The C standard formalized in Coq, what’s next?

Robbert Krebbers

Aarhus University, Denmark

May 13, 2016 © Cambridge Computer Laboratory, UK
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*  
  
  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
Why does C use underspecification that heavily?

**Pros** for optimizing compilers:
- More optimizations are possible
- High run-time efficiency
- Easy to support multiple architectures
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
The CH$_2$O project

C sources

CH$_2$O abstract C

CH$_2$O core C
The CH$_2$O project

C sources

CH$_2$O abstract C

CH$_2$O core C

OCaml part

Coq part

Operational semantics
$\Gamma, \delta \vdash S_1 \rightarrow S_2$

Soundness & Completeness

Type preservation & progress

Type soundness

Invariance

Typing judgment
$\Gamma \vdash S : f$

Refinement judgment
$S_1 \sqsubseteq f \Gamma S_2 : f$

Executable semantics
$S_2 \in \text{exec} \Gamma, \delta \vdash S_1$
The CH$_2$O project

C sources

CH$_2$O abstract C

CH$_2$O core C

Typing judgment
$\Gamma \vdash S : f_{\text{main}}$

Type soundness

Type preservation & progress

Operational semantics
$\Gamma, \delta \vdash S_1 \rightarrow S_2$

Soundness & Completeness

Type soundness

Type preservation & progress

Invariance

Soundness

Soundness & Completeness

Soundness & Completeness

Invariance

OCaml part

Coq part
The CH$_2$O project

C sources → CH$_2$O abstract C

Type soundness

Typing judgment
$\Gamma \vdash S : f_{\text{main}}$

Operational semantics
$\Gamma, \delta \vdash S_1 \rightarrow S_2$

Soundness & Completeness

Executable semantics
$S_2 \in \text{exec}_{\Gamma, \delta} S_1$

Type preservation & progress

Soundness

Type preservation & progress

Invariance

Type soundness
The CH$_2$O project

---

**OCaml part**

C sources

CH$_2$O abstract C

**Coq part**

CH$_2$O core C

Typing judgment

$\Gamma \vdash S : f_{\text{main}}$

Type soundness

Operational semantics

$\Gamma, \delta \vdash S_1 \rightarrow S_2$

Type preservation & progress

Soundness & Completeness

Pure expression evaluation

$[e]_{\Gamma, \rho, m} = \nu$

Soundness & Completeness

Executable semantics

$S_2 \in \text{exec}_{\Gamma, \delta} S_1$
The CH₂O project

C sources → CH₂O abstract C

CH₂O core C

Typing judgment
Γ ⊢ S : f_{main}

Type soundness

Operational semantics
Γ, δ ⊢ S₁ → S₂

Type preservation & progress

Pure expression evaluation
[e]_{Γ, ρ, m} = ν

Soundness & Completeness

Axiomatic semantics
R, J, T ⊢ Γ, δ
{P} s {Q}

Soundness

Soundness & Completeness

Executable semantics
S₂ ∈ exec_{Γ, δ} S₁

Soundness & progress

Type soundness & completeness

Invariance

Type soundness

Soundness

Invariance

Type soundness & completeness

Soundness
The CH₂O project

OCaml part

C sources

CH₂O abstract C

CH₂O core C

Typing judgment
\( \Gamma \vdash S : f_{\text{main}} \)

Type soundness

Type preservation & progress

Operational semantics
\( \Gamma, \delta \vdash S_1 \rightarrow S_2 \)

Soundness & Completeness

Pure expression evaluation
\( \llbracket e \rrbracket_{\Gamma, \rho, m} = \nu \)

Soundness & Completeness

Refinement judgment
\( S_1 \sqsubseteq^f S_2 : f_{\text{main}} \)

Invariance

Axiomatic semantics
\( R, J, T \vdash_{\Gamma, \delta} \{ P \} s \{ Q \} \)

Soundness

Executable semantics
\( S_2 \in \text{exec}_{\Gamma, \delta} S_1 \)

Soundness & Completeness

The CH₂O project

OCaml part

C sources

CH₂O abstract C

CH₂O core C

Typing judgment
\( \Gamma \vdash S : f_{\text{main}} \)

Type soundness

Type preservation & progress

Operational semantics
\( \Gamma, \delta \vdash S_1 \rightarrow S_2 \)

Soundness & Completeness

Pure expression evaluation
\( \llbracket e \rrbracket_{\Gamma, \rho, m} = \nu \)

Soundness & Completeness

Refinement judgment
\( S_1 \sqsubseteq^f S_2 : f_{\text{main}} \)

Invariance

Axiomatic semantics
\( R, J, T \vdash_{\Gamma, \delta} \{ P \} s \{ Q \} \)

Soundness

Executable semantics
\( S_2 \in \text{exec}_{\Gamma, \delta} S_1 \)

Soundness & Completeness

The CH₂O project

OCaml part

C sources

CH₂O abstract C

CH₂O core C

Typing judgment
\( \Gamma \vdash S : f_{\text{main}} \)

Type soundness

Type preservation & progress

Operational semantics
\( \Gamma, \delta \vdash S_1 \rightarrow S_2 \)

Soundness & Completeness

Pure expression evaluation
\( \llbracket e \rrbracket_{\Gamma, \rho, m} = \nu \)

Soundness & Completeness

Refinement judgment
\( S_1 \sqsubseteq^f S_2 : f_{\text{main}} \)

Invariance

Axiomatic semantics
\( R, J, T \vdash_{\Gamma, \delta} \{ P \} s \{ Q \} \)

Soundness

Executable semantics
\( S_2 \in \text{exec}_{\Gamma, \delta} S_1 \)

Soundness & Completeness

The CH₂O project
Non-local control flow and block scope variables
The C quiz, question 2

```c
int *p = NULL;
l: if (p) {
    return (*p);
} else {
    int j = 17;
p = &j;
goto l;
}
```
Non-local control flow and block scope variables

The C quiz, question 2

```c
int *p = NULL;
l: if (p) {
    return (*p);
} else {
    int j = 17;
p = &j;
goto l;
}
```

memory:

- `p` points to `NULL`
Non-local control flow and block scope variables

The C quiz, question 2

```c
int *p = NULL;
l: if (p) {
    return (*p);
} else {
    int j = 17;
    p = &j;
    goto l;
}
```

memory:
- `p` points to `NULL`
Non-local control flow and block scope variables
The C quiz, question 2

```c
int *p = NULL;
l: if (p) {
    return (*p);
} else {
    int j = 17;
    p = &j;
    goto l;
}
```

memory:

- `p`: `NULL`
- `j`: `17`
Non-local control flow and block scope variables
The C quiz, question 2

```c
int *p = NULL;
l: if (p) {
    return (*p);
} else {
    int j = 17;
p = &j;
goto l;
}
```

memory:

```
  p
```

```
  j
```

C11, 6.2.4p2: the value of a pointer becomes indeterminate when the object it points to (or just past) reaches the end of its lifetime. =⇒ Undefined behavior
Non-local control flow and block scope variables

The C quiz, question 2

```c
int *p = NULL;
l: if (p)
    return (*p);
} else {
    int j = 17;
p = &j;
goto l;
}
```
Non-local control flow and block scope variables

The C quiz, question 2

```c
int *p = NULL;
l: if (p) {
    return (*p);
} else {
    int j = 17;
p = &j;
goto l;
}
```

memory:

- `p`
Non-local control flow and block scope variables
The C quiz, question 2

```c
int *p = NULL;
l: if (p) {
    return (*p);
} else {
    int j = 17;
    p = &j;
    goto l;
}
```

C11, 6.2.4p2: the value of a pointer becomes indeterminate when the object it points to (or just past) reaches the end of its lifetime.

⇒ Undefined behavior
Non-local control flow and block scope variables

Goto considered harmful?

http://xkcd.com/292/
Non-local control flow and block scope variables

Goto considered harmful?

Not necessarily:

$p \vdash \{P\} \ldots \text{goto main}\_\text{sub3}; \ldots \{Q\}$

http://xkcd.com/292/
Non-local control flow and block scope variables

Separation logic for non-local control

Statement judgment:

\[ R, J, T \vdash \{ P \} s \{ Q \} \]
Non-local control flow and block scope variables

Separation logic for non-local control

**Statement judgment:**

\[ R, J, T \vdash \{P\} s \{Q\} \]

where:

- \( \{P\} s \{Q\} \) is a Hoare triple, as usual
Non-local control flow and block scope variables

Separation logic for non-local control

**Statement judgment:**

\[ R, J, T \vdash \{ P \} \cdot s \{ Q \} \]

where:

- \( \{ P \} \cdot s \{ Q \} \) is a Hoare triple, as usual
- \( R \) has to hold to execute a return
Non-local control flow and block scope variables
Separation logic for non-local control

**Statement judgment:**

\[ R, J, T \vdash \{ P \} s \{ Q \} \]

where:

- \( \{ P \} s \{ Q \} \) is a Hoare triple, as usual
- \( R \) has to hold to execute a \texttt{return}
- \( J \) maps labels to their jumping condition
  - When executing a \texttt{goto l}, the assertion \( J l \) has to hold
Non-local control flow and block scope variables
Separation logic for non-local control

Statement judgment:

\[ R, J, T \vdash \{ P \} s \{ Q \} \]

where:

- \( \{ P \} s \{ Q \} \) is a Hoare triple, as usual
- \( R \) has to hold to execute a return
- \( J \) maps labels to their jumping condition
  When executing a `goto l`, the assertion \( J l \) has to hold
- \( T \) maps breaks/continues to their jumping condition
Non-local control flow and block scope variables
Separation logic for non-local control

**Statement judgment:**

\[ R, J, T \vdash \{ P \} s \{ Q \} \]

where:

- \( \{ P \} s \{ Q \} \) is a Hoare triple, as usual
- \( R \) has to hold to execute a return
- \( J \) maps labels to their jumping condition
  - When executing a `goto l`, the assertion \( Jl \) has to hold
- \( T \) maps `breaks/continues` to their jumping condition

Example:

\[
R, J, T \vdash \{ J l \} \text{ goto } l \{ Q \} \quad R, J, T \vdash \{ J l \} l : \{ J l \}
\]
Non-local control flow and block scope variables

The block scope variable rule

\[
R \uparrow^* x_0 \mapsto -, J \uparrow^* (x_0 \mapsto - : \tau), T \uparrow^* (x_0 \mapsto - : \tau)
\vdash \{P \uparrow^* (x_0 \mapsto - : \tau)\} s \{Q \uparrow^* (x_0 \mapsto - : \tau)\}
\]

\[
\vdash \{P\} \text{local}_\tau s \{Q\}
\]

When entering a block:

- The De Bruijn indexes are lifted: (\_\_) \uparrow
- The memory is extended: (\_\_) \ast (x_0 \mapsto - : \tau)

When leaving a block: the reverse
Non-local control flow and block scope variables

The block scope variable rule

\[
R \uparrow^* x_0 \mapsto -, J \uparrow^* (x_0 \mapsto - : \tau), T \uparrow^* (x_0 \mapsto - : \tau)
\]
\[
\vdash \{ P \uparrow^* (x_0 \mapsto - : \tau) \} \ s \ \{ Q \uparrow^* (x_0 \mapsto - : \tau) \}
\]
\[
R, J, T \vdash \{ P \} \ \text{local}_\tau \ s \ \{ Q \}
\]

When entering a block:

- The De Bruijn indexes are lifted: \((\_\_) \uparrow\)
- The memory is extended: \((\_\_) \ast (x_0 \mapsto - : \tau)\)

When leaving a block: the reverse

Important:

- Symmetry matches gotos going both in and out
- Using De Bruijn indexes avoids shadowing
Non-determinism and sequence points

The C quiz, question 3

```c
int x = 0, y = 0, *p = &x;
int f() { p = &y; return 17; }
int main() {
    *p = f();
    printf("x=%d,y=%d\n", x, y);
}
```

Let us try some compilers

- Clang prints `x=0,y=17`
- GCC prints `x=17,y=0`

Non-determinism appears even in innocently looking code.
Non-determinism and sequence points
The C quiz, question 3

```c
int x = 0, y = 0, *p = &x;
int f() { p = &y; return 17; }
int main() {
    *p = f(); // p can become &x or &y
    printf("x=%d,y=%d\n", x, y);
}
```

Let us try some compilers
- Clang prints x=0, y=17
- GCC prints x=17, y=0

Non-determinism appears even in innocently looking code
Non-determinism and sequence points

Separation logic for C expressions

**Observation**: non-determinism corresponds to concurrency
**Idea**: use the separation logic rule for parallel composition

\[
\begin{align*}
\{P_1\} e_1 \{Q_1\} & \quad \{P_2\} e_2 \{Q_2\} \\
\{P_1 \ast P_2\} e_1 \odot e_2 \{Q_1 \ast Q_2\}
\end{align*}
\]

What does this mean:

- Split the memory into two disjoint parts
- Prove that \(e_1\) and \(e_2\) can be executed safely in their part
- Now \(e_1 \odot e_2\) can be executed safely in the whole memory

Disjointness \(\Rightarrow\) no sequence point violation (accessing the same location twice in one expression)
Non-determinism and sequence points
Separation logic for C expressions

Observation: non-determinism corresponds to concurrency
Idea: use the separation logic rule for parallel composition

\[
\begin{align*}
\{ P_1 \} e_1 \{ Q_1 \} & \quad \{ P_2 \} e_2 \{ Q_2 \} \\
\{ P_1 \ast P_2 \} e_1 \oslash e_2 \{ Q_1 \ast Q_2 \}
\end{align*}
\]

What does this mean:
- Split the memory into two disjoint parts
- Prove that \( e_1 \) and \( e_2 \) can be executed safely in their part
- Now \( e_1 \oslash e_2 \) can be executed safely in the whole memory
Non-determinism and sequence points
Separation logic for C expressions

Observation: non-determinism corresponds to concurrency
Idea: use the separation logic rule for parallel composition

\[
\begin{array}{c}
\{P_1\} e_1 \{Q_1\} \quad \{P_2\} e_2 \{Q_2\} \\
\{P_1 \ast P_2\} e_1 \Diamond e_2 \{Q_1 \ast Q_2\}
\end{array}
\]

What does this mean:

▶ Split the memory into two disjoint parts
▶ Prove that \(e_1\) and \(e_2\) can be executed safely in their part
▶ Now \(e_1 \Diamond e_2\) can be executed safely in the whole memory

Disjointness \(\Rightarrow\) no sequence point violation (accessing the same location twice in one expression)
Non-determinism and sequence points

Hoare triples

Expression judgment: \( \{P\} e \{Q\} \)
Non-determinism and sequence points

Hoare triples

Expression judgment: \{P\} e \{Q\}

\[Q: \text{val} \rightarrow \text{assert}\]

If \(P\) holds beforehand, then

- \(e\) does not crash
- \(Q\ v\) holds afterwards when terminating with \(v\)
Non-determinism and sequence points

Some actual rules

Binary operators:

\[
\forall v_1 v_2. (Q_1 v_1 \ast Q_2 v_2) \models \exists v'. (v_1 \oplus v_2) \Downarrow v' \land Q' v'
\]

\[
\{P_1 \ast P_2\} e_1 \otimes e_2 \{Q'\}
\]
Non-determinism and sequence points

Some actual rules

**Binary operators:**

\[
\forall v_1\ v_2 . (Q_1\ v_1 \ast Q_2\ v_2) \models \exists v' . ((v_1 \odot v_2) \downarrow v' \land Q'\ v')
\]

\[
\{ P_1 \ast P_2 \} e_1 \odot e_2 \{ Q' \}
\]

**Simple assignments:**

\[
\forall p\ v . (Q_1\ p \ast Q_2\ v) \models \exists v' . ((\tau) v \downarrow v' \land \\
((p \overset{\gamma}{\mu} - : \tau) \ast ((p \overset{\text{lock}}{\mu} \rightarrow \mid v' | \circ : \tau) \rightarrow Q'\ v')))
\]

\[
\{ P_1 \ast P_2 \} e_1 := e_2 \{ Q' \}
\]
Non-determinism and sequence points

Some actual rules

Binary operators:

\[
\begin{align*}
\forall v_1 \, v_2 . \ (Q_1 \, v_1 \, \ast \, Q_2 \, v_2) & \models \ \exists v' . \ (v_1 \, \odot \, v_2) \Downarrow \ v' \ \land \ Q' \ v' \\
\{ P_1 \ast \, P_2 \} \ e_1 \, \odot \, e_2 \ {\{ Q' \}}
\end{align*}
\]

Simple assignments:

\[
\begin{align*}
\forall p \, v . \ (Q_1 \, p \, \ast \, Q_2 \, v) & \models \ \exists v' . \ (\tau) \, v \, \Downarrow \ v' \ \land \\
& \quad \quad ((p \overset{\gamma}{\mu} \rightarrow - \tau) \ast \ ((p \overset{\text{lock}}{\mu} \rightarrow | \, v' | \, \circ : \tau) \rightarrow Q' \ v')) \\
\{ P_1 \ast \, P_2 \} \ e_1 : = \ e_2 \ {\{ Q' \}}
\end{align*}
\]

Comma:

\[
\begin{align*}
\{ P \} \ e_1 \, \{ \lambda \_ . \ P' \, \diamond \} \quad \{ P' \} \ e_2 \, \{ Q \} \\
\{ P \} \ (e_1 , \ e_2) \ {\{ Q \}}
\end{align*}
\]
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 x * 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;
}
```

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);
```
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
### Others (e.g. CompCert)

<table>
<thead>
<tr>
<th>Our approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory: a finite map of objects which consist of <strong>arrays</strong> of bytes</td>
</tr>
<tr>
<td>Pointers: pairs ((o, i)) where (o) identifies the object, and (i) the <strong>offset</strong> into that object</td>
</tr>
</tbody>
</table>

Too little information to formalize strict-aliasing
## 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 <em>arrays</em> of bytes</td>
<td>A finite map of objects which consist of well-typed <em>trees</em> of bits</td>
</tr>
<tr>
<td>Pointers: pairs ((o, i)) where (o) identifies the object, and (i) the offset into that object</td>
<td>Pairs ((o, r)) where (o) identifies the object, and (r) the path through the tree 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>Too little information to formalize strict-aliasing</td>
<td>A semantics for strict-aliasing</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:
Theorem (Strict-aliasing)

Given:
- addresses $\Gamma, \Delta \vdash a_1 : \sigma_1$ 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
Theorem (Strict-aliasing)

Given:

- addresses $\Gamma, \Delta \vdash a_1 : \sigma_1$ 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
\[ \begin{align*}
    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} := \text{si}^? \; k \\
    \tau & \in \text{ctype} := \text{void} | \text{def} \; x | \tau_1 | \tau^* \\
        & \quad | \tau \mapsto \tau | \tau[e] \\
        & \quad | \text{struct} \; x | \text{union} \; x \\
        & \quad | \text{enum} \; x | \text{typeof} \; e \\
    e & \in \text{ceexpr} := x | \text{const}_{\tau_i} \; z | \text{string} \; \vec{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 \; |\!| \; e_2 \\
        & \quad | (e_1, e_2) | e_1 \; ? \; e_2 : e_3 \\
    r & \in \text{crefseg} := [e] | \; . \; x \\
    l & \in \text{cinit} := e \mid \{ \vec{r} := l \} \\
    sto & \in \text{cstorage} := \text{static} | \text{extern} | \text{auto} \\
    s & \in \text{cstmt} := e \mid \text{return} \; e? \\
        & \quad | \text{goto} \; x \mid 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} \; \tau \; x | \text{union} \; \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})
\end{align*} \]
CH$_2$O abstract C
Translation to CH$_2$O core C in Coq

- Named variables to De Bruijn indices
CH$_2$O abstract C
Translation to CH$_2$O core C in Coq

- Named variables to De Bruijn indices
- Disambiguate l-values and r-values
Translation to CH₂O core C in Coq

- Named variables to De Bruijn indices
- Disambiguate l-values and r-values
- Sound/complete constant expression evaluation, e.g. in $\tau[e]$

\[
\llbracket e \rrbracket_{\Gamma, \text{locals } \mathcal{P}, m} = \nu \quad \text{iff} \quad S(\mathcal{P}, e, m) \rightarrow^* S(\mathcal{P}, \nu, m)
\]
**CH$_2$O abstract C**
Translation to CH$_2$O core C in Coq

- Named variables to De Bruijn indices
- Disambiguate l-values and r-values
- Sound/complete constant expression evaluation, e.g. in $\tau[e]$
  \[
  \llbracket e \rrbracket_{\Gamma, \text{locals } P, m} = v \quad \text{iff} \quad S(P, e, m) \rightarrow^* S(P, v, m)
  \]
- Simplification of loops, e.g. while($e$) $s$ becomes
  \[
  \text{catch (loop (if ($e'$) skip else throw 0 ; catch $s'$))}
  \]
CH₂O abstract C
Translation to CH₂O core C in Coq

► Named variables to De Bruijn indices
► Disambiguate l-values and r-values
► Sound/complete constant expression evaluation, e.g. in $\tau[e]

$$\llbracket e \rrbracket_{\Gamma, \text{locals } \mathcal{P}, m} = \nu \iff S(\mathcal{P}, e, m) \rightarrow^* S(\mathcal{P}, \nu, m)$$

► Simplification of loops, e.g. while(e) s becomes

  catch (loop (if (e') skip else throw 0; catch s'))

► Expansion of typedef and enum declarations
CH₂O abstract C
Translation to CH₂O core C in Coq

- Named variables to De Bruijn indices
- Disambiguate l-values and r-values
- Sound/complete constant expression evaluation, e.g. in $\tau[e]$
  $$\llbracket e \rrbracket_{\Gamma, \text{locals } P, m} = v \quad \text{iff} \quad S(P, e, m) \xrightarrow{*} S(P, v, m)$$
- Simplification of loops, e.g. $\text{while}(e) \ s$ becomes
  $$\text{catch (loop (if } e' \text{ skip else throw 0 ; catch } s'))$$
- Expansion of \texttt{typedef} and \texttt{enum} declarations
- Translation of constants like \texttt{INT_MIN}
CH₂O abstract C
Translation to CH₂O core C in Coq

- Named variables to De Bruijn indices
- Disambiguate l-values and r-values
- Sound.complete constant expression evaluation, e.g. in $\tau[e]$
  \[
  \llbracket e \rrbracket_{\Gamma, \text{locals } P, m} = v \quad \text{iff} \quad S(P, e, m) \rightarrow^* S(P, v, m)
  \]
- Simplification of loops, e.g. while($e$) $s$ becomes
catch (loop (if ($e'$) skip else throw 0 ; catch $s'$))
- Expansion of typedef and enum declarations
- Translation of constants like INT_MIN
- Translation of compound literals

Theorem (Type soundness)
Translation only produces well-typed CH₂O core C programs
CH$_2$O abstract C
Translation to CH$_2$O core C in Coq

- Named variables to De Bruijn indices
- Disambiguate l-values and r-values
- Sound/complete constant expression evaluation, e.g. in $\tau[e]$

$$\llbracket e \rrbracket_{\Gamma, \text{locals } \mathcal{P}, \mathcal{M}} = v \iff \mathbf{S}(\mathcal{P}, e, \mathcal{M}) \rightarrow^* \mathbf{S}(\mathcal{P}, v, \mathcal{M})$$

- Simplification of loops, e.g. while($e$) $s$ becomes

  catch (loop (if ($e'$) skip else throw 0; catch $s'$))

- Expansion of typedef and enum declarations
- Translation of constants like INT_MIN
- Translation of compound literals

Theorem (Type soundness)

*Translation only produces well-typed CH$_2$O core C programs*
Conclusion

- Large part of the C11 standard formalized in Coq
- Many oddities in the C11 standard text discovered
- Metatheory is important to establish sanity of specification
- Executable semantics important to test specification
- Extensions of separation logic developed
Future work
STW project “Sovereign” (Radboud University)

Modular and practical verification of C programs

- Develop design patterns to classify critical parts
- Formalize Misra C as a sublanguage of CH$_2$O
- Develop proof infrastructure
- Connect CH$_2$O to CompCert
- Methods for proving security properties
- Case studies at Nuclear Research Group and Rijkswaterstaat
Future work

More features

- Formalized parser and preprocessor
- Floating point arithmetic
- Bitfields and _Bool
- Untyped malloc
- Variadic functions
- Register storage class
- Type qualifiers
- External functions and I/O
Future work

Improve executable semantics

- Better error messages
- Use more efficient data structures
- Perform optimizations
- More desugaring in Coq instead of OCaml
- Use on large test suites (e.g. CSmith or Cerberus tests)
Future work
Symbolic execution for separation logic for expressions

**Expression judgment:** \( \Gamma \vdash \{ P \} e \{ Q \} \)

Invariant

Symbolic execution:

- Use static analysis to determine which objects are written to
- Put read-only objects in invariant:

\[
\frac{A_1 \cdot A_2 \vdash \{ P \} e \{ Q \}}{A_1 \vdash \{ A_2 \cdot P \} e \{ A_2 \cdot Q \}}
\]

- Invariant can be freely shared, but must be maintained by each atomic expression (in sequential C, function calls are atomic)
Future work

Concurrency

- Concurrency primitives: locks, message passing, ...
  - Rule out any racy concurrency
  - Well-understood and easy to reason about [Hobor, Appel, ...]
- Sequentially consistent concurrency
  - Thread-pool semantics
  - Difficult to reason about
  - Works well in separation logic [O’Hearn, Svendsen, Dinsdale-Young, Birkedal, Parkinson, Dreyer, Turon, ...]
  - Not sound with respect to C11 concurrency
- Weak memory concurrency
  - Still open problems w.r.t. semantics [Sewell, Batty, ...]
  - Very challenging in separation logic [Vafeiadis, ...]
Questions

The C standard formalized in Coq

PhD thesis & Coq sources:
http://robbertkrebbers.nl/thesis.html