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

```
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

```
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

```
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

What is this program supposed to do?

The C quiz, question 1

```
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)
  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, . . .* (+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

Why does C use underspecification that heavily?

Pros for optimizing compilers:

- More optimizations are possible
- High run-time efficiency
- Easy to support multiple architectures

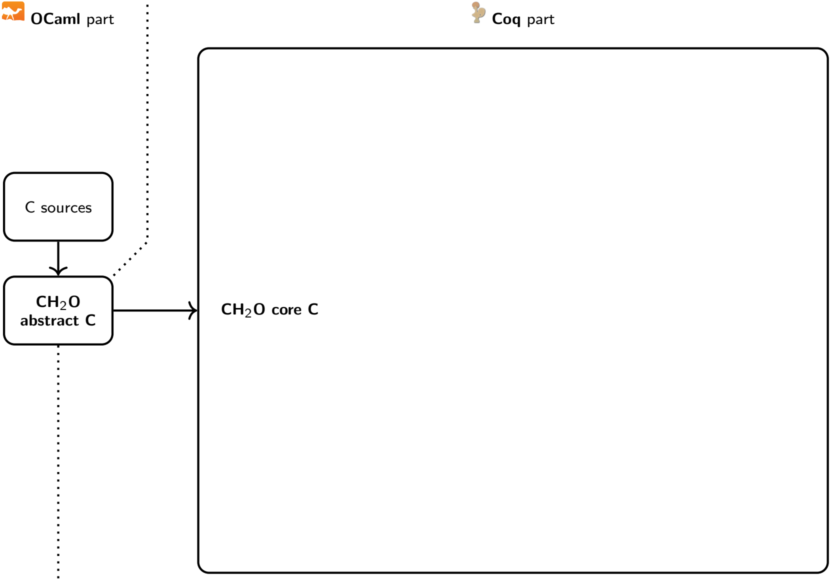
Cons for programmers/formal methods people:

- 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 $\mathbb{K}$
  - Describes most unspecified and undefined behavior
  - No proof assistant support
- **CH$_2$O** (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_2O project



OCaml part

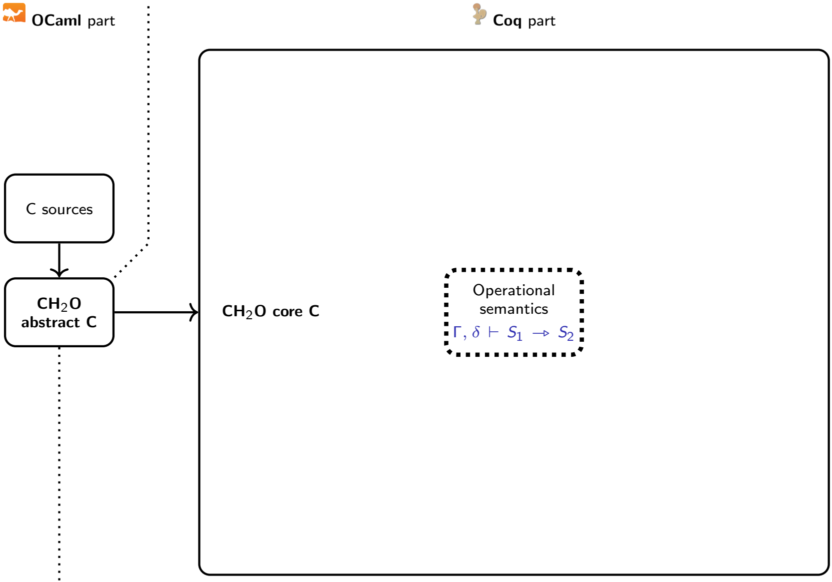
Coq part

C sources

CH_2O abstract C

CH_2O core C

The CH_2O project



OCaml part

Coq part

C sources

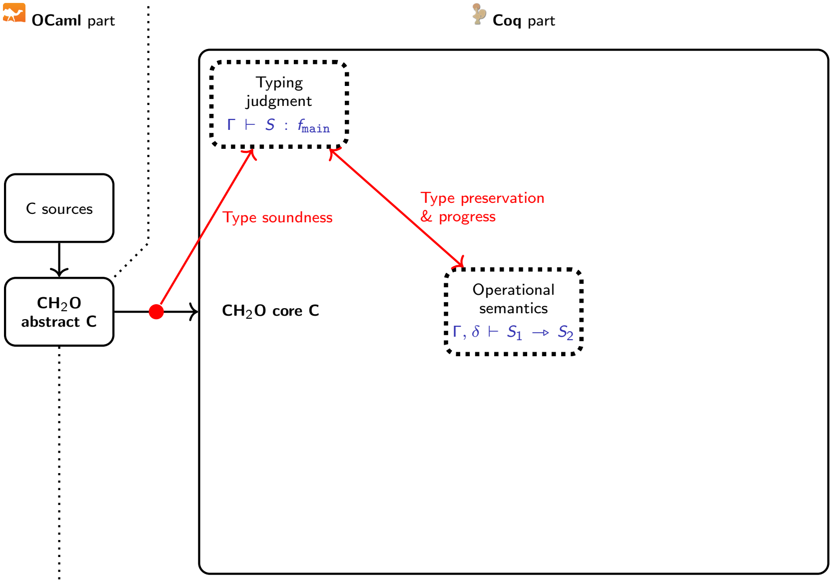
CH_2O abstract C

CH_2O core C

Operational semantics

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

The CH_2O project



OCaml part

Coq part

Typing judgment

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

C sources

CH_2O abstract C

Type soundness

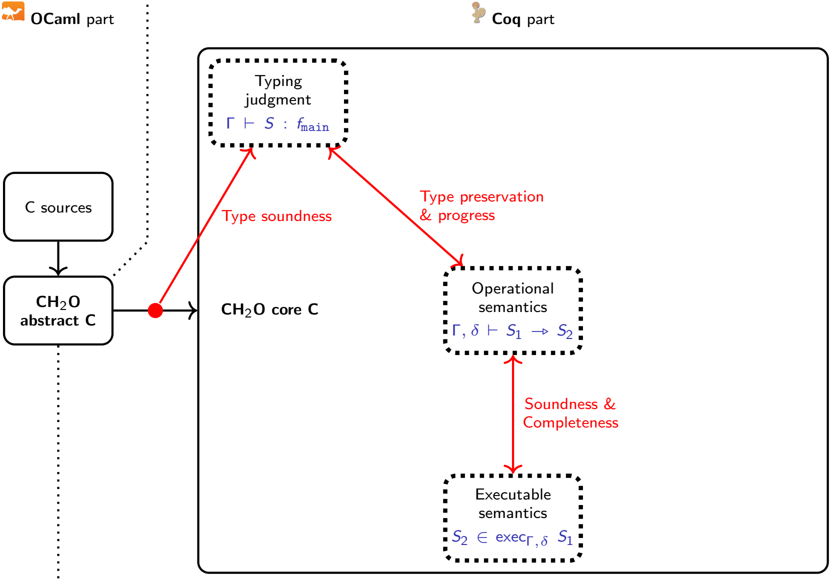
Type preservation & progress

CH_2O core C

Operational semantics

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

The CH_2O project



OCaml part

Coq part

C sources

CH_2O abstract C

CH_2O core C

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

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

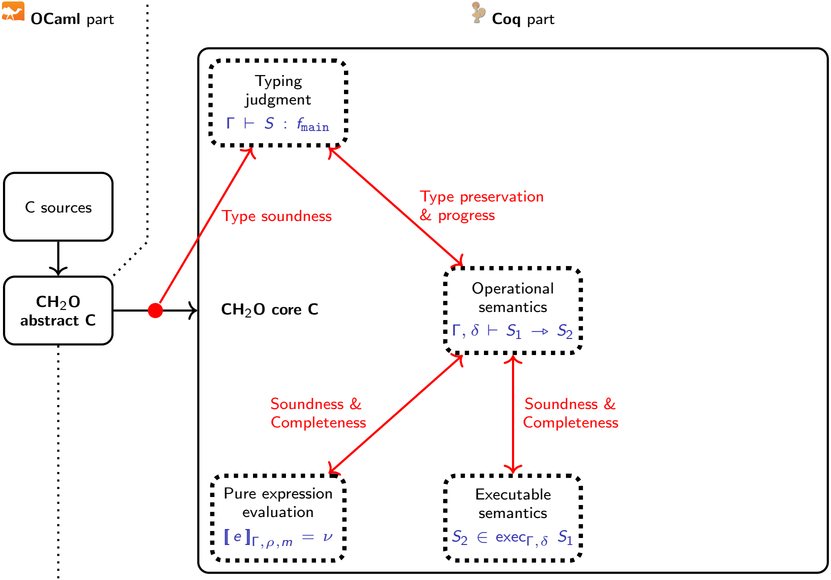
Executable semantics $S_2 \in \mathsf{exec}_{\Gamma,\delta}\ S_1$

Type soundness

Type preservation & progress

Soundness & Completeness

The CH_2O project



OCaml part

Coq part

C sources

CH_2O abstract C

CH_2O core C

Typing judgment

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

Type soundness

Type preservation & progress

Operational semantics

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

Soundness & Completeness

Soundness & Completeness

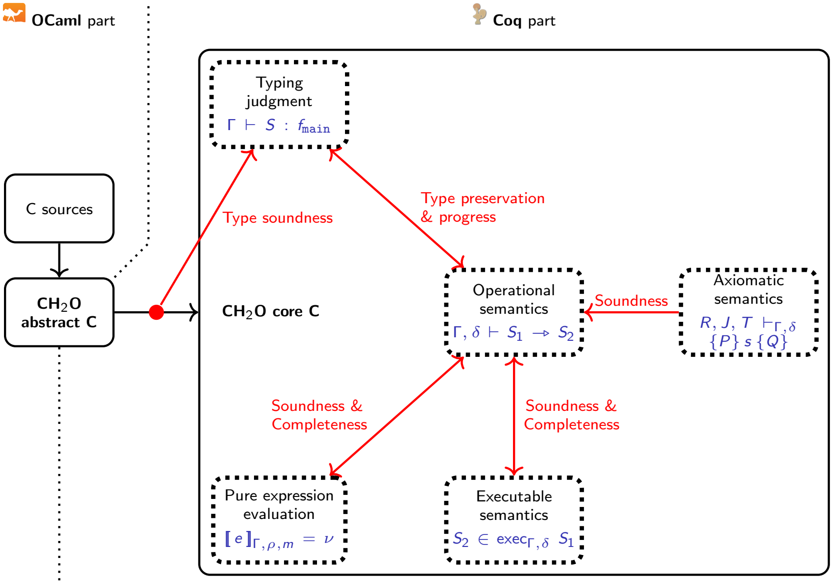
Pure expression evaluation

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

Executable semantics

$S_2 \in \mathsf{exec}_{\Gamma,\delta}\ S_1$

The CH_2O project



OCaml part

Coq part

C sources

CH_2O abstract C

CH_2O core C

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

Type soundness

Type preservation & progress

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

Soundness

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

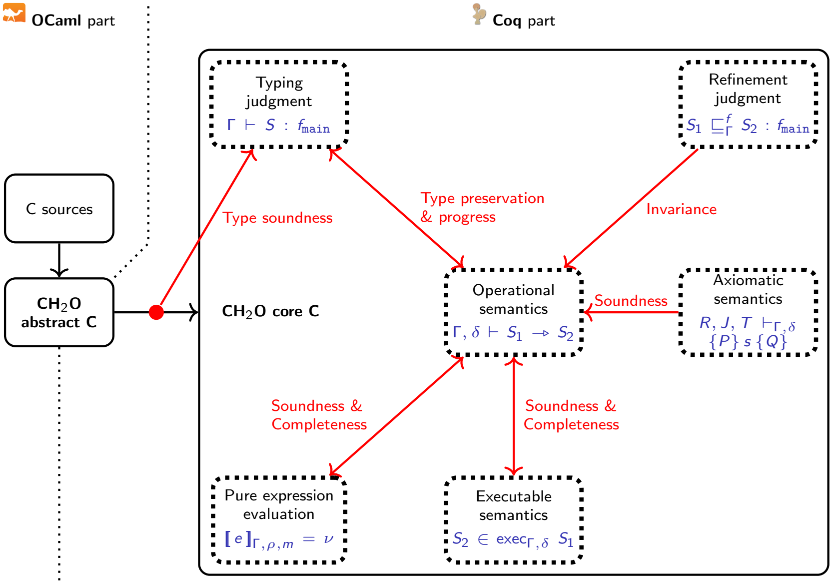
Soundness & Completeness

Soundness & Completeness

Pure expression evaluation $[\![e]\!]_{\Gamma,\rho,m} = \nu$

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

The CH_2O project



OCaml part

Coq part

C sources

CH_2O abstract C

CH_2O core C

Typing judgment

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

Refinement judgment

$S_1 \sqsubseteq^f_\Gamma S_2 : f_{\mathtt{main}}$

Operational semantics

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

Axiomatic semantics

$R, J, T \vdash_{\Gamma,\delta} \{P\}\ s\ \{Q\}$

Pure expression evaluation

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

Executable semantics

$S_2 \in \mathsf{exec}_{\Gamma,\delta}\ S_1$

Type soundness

Type preservation & progress

Invariance

Soundness

Soundness & Completeness

Soundness & Completeness

Non-local control flow and block scope variables

The C quiz, question 2

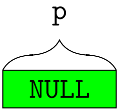
```
int *p = NULL;
l: if (p) {
  return (*p);
} else {
  int j = 17;
  p = &j;
  goto l;
}
```

Non-local control flow and block scope variables

The C quiz, question 2

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

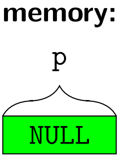
memory:



Non-local control flow and block scope variables

The C quiz, question 2

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



memory:

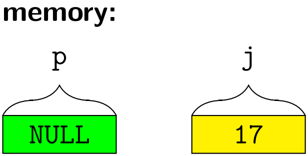
p

NULL

Non-local control flow and block scope variables

The C quiz, question 2

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



memory:

p

NULL

j

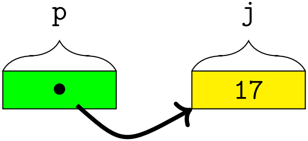
17

Non-local control flow and block scope variables

The C quiz, question 2

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

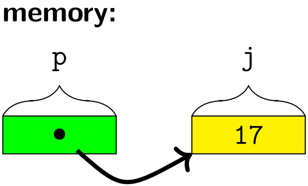
memory:



Non-local control flow and block scope variables

The C quiz, question 2

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

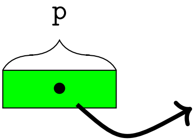


Non-local control flow and block scope variables

The C quiz, question 2

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

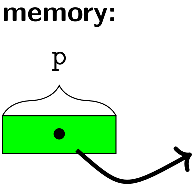
memory:



Non-local control flow and block scope variables

The C quiz, question 2

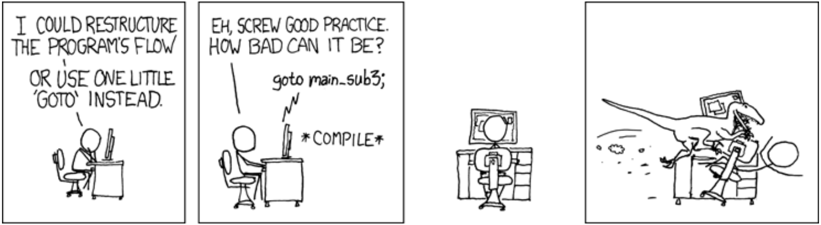
```
int *p = NULL;
l: if (p) {
  return (*p);
} else {
  int j = 17;
  p = &j;
  goto l;
}
```



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

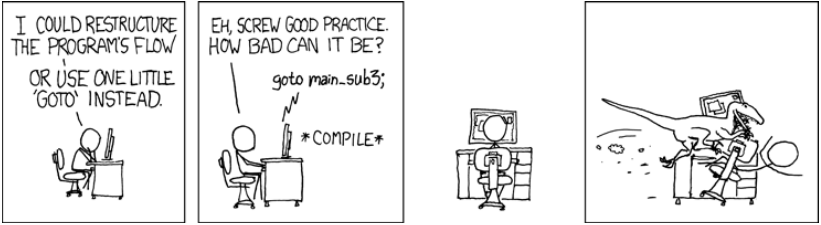
Goto considered harmful?



http://xkcd.com/292/

Non-local control flow and block scope variables

Goto considered harmful?



`http://xkcd.com/292/`

Not necessarily:

$$\vdash \{P\} \ldots \texttt{goto main\_sub3;} \ldots \{Q\}$$

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:

$$R, J, T \vdash \{P\}\, s\, \{Q\}$$

where:

- $\{P\}\, s\, \{Q\}$ 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 `return`
- J maps labels to their jumping condition
  When executing a `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 $J\,\texttt{l}$ has to hold
- T maps `break`s/`continue`s 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 J `l` has to hold
- T maps `breaks`/`continues` to their jumping condition

Example:

$$\frac{}{R, J, T \vdash \{J\, l\}\, \text{goto}\ l\, \{Q\}} \qquad \frac{}{R, J, T \vdash \{J\, l\}\, l :\ \{J\, l\}}$$

Non-local control flow and block scope variables

The block scope variable rule

$$\frac{\begin{array}{r} R\uparrow * x_0 \mapsto -,\ J\uparrow * (x_0 \mapsto - : \tau),\ T\uparrow * (x_0 \mapsto - : \tau) \\ \vdash \{P\uparrow * (x_0 \mapsto - : \tau)\}\ s\ \{Q\uparrow * (x_0 \mapsto - : \tau)\} \end{array}}{R, J, T \vdash \{P\}\ \mathsf{local}_\tau\ s\ \{Q\}}$$

When entering a block:

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

When leaving a block: the reverse

Non-local control flow and block scope variables

The block scope variable rule

$$\frac{\begin{array}{l} R\uparrow * x_0 \mapsto -, J\uparrow * (x_0 \mapsto - : \tau), T\uparrow * (x_0 \mapsto - : \tau) \\ \qquad\qquad \vdash \{P\uparrow * (x_0 \mapsto - : \tau)\}\, s\, \{Q\uparrow * (x_0 \mapsto - : \tau)\} \end{array}}{R, J, T \vdash \{P\}\, \mathsf{local}_\tau\, s\, \{Q\}}$$

When entering a block:

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

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

```
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);
}
```

Non-determinism and sequence points

The C quiz, question 3

```
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

$$\frac{\{P_1\}\, e_1\, \{Q_1\} \qquad \{P_2\}\, e_2\, \{Q_2\}}{\{P_1 * P_2\}\, e_1 \circledcirc e_2\, \{Q_1 * Q_2\}}$$

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

$$\frac{\{P_1\}\, e_1\, \{Q_1\} \qquad \{P_2\}\, e_2\, \{Q_2\}}{\{P_1 * P_2\}\, e_1 \circledcirc e_2\, \{Q_1 * Q_2\}}$$

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 $e_1 \circledcirc e_2$ 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

$$\frac{\{P_1\}\, e_1 \,\{Q_1\} \qquad \{P_2\}\, e_2 \,\{Q_2\}}{\{P_1 * P_2\}\, e_1 \circledcirc e_2 \,\{Q_1 * Q_2\}}$$

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 $e_1 \circledcirc e_2$ can be executed safely in the whole memory

Disjointness ⇒ 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: $\{P\}\, e\, \{Q\}$

$Q : \mathrm{val} \to \mathrm{assert}$

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:

$$\frac{\{P_1\}\, e_1\, \{Q_1\} \qquad \{P_2\}\, e_2\, \{Q_2\} \qquad \forall v_1\, v_2\, .\, (Q_1\, v_1 * Q_2\, v_2 \models \exists v'\, .\, (v_1 \circledcirc v_2) \Downarrow v' \wedge Q'\, v')}{\{P_1 * P_2\}\, e_1 \circledcirc e_2\, \{Q'\}}$$

Non-determinism and sequence points

Some actual rules

Binary operators:

$$\frac{\{P_1\}\, e_1\, \{Q_1\} \qquad \{P_2\}\, e_2\, \{Q_2\} \\ \forall v_1\, v_2 \,.\, (Q_1\, v_1 * Q_2\, v_2 \models \exists v' \,.\, (v_1 \circledcirc v_2) \Downarrow v' \land Q'\, v')}{\{P_1 * P_2\}\, e_1 \circledcirc e_2\, \{Q'\}}$$

Simple assignments:

$$\frac{\begin{array}{c} \{P_1\}\, e_1\, \{Q_1\} \qquad \{P_2\}\, e_2\, \{Q_2\} \qquad \text{Writable} \subseteq \text{kind}\ \gamma \\ \forall p\, v \,.\, \Big(Q_1\, p * Q_2\, v \models \exists v' \,.\, (\tau) v \Downarrow v' \land \\ \qquad\qquad ((p \xmapsto[\mu]{\gamma} - : \tau) * ((p \xmapsto[\mu]{\text{lock}\ \gamma} |\, v' \,|_\circ : \tau) \mathrel{-\!\!*} Q'\, v'))\Big) \end{array}}{\{P_1 * P_2\}\, e_1 := e_2\, \{Q'\}}$$

Non-determinism and sequence points

Some actual rules

Binary operators:

$$\frac{\{P_1\}\, e_1\, \{Q_1\} \qquad \{P_2\}\, e_2\, \{Q_2\} \qquad \forall v_1\, v_2\, .\, (Q_1\, v_1 * Q_2\, v_2 \models \exists v'\, .\, (v_1 \circledcirc v_2) \Downarrow v' \wedge Q'\, v')}{\{P_1 * P_2\}\, e_1 \circledcirc e_2\, \{Q'\}}$$

Simple assignments:

$$\frac{\{P_1\}\, e_1\, \{Q_1\} \qquad \{P_2\}\, e_2\, \{Q_2\} \qquad \text{Writable} \subseteq \text{kind}\ \gamma \qquad \forall p\, v\, .\, \Big(Q_1\, p * Q_2\, v \models \exists v'\, .\, (\tau)v \Downarrow v' \wedge ((p \xmapsto[\mu]{\gamma} - : \tau) * ((p \xmapsto[\mu]{\text{lock}\ \gamma} |\, v'\, |_\circ : \tau) \twoheadrightarrow Q'\, v'))\Big)}{\{P_1 * P_2\}\, e_1 := e_2\, \{Q'\}}$$

Comma:

$$\frac{\{P\}\, e_1\, \{\lambda\_\, .\, P'\, \Diamond\} \qquad \{P'\}\, e_2\, \{Q\}}{\{P\}\, (e_1, e_2)\, \{Q\}}$$

Strict-aliasing

What is aliasing?

Aliasing: multiple pointers referring to the same object

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

Strict-aliasing

What is aliasing?

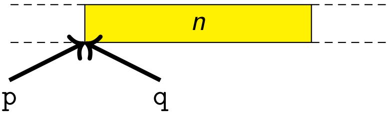
Aliasing: multiple pointers referring to the same object

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

Tricky with functions:

```
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?

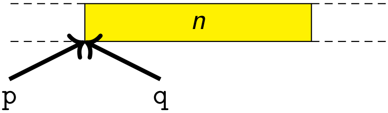
Aliasing: multiple pointers referring to the same object

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

Tricky with functions:

```
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



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:

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

Strict-aliasing

Alias analysis

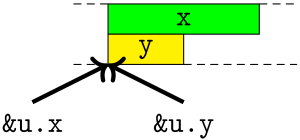
Alias analysis: to determine whether pointers can alias

Consider a similar function:

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

And call it with aliased pointers:

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



Strict-aliasing

Alias analysis

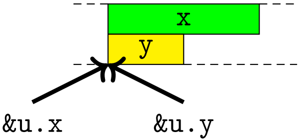
Alias analysis: to determine whether pointers can alias

Consider a similar function:

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

And call it with aliased pointers:

```
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

Others (e.g. CompCert)	**Our approach**
Memory: a finite map of objects which consist of arrays of bytes	
Pointers: pairs (o, i) where o identifies the object, and i the offset into that object	
Too little information to formalize strict-aliasing	

Strict-aliasing

How to treat pointers

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

Strict-aliasing

How to treat pointers

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

Strict-aliasing

How to treat pointers

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

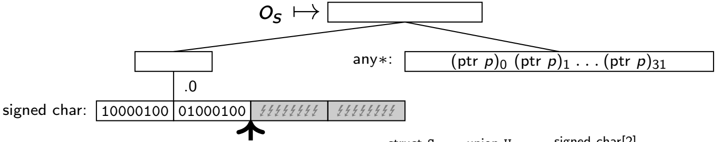
Strict-aliasing

Example of the memory as a structured forest

Consider:

```
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:



$$p = (o_s : \text{struct } \mathtt{S}, \xhookrightarrow{\text{struct } \mathtt{S}} 0 \xhookrightarrow{\text{union } \mathtt{U}} \bullet\ 0 \xhookrightarrow{\text{signed char[2]}} 0, 16)_{\text{signed char} >_* \text{void}}$$

Strict-aliasing

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
- σ_1 *not a subtype of* σ_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*

Strict-aliasing

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
- σ_1 *not a subtype of* σ_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

CH$_2$O abstract C

$$
\begin{aligned}
x \in \text{string} &::= \text{Set of strings} \\
k \in \text{cintrank} &::= \text{char} \mid \text{short} \mid \text{int} \\
&\mid \text{long} \mid \text{long long} \mid \text{ptr} \\
si \in \text{signedness} &::= \text{signed} \mid \text{unsigned} \\
\tau_\text{i} \in \text{cinttype} &::= si^? \; k \\
\tau \in \text{ctype} &::= \text{void} \mid \text{def } x \mid \tau_\text{i} \mid \tau * \\
&\mid \overrightarrow{\tau \, x^?} \to \tau \mid \tau[e] \\
&\mid \text{struct } x \mid \text{union } x \\
&\mid \text{enum } x \mid \text{typeof } e \\
e \in \text{cexpr} &::= x \mid \text{const}_{\tau_\text{i}} \; z \mid \text{string } \vec{z} \\
&\mid \text{sizeof } \tau \mid \text{alignof } \tau \\
&\mid \text{offsetof } \tau \; x \\
&\mid \tau \text{ min} \mid \tau \text{ max} \mid \tau \text{ bits} \\
&\mid \& e \mid * e \mid e \; . \; x \\
&\mid e_1 \; \alpha \; e_2 \\
&\mid e(\vec{e}) \mid \text{alloc}_\tau \; e \mid \text{free } e \\
&\mid \circledcirc_u \; e \mid e_1 \circledcirc e_2 \mid (\tau) I \\
&\mid e_1 \; \&\& \; e_2 \mid e_1 \; || \; e_2 \\
&\mid (e_1 , e_2) \mid e_1 \; ? \; e_2 : e_3 \\
r \in \text{crefseg} &::= [e] \mid .x
\end{aligned}
$$

$$
\begin{aligned}
I \in \text{cinit} &::= e \mid \{ \overrightarrow{\vec{r} := I} \} \\
sto \in \text{cstorage} &::= \text{static} \mid \text{extern} \mid \text{auto} \\
s \in \text{cstmt} &::= e \mid \text{return } e^? \\
&\mid \text{goto } x \mid x : s \\
&\mid \text{break} \mid \text{continue} \\
&\mid \{ s \} \\
&\mid \overrightarrow{sto} \; \tau \; x := I^? \; ; \; s \\
&\mid \text{typedef } x := \tau \; ; \; s \\
&\mid \text{skip} \mid s_1 \; ; \; s_2 \\
&\mid \text{while}(e) \; s \\
&\mid \text{do } s \text{ while}(e) \\
&\mid \text{for}(e_1 \; ; \; e_2 \; ; \; e_3) \; s \\
&\mid \text{if } (e) \; s_1 \text{ else } s_2 \\
d \in \text{decl} &::= \text{struct } \overrightarrow{\tau \, x} \mid \text{union } \overrightarrow{\tau \, x} \\
&\mid \text{enum } \overrightarrow{x := e^?} : \tau_\text{i} \\
&\mid \text{typedef } \tau \\
&\mid \text{global } I^? : \overrightarrow{sto} \; \tau \\
&\mid \text{fun } s : \overrightarrow{sto} \; \tau \\
\Theta \in \text{decls} &:= \text{list } (\text{string} \times \text{decl})
\end{aligned}
$$

CH_2O abstract C

Translation to CH_2O core C in Coq

- Named variables to De Bruijn indices

CH_2O abstract C

Translation to CH_2O core C in Coq

- Named variables to De Bruijn indices
- Disambiguate l-values and r-values

CH_2O abstract C

Translation to CH_2O core C in Coq

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

$$[\![\, e \,]\!]_{\Gamma,\mathsf{locals}\,\mathcal{P},m} = v \quad \text{iff} \quad \mathbf{S}(\mathcal{P},\, e,\, m) \rightharpoonup^{*} \mathbf{S}(\mathcal{P},\, v,\, m)$$

CH_2O abstract C

Translation to CH_2O core C in Coq

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

$$[\![e]\!]_{\Gamma,\mathrm{locals}\,\mathcal{P},m} = v \quad \text{iff} \quad \mathbf{S}(\mathcal{P},\, e,\, m) \rightharpoonup^{*} \mathbf{S}(\mathcal{P},\, v,\, m)$$

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

$$\mathsf{catch}\ (\mathsf{loop}\ (\mathsf{if}\ (e')\ \mathsf{skip}\ \mathsf{else}\ \mathsf{throw}\ 0\,;\ \mathsf{catch}\ s'))$$

CH_2O abstract C

Translation to CH_2O core C in Coq

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

$$[\![\, e \,]\!]_{\Gamma,\mathsf{locals}\,\mathcal{P},m} = v \quad \text{iff} \quad \mathbf{S}(\mathcal{P},\, e,\, m) \rightharpoonup^{*} \mathbf{S}(\mathcal{P},\, v,\, m)$$

- Simplification of loops, *e.g.* $\mathsf{while}(e)\; s$ becomes

$$\mathsf{catch}\;(\mathsf{loop}\;(\mathsf{if}\;(e')\;\mathsf{skip}\;\mathsf{else}\;\mathsf{throw}\;0\,;\;\mathsf{catch}\;s'))$$

- Expansion of `typedef` and `enum` declarations

CH_2O abstract C

Translation to CH_2O core C in Coq

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

$$[\![\, e \,]\!]_{\Gamma,\mathsf{locals}\,\mathcal{P},m} = v \quad \text{iff} \quad \mathbf{S}(\mathcal{P},\, e,\, m) \rightharpoonup^{*} \mathbf{S}(\mathcal{P},\, v,\, m)$$

- Simplification of loops, *e.g.* $\mathsf{while}(e)\ s$ becomes

$$\mathsf{catch}\ (\mathsf{loop}\ (\mathsf{if}\ (e')\ \mathsf{skip}\ \mathsf{else}\ \mathsf{throw}\ 0\,;\ \mathsf{catch}\ s'))$$

- Expansion of `typedef` and `enum` declarations
- Translation of constants like `INT_MIN`

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 $\tau[e]$

$$[\![e]\!]_{\Gamma,\mathsf{locals}\,\mathcal{P},m} = v \quad \text{iff} \quad \mathbf{S}(\mathcal{P},\, e,\, m) \rightharpoonup^* \mathbf{S}(\mathcal{P},\, v,\, m)$$

- Simplification of loops, *e.g.* $\mathsf{while}(e)\ s$ becomes

$$\mathsf{catch}\ (\mathsf{loop}\ (\mathsf{if}\ (e')\ \mathsf{skip}\ \mathsf{else}\ \mathsf{throw}\ 0\ ;\ \mathsf{catch}\ s'))$$

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

CH_2O abstract C

Translation to CH_2O core C in Coq

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

$$[\![e]\!]_{\Gamma,\mathrm{locals}\,\mathcal{P},m} = v \quad \text{iff} \quad \mathbf{S}(\mathcal{P},\, e,\, m) \rightharpoonup^{*} \mathbf{S}(\mathcal{P},\, v,\, m)$$

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

$$\text{catch (loop (if } (e') \text{ skip else throw } 0\,;\ \text{catch } s'))$$

- 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_2O core C programs

Conclusion

Formal methods can be applied to real programming languages

- 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_2O
- Develop proof infrastructure
- Connect CH_2O 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: $A \vdash \{P\}\, e\, \{Q\}$

Invariant

Symbolic execution:

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

$$\frac{A_1 * A_2 \vdash \{P\}\, e\, \{Q\}}{A_1 \vdash \{A_2 * P\}\, e\, \{A_2 * Q\}}$$

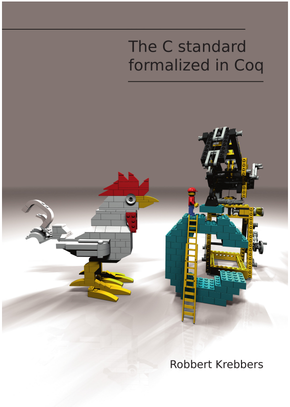
- Invariant can be freely shared, but must be maintained by each atomic expression (in sequential C, function calls are atomic)

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

Robbert Krebbers

PhD thesis & Coq sources:

`http://robbertkrebbers.nl/thesis.html`