Моделирование неопределённого поведения

в компиляторах на основе LLVM

Максим Казанцев

В этом докладе

- Что такое и для чего нужно неопределённое поведение (**UB**)
- UB в LLVM IR. Немедленное и отложенное
- Экскурс в историю: **undef** и проблемы с ним
- Состояние дел сейчас: **poison** и с чем его едят

О докладчике

- С 2013 года занимаюсь оптимизирующими компиляторами
- Intel, Azul System, Cadence, **Сбер**
- ~1000 патчей в **LLVM**, pytorch/glow, AOSP
- Курс по оптимизирующим компиляторам в НГУ

Очень важный дисклеймер

Этот доклад посвящён исключительно технической тематике. В нём не продвигаются и не рекламируются какие бы то ни было финансовые услуги и продукты. В рамках этого доклада «цена», «стоимость» и тому подобные слова относятся только и исключительно к расходам на исполнение компьютерных программ (с точки зрения машинных ресурсов), а не к финансовым продуктам банка.

Что такое неопределённое поведение и зачем оно нужно

Определение UB

Под неопределённым поведением (**undefined behavior**, **UB**) будем понимать ситуацию, когда поведение программы может быть непредсказуемым и вести к **любым, порой неожиданным результатам**, но в спецификации языка указано, что подобное **допустимо в данных условиях**.

Примеры UB в C++

- Знаковое целочисленное переполнение
- Бесконечный цикл без сторонних эффектов
- Деление на 0
- Разыменование нулевого или некорректного указателя
- Доступ за пределы массива/объекта
- Использование неинициализированной переменной
- https://en.cppreference.com/w/cpp/language/ub

Компилятор работает с IR

Чтобы выполнять анализы и оптимизации, компиляторы обычно предпочитают работать с промежуточным представлением (Intermediate Representation, IR), а не конструкциями оригинального языка.

В нашем случае будет рассматриваться clang и LLVM IR.

Совсем нельзя предсказать?

«Ну сложу я два числа, получу какое-то мусорное значение. Ну или прочитаю я неинициализированную переменную. Это же просто мусор со стека! Я могу просто распечатать это значение и узнать его свойства, и пользоваться им!»

Неизвестный автор

Совсем нельзя предсказать?

```
void print(int y);

void uninitialized() {
    int x;
    print(x == 10);
    print(x != 10);
    print(x == 1984);
    print(x != 1984);
}
```

```
define dso_local void @uninitialized()()
local_unnamed_addr #0 {
  tail call void @print(int)(i32 noundef 0)
  tail call void @print(int)(i32 noundef 0)
  tail call void @print(int)(i32 noundef 0)
  tail call void @print(int)(i32 noundef 0)
  ret void
}
```

https://godbolt.org/z/vcWoan3jz

clang -O2 -emit-llvm -g0

Совсем нельзя предсказать?

```
void bang();        // Говорит, что всё плохо
void print(int y); // Печатает значение y

void please_dont_bang_me(int x) {
    int y = x + 1;
    if (y > x) {
        bang();
    }
    print(y);
}

void test() {
    please_dont_bang_me(2147483647);
}
```

```
define … void @test()() … {
  call void @bang()()
  call void @print(int)(i32 … -2147483648)
  ret void
}
```

Мы открыли число, большее, чем SINT_MAX?

И оно равно SINT_MIN?

https://godbolt.org/z/7dcTbYssM

Что это было?

При неопределённом поведении вы можете наблюдать эффекты, которые **невозможны ни при каких значениях** переменных. Разные части вашей программы могут начать **противоречить друг другу**.

Неопределённое поведение – **сложнее и шире**, чем просто появление мусорных значений из воздуха.

UB - свойство динамическое

Если вы просто **написали в коде**, что читаете неинициализированную переменную, но **не выполнили** эту операцию, то в вашей программе нет неопределённого поведения.

Для возникновения UB необходимо **выполнить** опасную операцию (с опасными параметрами).

А зачем это вообще надо?

«Наверное, авторам языка просто было лень определить, что будет в той или иной ситуации. Им просто нравится смотреть, как мы стреляем себе в ногу, и смеяться над нами!»

Неизвестный автор

Определённость не бесплатна

- Нет никакой проблемы в том, чтобы определить поведение в этих случаях (во всяком случае, во многих)
  - Вся арифметика по модулю 2^N
  - При доступе к nullptr бросаем исключение
  - Во всех неинициализированных переменных лежит 0
  - и т.п.
- Но почему-то это не было сделано. Почему?

Определённость не бесплатна

```
int simplify_arithmetics_s(int x) {
    int y = x * 1000;
    int z = y / 25;
    return z;
}
```

```
unsigned simplify_arithmetics_u(unsigned x) {
    unsigned y = x * 1000;
    unsigned z = y / 25;
    return z;
}
```

```
define … i32 @simplify_arithmetics_s(i32 … %0) … {
  %2 = mul nsw i32 %0, 40
  ret i32 %2
}
```

```
define … @simplify_arithmetics_u (i32 … %0) … {
  %2 = mul i32 %0, 1000
  %3 = udiv i32 %2, 25
  ret i32 %3
}
```

Нельзя упростить из-за необходимости поддерживать арифметику по модулю 2^N

https://godbolt.org/z/bs9T35Tfv

Определённость не бесплатна

```
int simplify_noinit(int x, bool cond) {
    int y;
    if (cond)
        return x;
    else
        return y;
}
```

```
int simplify_init(int x, bool cond) {
    int y = 0;
    if (cond)
        return x;
    else
        return y;
}
```

https://godbolt.org/z/hjzrMfMsd

```
define … i32 @simplify_noinit(i32 … %0, i1 … %1) … {
  ret i32 %0
}
```

```
define … i32 @simplify_init(i32 … %0, i1 … %1) … {
  %3 = select i1 %1, i32 %0, i32 0
  ret i32 %3
}
```

Мы вынуждены проверять условие, чтобы поддержать инициализацию нулями

Определённость не бесплатна

Если определили поведение в некоторой **ошибочной** ситуации, то за это придётся **заплатить скоростью обработки корректных ситуаций**. Неопределённое поведение позволяет этого избежать.

Общее правило работы компилятора с UB: «делай так, как будто UB никогда не возникнет»

Неопределённое поведение в LLVM
немедленное и отложенное

Что можно компилятору

Задача оптимизирующего компилятора – создать максимально эффективный код, семантика которого **не противоречит** семантике исходной программы на исходном языке.

- Сценарии, в которых исполнялись инструкции, ведущие к UB, могут вести себя **как угодно** после оптимизации.
- Сценарии, в которых UB не было, обязаны вести себя предсказуемо.

Немедленное (Immedaite) UB

Некоторые операции ведут к UB в случае, если будут **исполнены** с определёнными данными:

- Деление на 0
- load/store из/в nullptr

Проще всего считать, что UB наступает **немедленно** в момент исполнения такой инструкции.
Такая ситуация называется **немедленным (Immediate) UB**.

Немедленное (Immedaite) UB

Вы не обязательно увидите странные эффекты именно в этом месте, **как только** эта инструкция будет выполнена. Но **весь ваш код** может вести себя непредсказуемо, если она выполнится.

UB может проявиться **сильно после** исполнения проблемной инструкции.

UB может также не иметь видимых проявлений.

Работа с немедленным UB

```
declare void @print(i32 %what)

define void @hoi
entry:
    br label %l op

loop:
    %iv = phi  2 [0, %                 next, %          ]
    br i1 %co  label          rue, label %ba

if.true:
    %y = sdiv  2 1000,
    call void  rint(i3
    br label %  ckedge

backedge:
    %iv.next =  dd i32 %iv, 1
    %loop.con   icmp slt i32 %iv.next, 1000
    br i1 %lo        , label %loop, label %done

done:
    ret void
}
```

```
declare void @print(i32 %what)

                                          {
entry:
    %y = sdiv i32 1000, %x
    br label %loop

lo                 32 [0,      try], [%iv.next,      ackedge]
                nd, labe          label %ba      edge

if
                rint(      %y)
          abe       kedg

backedge:
    %iv.next = add i32 %iv, 1
    %loop.cond = icmp slt i32 %iv.next, 10
                                              e

done:
    ret void
}
```

FAKE

Чего нельзя делать: немедленное UB

Компилятор не имеет права выполнить трансформацию, если **существует** сценарий, в котором до трансформации **никакая** инструкция с немедленным UB не исполнялась, а после трансформации **какая-то** такая инструкция начнёт исполняться.

UB при переполнении

LLVM IR позволяет выразить как сложение по модулю 2^N, так и сложение, ведущее к UB при переполнении. Инструкции в этом случае выглядят так:

add i32 %x, %y – сложить по модулю 2^N

add **nsw** i32 %x, %y – **no sign wrap** позволяет UB при знаковом переполнении

UB при переполнении

```
define i32 @arith_overflowing(i32 %x) {
    %y = mul i32 %x, 1000
    %z = sdiv i32 %y, 25
    ret i32 %z
}

define i32 @arith_non_overflowing(i32 %x) {
    %y = mul nsw i32 %x, 1000
    %z = sdiv i32 %y, 25
    ret i32 %z
}
```

```
define i32 @arith_overflowing(i32 %x) {
  %y = mul i32 %x, 1000
  %z = sdiv i32 %y, 25
  ret i32 %z
}

define i32 @arith_non_overflowing(i32 %x) {
  %z = mul nsw i32 %x, 40
  ret i32 %z
}
```

https://godbolt.org/z/nYsYb6r8G

opt -passes=instcombine

Тут UB нет

```
declare void @print(i32 %what)

define void @hoist_add(i32 %x, i1 %cond) {
entry:
    br label %loop

loop:
    %iv = phi i32 [0, %entry], [%iv.next, %backedge]
    br i1 %cond, label %if.true, label %backedge

if.true:
    %y = add i32 %x, 1000000
    call void @print(i32 %y)
    br label %backedge

backedge:
    %iv.next = add i32 %iv, 1
    %loop.cond = icmp slt i32 %iv.next, 1000
    br i1 %loop.cond, label %loop, label %done

done:
    ret void
}
```

```
declare void @print(i32 %what)

define void @hoist_add(i32 %x, i1 %cond) {
entry:
    %y = add i32 %x, 1000000
    br label %loop

loop:
    %iv = phi i32 [0, %entry], [%iv.next, %backedge]
    br i1 %cond, label %if.true, label %backedge

if.true:
    call void @print(i32 %y)
    br label %backedge

backedge:
    %iv.next = add i32 %iv, 1
    %loop.cond = icmp slt i32 %iv.next, 1000
    br i1 %loop.cond, label %loop, label %done

done:
    ret void
}
```

Так можно

UB при переполнении

```
declare void @print(i32 %what)

define void @hoist_add(i32 %x, i1 %cond) {
entry:
    br label %loop

loop:
    %iv = phi i32 [0, %entry], [%iv.next, %backedge]
    br i1 %cond, label %if.true, label %backedge

if.true:
    %y = add nsw i32 %x, 1000000
    call void @print(i32 %y)
    br label %backedge

backedge:
    %iv.next = add i32 %iv, 1
    %loop.cond = icmp slt i32 %iv.next, 1000
    br i1 %loop.cond, label %loop, label %done

done:
    ret void
}
```

```
declare void @print(i32 %what)

define void @hoist_add(i32 %x, i1 %cond) {
entry:
    %y = add nsw i32 %x, 1000000
    br label %loop

loop:
    %iv = phi i32 [0, %entry], [%iv.next, %backedge]
    br i1 %cond, label %if.true, label %backedge

if.true:
    call void @print(i32 %y)
    br label %backedge

backedge:
    %iv.next = add i32 %iv, 1
    %loop.cond = icmp slt i32 %iv.next, 1000
    br i1 %loop.cond, label %loop, label %done

done:
    ret void
}
```

А так можно?

Есть нюанс

```
define void @hoist_division(i32 %x, i1 %cond) {
entry:
    %y = sdiv i32 1000, %x
    br label %loop

loop:
    %iv = phi i32 [0, %entry], [%iv.next, %backedge]
    br i1 %cond, label %if.true, label %backedge

if.true:
    call void @print(i32 %y)
    br label %backedge

backedge:
    %iv.next = add i32 %iv, 1
    %loop.cond = icmp slt i32 %iv.next, 1000
    br i1 %loop.cond, label %loop, label %done

done:
    ret void
}
```

```
define void @hoist_add(i32 %x, i1 %cond) {
entry:
    %y = add nsw i32 %x, 1000000
    br label %loop

loop:
    %iv = phi i32 [0, %entry], [%iv.next, %backedge]
    br i1 %cond, label %if.true, label %backedge

if.true:
    call void @print(i32 %y)
    br label %backedge

backedge:
    %iv.next = add i32 %iv, 1
    %loop.cond = icmp slt i32 %iv.next, 1000
    br i1 %loop.cond, label %loop, label %done

done:
    ret void
}
```

Отложенное (deferred) UB

Есть инструкция, которая может вести к UB, но сама она **может быть безопасно выполнена**. UB может быть **отложено** до того момента, когда «плохое» значение, полученное в результате этой инструкции, будет **использовано** неподобающим образом.

Обычно эти инструкции имеют версию без UB (например, арифметика без флагов по модулю 2^N).

Что такое нюанс

```
declare void @print(i32 %what)

define void @hoist_division(i32 %x, i1 %cond) {
entry:
    %y = sdiv i32 1000, %x
    br label %loop

loop:
    %iv = phi i32 [0, %entry], [%iv.next, %backedge]
    br i1 %cond, label %if.true, label %backedge

if.true:
    call void @print(i32 %y)
    br label %backedge

backedge:
    %iv.next = add i32 %iv, 1
    %loop.cond = icmp slt i32 %iv.next, 1000
    br i1 %loop.cond, label %loop, label %done

done:
    ret void
}
```

Поведение станет неопределённым, как только выполнится деление при x = 0

Что такое нюанс

```
declare void @print(i32 %what)

define void @hoist_add(i32 %x, i1 %cond) {
entry:
    %y = add nsw i32 %x, 1000000
    br label %loop

loop:
    %iv = phi i32 [0, %entry], [%iv.next, %backedge]
    br i1 %cond, label %if.true, label %backedge

if.true:
    call void @print(i32 %y)
    br label %backedge

backedge:
    %iv.next = add i32 %iv, 1
    %loop.cond = icmp slt i32 %iv.next, 1000
    br i1 %loop.cond, label %loop, label %done

done:
    ret void
}
```

Поведение станет неопределённым, только если

- add переполнится
- Результат будет распечатан

Отложенное UB Шрёдингера

Возможно, вы и вычислили «плохое» значение, но вы в безопасности, пока его не пронаблюдаете (т.е. не исполните с ним опасную операцию).

Кот Шрёдингера тоже в безопасности, пока его не пронаблюдают.

Моделирование отложенного UB через undefined value

В начале было слово undef

Undefined value – типизированное значение, имеющее неизвестный и меняющийся **в произвольный момент времени** битовый паттерн (например, i32 undef).

Компилятор **в любой момент** может превратить **undef** в **любое** значение этого типа, если считает это выгодным.

Оптимизации с undef

```
define i32 @optimize_undef_example(i1 %cond) {
entry:
    %ret_value = select i1 %cond, i32 10, i32 undef
    ret i32 %ret_value
}
```

```
define i32 @optimize_undef_example(i1 %cond) {
  ret i32 10
}
```

Если **cond = true**, надо вернуть 10.

Если **cond = false**, надо вернуть какой-нибудь любой битовый паттерн.

Например, 10.

https://godbolt.org/z/hdKx8KEqh

```
opt -passes=instcombine
```

Оптимизации с undef

```
define i32 @optimize_undef_example(i32 %x, i1 %cond) {
entry:
    %ret_value = select i1 %cond, i32 %x, i32 undef
    ret i32 %ret_value
}
```

```
define i32 @optimize_undef_example(i32 noundef %x, i1 %cond) {
    ret i32 %x
}
```

Если **cond = true**, надо вернуть x.

Если **cond = false**, надо вернуть какой-нибудь любой битовый паттерн.

Например, он всегда равен x.

https://godbolt.org/z/39M4a6Evv

```
opt -passes=instcombine
```

Оптимизации с undef

```
define i32 @optimize_undef_example(i32 noundef %x,
                                   i32 noundef %y,
                                   i32 noundef %z) {

entry:
    %v1 = add i32 %x, 1
    %v2 = add i32 %y, undef
    %v3 = add i32 %z, %x
    %sum1 = add i32 %v1, %v2
    %sum2 = add i32 %sum1, %v3
    ret i32 %sum2
}
```

```
define i32 @optimize_undef_example(i32 noundef %x,
                                   i32 noundef %y,
                                   i32 noundef %z) {

  ret i32 undef
}
```

Здесь undef замешалось в цепочку вычислений, и с этого момента можно получить любой битовый паттерн в v2, sum1 и sum2

https://godbolt.org/z/Psd683fnM

```
opt -passes=instcombine
```

Undef и отложенное UB

С самого начала в LLVM считалось, что достаточно считать результат инструкции с отложенным UB при соответствующих данных **undefined value.**

Переход к немедленному UB

Некоторые операции считаются **безопасными**: имея **undef** в качестве операнда, они опять произведут **undef**. Пример: add, sub и т.п.

Некоторые операции считаются **опасными**: имея **undef** в качестве операнда, они при исполнении породят **немедленное UB**. Пример: load, store, call и т.п.

Свойства undef

Операция	Результат
`i32 undef == undef`	`i1 undef`
`i32 undef != undef`	`i1 undef`
`i32 undef > 100`	`i1 undef`
`i32 undef ^ 10`	`i32 undef`
`i32 undef + 1000`	`i32 undef`

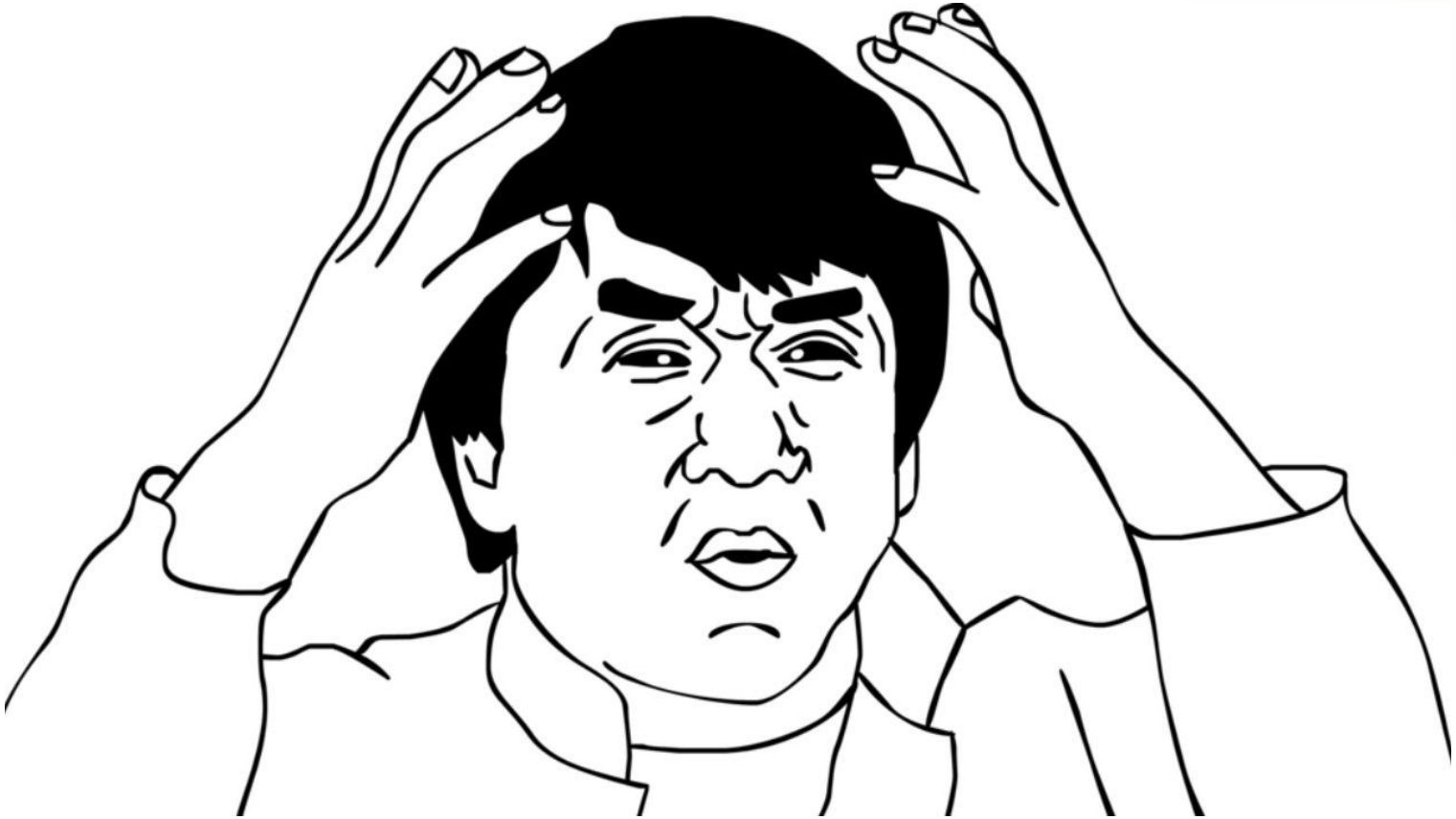
Свойства undef

Операция	Результат
`i32 undef > INT_MAX`	`false`
`i32 undef < INT_MIN`	`false`
`i32 undef <= INT_MAX`	`true`
`i32 undef & 0`	`0`
`i32 undef | 0xffffffff`	`0xffffffff`

Свойства undef

Операция	Результат
`undef % 100`	Что-то из [0, 100)?
`undef & 0xff0000ff`	Что-то, у чего старшие и младшие биты произвольные, а остальные нули?
`undef / 2`	Что-то из [0, INT_MAX/2)?
`undef * 2`	Произвольное чётное число?
`undef * 17`	Произвольное число, делящееся на 17?

Свойства undef



Ограничения undef

- Если использовать undef для моделирования отложенного UB, то у оптимизатора довольно много ограничений (в т.ч. нельзя превратить “add nsw x, 1 > x” в true).
- Получится ли в результате undef, true или произвольное значение со сложными свойствами - становится сложной задачей.
- Это не полноценная модель UB!
- К сожалению, проблемы undef на этом не заканчиваются.

Что такое if (undef)?

- `if (undef) { do_smth } else { do_smth_else }`
- Интуитивно кажется, что такое ветвление должно выполнить `do_smth` либо `do_smth_else`, но не является немедленным UB
- Например, кажется, что такая конструкция не может привести к крашу или зависанию программы

Контрпример 1

```
void bad_division(int x, int y) {
    int z = x + 1000000000; // переполнение
    // x -- нормальное число
    // y > 0
    if (y == z) {
        foo(x / y); // нет UB
    }
}
```

```
void bad_division(int x, int y) {
    int z = x + 1000000000; // переполнение
    // x -- нормальное число
    // y > 0
    if (y == z) {
        foo(x / z); // UB
    }
}
```

В оригинальной программе немедленного UB не было. Откуда оно взялось?

Что такое if (undef)?

Корректные оптимизации не создают UB из воздуха.

Данная оптимизация (замена y на z при их равенстве) очевидно корректная.

Следовательно, UB было в изначальной программе.

В стандарте LLVM записали, что br i1 undef - это немедленное UB

Контрпример 2

```
bool early_exit();

void bad_unswitching(int x, int y) {
    bool cond = x + 1000000000 > y; // undef
    for (int i = 0; i < 1000; i++) {
        if (early_exit()) break; // всегда выйдет
        if (cond) { // никогда не исполнится
            // do_smth
        } else {
            // do_smth_else
        }
    }
}
```

Контрпример 2

```
bool early_exit();

void bad_unswitching(int x, int y) {
    bool cond = x + 1000000000 > y; // undef
    if (cond) { // UB???
        for (int i = 0; i < 1000; i++) {
            if (early_exit()) break; // всегда выйдет
            // do_smth
        }
    } else {
        for (int i = 0; i < 1000; i++) {
            if (early_exit()) break; // всегда выйдет
            // do_smth_else
        }
    }
}
```

Что такое if (undef)?

Если определить ветвление по undef как немедленное UB, то Loop Unswitching становится некорректной оптимизацией (если не доказано, что undef не возникает).

Но при этом это тоже очевидно корректная и нужная оптимизация!

Что же делать?

Poison – решение всех проблем

* не является медицинской рекомендацией

Undef не справился с задачей

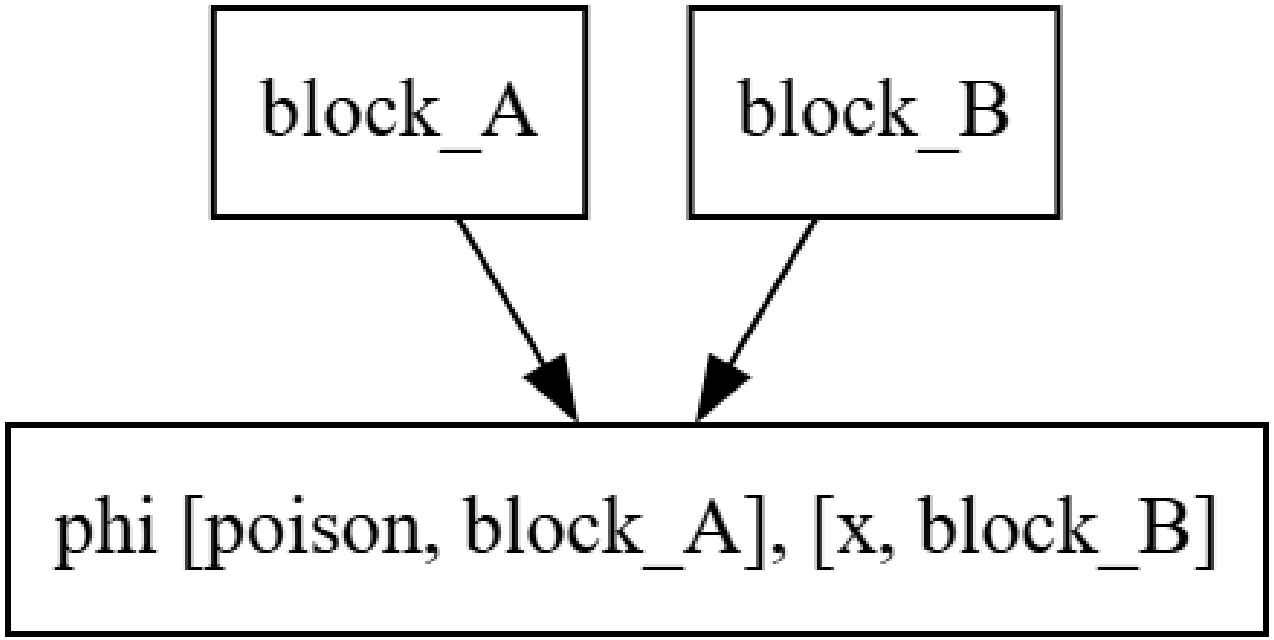
Моделирование **отложенного UB** через **undefined value** оказалось в целом несостоятельной идеей.

Такая модель оказалась и **неполной** (нельзя представить число больше MAX_INT), и **внутренне противоречивой** (очевидно корректные оптимизации оказывались некорректными).

Poison как средство от проблемы

- Появился в LLVM 12.0 (2021 год)
- Пусть в результате операции с отложенным UB возникает не **undef**, а **poison**
- Любая* операция с **poison** – немедленное UB или **poison**
- **Poison** в любой момент может быть превращён в **undef** (но не наоборот)
- Ветвление по **undef** – не UB
- Ветвление по **poison** – UB

Разумное исключение



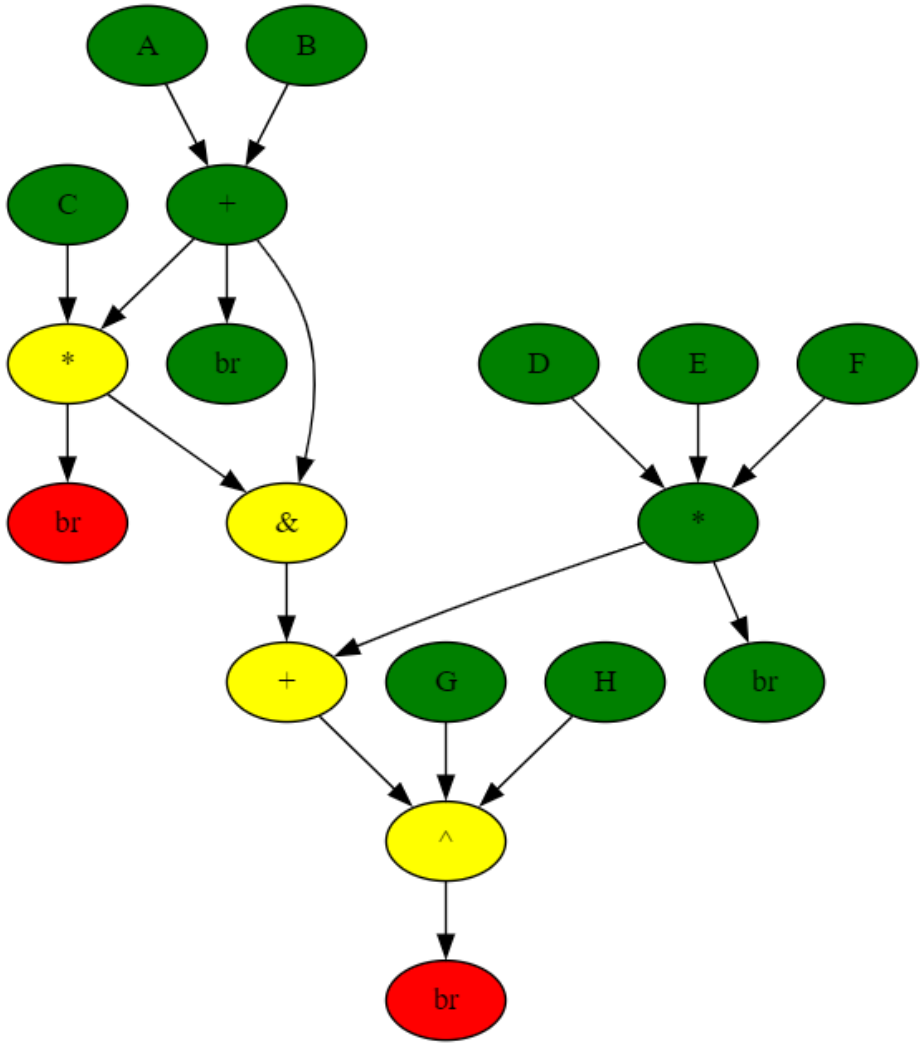
Разумное исключение

- `phi [poison, block_A] [%x, block_B]` – не poison, если мы пришли из блока B
- `select cond, poison, %x` – не poison, если cond == false
- Чисто формально poison выступает здесь операндом, но динамически он не оказывается выбран, поэтому результат не становится poison

Свойства poison

Операция	**Результат**
`poison == poison`	`poison`
`poison > SINT_MAX`	`poison`
`poison & 0`	`poison`
`false ? poison : X`	`X`
`Φ[poison, bb1], [X, bb2] при входе из bb2`	`X`

Poison распространяется по коду



Решён ли контрпример 1?

```
void bad_division(int x, int y) {
    int z = x + 1000000000; // переполнение
    // x -- нормальное число
    // y > 0
    if (y == z) {   // UB
        foo(x / y);
    }
}
```

Решён ли контрпример 2?

```
bool early_exit();

void bad_unswitching(int x, int y) {
    bool cond = x + 1000000000 > y; // poison
    if (cond) { // ОПЯТЬ UB???
        for (int i = 0; i < 1000; i++) {
            if (early_exit()) break; // всегда выйдет
            // do_smth
        }
    } else {
        for (int i = 0; i < 1000; i++) {
            if (early_exit()) break; // всегда выйдет
            // do_smth_else
        }
    }
}
```

Решение проблем с Unswitching

Единственная проблема, которую poison не решил – это раннее использование отложенное UB. Оно по-прежнему создавало немедленное UB там, где его не было.

Нас бы устроило «как в ветвлении по undef».

Заморозка deferred UB

Для решения подобных проблем придумали инструкцию freeze, которая «замораживает» инструкции undef и poison, лишая их части волшебных свойств.
После заморозки это неизвестный, но зафиксированный набор битов, который не меняется в произвольные моменты.

Операция	**Результат**
`freeze(X), если X не poison/undef`	`X`
`freeze(X), если X poison или undef`	`«замороженный» undef`

Контрпример 2 решён!

```
bool early_exit();

void bad_unswitching(int x, int y) {
    bool cond = x + 1000000000 > y; // poison
    if (freeze(cond)) { // Пойдём в одну из веток
        for (int i = 0; i < 1000; i++) {
            if (early_exit()) break; // всегда выйдет
            // do_smth
        }
    } else {
        for (int i = 0; i < 1000; i++) {
            if (early_exit()) break; // всегда выйдет
            // do_smth_else
        }
    }
}
```

Весь зоопарк

Свойство	Обычное значение	Замороженный undef	undef	poison
Произвольный набор битов своего типа	Нет	Да	Да	Выходит за рамки своего типа
Если используется в безопасной операции	Обычное значение	Обычное значение	undef, обычное значение или «нечто странное»	poison
Если используется в операции с отложенным UB	Обычное значение или poison	Обычное значение или poison	undef, обычное значение, нечто странное» или poison	poison
Может менять значения в рамках одной инструкции	Нет	Нет	Да	Да
br X	true- или false-ветка	true- или false-ветка	true- или false-ветка	UB
load X	Загрузка или UB	UB	UB	UB

Выводы

- Моделирование отложенного неопределённого поведения – сложная задача, которая в LLVM была решена не с первого раза
- В настоящее время почти во всех случаях используется poison, undef со временем может исчезнуть
- Не нужно недооценивать то, что компилятор может сделать при неаккуратной работе с UB

Контакты

Telegram

Хабр

Спасибо!