Semantics for Compiler IRs: Undefined Behavior is not Evil!

Nuno Lopes (Microsoft Research)

Joint work with: Juneyoung Lee, Yoonseung Kim, Youngju Song, Gil Hur (SNU);
David Majnemer, Sanjoy Das (Google); Ralf Jung (MPI-SWS); Zhengyang Liu,
John Regehr (U. Utah)

With Undefined Behavior, Anything is Possible

Raph Levien’s blog

Nasal demons

From: John F. Woods

Newsgroups: comp.std.c

Subject: Re: Why is this legal?

Date: 25 Feb 1992

In short, you can't use sizeof() on a structure whose elements haven't been defined, and if you do, **demons may fly out of your nose**.

OK, OK; so **the Standard doesn't *ACTUALLY* mention demons** or noses. Not as such, anyway.

Linux kernel bug in 2009

```
static unsigned int tun_chr_poll(struct file *file, poll_table * wait)
{
    struct tun_file *tfile = file->private_data;
    struct tun_struct *tun = __tun_get(tfile);
    struct sock *sk = tun->sk;    Implies tun != NULL
    unsigned int mask = 0;

    if (!tun)  <---  Always false
      return POLLERR;
```

UB is **not** a demon

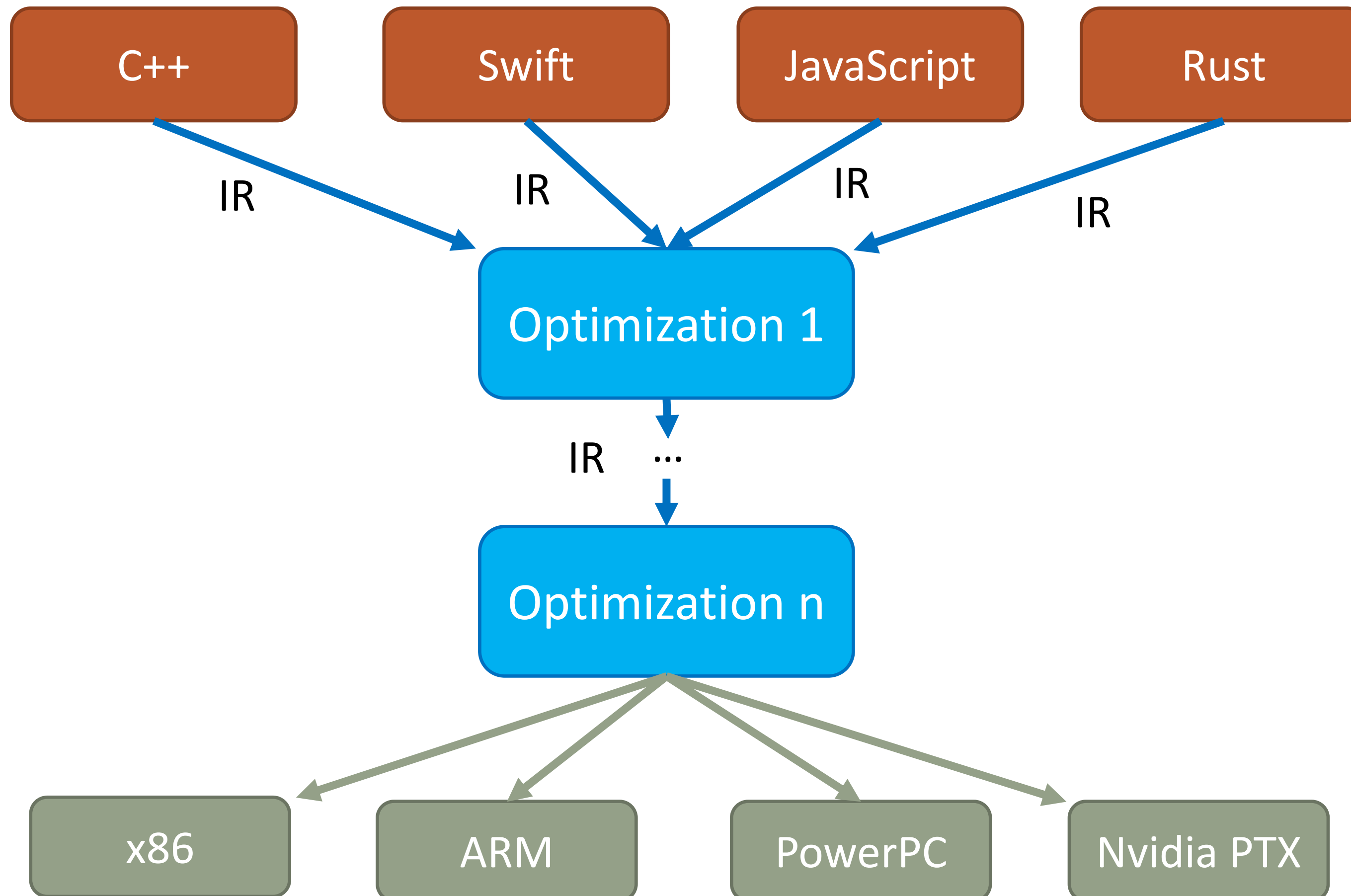
- Not at IR level, at least

- UB simply restricts domain of functions
- UB is an efficient way to represent assumptions
- Even “safe” languages benefit from UB in the IR

This talk is about Compiler IRs

NOT SOURCE LANGUAGES

Typical compiler



<u>IR must be:</u>

- Expressible
- Support desired optimizations
- Block wrong transformations
- Efficient transformations
- Efficient analyses
- Efficient encoding of assumptions from source language
- Efficiently cache derived facts
- Efficient lowering to ASM

??

UNSAT!

And SSA was born (late 80's)

Before:

```
x = 2;
x = 3;
y = x + 1;
if (y < 0)
  y = 0;
return y;
```

?

SSA:

```
x₀ = 2;
x₁ = 3;
y₀ = x₁ + 1;
if (y₀ < 0)
  y₁ = 0;
y₂ = ϕ(y₀, y₁)
return y₂;
```

Pros:

- Fast operand lookup (caches reaching definitions)
- Avoids mistakes in operand lookup
- Enables sparse flow-sensitive analysis

Cons:

- Takes time to build, maintain & undo
- Consumes more memory

What about sparse **path-sensitive**?

- SSA only gives sparse **flow-sensitive** analysis
- SSI is born (1999)!

SSA:

```
y₀ = x₀ + w₀;
if (y₀ < 0)
  y₁ = -y₀;
y₂ = ϕ(y₀, y₁)
return y₂;
```

$y_0 \in [-2^{n-1}, 2^{n-1})$

$y_1 \in [-2^{n-1}, 2^{n-1})$

$y_2 \in [-2^{n-1}, 2^{n-1})$

SSI:

```
y₀ = x₀ + w₀;
if (y₀ < 0)
  y₁ = π(y₀)
  y₂ = -y₁;
else
  y₃ = π(y₀)
y₄ = ϕ(y₂, y₃)
return y₄;
```

$y_0 \in [-2^{n-1}, 2^{n-1})$

$y_1 \in [-2^{n-1}, 0)$

$y_2 \in (0, 2^{n-1})$

$y_3 \in [0, 2^{n-1})$

$y_4 \in [0, 2^{n-1})$

What about overflows?

```
y₀ = x₀ + w₀;
if (y₀ < 0)
  y₁ = π(y₀)
  y₂ = 0 - y₁;
```

$y_0 \in [-2^{n-1}, 2^{n-1})$

$y_1 \in [-2^{n-1}, 0)$

$y_2 \in (0, 2^{n-1})$

- '0 – INT_MIN' overflows
  - in two's complement = INT_MIN
  - $\mathrm{INT_MIN} \notin [0, 2^{n-1})$
- What now? Back to $[-2^{n-1}, 2^{n-1})$?
- But:
  - In C++, signed overflow is UB
  - In Swift, program crash on overflow

What about memory?

- SSA/SSI is only about local variables (“registers”)
- GCC (and later LLVM) added MemorySSA
- Memory gets functional

```
// x, y point to disjoint objects

m1 = store(m0, x0, 3);  // *x = 3;
m2 = store(m0, y0, 4);  // *y = 4;
w = load(m2, y0);       // w = *y;
v = load(m1, x0);       // v = *x;
```

Pros:

- Easier to reorder memory operations
- Easier to detect redundant loads
- Easier to do store forwarding

Cons:

- (Still?) expensive in practice: GCC & LLVM implement only a lightweight version

What’s next?

- I don’t know! ☺ Research ongoing

- This talk: exploiting UB as an efficient means to encode assumptions

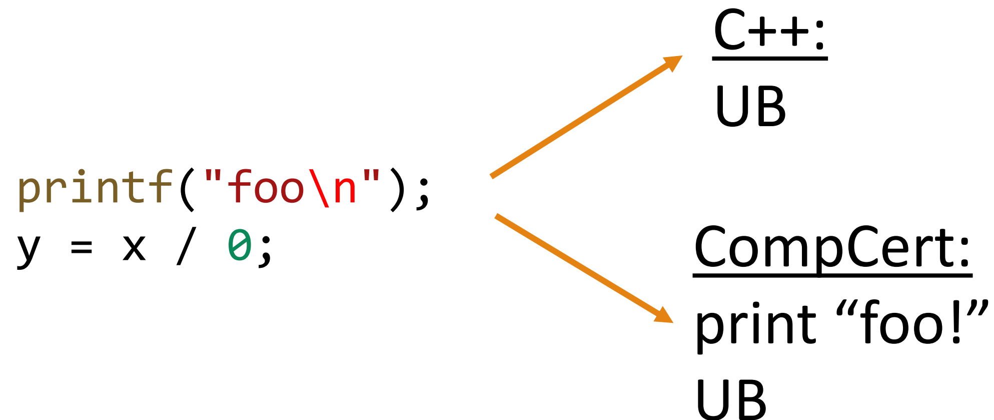
<u>IR must be:</u>

- Expressible
- Support desired optimizations
- Block wrong transformations
- Efficient transformations
- Efficient analyses
- Efficient encoding of assumptions from source language
- Efficiently cache derived facts
- Efficient lowering to ASM

What’s undefined behavior (UB)?

There are two mainstream definitions:

- C/C++/GCC/LLVM: If the program executes UB, it has no specified semantics
  - UB can travel back in time
- CompCert: Program is defined until UB is executed



UB restricts domain of functions

```
int f(int x, int y) {
  return x / y;
}
```

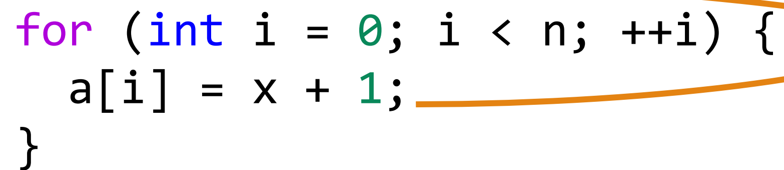
- What’s the value of `f(3, 0)`? **UB**
- What’s the value of `f(INT_MIN, -1)` ? **UB**
  - Range of int is $[-2^{n-1}, 2^{n})$
  - E.g., with 3 bits is: $[-4, 3]$, so -4/-1 not in range

C/C++ uses of UB

- For operations that trap on some ISA (e.g., division by 0) performance
- For operations with different results in different ISAs (e.g., shift overflow) performance
- To enable optimizations (e.g., memory model) performance

Importing all of C++'s UB to IR…

- … is not a good idea!
- Semantics for source languages have different goals than for IRs

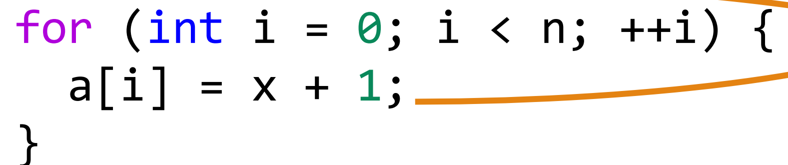


```
for (int i = 0; i < n; ++i) {
  a[i] = x + 1;
}
```

- We want to hoist x + 1
- In C++, x + 1 is UB if x = INT_MAX
- Take n = 0, x = INT_MAX
  - Original: well defined
  - Hoisted: UB

UB constrains movement

- Instructions that may raise UB are hard to move
- So: get rid of UB?



We cannot get rid of UB!

- It’s easy for arithmetic
  - E.g., define semantics division by zero and pay the price when lowering to ASM
- But not for memory (in IRs supporting non-memory safe languages)
  - Easy: load from NULL
  - Nearly impossible: load from non-allocated memory
    - Unless you make the language memory safe…

```
int f(int *p) {
  return *p;
}
```

- Is p in bounds?
- Is the object pointed by p still alive?
- Is the access properly aligned?

LLVM’s design

- UB: operations that trap in mainstream ISAs
  - Arithmetic errors (division by zero, INT_MIN/-1, etc)
  - Memory operations (out-of-bounds accesses, etc)

- “Delayed” UB (allows free movement):
  - Signed arithmetic overflows: for loop optimizations
  - Shift overflows: ISA result mismatch
  - Uninitialized memory reads: for performance

Delayed UB

- At least 3 forms in current compilers:
  - Poison value [LLVM]: Taint value like NaN (“undef” in CompCert)
  - Undef value [LLVM]: arbitrary value of the type; different value every time it’s read
  - Arbitrary value [ICC]: consistent arbitrary value of the type (“freeze poison” in LLVM)

```
poison * x → poison

undef + x → undef
undef * x → ? (not undef)

x = undef
x + x → undef

x = freeze poison
x + x → even number (not undef)
```

Benefits of delayed UB

- Still constrains domain of functions like immediate UB
- Allows free movement
- Allows speculative execution

Undef values are nice!

- They make local transformations really easy
- Freeze poison is harder

```
x = undef;
a = x * y;
b = x | y;
return a + b;
```

assume → x = 0

```
x = undef;
a = 0;
b = x | y;
return a + b;
```

assume → x = -1

```
x = undef;
a = 0;
b = -1;
return a + b;
```

constant → fold

```
return -1;
```

```
x = freeze poison;
a = x * y;
b = x | y;
return a + b;
```

assume → x = 0

```
x = 0;
a = 0;
b = y;
return a + b;
```

constant → fold

```
return y;
```

(BTW ‘return -1’ isn’t necessarily better/faster/smaller than ‘return y’)

But undef values are tricky

Wrong:

```
• x * 2 → x + x       // even number vs any number if x is undef
• x ? undef : y → y   // y may be poison
```

OK:

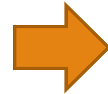
```
• x * 2 → x << 1
• x + x → x * 2
• x ? poison : y → y
```

Trying to get rid of undef values in LLVM ATM…

Hoisting with undef

```
if (k != 0) {
  while (...) {
    use(1 / k);
  }
}
```

k != 0, so safe to hoist division?



```
if (k != 0) {
  int tmp = 1 / k;
  while (...) {
    use(tmp);
  }
}
```

<u>If k = undef</u>

"k != 0" may be true **and**

"1 / k" trigger UB

Poison taints too much sometimes

- Bitfields of C are tricky to implement

```
struct t {
  int a:2;
  int b:3;
}

x->a = y;
```

```
tmp = load x             // load a, b
tmp = tmp & ~3;          // discard old a
tmp = tmp | (y & 3)      // add new a
store tmp, x             // store new a, old b
```

What if x->b was poison?

Summary so far

- Immediate UB is a necessary evil, but inhibits movement
- Delayed UB enables movement but:
  - `undef` has surprising effects
  - `freeze poison` is hard to optimize locally
  - `poison` taints too much sometimes

Let’s get rid of delayed UB?

- That’s possible, unlike immediate UB
- But… it’s a good way of expressing assumptions

Compiling shift

```
int f(int x, int y) {
  return x << y;
}
```

- What's the value of `f(1, 32)`?
  - X86: 1 (x << (y & 31))
  - ARM: 0 (x >= 32 ? 0 : x << y)
  - C/C++: UB

Compiling shift #2

- If the result of x << 32 is defined in source language as 0, then on x86 we need:
  - x >= 32 ? 0 : x << y;

```
mov %edx, %eax
shl %cl, %eax
cmp $0x20, %edx
mov $0x0, %edx
cmovge %edx, %eax
```

- Languages like C++ don’t want this cost, so define this case as UB

Shift in LLVM/CompCert

- Overflow yields poison: `x << 32 → poison`
- Pushes semantics of overflow case to front-ends
- Benefits: efficient lowering & free movement

```
for (int i = 0; i < n; ++i) {
  a[i] = x << y;
}
```

```
tmp = x << y;
for (int i = 0; i < n; ++i) {
  a[i] = tmp;
}
```

Undef/Poison for SSA construction

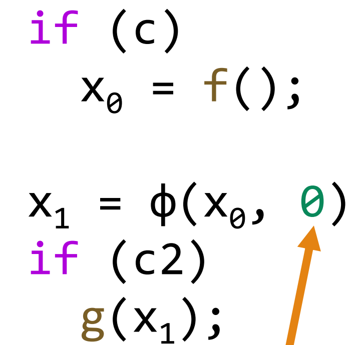
C code:

```
if (c)
  x = f();

if (c2)
  g(x);
```

SSA:

```
if (c)
  x0 = f();

x1 = φ(x0, undef)
if (c2)
  g(x1);
```

SSA without undef:



```
if (c)
  x0 = f();

x1 = φ(x0, 0)
if (c2)
  g(x1);
```

Code size increases by 2 bytes:
“xorl %eax, %eax”

Poison as cache for inferred facts

```
int w, x, y;

if (x >= 0 && y >= 0) {
  w = x - y;
  ...
}
```

1. Range analysis proves x – y doesn't overflow
2. How to cache this information?
   - In a side map?
   - In the instruction?

LLVM has overflow attributes:

- nsw: no signed overflow (wrap)
- nuw: no unsigned overflow

No signed overflow

```
int w, x, y;

if (x >= 0 && y >= 0) {
  w = x - y;
  ...
}
```

```
int w, x, y;

if (x >= 0 && y >= 0) {
  w = x -nsw y;
  ...
}
```

$$x -_{\text{nsw}} y = \begin{cases} x - y, & \text{if no signed overflow} \\ \text{poison}, & \text{otherwise} \end{cases}$$

- Analyses only need to tag instructions with derived facts
- No invalidation of on-the-side metadata
- Tags are first-cast citizens: front-ends can also insert them

Integer overflow in C++

```
for (int i = 0; i <= n; ++i) {
  a[i] = 42;
}
```

Mismatch between pointer and index types on x86-64

Index increment can overflow in 32 bits
So cannot be trivially changed to 64 bits

IR for x86-64:

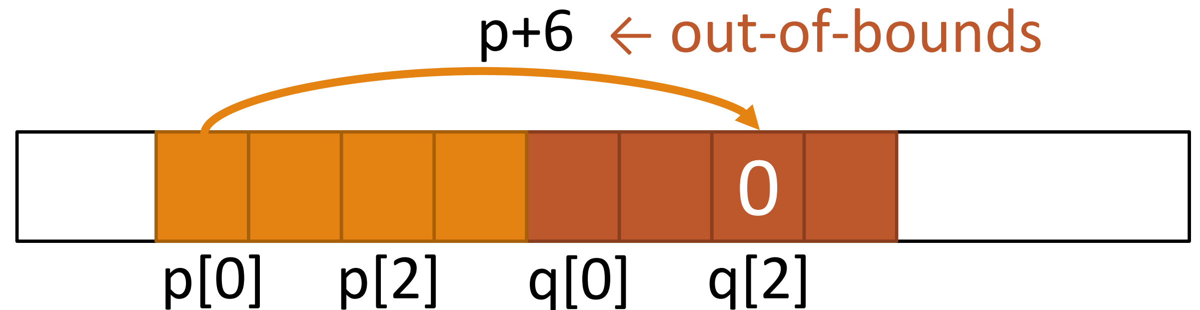
```
for (int i = 0; i <= n; ++i) {
  *(a + sign_ext64(i)) = 42;
}
```

Hoisting sext gives 39% speedup on my desktop!

32 bits /	64 bits:
i = 1	1
i = 2	2
…	
i = INT_MAX	INT_MAX
i = INT_MIN	INT_MAX+1
…	

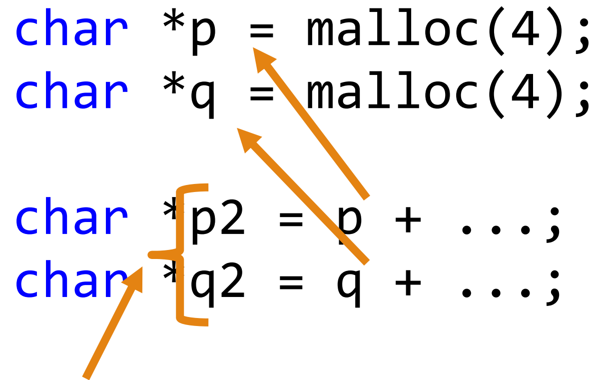
C++ pointers: data-flow provenance

```
char *p = malloc(4);
char *q = malloc(4);
char *q2 = q + 2;
char *p6 = p + 6;

*q2 = 0;
*p6 = 1;    UB

print(*q2);   print(0)
```



Pointer must be inbounds of object found in use-def chain!

C++ pointers: simple no-alias detection



If 2 pointers are derived from different objects, they don’t alias!

Programs cannot guess the memory layout,
but can observe (with pointer-to-integer casts)

Tradeoffs

BECAUSE THERE’S NO FREE LUNCH

Associativity w/ NSW

(with 8 bits)

a = 50

b = -50

c = -100

$v = (a +_{nsw} b) +_{nsw} c$

$v = a +_{nsw} (b +_{nsw} c)$

v = -100

$v = a +_{nsw} (-150)$

$v = a +_{nsw} \text{poison}$

v = poison

Goals for select in LLVM

```
select %c, %a, %b   // (c ? a : b)
```

Should allow:

- control-flow → select
- select → control-flow
- select → arithmetic
- select removal
- select hoisting through arithmetic
- easy movement

Which one?

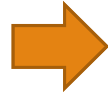
```
select %c, %a, %b
```

	UB if c poison + conditional poison	UB if c poison + poison if either a/b poison	Conditional poison + non-det choice if c poison	Conditional poison + poison if c poison	Poison if any of a/b/c poison
control-flow → select	✓		✓	✓	
select → control-flow	✓	✓			
select → arithmetic		✓			✓
select removal	✓	✓		✓	✓
select hoisting	✓	✓	✓		
easy movement			✓	✓	✓

(assuming branch on poison is UB)

GVN vs Loop unswitching

```
while (c) {
  if (c2) { foo }
  else    { bar }
}
```



```
if (c2) {
  while (c) { foo }
} else {
  while (c) { bar }
}
```

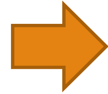
Loop unswitch

Branch on poison **cannot** be UB

Otherwise, wrong if loop never executed

GVN vs Loop unswitching

```
t = x + 1;
if (t == y) {
  w = x + 1;
  foo(w);
}
```



```
t = x + 1;
if (t == y) {
  foo(y);
}
```

Contradiction with loop unswitching!

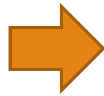
GVN

Branch on poison **must** be UB

Otherwise, wrong if y poison but not x

Fixing loop unswitch

```
while (c) {
  if (c2) { foo }
  else    { bar }
}
```



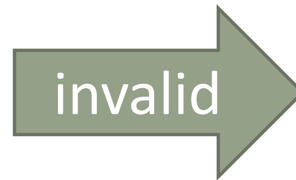
```
if (freeze(c2)) {
  while (c) { foo }
} else {
  while (c) { bar }
}
```

GVN doesn't need any change!

Malloc and pointer comparison

- Pointer comparison should move freely
- It's only valid to compare pointers with overlapping liveness ranges
- Potentially illegal to trim liveness ranges

```
char *p = malloc(4);
char *q = malloc(4);

// valid
if (p == q) { ... }

free(p);
```




```
char *p = malloc(4);
free(p);

char *q = malloc(4);

// poison
if (p == q) { ... }
```

There’s no perfect semantics

- Must balance tradeoffs
- Enable most useful optimizations (which?)
- Change semantics throughout the pipeline
  - Later stages do different optimizations
  - May confuse compiler developers?

UB is not only for unsafe languages

- 1/8 of additions in Swift have nsw/nuw attributes
- If a language is memory safe: use UB in memory operations
- Type-based alias analysis

- Actually: the safer the language, the more assumptions can be given by the front-end!

UB is awesome, but…

UB May Result in Security Vulnerabilities

NaCl/x86 appears to leave return addresses unaligned when returning through the springboard.

Project Member Reported by cbiffle@google.com, Jan 13 2010

```
When returning from trusted to untrusted code, we must sanitize the return
address before taking it.  This ensures that untrusted code cannot use the
syscall interface to vector execution to an arbitrary address.  This role
is entrusted to the function NaClSandboxAddr, in sel_ldr.h.  Unfortunately,
since r572, this function has been a no-op on x86.


-- What happened?

During a routine refactoring, code that once read
  aligned_tramp_ret = tramp_ret & ~(nap->align_boundary - 1);

was changed to read
  return addr & ~(uintptr_t)((1 << nap->align_boundary) - 1);

Besides the variable renames (which were intentional and correct), a shift
was introduced, treating nap->align_boundary as the log2 of bundle size.

We didn't notice this because NaCl on x86 uses a 32-byte bundle size.  On
x86 with gcc, (1 << 32) == 1.  (I believe the standard leaves this behavior
```

```
This change had four listed reviewers and was explicitly LGTM'd by two.
Nobody appears to have noticed the change.
```

Preventing accidental UB

- Formal spec in K – covers most UB in C99 spec
- Clang Ubsan: -fsanitize=undefined – runtime verification; covers many things that compilers exploit at the moment
- GCC 4.9+ also includes a version of ubsan

Compiler developers also get confused

"Every transformation above seems of no problem, but the composition result is wrong. It is still not clear which transformation to blame."
— LLVM developer

Result: LLVM miscompiled itself!

72239989 Clang++ Compilation Optimization Bug

```
int kOne = 1;

__attribute__((weak))
void check(int x) {
  if (x != 9) {
    printf("ERROR: x = %d\n", x);
  }
  exit(0);
}

__attribute__((weak))
void buggy(void) {
  unsigned char dst[1] = {42};
  unsigned char src[1] = {9};
  unsigned char *dp = dst;
  unsigned char *sp = src;

  while (1) {
    if (kOne) {
      dp[0] = 9;
    } else {
      memcpy(dp, sp, 16);
      sp += 16;
      dp += 16;
    }
    check(dst[0]);
  }
```

PR36228: miscompiles Android: API usage mismatch between AA and AliasSetTracker

PR34548: incorrect Instcombine fold of inttoptr/ptrtoint

```
Example of an end-to-end miscompilation by clang

$ cat c.c
#include <stdio.h>

void f(int*, int*);

int main()
{
  int a=0, y[1], x = 0;
  uintptr_t pi = (uintptr_t) &x;
  uintptr_t yi = (uintptr_t) (y+1);
  uintptr_t n = pi != yi;

  if (n) {
    a = 100;
    pi = yi;
  }

  if (n) {
    a = 100;
    pi = (uintptr_t) y;
  }

  *(int *)pi = 15;

  printf("a=%d x=%d\n", a, x);

  f(&x,y);

  return 0;
}
```

```
pub fn test(gp1: &mut usize, gp2: &mut usize, b1:
bool, b2: bool) -> (i32, i32) {
  let mut g = 0;
  let mut c = 0;
  let y = 0;
  let mut x = 7777;
  let mut p = &mut g as *const _;

  {
    let mut q = &mut g;
    let mut r = &mut 8888;

    if b1 {
      p = (&y as *const _).wrapping_offset(1);
    }

    if b2 {
      q = &mut x;
    }

    *gp1 = p as usize + 1234;
    if q as *const _ == p {
      c = 1;
      *gp2 = (q as *const _) as usize + 1234;
      r = q;
    }
    *r = 42;
  }
  return (c, x);
}
```

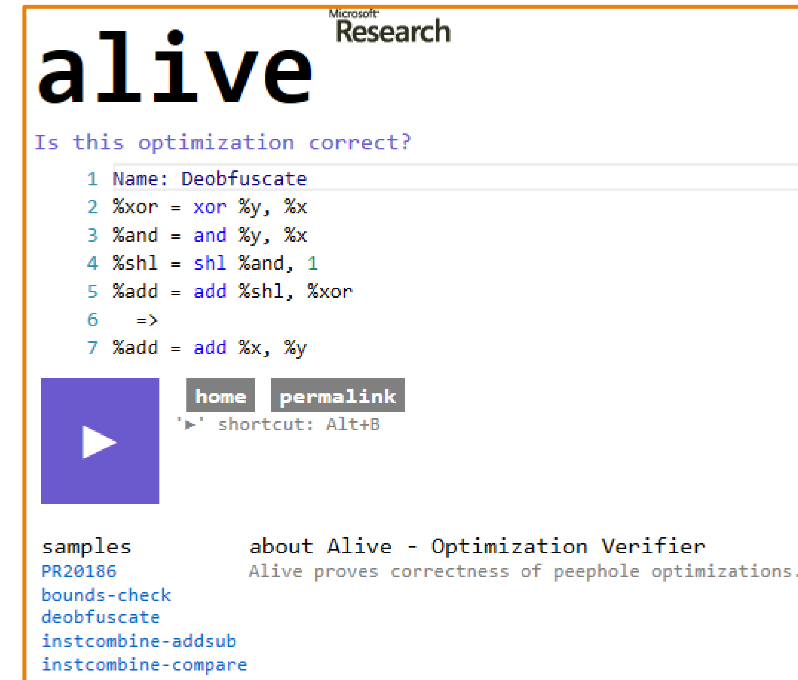
Safe Rust program miscompiled by GVN

The need for tools

- We need tools to validate conformance with specified semantics
- Otherwise:
  - Bugs in the compiler
  - Specification is worthless
  - Impossible to ensure correctness of spec & tools
- Anecdote:
  - First run of Alive2 TV found a bug in LLVM’s own unit tests!
  - Unit test was checking for an invalid optimization

The need for tools: select in LLVM

- Select has many possible reasonable semantics
- We found all of them implemented in different parts of LLVM
- We did exhaustive test case generation
  - ~45M programs of 3 instructions & 2-bits
  - Found dozens of bugs in LLVM
  - Found dozens of semantics inconsistencies
- Secret: We choose one semantics and implemented in on-line Alive, which is now the *de facto* oracle



Compiler bugs may be security hazards

3 Deniable Backdoors Using Compiler Bugs

by Scott Bauer, Pascal Cuoq, and John Regehr

Do compiler bugs cause computer software to become insecure? We don't believe this happens very often in the wild because (1) most code is not miscompiled and (2) most code is not security-critical. In this article we address a different situation: we'll play an adversary who takes advantage of a naturally occurring compiler bug.

Do production-quality compilers have bugs?

ery new tool tends to find different bugs. This has been demonstrated recently by running **`afl-fuzz`** against Clang/LLVM.[3] A final way to get good compiler bugs is to introduce them ourselves by submitting bad patches. As that results in a "Trusting Trust" situation where almost anything is possible, we won't consider it further.

So let's build a backdoor! The best way to do

Conclusions

- There are no demons in UB!

- Undefined behavior is an efficient way for:
  - Passing assumptions from the front-end to optimizers
  - Cache derived facts
  - Restrict domain of functions

- Further research needed to find better IRs