Undefinedness and Non-determinism in C

Nabil M. Al-Rousan

Nov. 21, 2018 @ UBC

Based on slides from Robbert Krebbers
Aarhus University, Denmark
What is this program supposed to do?
The C quiz, question 1

```c
int main() {
    int x;
    int y = (x = 3) + (x = 4);
    printf("x=%d,y=%d\n", x, y);
}
```
What is this program supposed to do?
The C quiz, question 1

```c
int main() {
    int x;
    int y = (x = 3) + (x = 4);
    printf("x=%d,y=%d\n", x, y);
}
```

Let us try some compilers

- **Clang** prints $x=4,y=7$, seems just left-right
What is this program supposed to do?

The C quiz, question 1

```c
int main() {
    int x;
    int y = (x = 3) + (x = 4);
    printf("x=%d,y=%d\n", x, y);
}
```

Let us try some compilers

- Clang prints `x=4,y=7`, seems just left-right
- GCC prints `x=4,y=8`, does not correspond to any order

This program violates the sequence point restriction due to two unsequenced writes to `x` resulting in undefined behavior; thus both compilers are right.
What is this program supposed to do?
The C quiz, question 1

```c
int main() {
    int x;
    int y = (x = 3) + (x = 4);
    printf("x=%d,y=%d\n", x, y);
}
```

Let us try some compilers

- **Clang** prints `x=4,y=7`, seems just left-right
- **GCC** prints `x=4,y=8`, does not correspond to any order

This program violates the sequence point restriction

- due to two unsequenced writes to `x`
- resulting in **undefined behavior**
- thus both compilers are right
Underspecification in C11

- **Unspecified behavior**: two or more behaviors are allowed
  For example: order of evaluation in expressions (+57 more)

- **Implementation defined behavior**: like unspecified behavior, but the compiler has to document its choice
  For example: size and endianness of integers (+118 more)

- **Undefined behavior**: the standard imposes no requirements at all, the program is even allowed to crash
  For example: dereferencing a NULL or dangling pointer, signed integer overflow, ... (+201 more)
Underspecification in C11

- **Unspecified behavior**: two or more behaviors are allowed
  
  *For example: order of evaluation in expressions* (+57 more)
  
  Non-determinism

- **Implementation defined behavior**: like unspecified behavior, but the compiler has to document its choice
  
  *For example: size and endianness of integers* (+118 more)

- **Undefined behavior**: the standard imposes no requirements at all, the program is even allowed to crash
  
  *For example: dereferencing a NULL or dangling pointer, signed integer overflow, ...* (+201 more)

  No semantics/crash state
Why does C use underspecification that heavily?

**Pros** for optimizing compilers:
- More optimizations are possible
- High run-time efficiency
- Easy to support multiple architectures

**Cons** for programmers/formal methods people:
- Portability and maintenance problems
- Hard to capture precisely in a semantics
- Hard to formally reason about
Why does C use underspecification that heavily?

**Pros** for optimizing compilers:
- More optimizations are possible
- High run-time efficiency
- Easy to support multiple architectures

**Cons** for programmers/formal methods people:
- Portability and maintenance problems
- Hard to capture precisely in a semantics
- Hard to formally reason about
Approaches to underspecification

- **CompCert** (Leroy *et al.*) / **VST** (Appel *et al.*)
  - Main goal: verification of/w.r.t. CompCert compiler in Coq
  - Semantics only needs to be correct for CompCert compiler
    *For example: integer overflow and aliasing violations not UB*

- **KCC** (Ellison & Rosu, Hathhorn *et al.*)
  - Main goal: compiler independent C11 semantics in **K**
  - Describes *most* unspecified and undefined behavior
  - No proof assistant support

- **CH₂O** (Krebbers & Wiedijk)
  - Main goal: compiler independent C11 semantics in Coq
  - Describes *all* unspecified and undefined behavior
  - Describes *some* implementation-defined behavior
    *For example: no legacy architectures with 1s' complement*

- **Cerberus** (Sewell *et al.*)
  - Main goal: ‘defacto’ C11 semantics in LEM
  - Improve standard to match the way C is used in practice
Strict-aliasing
What is aliasing?

**Aliasing:** multiple pointers referring to the same object

```c
int x, *p = &x, *q = &x; // p and q are aliased
```
Strict-aliasing

What is aliasing?

**Aliasing**: multiple pointers referring to the same object

```c
int x, *p = &x, *q = &x; // p and q are aliased
```

Tricky with functions:

```c
int f(int *p, int *q) {
    int x = *q; *p = 17; return x;
}
```

If `p` and `q` alias, the original value `n` of `*p` is returned
Strict-aliasing

What is aliasing?

**Aliasing**: multiple pointers referring to the same object

```c
int x, *p = &x, *q = &x; // p and q are aliased
```

Tricky with functions:

```c
int f(int *p, int *q) {
    int x = *q; *p = 17; return *p * q;
}
```

If `p` and `q` alias, the original value `$n$` of `$p$` is returned

```
    n
  ________________
     |             |
   p   q
```

Eliminating `x` is unsound: `17` would be returned
Strict-aliasing

Alias analysis

**Alias analysis:** to determine whether pointers can alias
Strict-aliasing
Alias analysis

**Alias analysis**: to determine whether pointers can alias

Consider a similar function:

```c
short g(int *p, short *q) {
    short x = *q; *p = 17; return x;
}
```
Strict-aliasing

Alias analysis

**Alias analysis**: to determine whether pointers can alias

Consider a similar function:

```c
short g(int *p, short *q) {
    short x = *q; *p = 17; return x;
}
```

And call it with aliased pointers:

```c
union { int x; short y; } u;
u.y = 3;
g(&u.x, &u.y);
```
Strict-aliasing

Alias analysis

**Alias analysis:** to determine whether pointers can alias

Consider a similar function:

```c
short g(int *p, short *q) {
    short x = *q; *p = 17; return x;
}
```

And call it with aliased pointers:

```c
union { int x; short y; } u;
    u.y = 3;
    g(&u.x, &u.y);
```

C99/C11 allow **type-based alias analysis:** reads/writes with "the wrong type" cause undefined behavior

⇒ A compiler can **assume** that p and q do not alias
## Strict-aliasing

How to treat pointers

<table>
<thead>
<tr>
<th>Others (e.g. CompCert)</th>
<th>Our approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory: a finite map of objects which consist of <strong>arrays</strong> of bytes</td>
<td><strong>Too little information to formalize strict-aliasing</strong></td>
</tr>
<tr>
<td>Pointers: pairs ((o, i)) where (o) identifies the object, and (i) the <strong>offset</strong> into that object</td>
<td></td>
</tr>
</tbody>
</table>
## Strict-aliasing

### How to treat pointers

<table>
<thead>
<tr>
<th>Others (e.g. CompCert)</th>
<th>Our approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory: a finite map of objects which consist of arrays of bytes</td>
<td>A finite map of objects which consist of well-typed trees of bits</td>
</tr>
<tr>
<td>Pointers: pairs ((o, i)) where (o) identifies the object, and (i) the offset into that object</td>
<td></td>
</tr>
<tr>
<td>Too little information to formalize strict-aliasing</td>
<td></td>
</tr>
</tbody>
</table>
## Strict-aliasing

### How to treat pointers

<table>
<thead>
<tr>
<th>Others (e.g. CompCert)</th>
<th>Our approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory: a finite map of objects which consist of <strong>arrays</strong> of bytes</td>
<td>A finite map of objects which consist of well-typed <strong>trees</strong> of bits</td>
</tr>
<tr>
<td>Pointers: pairs ((o, i)) where (o) identifies the object, and (i) the <strong>offset</strong> into that object</td>
<td>Pairs ((o, r)) where (o) identifies the object, and (r) the <strong>path through the tree</strong> in that object</td>
</tr>
<tr>
<td>Too little information to formalize strict-aliasing</td>
<td></td>
</tr>
</tbody>
</table>
### Strict-aliasing

**How to treat pointers**

<table>
<thead>
<tr>
<th>Others (e.g. CompCert)</th>
<th>Our approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory: a finite map of objects which consist of <strong>arrays</strong> of bytes</td>
<td>A finite map of objects which consist of well-typed <strong>trees</strong> of bits</td>
</tr>
<tr>
<td>Pointers: pairs ((o, i)) where (o) identifies the object, and (i) the <strong>offset</strong> into that object</td>
<td>Pairs ((o, r)) where (o) identifies the object, and (r) the <strong>path through the tree</strong> in that object</td>
</tr>
<tr>
<td><strong>Too little information to formalize strict-aliasing</strong></td>
<td><strong>A semantics for strict-aliasing</strong></td>
</tr>
</tbody>
</table>
Strict-aliasing
Example of the memory as a structured forest

Consider:

```c
struct S {
    union U {
        signed char x[2]; int y;
    } u;
    void *p;
} s = { { .x = {33,34} }, s.u.x + 2 }
```

The object in memory looks like:
Problem: Type-Punning

```c
union int_or_short { int x; short y; } u = { .x = 3 }; printf("%d\n", u.y);

union int_or_short { int x; short y; } u = { .x = 3 }; short *p = &u.y; printf("%d\n", *p);
```
Theorem (Strict-aliasing)

Given:

- \( \text{addresses } \Gamma, \Delta \vdash a_1 : \sigma_1 \text{ and } \Gamma, \Delta \vdash a_2 : \sigma_2 \)
- with annotations that do not allow type-punning
- \( \sigma_1, \sigma_2 \neq \text{unsigned char} \)
- \( \sigma_1 \) not a subtype of \( \sigma_2 \) and vice versa

Then there are two possibilities:

1. \( a_1 \) and \( a_2 \) do not alias
2. accessing \( a_1 \) after \( a_2 \) (and vice versa) is undefined
Theorem (Strict-aliasing)

Given:

- addresses $\Gamma, \Delta \vdash a_1 : \sigma_1 \text{ and } \Gamma, \Delta \vdash a_2 : \sigma_2$
- with annotations that do not allow type-punning
- $\sigma_1, \sigma_2 \neq$ unsigned char
- $\sigma_1$ not a subtype of $\sigma_2$ and vice versa

Then there are two possibilities:

1. $a_1$ and $a_2$ do not alias
2. accessing $a_1$ after $a_2$ (and vice versa) is undefined

Corollary

Compilers can perform type-based alias analysis
CH₂O abstract C

\[ x \in \text{string} := \text{Set of strings} \]
\[ k \in \text{cintrank} := \text{char} | \text{short} | \text{int} \]
\[ \quad | \text{long} | \text{long long} | \text{ptr} \]
\[ si \in \text{signedness} := \text{signed} | \text{unsigned} \]
\[ \tau_i \in \text{cinttype} := si? k \]
\[ \tau \in \text{ctype} := \text{void} | \text{def} x \ | \tau_i \ | \tau \ast \]
\[ \quad | \tau x \rightarrow \tau \ | \tau[e] \]
\[ \quad | \text{struct} x \ | \text{union} x \]
\[ \quad | \text{enum} x \ | \text{typeof} e \]
\[ e \in \text{cexpr} := x \ | \text{const}_{\tau_i} z \ | \text{string} \bar{z} \]
\[ \quad | \text{sizeof} \tau \ | \text{alignof} \tau \]
\[ \quad | \text{offsetof} \tau x \]
\[ \quad | \tau \text{min} \ | \tau \text{max} \ | \tau \text{bits} \]
\[ \quad | \&e \ | \ast e \ | e . x \]
\[ \quad | e_1 \alpha e_2 \]
\[ \quad | e(e') \ | \text{alloc}_{\tau} e \ | \text{free} e \]
\[ \quad | \odot_u e \ | e_1 \odot e_2 \ | (\tau)l \]
\[ \quad | e_1 \&\& e_2 \ | e_1 \mid e_2 \]
\[ \quad | (e_1, e_2) \ | e_1 ? e_2 : e_3 \]
\[ r \in \text{crefseg} := [e] \ | .x \]
\[ l \in \text{cinit} := e \ | \{ \overrightarrow{r} := l \} \]
\[ sto \in \text{cstorage} := \text{static} | \text{extern} | \text{auto} \]
\[ s \in \text{cstmt} := e \ | \text{return} e? \]
\[ \quad | \text{goto} x \ | x : s \]
\[ \quad | \text{break} \ | \text{continue} \]
\[ \quad | \{s\} \]
\[ \quad | \overrightarrow{sto} \tau x := l? \ ; s \]
\[ \quad | \text{typedef} x := \tau \ ; s \]
\[ \quad | \text{skip} \ | s_1 ; s_2 \]
\[ \quad | \text{while}(e) s \]
\[ \quad | \text{do} s \text{ while}(e) \]
\[ \quad | \text{for}(e_1 \ ; e_2 \ ; e_3) s \]
\[ \quad | \text{if} (e) s_1 \text{ else } s_2 \]
\[ d \in \text{decl} := \text{struct} \overrightarrow{\tau x} \ | \text{union} \overrightarrow{\tau x} \]
\[ \quad | \text{enum} x := e? : \tau_i \]
\[ \quad | \text{typedef} \tau \]
\[ \quad | \text{global} l? : \overrightarrow{sto} \tau \]
\[ \quad | \text{fun} s : \overrightarrow{sto} \tau \]
\[ \Theta \in \text{decls} := \text{list} (\text{string} \times \text{decl}) \]
Problem: End-of-Array Pointers

```c
int a[4] = { 0, 1, 2, 3 }; int *p = a; while (p < a + 4) { *p += 1; p += 1; }
```

```c
int compare(int *p, int *q) {
    // some code to confuse the optimizer
    return p == q;
}
int main() {
    int x, y;
    if (&x + 1 == &y) printf("x and y are adjacent\n");
    if (compare(&x + 1, &y)) printf("x and y are still adjacent\n");
}
```

- Comparison of pointers in the same block is defined only if both are *weakly valid*. A pointer is weakly valid if it is valid or end-of-array.
- Comparison of pointers with different block identifiers is defined for valid pointers only.
Problem: Byte-Level Operations and Object Representation

```c
struct S { short x; short *r; } s1 = { 10, &s1.x };
unsigned char *p = (unsigned char*)&s1;
```

On 32-bit computing architectures such as x86 (with _Alignof(short*) = 4), the object representation of s1 might be:

```
| 01010000 | 00000000 | padding | padding | padding |
```

```
struct S { short x; short *r; } s1 = { 10, &s1.x }; struct S s2;
for (size_t i = 0; i < sizeof(struct S); i++)
    ((unsigned char*)&s2)[i] = ((unsigned char*)&s1)[i];
```
Problem: Padding of Structs and Unions

```c
struct S { char x; char y; char z; };
void f(struct S *p) { p->x = 0; p->y = 0; p->z = 0; }

struct S s = { 1, 1, 1 };
((unsigned char*)&s)[3] = 10;
f(&s);
return ((unsigned char*)&s)[3];
```
Problem: Indeterminate Memory and Pointers

```c
int *p = malloc(sizeof(int)); assert (p != NULL);
free(p);
int *q = malloc(sizeof(int)); assert (q != NULL);
if (p == q) { // undefined, p is indeterminate due to the free
    *q = 10;
    *p = 14;
    printf("%d\n", *q); // p and q alias, expected to print 14
}
```
Conclusion

- Formal methods *can* be applied to real programming languages
- It is hard to write semantics to behaviors that were previously undefined