Undefined Behavior in LLVM

John Regehr
Trust-in-Soft / University of Utah
• $\sqrt{-1} = ?$
  - $i$
  - NaN
  - Arbitrary value
  - Exception
  - Undefined behavior

• **Undefined behavior** (UB) is a design choice
  - System designers use UB when they don’t feel like committing (or can’t commit) to any particular semantics
Undefined behavior is undefined

• Technically, anything can happen next
  – “Permissible undefined behavior ranges from ignoring the situation completely with unpredictable results, to having demons fly out of your nose.”

• In practice, UB is implemented lazily: by assuming it will never happen
Common consequences include...

- Predictable and useful result on one platform, different result on another platform
- Unpredictable or nonsensical result
- Memory corruption
- Remote code execution
- Trap or fault
- No consequences at all
• **AVR32 (embedded CPU):**

  D - Debug State

  The processor is in debug state when this bit is set. The bit is cleared at reset and should only be modified by debug hardware, the _breakpoint_ instruction or the _retd_ instruction. Undefined behaviour may result if the user tries to modify this bit using other mechanisms.

• **Scheme R6RS:**

  value. The effect of passing an inappropriate number of values to such a continuation is undefined.

• **C/C++ have tons and tons of undefined behaviors**
  
  – divide by zero, use of dangling pointer, shift past bitwidth, signed integer overflow, ...

• **LLVM has undefined behavior too**
int foo (int x) {
    return (x + 1) > x;
}

int main () {
    printf("%d\n", (INT_MAX + 1) > INT_MAX);
    printf("%d\n", foo(INT_MAX));
    return 0;
}

$ gcc -O2 intmax-overflow.c ; ./a.out
0
1
```c
int main() {
    int *p = (int*)malloc(sizeof(int));
    int *q = (int*)realloc(p, sizeof(int));
    *p = 1;
    *q = 2;
    if (p == q)
        printf("%d %d\n", *p, *q);
}

$ clang -O realloc.c ; ./a.out
1 2
```
void foo(char *p) {
    #ifndef DEBUG
        _foo:
        printf("%s\n", p);
    #endif
    if (p != 0)
        bar(p);
}

_without -DDEBUG

void foo(char *p) {
    #ifdef DEBUG
        _foo:
        printf("%s\n", p);
    #endif
    if (p != 0)
        bar(p);
}

_without -DDEBUG

typedef void (*foo_t)(char *);

foo_t bar = foo;

_without -DDEBUG

typedef void (*foo_t)(char *);

foo_t bar = foo;

_without -DDEBUG

typedef void (*foo_t)(char *);

foo_t bar = foo;
void foo(char *p) {
    _foo:
    #ifdef DEBUG
        printf("%s\n", p);
    #endif
    if (p != 0)
        bar(p);
}

With -DDEBUG

pushq %rbx
movq %rdi, %rbx
call _puts
movq %rbx, %rdi
popq %rbx
jmp _bar
As developers, what can do we about undefined behavior in C and C++?

• Only use these languages appropriately
• Use modern coding style
• Dynamic tools
  – UBSan, ASan, Valgrind
  – And test like crazy, use fuzzers, etc.
• Static analysis tools
  – Enable and heed compiler warnings
  – Lots more
Facts About UB in LLVM

• It exists to support generation of good code
• It is independent of undefined behavior in source or target languages
  – You can compile an UB-free language to LLVM
• It comes in several flavors
• Reasoning about optimizations in the presence of UB is very difficult
• Compilers transform source programs to target programs in a series of steps, e.g.
  – Swift ➔ SIL
  – SIL ➔ LLVM
  – LLVM ➔ ARMv8

• At each step
  – OK to remove UB
  – Must not add UB
  – This is refinement

• Example: Shift instructions are defined for shifts past bitwidth
  – But different processors define it differently
LLVM has three kinds of UB

1. Undef
   - Explicit value in the IR
   - Acts like a free-floating hardware register
     • Takes all possible bit patterns at the specified width
     • Can take a different value every time it is used
   - Comes from uninitialized variables
   - Further reading
     • [http://sunfishcode.github.io/blog/2014/07/14/undefined-introduction.html](http://sunfishcode.github.io/blog/2014/07/14/undefined-introduction.html)
• We want this optimization:
  \[
  \text{\%add} = \text{add nsw i32 \%a, \%b}
  \text{\%cmp} = \text{icmp sgt i32 \%add, \%a}
  \]
  \[
  \Rightarrow
  \text{\%cmp} = \text{icmp sgt i32 \%b, 0}
  \]
• But undefined doesn’t let us do it:
  \[
  \text{\%add} = \text{add nsw i32 \%INT_MAX, \%1}
  \text{\%cmp} = \text{icmp sgt i32 undefined, \%INT_MAX}
  \]
• There’s no bit pattern we can substitute for the undefined that makes \%cmp = true
LLVM has three kinds of UB

2. Poison

– Ephemeral effect of math instructions that violate
  • nsw – no signed wrap for add, sub, mul, shl
  • nuw – no unsigned wrap for add, sub, mul, shl
  • exact – no remainder for sdiv, udiv, lshr, ashr

– Designed to support speculative execution of operations that might overflow

– Poison propagates via instruction results

– If poison reaches a side-effecting instruction, the result is true UB
LLVM has three kinds of UB

3. True undefined behavior
   - Triggered by
     - Divide by zero
     - Illegal memory accesses
   - Anything can happen as a result
     - Typically results in corrupted execution or a processor exception
• Which of these transformations is OK?

```assembly
%result = add nsw i32 %a, %b
=>
%result = add i32 %a, %b
```

I’m OK
• Use Alive to do automated proofs about LLVM peephole optimizations:
  – [https://github.com/nunoplopes/alive](https://github.com/nunoplopes/alive)
  – Alive understands all three kinds of UB

```
$ ./alive.py add.opt
----------------------------------------
Optimization: 1
Precondition: true
  %result = add nsw i32 %a, %b
=>
  %result = add i32 %a, %b

Done: 1
Optimization is correct!
```
$ ./alive.py add-bad.opt

Optimization: 1
Precondition: true
  \texttt{%result = add i32 \%a, \%b}
=>
  \texttt{%result = add nsw i32 \%a, \%b}

ERROR: Domain of poisoness of Target is smaller than Source's for \texttt{i32 \%result}

Example:
\texttt{\%a i32 = 0x7FFEFFFF (2147479551)}
\texttt{\%b i32 = 0x7FFFFBFFF (2147482623)}
Source value: 0xFFFFFEBFBE (4294962174, -5122)
Target value: poison
• We translated a bunch of InstCombine patterns into Alive
  – Found some wrong ones, reported bugs
  – Found some missed opportunities to preserve UB flags (nsw, nuw, exact)

• Details can be found in a paper

• Please try out Alive if you reason about peephole optimizations in LLVM
Conflicting design goals for LLVM UB

1. Enable all optimizations that we want to perform
2. Be internally consistent
3. Be consistent with the LLVM implementation

The current scheme generally works fine

• But it’s not clear that it actually meets any of these three goals
• Nuno Lopes is heading an effort to rework poison and undef
  – Currently they are (we think) unnecessarily complicated
  – Goal is to make undef a bit stronger and drop poison entirely
  – No change to “true UB”

• Other compilers (GCC, Microsoft) have similar UB-related concepts
  – Detailed specifications are hard to find
  – Same motivation: support efficient code gen
Thanks!