How defined is Rust?

Florian “Florob” Zeitz

2019-08-07

Previously

- this talk is based around an older one “What you thought you knew about C”
- stripped down the C parts
- added Rust’s perspective
- not meant to be C bashing/zealotry
- if you care about C look on media.ccc.de:
  - sigint12 (english)
  - CCCAC 2015 (german)

Disclaimer: What are we talking about?

- C99 and/or C11
- not necessarily C++
- but Objective-C, as it works as a real superset
- Rust as of today (many things still being discussed)

What's this?

Multiple types of "behaviour":

implementation-defined behaviour

documented implementation choice (e. g. signedness of `char`)

What's this?

Multiple types of "behaviour":

implementation-defined behaviour

documented implementation choice (e. g. signedness of `char`)

unspecified behaviour

more than one possibility (e. g. evaluation of function arguments)

What's this?

Multiple types of "behaviour":

implementation-defined behaviour

documented implementation choice (e. g. signedness of `char`)

unspecified behaviour

more than one possibility (e. g. evaluation of function arguments)

undefined behaviour

everything goes, input program is considered erroneous (e. g. use-after-free)

The "evil compiler (writers)"

- the compiler looks for undefined behavior to subsequently break your code
- all in the name of faster benchmarks
- don't care about normal users, just numbers

The "evil compiler (writers)"

- the compiler looks for undefined behavior to subsequently break your code
- all in the name of faster benchmarks
- don't care about normal users, just numbers
- this is (usually) not how things work
- compilers never look for undefined behavior
- optimizations work under the assumption there is no undefined behavior
- we'll see some examples

What does this snippet usually print?
(optimized)

```
1  int a, b;
2  if (a) {
3    a = 3;
4  } else {
5    b = 4;
6  }
7  printf("%i\n", a + b);
```

3

4

5

7

What does this snippet usually print?
(optimized)

```
1  int a, b;
2  if (a) {
3    a = 3;
4  } else {
5    b = 4;
6  }
7  printf("%i\n", a + b);
```

3

4

5

7

Uninitialized in safe Rust

- Impossible in safe Rust
- compiler forbids usage of possibly uninitialized variables

Uninitialized in unsafe Rust


```
1  fn main() {
2    let mut a = MaybeUninit::<i32>::uninit();
3    let mut b = MaybeUninit::<i32>::uninit();
4
5    if std::env::var("A").is_ok() {
6        unsafe { a.as_mut_ptr().write(3) };
7    } else {
8        unsafe { b.as_mut_ptr().write(4) };
9    }
10
11   println!(
12     "{}",
13     unsafe { a.assume_init() + b.assume_init() }
14   );
15 }
```

What does this snippet usually print when size is `INT_MAX`? (Optimized with -O3)

```
1 int size = ...;
2 if (size > size+1) {
3   puts("Aborted")
4   abort();
5 }
6 puts("Fetching memory");
7 malloc(size+1);
```

Nothing

"Aborted"

"Fetching memory"

size

What does this snippet usually print when size is `INT_MAX`? (Optimized with -O3)

```
1 int size = ...;
2 if (size > size+1) {
3   puts("Aborted")
4   abort();
5 }
6 puts("Fetching memory");
7 malloc(size+1);
```

Nothing

"Aborted"

"Fetching memory"

size

Signed integer overflow (C)

- unsigned integer overflow is well-defined: `UINT_MAX + 1 == 0`
- signed integer overflow is not: `INT_MAX + 1 == /* undef */`
- rumours aside `INT_MAX + 1` is not `INT_MIN`
- Check equality against `INT_MAX`

```
1  int size = ...;
2  if (size > size+1) {
3    puts("Aborted")
4    abort();
5  }
6  puts("Fetching memory");
7  malloc(size+1);
```

- Only defined behavior is considered
- `size > size + 1` is always false
- Optimization removes the branch

Signed integer overflow (Rust)

- same behaviour for all integer types, signed and unsigned
- debug: panic on overflow
- release: wrap around on overflow
- individual methods for specific requirements:
  - `checked_add()`
  - `saturating_add()`
  - `wrapping_add()`
  - `overflowing_add()`

What does this snippet usually print?
(Unoptimized, on an x86 system)

```
1 uint32_t shifty = 1;
2 shifty = shifty << 32;
3 printf("%"PRIu32"\n", shifty);
```

0

1

42

neither

What does this snippet usually print?
(Unoptimized, on an x86 system)

```
1 uint32_t shifty = 1;
2 shifty = shifty << 32;
3 printf("%"PRIu32"\n", shifty);
```

0	1
42	neither

Oversized shift amounts (C)

> *If the value of the right operand is negative or is greater than or equal to the width of the promoted left operand, the behavior is undefined.*

- set variables to zero instead
- easily checked when type width is known

Oversized shift amounts (Rust)

- debug: panic on oversized shift amount
- release: mask right operand to bit width
- individual methods for specific requirements:
  - `checked_shl()`
  - `wrapping_shl()`
  - `overflowing_shl()`

What does this snippet usually print?
(Optimized, clang or gcc)

```
1 signed int s = -1;
2 unsigned int u = 1;
3 if (s < u)
4   puts("True");
5 else
6   puts("False");
```

"True"

Nothing

"False"

"Trlse"

What does this snippet usually print?
(Optimized, clang or gcc)

```
1 signed int s = -1;
2 unsigned int u = 1;
3 if (s < u)
4   puts("True");
5 else
6   puts("False");
```

"True"

Nothing

"False"

"Trlse"

What does this snippet print?

```
1 unsigned int u = 1;
2 signed int s1 = -2;
3 signed int s2 = u + s1;
4 printf("%i\n", s2);
```

-1

0

4294967295

1

What does this snippet print?

```
1 unsigned int u = 1;
2 signed int s1 = -2;
3 signed int s2 = u + s1;
4 printf("%i\n", s2);
```

- -1
- 0
- 4294967295
- 1

Usual arithmetic conversions

Applied for certain operations:

- multiplicative (`*`, `/`, `%`)
- additive (`+`, `-`)
- relational (`<`, `>`, `<=`, `>=`)
- equality (`==`, `!=`)
- bitwise (`&`, `|`, `^`)
- conditional (`a ? b : c`, only to the second and third operand)

Usual arithmetic conversions - Example


```
1 unsigned int a = 1;
2 signed int b = -1, c = a + b;
3 if (a > b) printf("True\n");
```

- `a` and `b` have the same rank
- For both `+` and `>`, `b` is converted to **`unsigned int`**
- Effect: `a > b` is false
- $a + b_{\text{signed}} \equiv a + b_{\text{unsigned}} \pmod{(\texttt{UINT\_MAX} + 1)}$, hence `a + b` is 0

The problem

- depends on whom you ask
- the rules for integer promotions and usual arithmetic conversions
- having implicit conversions at all (cf. problems in JavaScript)

Coercions and Rust

- Rust has almost no coercions
- those that exist are between lifetimes not types

What does this snippet usually print?
(Optimized, clang or gcc)

```c
void f(int *i, float *f) {
  *i = 42;
  *f = 13;
  printf("%i\n", *i);
}
```

```c
int main(void) {
  int var;
  f(&var, &var);
  return 0;
}
```

42

13

0

1095761920

What does this snippet usually print?
(Optimized, clang or gcc)

```c
void f(int *i, float *f) {
  *i = 42;
  *f = 13;
  printf("%i\n", *i);
}
```

```c
int main(void) {
  int var;
  f(&var, &var);
  return 0;
}
```

42

13

0

1095761920

Strict Aliasing Rule

- C allows aliasing
- `int *pa = &a`, `*pa` aliases `a`
- not all expressions may be used to access an object
- expression and object type must match
- this restriction is commonly called the *strict aliasing rule*
- with `a` declared as a `float`, `*pa` may be neither read nor written

Exceptions

- different signedness
- different qualifiers
- struct, array or union type with a member of one of the aforementioned types
- character type

Why have this rule?

- makes it harder to create trap/hiche values
- avoids unaligned writes
- restricts aliasing (potential for optimizations)

(Strict) Aliasing and Rust

- type punning is not possible in safe Rust
- generally Rust's mutability XOR aliasing system gives stronger guarantees than this rule
- arbitrary aliasing is possible in unsafe Rust
- discussion is focussing more on maintaining mutability XOR aliasing and provenance
- maybe we don't need (purely) type based aliasing rules in Rust?
- certainly not for optimization

How many possible results does this function have?

```
1  int f(signed int *i1, unsigned int *i2, float *f, char *c) {
2    *i1 = 42;
3    *i2 = 43;
4    *f = 13.;
5    *c = 1;
6    return *i1 + *i2 + *f + *c;
7  }
```

19

1

2^{104}

214

How many possible results does this function have?

```
1  int f(signed int *i1, unsigned int *i2, float *f, char *c) {
2    *i1 = 42;
3    *i2 = 43;
4    *f = 13.;
5    *c = 1;
6    return *i1 + *i2 + *f + *c;
7  }
```

- 19
- 1
- 2^{104}
- 214

```c
int f(signed int *i1,
      unsigned int *i2,
      float *f,
      char *c)
{
  *i1 = 42;
  *i2 = 43;
  *f = 13.;
  *c = 1;
  return *i1 + *i2 + *f + *c;
}
```

```
.LCPI0_0:
  .long 1065353216 # float 1
f:
  movl $42, (%rdi)
  movl $43, (%rsi)
  movl $1095761920, (%rdx)
  movb $1, (%rcx)
  movl (%rsi), %eax
  addl (%rdi), %eax
  cvtsi2ssq %rax, %xmm0
  addss (%rdx), %xmm0
  addss .LCPI0_0(%rip), %xmm0
  cvttss2si %xmm0, %eax
  retq
```

restrict

- C99 added the `restrict` qualifier
- can only be applied to pointer types
- the pointee may only be accessed via an expression based on the pointer
- restricts aliasing

```c
int f(signed int *restrict i1,
      unsigned int *restrict i2,
      float *restrict f,
      char *restrict c)
{
  *i1 = 42;
  *i2 = 43;
  *f = 13.;
  *c = 1;
  return *i1 + *i2 + *f + *c;
}
```

```asm
f:
  movl $42, (%rdi)
  movl $43, (%rsi)
  movl $1095761920, (%rdx)
  movb $1, (%rcx)
  movl $99, %eax
  retq
```

`&mut T`

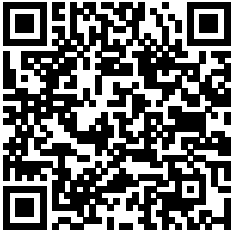
- Rust's `&mut T` has stronger guarantees then **`restrict`**
- no access via any other reference
- enables even more optimizations

&mut T


```rust
fn f(
    i1: &mut i32, i2: &mut u32,
    f: &mut f32, c: &mut i8,
) -> i32 {
    *i1 = 42;
    *i2 = 43;
    *f = 13.;
    *c = 1;
    (
        *i1 as f32 + *i2 as f32
            + *f + *c as f32
    ) as i32
}
```

```asm
f:
    movl $42, (%rdi)
    movl $43, (%rsi)
    movl $1095761920, (%rdx)
    movb $1, (%rcx)
    cvtsi2ssl (%rdi), %xmm0
    movl (%rsi), %eax
    cvtsi2ssq %rax, %xmm1
    addss %xmm0, %xmm1
    addss (%rdx), %xmm1
    addss .LCPI4_0(%rip), %xmm1
    cvttss2si %xmm1, %eax
    retq
```

Thank you for your attention. Any questions?



https://babelmonkeys.de/~florob/talks/RC-2019-08-07-rust-defined.pdf