А) количество лексем из всех исходных файлов	14,037,713	2,601,184	107,671
Б) количество лексем из всех заголовочных файлов	667,775,667	30,542,791	103,001,359
Отношение количества А к Б	0.021	0.085	0.001
Максимальное количество лексем в одном исходном файле	406,213	143,556	5,984
Максимальное количество лексем в заголовочных файлах, подключенных в один исходный	877,804	180,374	1,406,616

Программист в процессе работы обычно дополняет или изменяет уже имеющийся код, редко изменяя подключенные заголовочные файлы. Поэтому если сохранить результаты анализа заголовочных файлов, то можно существенно сократить время, затрачиваемое на повторный запуск анализа. Рассмотрим алгоритм подробнее

4. Описание алгоритма.

Для каждого открываемого в среде разработки файла анализатор будет создавать кэш следующего вида. Анализатор обрабатывает входную последовательность символов, отслеживая момент появления первой лексемы непосредственно из текста файла. На рисунке это место отмечено пунктирной линией.

```
#include <stdio.h>
#include <stdlib.h>
-------------------------------------------------
typedef char number;

#include <inlined.h>
```

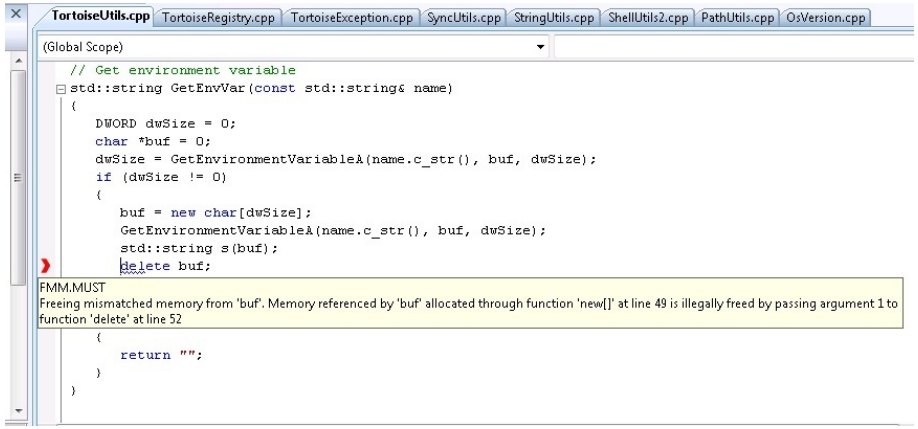
Заметим, что в данном примере последний заголовок не попадет в кэш, так как директива его подключения находится после первой лексемы из исходного файла, но это редко встречающаяся ситуация. В момент обнаружения такой лексемы в памяти сохраняется "снимок" информации, обработанной к этому моменту. В него включены таблица символов препроцессора, таблица символов парсера, необходимые семантические элементы, а также состояние области видимости идентификаторов. Также сохраняется состояние работы чекеров, которым для обнаружения дефекта нужна информация о файле целиком. Затем анализируется оставшаяся часть файла. После того как пользователь изменяет исходный файл, с некоторой задержкой запускается поиск дефектов.

Вначале необходимо перевести анализатор в состояние, которое он имел на момент создания снимка. Например, для семантического анализатора это означает удаление из областей видимости заголовков, тех элементов, которые были внесены при анализе самого исходного файла. Также необходимо освободить память, выделенную динамически для структур из основной части текста исходного файла. Для того, чтобы упростить переход к состоянию до анализа исходного файла, можно создать пулы памяти - один для всех заголовочных файлов, и один для основной части файла. Тогда память будет освобождаться в одно действие, при условии, что из пула, соответствующего заголовочным файлам, будут удалены все ссылки на элементы во втором пуле. Затем синтаксический анализатор начинает анализ с первой лексемы после заголовочных файлов. Построенное по файлу дерево синтаксиса инкрементально дополняет сохраненное дерево для заголовков. В результате

полное дерево синтаксиса не будет отличаться от построенного при полной сборке.

Поиск дефектов по дереву синтаксиса происходит довольно быстро, поэтому не имеет смысла ограничивать область поиска только областью файла, в которой произошли изменения - например, классом или функцией. Но эта оптимизация имеет большое значение для ускорения построения промежуточного представления и анализа потоков данных. Для этого при обходе дерева для каждой функции сохраняется контрольная сумма, посчитанная по лексемам. При последующих итерациях анализ ограничивается функциями с изменившимися контрольными суммами, и зависящими от них объектами.

После обнаружения дефектов они отображаются в окне с исходным кодом в виде колонки с маркерами либо подчеркиваются прямо в коде. На рисунке показано, как отображаются дефекты в плагине Klocwork для Visual Studio.



Имеет смысл запускать анализ с небольшой задержкой (например, полсекунды) после того, как пользователь прекратит набор текста. Зачастую, исходный код программы в этот момент синтаксически некорректен. Поэтому большое значение приобретает обработка ошибок. Синтаксический анализатор должен правильно восстановиться после обнаруженной ошибки, а также запретить дальнейший поиск дефектов, иначе велика вероятность ложных срабатываний и пропуска настоящих дефектов. Необходимо также проверять, не изменился ли контекст с момента сохранения "снимка" для заголовочных файлов. В данном случае под контекстом понимается набор опций компилятора, содержание заголовочных файлов и набор дефектов для поиска. Если один из этих компонентов меняется, следует запустить анализ для заголовочных файлов заново.

Постоянное хранение кэша в памяти требует аккуратного управления ресурсами. Когда пользователь открывает все новые и новые файлы в среде разработки, необходимо поддерживать объем доступной памяти не ниже

определенного порога. Он может определяться, например, так: все запущенные процессы анализа должны занимать не более 75% свободной памяти. Конечно, выгружать лишние процессы необходимо так, чтобы пользователь имел возможность быстро получать результаты анализа для активно редактируемых файлов. Если файлы проекта при сборке явно используют один и тот же предварительно откомпилированный заголовок, то все эти файлы можно обрабатывать одним процессом с общим кэшем, что облегчает задачу освобождения памяти. Так часто обстоит дело с проектами Microsoft Visual Studio. Подобная техника оптимизации времени построения синтаксического дерева может применяться не только для статического анализа, но и для задач автоматического рефакторинга кода [5]. Большинство правил рефакторинга – изменения структуры кода, не затрагивающего поведения программы – можно реализовать, используя синтаксическое дерево с атрибутами. Если пользователь будет проводить много операций по изменению кода на одном файле, то быстрое обновление синтаксического дерева будет очень полезно.

5. Результаты.

Результаты работы описанного метода оценивались следующим образом. Была написана программа, которая запускает статический анализ на файле один раз и измеряет время первого запуска. Затем следуют еще 9 итераций, на которых в файл вносятся произвольные изменения и измеряется время повторного анализа.

проект	firefox-9.0	httpd-2.2.22	boost 1.47
среднее время первого анализа, с	1.6	0.8	3
среднее время последующих стадий анализа, с	0.16	0.01	0.1
среднее количество обнаруженных дефектов	77	31	37

Как видно из таблицы, поставленная цель достигнута: за счет инкрементального анализа файла время повторных стадий анализа более чем в 10 раз меньше времени первого анализа, что позволяет перезапускать анализ практически на каждое нажатие клавиши.

6. Заключение.

Описанная техника существенно сокращает время поиска дефектов на исходном файле. Это позволяет программисту своевременно исправлять

возможные дефекты. Технология была реализована нами в коммерческом продукте для статического анализа Klocwork 9.5. Дальнейшим развитием оптимизации времени анализа может стать дополнение инкрементального анализа «ленивым». При таком подходе анализ областей видимости блоков кода откладывается до момента непосредственного обращения к элементам из этой области.

Литература.

[1] Aho, Sethi, Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.
[2] С.В. Сыромятников. Декларативный интерфейс поиска дефектов по синтаксическим деревьям: язык KAST.
[3] MS Visual Studio precompiled headers http://msdn.microsoft.com/en-us/library/szfdksca(v=vs.71).aspx.
[4] GCC Using precompiled headers http://gcc.gnu.org/onlinedocs/gcc/Precompiled-Headers.html.
[5] Martin Fowler. Refactoring: Improving the Design of Existing Code.
[6] Andrew Hunt and David Thomas The Pragmatic Programmer: From Journeyman to Master Addison-Wesley, Oct 1999.
[7] J.-M. Larcheveque *Optimal Incremental Parsing* ACM Transactions on Programming Languages and Systems, Vol. 17, No 1, January 1995.
[8] Tim A. Wagner and Susan L. Graham *Incremental Analysis of Real Programming Languages* 1997 ACM.
[9] Dashing Yeh, Uwe Kastens *Automatic Construction of Incremental LR(1) Parsers*
[10] Carlo Ghezzi, Dino Mandrioli *Incremental Parsing* ACM Transactions on Programming Languages and Systems, Vol. 1, No. 1, July 1979.
[11] Mats Wirdn *Minimal Change and Bounded Incremental Parsing* .

Incremental source code analysis for C/C++ languages.

V. O. Savitsky, D. V. Sidorov
ssavitsky@ispras.ru, sidorov@ispras.ru
ISP RAS, Moscow, Russia

Abstract. The article describes a way of implementing static code analysis for C/C++ languages that allows to reduce dramatically the time of the second and following analysis iterations. Main idea behind algorithm based on fact that programs on C/C++ languages widely use common parts of code placed inside header files with declarations and such header files included inside of several compilation units of projects. The ratio of the number of tokens in the source file to the number of tokens in the included headers is small. It is possible to reduce time of analysis using cache of compilation results for rarely changed part of compilation unit and use it in following iterations of analysis during work inside modern IDEs. The method is implemented in a commercial product for MS Visual Studio. The result of implementation described method is reduced more that in ten times time needed to analyze changed source code. Such performance improvement allows to re-analyze code on every changed symbol inside IDE and issue defects found in source code on-the-fly while developer creates program. This method is implemented inside of product Klocwork inSight 9.5

Keywords. static code analysis, second and following analysis iterations, C/C++

References

[1]. Aho, Sethi, Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.
[2]. S.V. Syromyatnikov. Deklarativnyj interfejs poiska defektov po sintaksicheskim derev'yam: yazyk KAST [Declarative Interface of Detecting Defects on Syntax Trees: KAST Language]. *Trudy ISP RAN* [*The Proceedings of ISP RAS*]. 2011, vol. 20, pp. 51-68 (in Russian).
[3]. MS Visual Studio precompiled headers http://msdn.microsoft.com/en-us/library/szfdksca(v=vs.71).aspx.
[4]. GCC Using precompiled headers http://gcc.gnu.org/onlinedocs/gcc/Precompiled-Headers.html.
[5]. Martin Fowler. Refactoring: Improving the Design of Existing Code.
[6]. Andrew Hunt and David Thomas The Pragmatic Programmer: From Journeyman to Master Addison-Wesley, Oct 1999.
[7]. J.-M. Larcheveque Optimal Incremental Parsing ACM Transactions on Programming Languages and Systems, Vol. 17, No 1, January 1995.
[8]. Tim A. Wagner and Susan L. Graham Incremental Analysis of Real Programming Languages 1997 ACM.
[9]. Dashing Yeh, Uwe Kastens Automatic Construction of Incremental LR(1) Parsers
[10]. Carlo Ghezzi, Dino Mandrioli Incremental Parsing ACM Transactions on Programming Languages and Systems, Vol. 1, No. 1, July 1979.
[11]. Mats Wirdn Minimal Change and Bounded Incremental Parsing .