

Formal C semantics: CompCert and the C standard

Robbert Krebbers¹ Xavier Leroy² Freek Wiedijk¹

¹ICIS, Radboud University Nijmegen, The Netherlands

²Inria Paris-Rocquencourt, France

July 17, 2014 @ ITP, Vienna, Austria

Underspecification in C

- ▶ **Unspecified behavior:** two or more behaviors are allowed
For example: order of evaluation in expressions
- ▶ **Implementation defined behavior:** like unspecified behavior, but the compiler has to document its choice
For example: size and endianness of integers
- ▶ **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, ...

Underspecification in C

- ▶ **Unspecified behavior:** two or more behaviors are allowed
For example: order of evaluation in expressions
Non-determinism
- ▶ **Implementation defined behavior:** like unspecified behavior, but the compiler has to document its choice
For example: size and endianness of integers
Parametrization
- ▶ **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, ...
No semantics/crash state

Pros and cons of underspecification

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 formally reason about

Approaches to underspecification

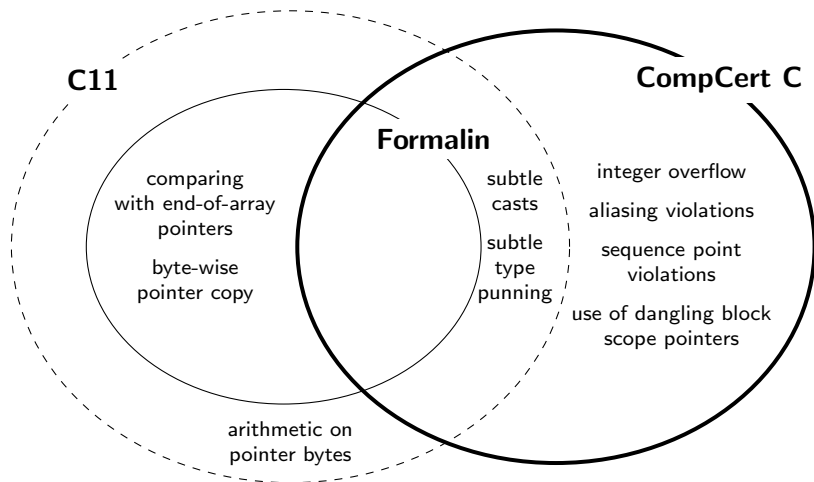
■ ■ **CompCert** (Leroy *et al.*)

- ▶ Main goal: verified optimizing compiler in 🧑
- ▶ Specific choices for unspecified/impl-defined behavior
For example: 32-bits ints
- ▶ Describes **some** undefined behavior
For example: dereferencing NULL, integer overflow defined
- ▶ Compiler correctness proof only for programs without undefined behavior

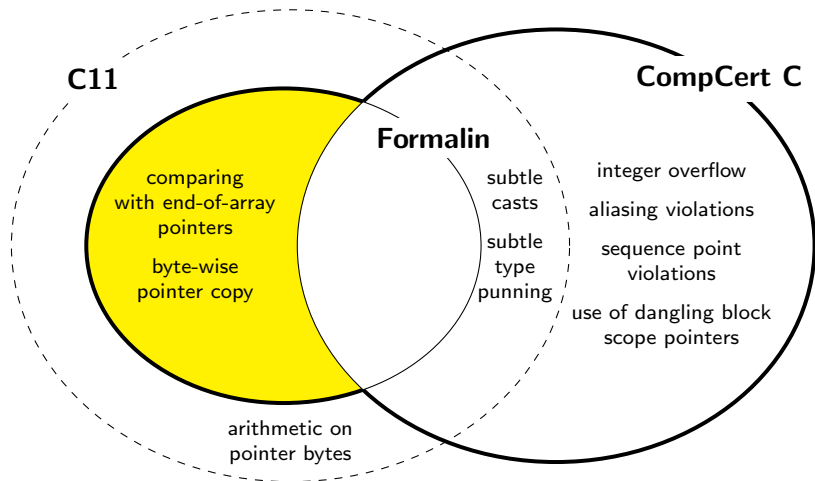
■ ■ **Formalin** (Krebbers & Wiedijk)


- ▶ Main goal: compiler independent separation logic in 🧑
- ▶ Describes **some** implementation-defined behavior
For example: no legacy architectures with 0's complement
- ▶ Aims to describe **all** unspecified and undefined behavior

Defined behaviors in C11, Formalin and CompCert C



Defined behaviors in C11, Formalin and CompCert C



This talk: add  to **CompCert** so we get **Formalin** \subseteq **CompCert**

Comparing with end-of-array pointers (problem)

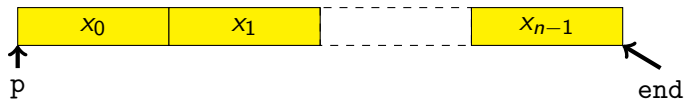
Useful:

```
void inc_array(int *p, int n) {  
    int *end = p + n;  
    while (p < end) (*p++)++;  
}
```


Comparing with end-of-array pointers (problem)

Useful:

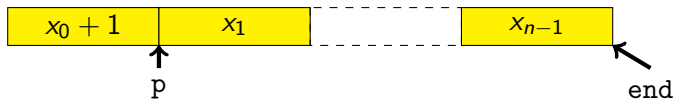
```
void inc_array(int *p, int n) {  
    int *end = p + n;  
    while (p < end) (*p++)++;  
}
```



Comparing with end-of-array pointers (problem)

Useful:

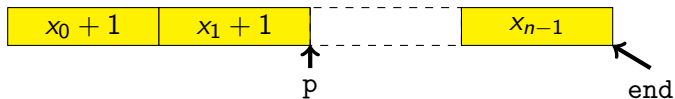
```
void inc_array(int *p, int n) {  
    int *end = p + n;  
    while (p < end) (*p++)++;  
}
```



Comparing with end-of-array pointers (problem)

Useful:

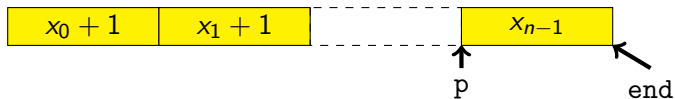
```
void inc_array(int *p, int n) {  
    int *end = p + n;  
    while (p < end) (*p++)++;  
}
```



Comparing with end-of-array pointers (problem)

Useful:

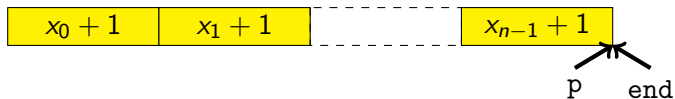
```
void inc_array(int *p, int n) {  
    int *end = p + n;  
    while (p < end) (*p++)++;  
}
```



Comparing with end-of-array pointers (problem)

Useful:

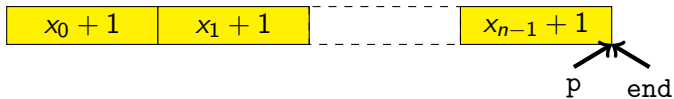
```
void inc_array(int *p, int n) {  
    int *end = p + n;  
    while (p < end) (*p++)++;  
}
```



Comparing with end-of-array pointers (problem)

Useful:

```
void inc_array(int *p, int n) {  
    int *end = p + n;  
    while (p < end) (*p++)++;  
}
```



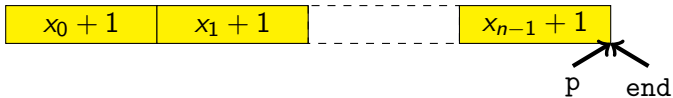
Bizarre:

```
int x, y;  
if (&x + 1 == &y) printf("x and y are adjacent\n");
```

Comparing with end-of-array pointers (problem)

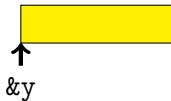
Useful:

```
void inc_array(int *p, int n) {  
    int *end = p + n;  
    while (p < end) (*p++)++;  
}
```



Bizarre:

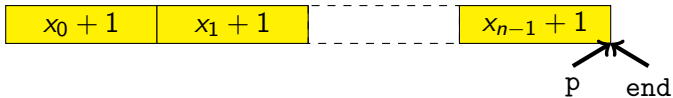
```
int x, y;  
if (&x + 1 == &y) printf("x and y are adjacent\n");
```



Comparing with end-of-array pointers (problem)

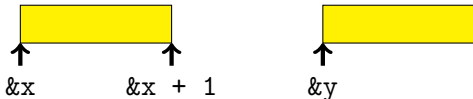
Useful:

```
void inc_array(int *p, int n) {  
    int *end = p + n;  
    while (p < end) (*p++)++;  
}
```



Bizarre:

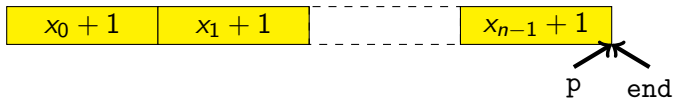
```
int x, y;  
if (&x + 1 == &y) printf("x and y are adjacent\n");
```



Comparing with end-of-array pointers (problem)

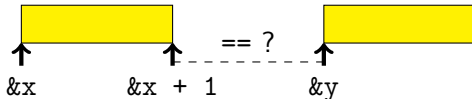
Useful:

```
void inc_array(int *p, int n) {  
    int *end = p + n;  
    while (p < end) (*p++)++;  
}
```



Bizarre:

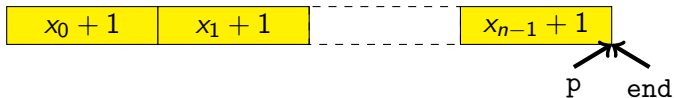
```
int x, y;  
if (&x + 1 == &y) printf("x and y are adjacent\n");
```



Comparing with end-of-array pointers (problem)

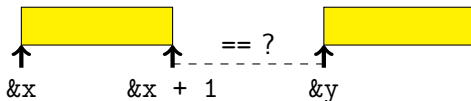
Useful:

```
void inc_array(int *p, int n) {  
    int *end = p + n;  
    while (p < end) (*p++)++;  
}
```



Bizarre:

```
int x, y;  
if (&x + 1 == &y) printf("x and y are adjacent\n");
```

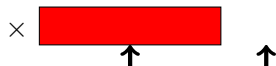


Both undefined behavior in CompCert (1.12 and before)

Comparing with end-of-array pointers (solution)

Solution: Comparison of pointers is defined if:

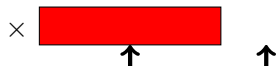
- ▶ Same block: both should **within** block bounds



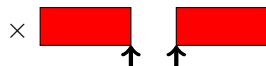
Comparing with end-of-array pointers (solution)

Solution: Comparison of pointers is defined if:

- ▶ Same block: both should **within** block bounds



- ▶ Different block: both should be **strictly within** block bounds



Comparing with end-of-array pointers (solution)

Solution: Comparison of pointers is defined if:

- ▶ Same block: both should **within** block bounds



- ▶ Different block: both should be **strictly within** block bounds



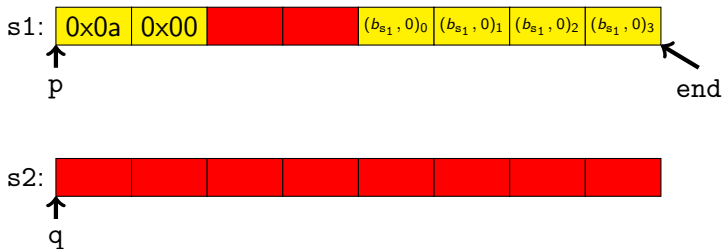
Stable under compilation and gives a semantics to common programming practice with end-of-array pointers

Byte-wise copying of objects (problem)

```
struct { short x; short *r; } s1 = {10, &s.x}, s2;  
unsigned char *p = &s1, *q = &s2;  
unsigned char *end = p + sizeof(s1);  
while (p < end) *p++ = *q++;
```

Byte-wise copying of objects (problem)

```
struct { short x; short *r; } s1 = {10, &s.x}, s2;  
unsigned char *p = &s1, *q = &s2;  
unsigned char *end = p + size_of(s1);  
while (p < end) *p++ = *q++;
```



Byte-wise copying of objects (problem)

```
struct { short x; short *r; } s1 = {10, &s.x}, s2;  
unsigned char *p = &s1, *q = &s2;  
unsigned char *end = p + size_of(s1);  
while (p < end) *p++ = *q++;
```



Byte-wise copying of objects (problem)

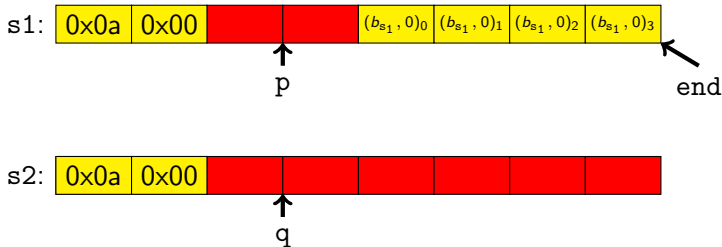
```
struct { short x; short *r; } s1 = {10, &s.x}, s2;  
unsigned char *p = &s1, *q = &s2;  
unsigned char *end = p + size_of(s1);  
while (p < end) *p++ = *q++;
```



Previously undefined, need to allow copying indeterminate bytes

Byte-wise copying of objects (problem)

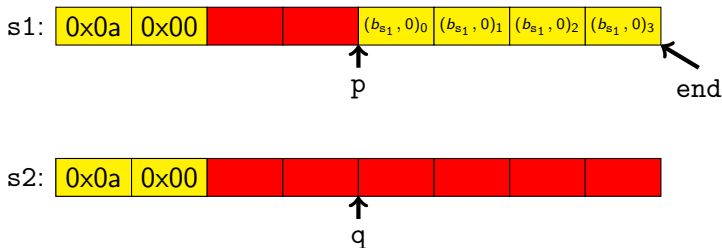
```
struct { short x; short *r; } s1 = {10, &s.x}, s2;  
unsigned char *p = &s1, *q = &s2;  
unsigned char *end = p + size_of(s1);  
while (p < end) *p++ = *q++;
```



Previously undefined, need to allow copying indeterminate bytes

Byte-wise copying of objects (problem)

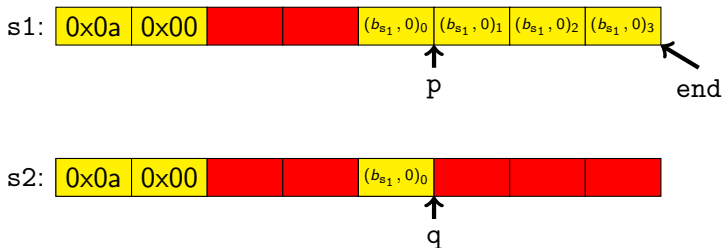
```
struct { short x; short *r; } s1 = {10, &s.x}, s2;  
unsigned char *p = &s1, *q = &s2;  
unsigned char *end = p + size_of(s1);  
while (p < end) *p++ = *q++;
```



Previously undefined, need to allow copying symbolic pointer bytes

Byte-wise copying of objects (problem)

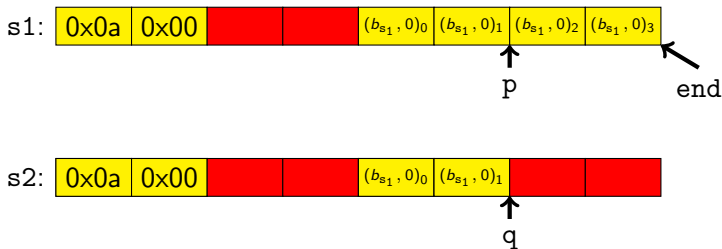
```
struct { short x; short *r; } s1 = {10, &s.x}, s2;  
unsigned char *p = &s1, *q = &s2;  
unsigned char *end = p + size_of(s1);  
while (p < end) *p++ = *q++;
```



Previously undefined, need to allow copying symbolic pointer bytes

Byte-wise copying of objects (problem)

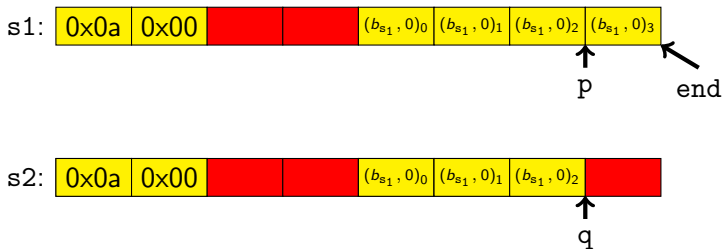
```
struct { short x; short *r; } s1 = {10, &s.x}, s2;  
unsigned char *p = &s1, *q = &s2;  
unsigned char *end = p + size_of(s1);  
while (p < end) *p++ = *q++;
```



Previously undefined, need to allow copying symbolic pointer bytes

Byte-wise copying of objects (problem)

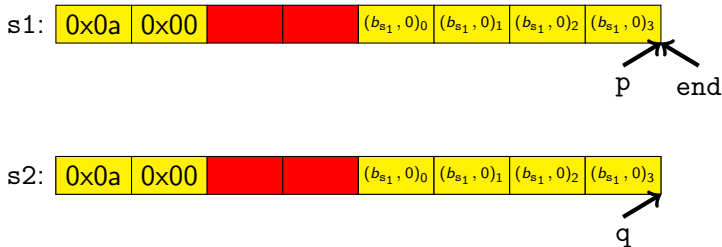
```
struct { short x; short *r; } s1 = {10, &s.x}, s2;  
unsigned char *p = &s1, *q = &s2;  
unsigned char *end = p + size_of(s1);  
while (p < end) *p++ = *q++;
```



Previously undefined, need to allow copying symbolic pointer bytes

Byte-wise copying of objects (problem)

```
struct { short x; short *r; } s1 = {10, &s.x}, s2;  
unsigned char *p = &s1, *q = &s2;  
unsigned char *end = p + size_of(s1);  
while (p < end) *p++ = *q++;
```



Previously undefined, need to allow copying symbolic pointer bytes

Byte-wise copying of objects (solution)

Solution: extend values with *pointer fragment* values

```
Inductive val: Type :=
  | Vundef: val
  | Vint: int -> val
  | Vlong: int64 -> val
  | Vfloat: float -> val
  | Vptr: block -> int -> val
  | Vptrfrag: block -> int -> nat -> val.
```

Subtleties:

- ▶ Dealing with arithmetic on pointer fragments
- ▶ Dealing with implicit casts (at assignments)
- ▶ More values possible, need to extend static analysis

Conclusion and future work

- ▶ Semantics to useful behaviors that were previously undefined
 - ▶ Comparing with end-of-array pointers
 - ▶ Byte-wise pointer copy
- ▶ CompCert proofs adapted for these extensions
 - ▶ Small changes to the semantics
 - ▶ Involves proofs of many compilation passes
- ▶ Needed for cross validation of CompCert and Formalin
- ▶ Call-by-reference passing of struct values future work

Questions

Sources: `http://github.com/robertkrebbers`