электронный журнал
МОЛОДЕЖНЫЙ НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК
Издатель ФГБОУ ВПО "МГТУ им. Н.Э. Баумана". Эл No. ФС77-51038.

УДК 004.432.2

Методы обнаружения неопределённого и оптимизационно-нестабильного поведений в коде на языке C

А. С. Красиков, *студент*
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана
кафедра «Программное обеспечение ЭВМ и информационные технологии»
nyaapa@gmail.com

Научный руководитель: И. В. Ломовской, старший преподаватель
«Программное обеспечение ЭВМ и информационные технологии»
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана
irudakov@bmstu.ru

Неопределённое поведение (англ. undefined behavior) – такое поведение, которые может возникнуть при использовании ошибочных или некорректных программных конструкций или данных, на которое международная организация по стандартам в стандарте Programming languages — C не накладывает никаких ограничений [1].

Так как ограничений никаких стандартом не накладывается, то неопределённое поведение полностью зависит лишь от реализации компилятора и случайных параметров наподобие состояния памяти или сработавшего прерывания. Оно может варьироваться между полным игнорированием, выдачей предупреждающих сообщений и завершением программы.

Программы с неопределённым поведением заведомо не могут являться правильными программами (*англ.*well-formed) языка C, так как их поведение недетерминировано [1]. Стоит подчеркнуть, что стандарт языка C явно выделяет случаи неопределённого поведения.

Неопределённое поведение используется для того, чтобы упростить структуры компиляторов языка и ускорить целевой код. Например, стандарт языка C 99 не определяет результат деления на 0, численное переполнение, обращение по нулевому указателю и выход за пределы массивов. Таким образом, в программах на языке C, деление на 0 иногда может не приводить к ошибкам времени выполнения, как, например, происходит на PowerPC.

Идея неопределённого поведения заключается в том, что оно должно отлавливаться

на уровне логики программистом, а не на уровне скомпилированного кода.

Оптимизирующие компиляторы

Разработчики оптимизирующих компиляторов делают предположение, что поданный на вход оптимизирующему компилятору код не содержит неопределённых или неправильных конструкций, надеясь что все они были отловлены на предыдущих этапах компиляции.

Листинг 1. Код полагающийся на неопределённое поведение численного переполнения

```
1 int x;
2 ...
3 if (x + 1 < x) { ... }
```

К примеру, в листинге 1 приведёна приведена простая проверка на переполнение целочисленной переменной x, но выражение x + 1 никогда не будет меньше x, кроме случая переполнения. С точки зрения оптимизирующего компилятора, ему на вход подан валидный код, значит в нём нет и неопределённого поведения, поэтому он может удалить этот блок, т.к. условие никогда не сработает. Код с неопределённым поведением после прогона через оптимизирующий компилятор становится потенциально оптимизационно-нестабильным [2].

Для поиска неопределённого и оптимизационно-нестабильного поведений в исходных кодах на языке C в последнее время было предложено несколько независимых подходов.

Доопределение стандарта

Попытка доопределения стандарта языка C [3] была основана на фреймворке для создания символьных интерпретаторов K [4]. K на основе поданных ему правил преобразований создаёт семантически-переписывающий решатель, который преобразует исходный код на основе переписывающей логики. Переписывающая логика объединяет переписывающие подходы в логику с доказательной основой и семантикой.

Основная идея, лежащая в переписывающей логике, заключается в том, что все программы могут быть полностью написаны в виде переписывающих систем, которые в свою очередь состоят из термов и правил преобразования термов, которые и отображают шаги вычислений. Подавая на вход такой системе исходный код в виде термов и правила их преобразования, на выходе можно получить систему переписывающих правил.

Авторы уже ранее сделали попытку создания правил преобразования языка C [5], которая, по указанным ими результатам, вполне удалась: было пройдено 99.2% тестов из

набора тестов GCC. Таким образом, доопределение стандарта является расширением правил преобразования добавлением в них новых правил для выделенных стандартом неопределённых поведений.

Например, правило деления $\frac{< I / J... >_k}{I /_{\text{Int}} J}$ дополняется правилом

$\frac{< I / J... >_k}{\text{reportError}(\text{'Division by zero'})}$ when J = 0 для определения деления на 0.

К сожалению, авторы ограничиваются только результатами проверок на тестовых наборах, без информации о том, насколько применим данный подход к поиску неопределённого поведения в реальных исходных кодах. Основным недостатком данного метода, как средства поиска неопределённого поведения в исходном коде, является то, что появляется необходимость существования тестовых наборов для проверки наличия в результирующей программе неопределённого поведения, т.к. все проверки на возможное неопределённое поведения происходят только в момент интерпретации фреймворком.

Поиск оптимизационно-нестабильного поведения

Был сделан подход, пробующий создать для языка C* все оптимизации, но более трепетно относящиеся к неопределённому поведению [2]. Авторы так же формально определяют язык C*, как и предыдущие авторы, но лишь для того, чтобы создать оптимизирующие методы, которые не смогут приводить к появлению оптимизационно-нестабильного поведения. Авторы утверждают, что если в коде после прогона оптимизирующими компиляторами C и C* есть различия, то это означает, что оптимизация каких-то участков в доопределённом языке C* невозможна, т.е. оптимизирующие компиляторы языка C добавляют коду оптимизационно-нестабильное поведение. Для языка C* были реализованы оптимизации удаления недостижимого кода и неиспользуемых результатов вычислений. Была создана реализация данного подхода под названием Stack, принцип работы которой отображён на рисунке 1.

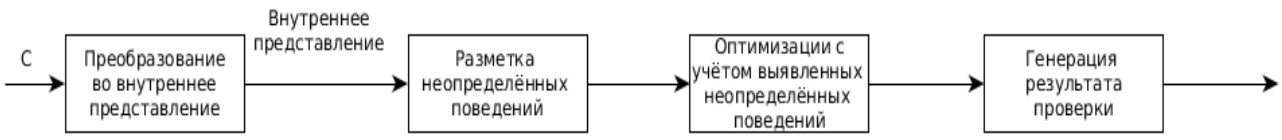

Рис. 1. Принцип работы анализатора Stack

Полученные результаты, а именно, найденные источники оптимизационно-нестабильного поведения, доказывают применимость данного подхода. Тем не менее, доопределение языка C, а также воссоздание и поддержка оптимизирующих методов для него, является сложной задачей, а если учитывать, что в новом стандарте языка C любая программа с гонками данных обладает неопределённым поведением [1], то становится очевидно, что эти недостатки устранимы крайне нетривиально, если вообще устранимы в рамках языка C.

Обобщённый поиск оптимизационно-нестабильного поведения

Принимая во внимания недостатки предыдущих подходов, а именно невозможность применения в общем случае для любых видов оптимизаций, была сделана попытка решить проблему появления оптимизационно-нестабильного поведения с меньшей эффективностью, но для общего случая. Проблема оптимизационно-нестабильного поведения заключается в том, что оптимизирующий компилятор преобразует код с неопределённым поведением, то и сравнивать нам надо неопределённые поведения до оптимизации и после. При рассмотрении оптимизационно-нестабильного поведения используют исходный код на языке C. Но, что исходный код после первого прогона оптимизирующего компилятора как-минимум крайне сложно привести в состояние, хотя бы похожее на первоначальное.

Сравнение исходных кодов можно рассматривать как выделение ключевых характеристик у сравниваемых кодов и их последующее сопоставление. Эти же ключевые состояния можно выделять и на внутреннем представлении компилятора. Таким образом, можно говорить о том, что исходный код преобразуется один в один без изменений во внутреннее представление компилятора после семантического прогона компилятора, кроме случаев примитивной оптимизации, как удаление кода следующего после `exit` вызова. Тогда следующим шагом в полученном внутреннем представлении мы должны выделить неопределённые поведения.

Можно рассмотреть листинг 2, в котором приведён пример кода, обладающего неопределённым поведением, а именно разыменованием указателя после передачи его в качестве аргумента при вызове `realloc`. Оптимизирующий компилятор Clang обладая этой информацией разделяет значения `p` и `q`, а после этого использует подстановку констант. Таким образом, хотя указатели и будут одинаковы, вызов `printf` выведет 1 2.

Листинг 2. Код с неопределённым поведением

```
1 int *p = malloc(sizeof(int));
2 int *q = realloc(p, sizeof(int));
3 *p = 1;
4 *q = 2;
5 if (p == q)
6     printf("%d %d\n", *p, *q);
```

Косвенно отловить оптимизационно-нестабильное поведение в данном случае можно разметив множества неопределённых поведений и сравнив их. После оптимизаций код будет выглядеть примерно как в листинге 3. В нём при вызове printf отсутствует вызов переменной, которая является неопределённой после вызова realloc.

Листинг 3. Код с оптимизационно-нестабильным поведением

```
1 int *p = malloc(sizeof(int));
2 int *q = realloc(p, sizeof(int));
3 *p = 1;
4 *q = 2;
5 if (p == q)
6     printf("%d %d\n", 1, 2);
```

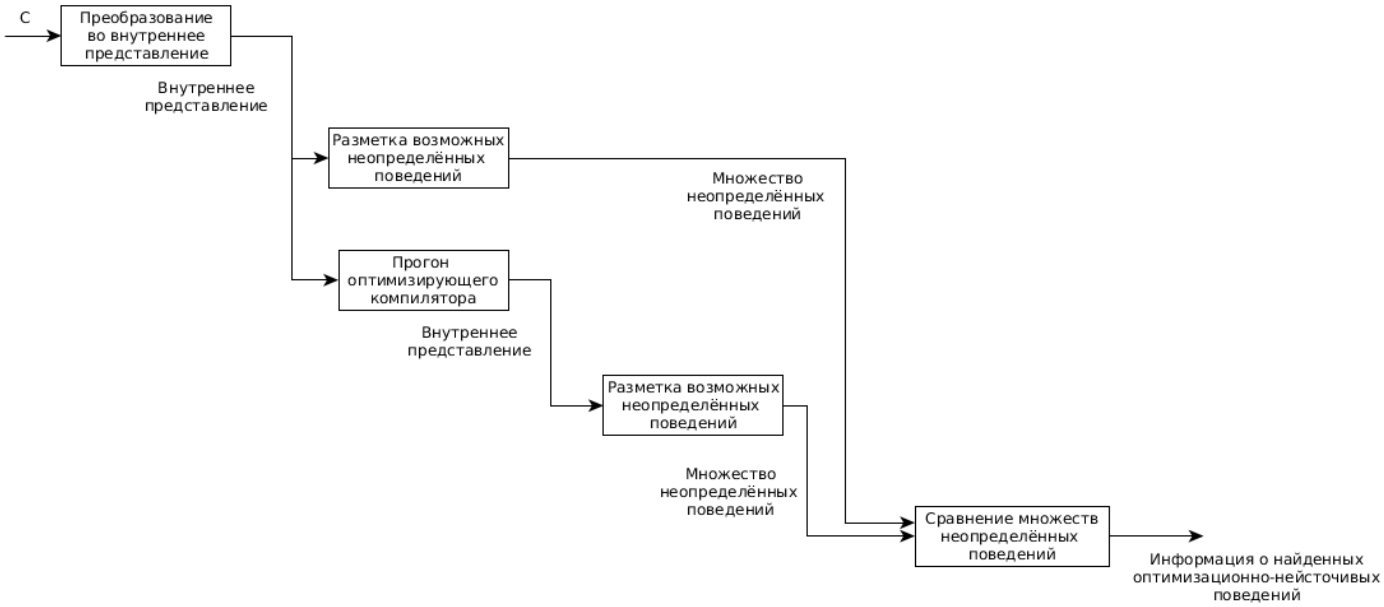

Рис. 2. Обобщённый поиск оптимизационно-нестабильного поведения

Таким образом, мы можем говорить о том, что оптимизирующий компилятор не добавил оптимизационно-нестабильного поведения если код до оптимизирующих прогонов и после обладает сопоставимыми множествами неопределённых поведений. Ключевые этапы и их последовательность предложенного способа изображены на рисунке 2.

Обобщённый способ поиска оптимизационно-нестабильного поведения также требует поддержания в актуальном состоянии механизма определения неопределённого поведения, но в отличии от остальных подходов не требует поддержания в актуальном состоянии соответсвия всем пунктам стандарта языка C.

Работа выполнена при частичной поддержке Российского фондафундаментальных исследований (грант № 13-07-00918).

Список литературы

1. Internation Standard Organization C Standard 1999 / ISO, ISO/IEC 9899:1999 draft. WG14 ed., 1999. Available at: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf, accessed: 27.11.2014.
2. Xi Wang, Zeldovich N., Frans Kaashoek M., Armando Solar-Lezama Towards

Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior. MIT CSAIL, 2013. Available at: http://dl.acm.org/citation.cfm?doid=2517349.2522728, accessed: 27.11.2014.

3. Ellison, C. Defining the undefinedness of C. University of Illinois, 2012. Available at: http://hdl.handle.net/2142/30780, accessed: 27.11.2014.
4. Rosu G. K rewrite-based executable semantic framework, ICSE'11, 2011. Available at: http://www.kframework.org, accessed: 27.11.2014.
5. Ellison C., Rosu G. An executable formal semantics of C with applications, POPL'12, ACM. Available at: http://fsl.cs.illinois.edu/pubs/ellison-rosu-2012-popl.pdf, accessed: 27.11.2014.