The Whacky World of Undefined Behaviour

Beck, Calvin
hobbes@seas.upenn.edu

October 17, 2019

What is this Talk about?

Undefined behaviour! With a smack of LLVM.

We'll cover things like:

- What is undefined behaviour?
- What happens when you encounter UB?
- How is UB useful? Should we avoid it?
  - Optimizations?
- UB in LLVM (and indeterminate values)
- How this all fits into Vellvm

Not for anything in particular! It's just a fun topic, and hopefully talking about it will clarify some things for myself and you!

What is undefined behaviour (UB)?

It's behaviour...

That's undefined.

Done.

What is undefined behaviour (UB)?

Why does this seem hard?

- Easy to conflate with things like implementation defined behaviour... Which is sort of different.
- Language dependent.
  - Array out of bounds in Python? Exception, not UB.
  - Array out of bounds in C? ... Pray.

What happens when you encounter UB?

ANYTHING.

Yes. Anything.

What happens when you encounter UB?

Compiler will do whatever it finds easiest or most efficient.

- noop, and then continue
- halt
- halt **and** catch fire
- erase the hard drive
  - No, seriously. Erase the hard drive.
  - `https://kristerw.blogspot.com/2017/09/why-undefined-behavior-may-call-never.html`
- time travel
  - no, really.
  - `https://devblogs.microsoft.com/oldnewthing/?p=633`
- nasal demons?
  - `https://en.wikipedia.org/wiki/Nasal_demons`
  - So far I'm pretty sure this is just a joke, but I wouldn't rule it out.

Why is this useful?

Why have UB at all? Isn't it... crazy?

- PL without UB might have a lot of dynamic sanity checks.
  - Bounds checking.
- What about type systems?
  - Static checks can eliminate some dynamic checks
  - Bounds checking still common.

Instead, why not...

Do nothing?

Why is this useful?

UB may seem somewhat unprincipled, but it has advantages:

- Gives compiler an axiom.
- Puts burden on programmer, or other tools

UB can reflect programmer intent

I want to change this...

```
a + b < a + c
```

To this:

```
b < a
```

After all, who wants to do 2 extra additions?

But this is sort of wrong...

```
1 + INT_MAX < 1 + 3
// This evaluates to
INT_MIN < 4 == True

// But...
INT_MAX < 3 == False
```

Pointer aliasing — optimizations with undecidability

Undefined behavour allows compilers to make optimizations based on undecidable invariants.

It might be nice to optimize this:

```
void sum(double *total, double *array, size_t len )
{
    *total = 0;
    for (size_t i=0; i<len; i++) {
        *total += array[i];
    }
}
```

To this:

```
void sum(double *total, double *array, size_t len )
{
    double local_total = 0;
    for (size_t i=0; i<len; i++) {
        local_total += array[i];
    }

    *total = local_total;
}
```

C's restrict keyword

```
void sum(double* restrict total, double* restrict array, size_t len )
{
    *total = 0;
    for (size_t i=0; i<len; i++) {
        *total += array[i];
    }
}
```

Allows optimization to

```
void sum(double *total, double *array, size_t len )
{
    double local_total = 0;
    for (size_t i=0; i<len; i++) {
        local_total += array[i];
    }

    *total = local_total;
}
```

Because restrict says "these pointers don't alias with anything". If they happen to alias, then it's UB, and the program can do whatever.

WHERE WE'RE GOING WE DON'T NEED ROADS

So, how powerful is undefined behaviour?

Can we optimize this:

```
char inp = getchar();
if ('A' == inp) {
    printf("Hello, world!\n");
    x = 1 / 0;
}
```

To this?

```
getchar();
```

Or maybe just this?

```
char inp = getchar();
if ('A' == inp) {
    printf("Hello, world!\n");
}
```

Does UB mean **cannot** happen or **anything** can happen?

Depends who you ask

Supposedly the C++ standard says this:

```
However, if any such execution contains an undefined operat
this International Standard places no requirement on the
implementation executing that program with that input
(not even with regard to operations preceding the
first undefined operation).
```

So, C++ seems to explicitly allow this kind of time travel.

In general UB seems to be somewhat of an underspecified concept.
Both options seem reasonable, depending on what you want to do.

Undefined behaviour in IR

Now we know a few things about UB.

- Seems to be useful for optimizations.
- With languages like C, passes burden onto the programmer :(.

Whether or not you like UB in a programming language... It seems to make some sense for an IR.

- Again, optimizations
- Way to pass down invariants from higher levels
  - Either the programmer proves something (or pretends to have proved something)
  - Type systems when the types get erased
  - Other external tools / checkers
    - Maybe you even use something like Coq to verify your C. Why not get better optimizations in that case?

Expressive UB in IRs?

Why not have a lot of control over exactly what counts as UB?

LLVM Has some of this already:

```
<result> = add <ty> <op1>, <op2>
<result> = add nuw <ty> <op1>, <op2>
<result> = add nsw <ty> <op1>, <op2>
<result> = add nuw nsw <ty> <op1>, <op2>
```

Lets compilers pick the option that best suits the situation.

Indeterminate values in LLVM

LLVM has indenterminate values in the form of `undef` and `poison`.

- Not undefined behaviour themselves
  - But, closely related
- "it doesn't matter what value this has"
- "this value won't be used"

`undef` and `poison` are sort of messy. We're doing our best to describe what we believe LLVM to be in Vellvm, but it's not always clear with these two.

Undef

`undef` is more like “undefined value” than “undefined behaviour”.

- Unspecified / arbitrary bit pattern
  - uninitialized variables
  - loads from uninitialized memory

Undef's laziness

`undef` is a little weird.

```
%x = i32 undef
%y = add i32 %x %x
```

Is the same as this:

```
%x = i32 undef
%y = i32 undef   ;; Not just even numbers
```

Why?

`undef` isn't just "I don't care what value this variable has" it's more like "I don't even care if this variable has a specific value."

No need to save an arbitrary bit pattern if it shouldn't change the behaviour of the program.

Undef gets a little weirder...

But `undef` does not always beget undef...

```
%x = i32 undef
%y = mul i32 %x 2
```

What's `y`?

It's also an indeterminate value. The set of all even numbers, and...

```
%z = add i32 %y %y
```

Also the set of even numbers. Not some multiple of 4. LLVM again allows `y` to have a different value each time, but it must be a multiple of 2.

How is undef useful?

```
int x;
if (cond) {
    x = f();
}

if (cond2) {
    g(x);
}
```

Why force a value for `x`? What if $\mathrm{cond2} \implies \mathrm{cond}$?

We can assume `x` is assigned before use, and avoid an arbitrary assignment to `x`.

undef under Vellvm

Keep full `undef` expressions to avoid early concretization.

```
(undef + 2) * (3 * undef)  // Left in this complicated form
```

Makes interpreter / model much more complicated.

We have "pick" events whenever we have to concretize `undef`, when side-effecting operations occur. Even something like `div` forces pick events.

Spreading poison

Sometimes we need something a little stronger than `undef`: `poison`.

- Can always relax `poison` to `undef`.
- Poison is much simpler.
- Operations on poison result in poison
- If you use poison with something that causes side-effects you get UB.

How can we use poison?

poison is really useful as a kind of deferred undefined behaviour. Can use it for a kind of speculative execution.

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

Can we optimize this? Maybe to this?

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

Want to delay UB from $x + 1$, so LLVM gives overflows the value of poison.

More uses of poison

Similarly, can we optimize this?

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

To this?

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

Poisoning Vellvm

Poison in Vellvm is much simpler than `undef`. Basically just have operations return a `poison` if they're given `poison`.

Certain operations on poison will trigger UB.

poison vs undef

- Don't want to get too hung up on this, because it is confusing.
- `poison` and `undef` are similar, but slightly different
- Justify different optimizations

```
a + 1 > a
```

To...

```
true // Or I guess 1 in C...
```

only works with poison for $a + 1$ overflowing.

Refinement and Vellvm

```
(* Refinement relation for uvalues *)
Inductive refine_uvalue: uvalue -> uvalue -> Prop :=
| UndefPoison: forall t, refine_uvalue (UVALUE_Undef t)
| RefineConcrete: forall uv1 uv2, (forall dv, concretiz
.

(* Refinement of uninterpreted mcfg *)
Definition refine_L0: relation (itree L0 uvalue) := eut

(* Refinement of mcfg after globals *)
Definition refine_res1 : relation (global_env * uvalue)
  := TT x refine_uvalue.

(* ... *)

  (* Refinement for after interpreting pick. *)
Definition refine_L4 : relation ((itree L4 (memory * (I
  := fun ts ts' => forall t, ts t -> exists t', ts' t'
```

Better future

More clear semantics for undef / poison? Freeze thaw?

- remove undef. Only poison.
- freeze of poison is nondeterministic choice.

References


In: ().


.


.

These are all good resources! You should look at them!