Aliasing restrictions of C11 formalized in Coq

Robbert Krebbers

Radboud University Nijmegen

December 11, 2013 @ CPP, Melbourne, Australia

Aliasing

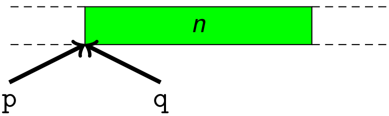
Aliasing: multiple pointers referring to the same object

Aliasing

Aliasing: multiple pointers referring to the same object

```
int f(int *p, int *q) {
  int x = *p; *q = 314; return x;
}
```

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

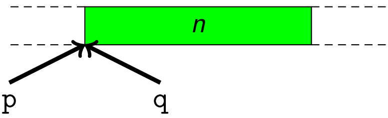


Aliasing

Aliasing: multiple pointers referring to the same object

```
int f(int *p, int *q) {
  int x = *p; *q = 314; return x *p;
}
```

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



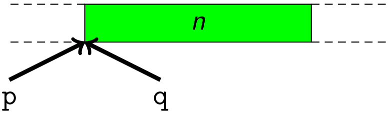
Optimizing `x` away is unsound: 314 would be returned

Aliasing

Aliasing: multiple pointers referring to the same object

```
int f(int *p, int *q) {
  int x = *p; *q = 314; return x;
}
```

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



Optimizing `x` away is unsound: 314 would be returned

Alias analysis: to determine whether pointers can alias

Aliasing with different types

Consider a similar function:

```
int h(int *p, float *q) {
  int x = *p; *q = 3.14; return x;
}
```

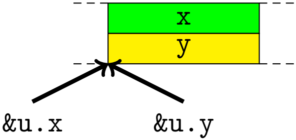
Aliasing with different types

Consider a similar function:

```
int h(int *p, float *q) {
  int x = *p; *q = 3.14; return x;
}
```

It can still be called with aliased pointers:

```
union { int x; float y; } u;
u.x = 271;
return h(&u.x, &u.y);
```



C89 allows p and q to be aliased, and thus requires it to return 271

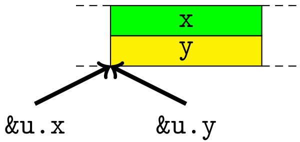
Aliasing with different types

Consider a similar function:

```
int h(int *p, float *q) {
  int x = *p; *q = 3.14; return x;
}
```

It can still be called with aliased pointers:

```
union { int x; float y; } u;
u.x = 271;
return h(&u.x, &u.y);
```



C89 allows p and q to be aliased, and thus requires it to return 271

C99/C11 allows **type-based alias analysis**:

- A compiler can assume that p and q do not alias
- Reads/writes with "the wrong type" yield undefined behavior

Undefined behavior in C

"Garbage in, garbage out" principle

- Programs with undefined behavior are not statically excluded
- Instead, these may do literally anything when executed
- Compilers are allowed to assume no undefined behavior occurs
- Allows them to omit (expensive) dynamic checks

Undefined behavior in C

"Garbage in, garbage out" principle

- Programs with undefined behavior are not statically excluded
- Instead, these may do literally anything when executed
- Compilers are allowed to assume no undefined behavior occurs
- Allows them to omit (expensive) dynamic checks

A formal C semantics should account for undefined behavior

Bits and bytes

Interplay between low- and high-level

- Each object *should be* represented as a sequence of bits
  . . . which can be inspected and manipulated *in C*
- Each object can be accessed using typed expressions
  . . . that are used by compilers for optimizations

Hence, the formal memory model needs to keep track of more information than present in the memory of an actual machine

Bits and bytes

Interplay between low- and high-level

- Each object *should be* represented as a sequence of bits
  ... which can be inspected and manipulated *in C*
- Each object can be accessed using typed expressions
  ... that are used by compilers for optimizations

Hence, the formal memory model needs to keep track of more information than present in the memory of an actual machine

The standard is unclear on many of such difficulties
Opportunities for a formal semantics to resolve this unclarity!

Contribution

An abstract formal memory for C supporting

- Types (arrays, structs, unions, . . .)
- Strict aliasing restrictions (effective types)
- Byte-level operations
- Type-punning
- Indeterminate memory
- Pointers "one past the last element"
- Parametrized by an interface for integer types
- Formalized together with essential properties in Coq

How to treat pointers

Others (e.g. CompCert)	Our approach
Memory: a finite map of cells which consist of arrays of bytes	
Pointers: pairs (x, i) where x identifies the cell, and i the offset into that cell	
Too little information to capture strict aliasing restrictions	

How to treat pointers

Others (e.g. CompCert)	**Our approach**
Memory: a finite map of cells which consist of arrays of bytes	A finite map of cells which consist of well-typed trees of bits
Pointers: pairs (x, i) where x identifies the cell, and i the offset into that cell	
Too little information to capture strict aliasing restrictions	

How to treat pointers

Others (e.g. CompCert)	**Our approach**
Memory: a finite map of cells which consist of arrays of bytes	A finite map of cells which consist of well-typed trees of bits
Pointers: pairs (x, i) where x identifies the cell, and i the offset into that cell	Pairs (x, r) where x identifies the cell, and r the path through the tree in that cell
Too little information to capture strict aliasing restrictions	

How to treat pointers

Others (e.g. CompCert)	**Our approach**
Memory: a finite map of cells which consist of arrays of bytes	A finite map of cells which consist of well-typed trees of bits
Pointers: pairs (x, i) where x identifies the cell, and i the offset into that cell	Pairs (x, r) where x identifies the cell, and r the path through the tree in that cell
Too little information to capture strict aliasing restrictions	A semantics for strict aliasing restrictions

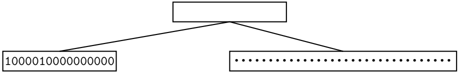
Three kinds of values

Our formal description has three kinds of values. For

```
struct { short x, *p; } s = { 33; &s.x }
```

we have:

- *A memory value* with arrays of bits as leafs:



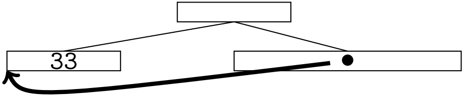
Three kinds of values

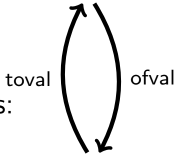
Our formal description has three kinds of values. For

```
struct { short x, *p; } s = { 33; &s.x }
```

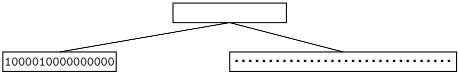
we have:

- *An abstract value* with machine integers and pointers as leafs:





- *A memory value* with arrays of bits as leafs:



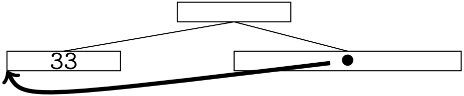
Three kinds of values

Our formal description has three kinds of values. For

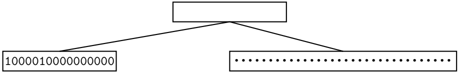
```
struct { short x, *p; } s = { 33; &s.x }
```

we have:

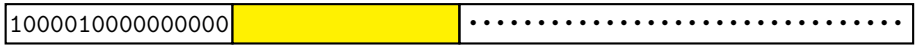
- *An abstract value* with machine integers and pointers as leafs:

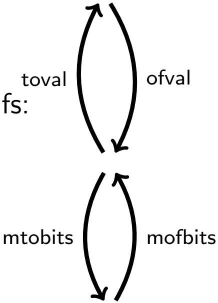


- *A memory value* with arrays of bits as leafs:



- An array of bits:





Bits and memory values

- *Bits* are represented symbolically (à la bytes in CompCert):

$$b ::= 0 \mid 1 \mid (\mathsf{ptr}\ p)_i \mid \mathsf{indet}$$

- Gives *"the best of both worlds"*: allows bitwise hacking on integers while keeping the memory abstract

Bits and memory values

- *Bits* are represented symbolically (à la bytes in CompCert):

$$b ::= 0 \mid 1 \mid (\mathsf{ptr}\ p)_i \mid \mathsf{indet}$$

- Gives *"the best of both worlds"*: allows bitwise hacking on integers while keeping the memory abstract
- *Memory values* are defined as:

$$\begin{aligned} w ::= {} & \mathsf{base}_{\tau_\mathrm{b}}\ \vec{b} \mid \mathsf{array}\ \vec{w} \\ & \mid \mathsf{struct}_s\ \vec{w} \mid \mathsf{union}_s\ (i, w) \mid \overline{\mathsf{union}}_s\ \vec{b} \end{aligned}$$

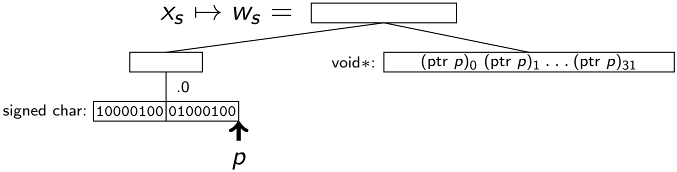
- Memory values have (unique) types

Example

Consider:

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

As a picture:

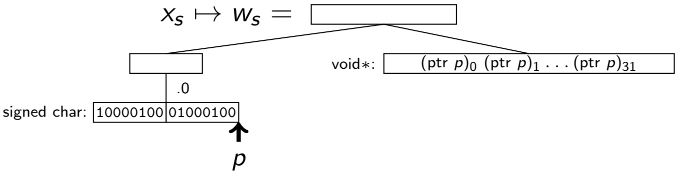


Example

Consider:

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

As a picture:



Here we have:

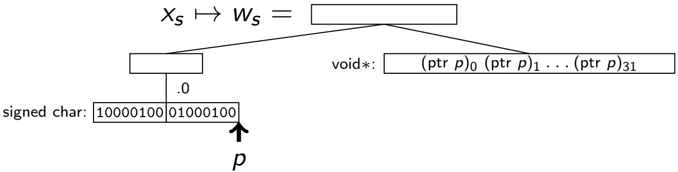
- $p = (x_s, \overset{\mathtt{T}}{\rightsquigarrow} 0 \overset{\mathtt{U}}{\rightsquigarrow} 0, 2)_{\text{signed char} > \text{void}}$

Example

Consider:

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

As a picture:



Here we have:

- $p = (x_s, \overset{\texttt{T}}{\rightsquigarrow} 0 \overset{\texttt{U}}{\rightsquigarrow} 0, 2)_{\mathsf{signed\ char} > \mathsf{void}}$
- mtobits $w_s =$

1000010001000100	indet . . . indet	$(\mathsf{ptr}\ p)_0\ (\mathsf{ptr}\ p)_1 \ldots (\mathsf{ptr}\ p)_{31}$

Memory values to bits?

Let us reconsider memory values:

$$w ::= \mathsf{base}_{\tau_\mathsf{b}}\ \vec{b} \mid \mathsf{array}\ \vec{w} \\ \mid \mathsf{struct}_s\ \vec{w} \mid \mathsf{union}_s\,(i, w) \mid \overline{\mathsf{union}}_s\ \vec{b}$$

How to do the conversion of bits?

Memory values to bits?

Let us reconsider memory values:

$$
\begin{aligned}
w ::= {} & \mathsf{base}_{\tau_\mathrm{b}}\ \vec{b} \mid \mathsf{array}\ \vec{w} \\
& \mid \mathsf{struct}_s\ \vec{w} \mid \mathsf{union}_s\,(i, w) \mid \overline{\mathsf{union}}_s\ \vec{b}
\end{aligned}
$$

How to do the conversion of bits?

- Seems impossible
- Given bits of

```
union U { int x; int y; }
```

should we choose the variant `x` or `y`?

Memory values to bits?

Let us reconsider memory values:

$$\begin{aligned} w ::= {} & \mathsf{base}_{\tau_\mathrm{b}}\ \vec{b} \mid \mathsf{array}\ \vec{w} \\ & \mid \mathsf{struct}_s\ \vec{w} \mid \mathsf{union}_s\ (i, w) \mid \overline{\mathsf{union}}_s\ \vec{b} \end{aligned}$$

How to do the conversion of bits?

- Seems impossible
- Given bits of

```
union U { int x; int y; }
```

  should we choose the variant `x` or `y`?

Solution: postpone this choice by storing it as a $\overline{\mathsf{union}}_{\mathrm{U}}\ \vec{b}$
. . . and change it into a $\mathsf{union}_{\mathrm{U}}\ (i, w)$ node on a lookup

Memory values to bits?

Let us reconsider memory values:

$$w ::= \mathsf{base}_{\tau_b}\ \vec{b} \mid \mathsf{array}\ \vec{w}$$
$$\mid \mathsf{struct}_s\ \vec{w} \mid \mathsf{union}_s\,(i, w) \mid \overline{\mathsf{union}}_s\ \vec{b}$$

How to do the conversion of bits?

- Seems impossible
- Given bits of

```
union U { int x; int y; }
```

  should we choose the variant `x` or `y`?

Solution: postpone this choice by storing it as a $\overline{\mathsf{union}}_{\mathtt{U}}\ \vec{b}$
...and change it into a $\mathsf{union}_{\mathtt{U}}\,(i, w)$ node on a lookup

Hard part: dealing with this choice in *abstract values* and the various operations

Type-punning

Type-punning: reading a union using a pointer to another variant

C11: vaguely mentioned in a footnote

Type-punning

Type-punning: reading a union using a pointer to another variant

C11: vaguely mentioned in a footnote
GCC: allowed if “the memory is accessed through the union type”

Given:

```
union U { int x; float y; } t;
```

Defined behavior:

```
t.y = 3.0; return t.x; // OK
```

Type-punning

Type-punning: reading a union using a pointer to another variant

C11: vaguely mentioned in a footnote
GCC: allowed if "the memory is accessed through the union type"

Given:

```
union U { int x; float y; } t;
```

Defined behavior:

```
t.y = 3.0; return t.x; // OK
```

Undefined behavior:

```
int *p = &t.x; t.y = 3.0; return *p; // UB
```

Type-punning

Type-punning: reading a union using a pointer to another variant

C11: vaguely mentioned in a footnote
GCC: allowed if "the memory is accessed through the union type"

Given:

```
union U { int x; float y; } t;
```

Defined behavior:

```
t.y = 3.0; return t.x; // OK
```

Undefined behavior:

```
int *p = &t.x; t.y = 3.0; return *p; // UB
```

Formalized by decorating pointers with annotations

Strict-aliasing Theorem

Theorem (Strict-aliasing)

Given:

- *addresses* $m \vdash a_1 : \sigma_1$ *and* $m \vdash a_2 : \sigma_2$
- *with annotations that do not allow type-punning*
- $\sigma_1, \sigma_2 \neq$ unsigned char
- σ_1 *not a subtype of* σ_2 *and vice versa*

Then there are two possibilities:

- a_1 *and* a_2 *do not alias*
- *accessing* a_1 *after* a_2 *(and vice versa) has undefined behavior*

Strict-aliasing Theorem

Theorem (Strict-aliasing)

Given:

- *addresses* $m \vdash a_1 : \sigma_1$ *and* $m \vdash a_2 : \sigma_2$
- *with annotations that do not allow type-punning*
- $\sigma_1, \sigma_2 \neq$ unsigned char
- σ_1 *not a subtype of* σ_2 *and vice versa*

Then there are two possibilities:

- a_1 *and* a_2 *do not alias*
- *accessing* a_1 *after* a_2 *(and vice versa) has undefined behavior*

Corollary Compilers can perform type based alias analysis

Memory extensions

To prove program transformations correct one has to relate:

- the memory of the original program to
- the memory of the the transformed program

Memory extensions

To prove program transformations correct one has to relate:

- the memory of the original program to
- the memory of the the transformed program

We have $m_1 \sqsubseteq m_2$ if m_2 allows more behaviors than m_1:

- More memory content determinate

$$\mathsf{indet} \sqsubseteq b$$

Memory extensions

To prove program transformations correct one has to relate:

- the memory of the original program to
- the memory of the the transformed program

We have $m_1 \sqsubseteq m_2$ if m_2 allows more behaviors than m_1:

- More memory content determinate

$$\mathsf{indet} \sqsubseteq b$$

- Fewer restrictions on effective types

$$\mathsf{union}_u\,(i, w) \sqsubseteq \overline{\mathsf{union}}_u\,\vec{b}$$

Memory extensions

To prove program transformations correct one has to relate:

- the memory of the original program to
- the memory of the the transformed program

We have $m_1 \sqsubseteq m_2$ if m_2 allows more behaviors than m_1:

- More memory content determinate

$$\text{indet} \sqsubseteq b$$

- Fewer restrictions on effective types

$$\text{union}_u\,(i, w) \sqsubseteq \overline{\text{union}_u}\,\vec{b}$$

Theorem "copy by assignment" $\sqsubseteq$ "byte-wise copy"

Formalization in Coq

Type theory is ideal for the combination programming/proving

- *The devil is in the details*, Coq is extremely useful for debugging of definitions
- Useful to prove meta-theoretical properties
- Use of type classes for parametrization by machine integers
- Use of type classes for overloading of notations
- 8.500 lines of code

Future research

- Integration into our operational semantics [K, POPL'14]
  ... and make it (reasonably efficiently) executable
- Memory injections à la CompCert
- Integration into our axiomatic semantics [K, POPL'14]
- Floating point numbers, bit fields, variable length arrays
- The `const`, `volatile` and `restrict` qualifier
- Verification Condition generator in Coq

Questions

Sources: see `http://robbertkrebbers.nl/research/ch2o/`