Formalization of C:
What we have learned and beyond

Robbert Krebbers

Radboud University, The Netherlands

Now at: Aarhus University, Denmark

December 2, 2015

What is this program supposed to do?

```
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?

```
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?

```
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?

```
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?

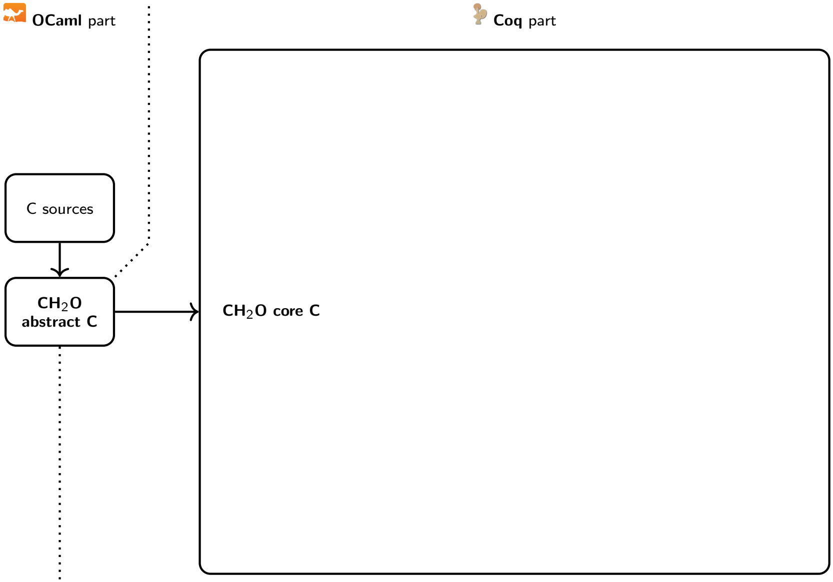
Pros for optimizing compilers:

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

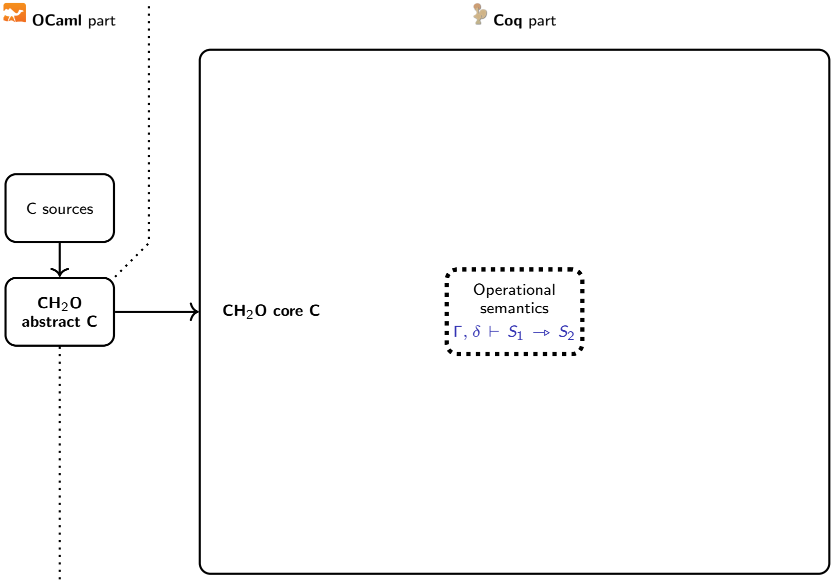
Cons for programmers/formal methods people:

- Portability and maintenance problems
- Hard to capture precisely in a semantics
- Hard to formally reason about

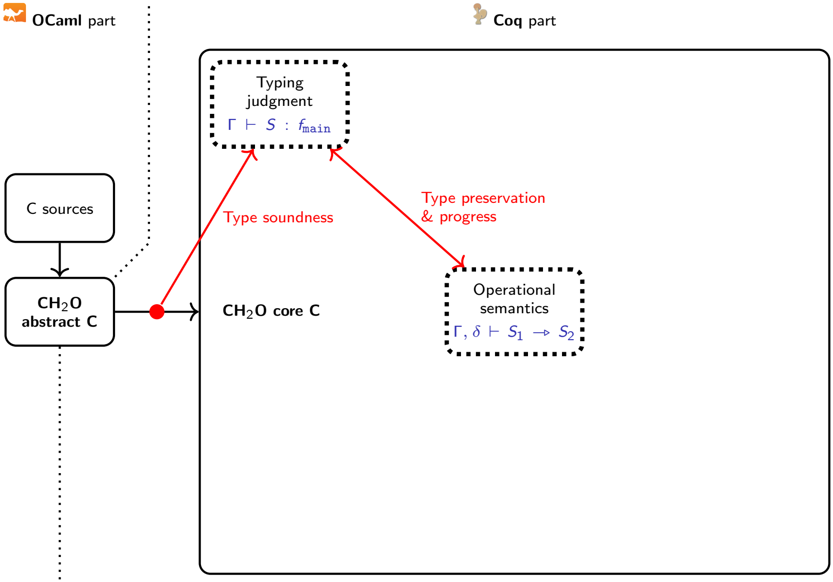
The CH_2O project



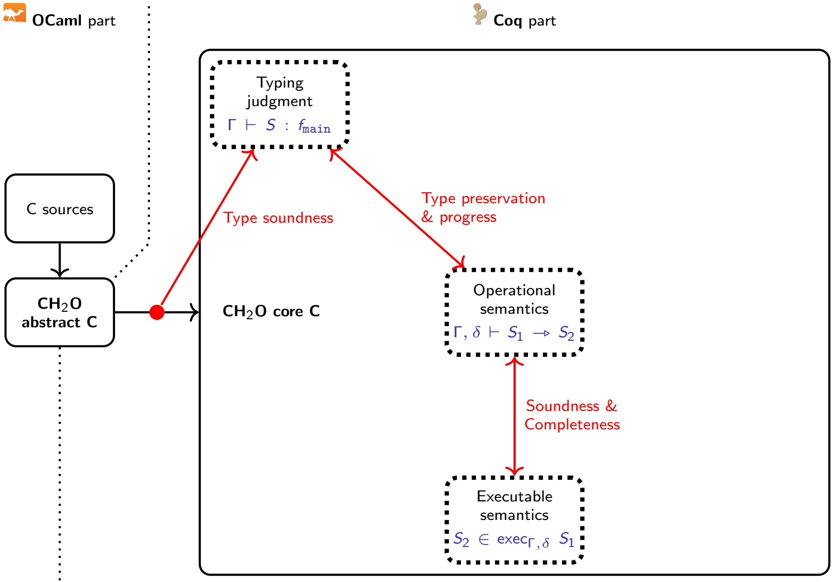
The CH_2O project



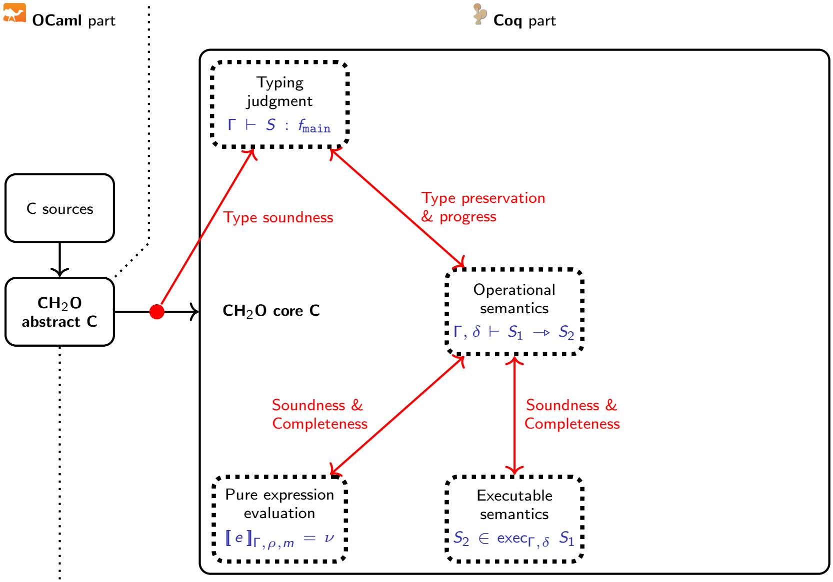
The CH_2O project



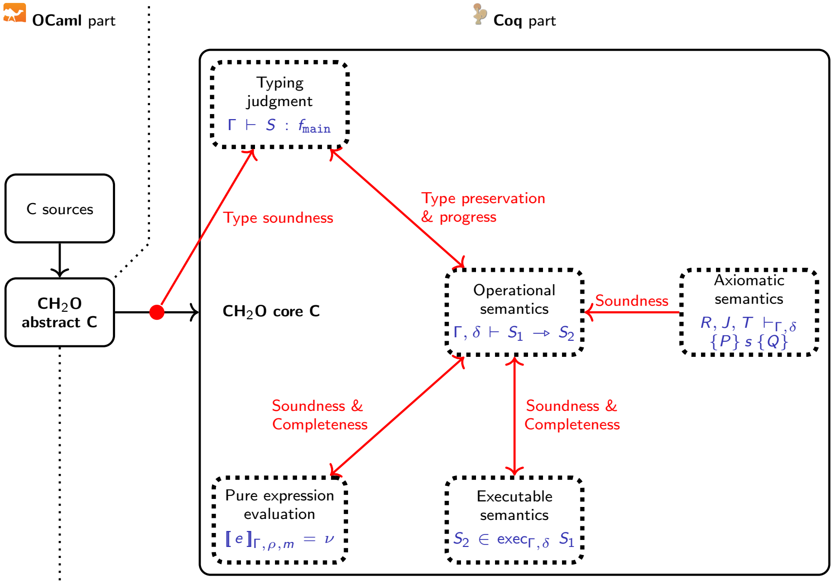
The CH_2O project



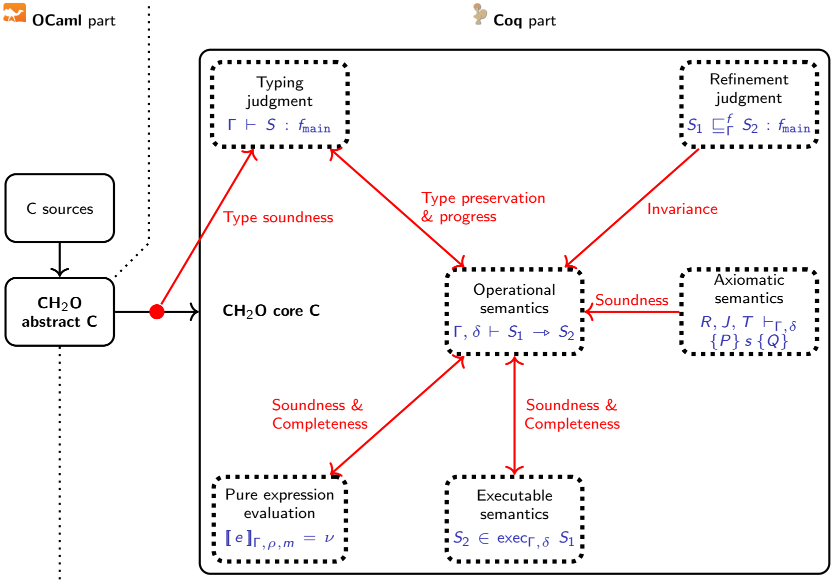
The CH_2O project



The CH_2O project



The CH_2O project



Part 1

Our experience with standardization

Does this have to print the same value?

```
int a[1];
/* intentionally uninitialized */

printf("%d\n", a[0]);
printf("%d\n", a[0]);
```

Does this have to print the same value?

```
unsigned char a[1];
/* intentionally uninitialized */

printf("%d\n", a[0]);
printf("%d\n", a[0]);
```

Does this have to print the same value?

```
unsigned char a[1];
/* intentionally uninitialized */

printf("%d\n", a[0]);
printf("%d\n", a[0]);
```

For types without trap values (*e.g.* `unsigned char` or `int32_t`):

indeterminate value = unspecified value

What can we do with these values?

Defect Report # 260

Question (2001-09-07):

> *If an object holds an indeterminate value, can that value change other than by an explicit action of the program?*

Defect Report # 260

Question (2001-09-07):

> *If an object holds an indeterminate value, can that value change other than by an explicit action of the program?*

Answer (2003-03-06):

> *An object with indeterminate value has a bit pattern representation which remains constant during its lifetime.*

Answer (2004-09-28):

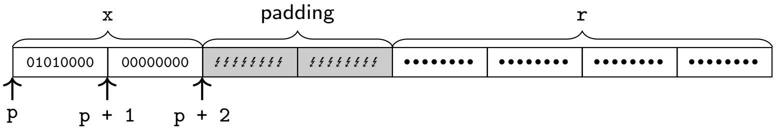
> *In the case of an indeterminate value [. . .] the actual bit-pattern may change without direct action of the program.*

Status of Defect Report # 260

- Decided no change to the standard text was needed
- Defect report about C99
- Defect report superseded by C11
- All relevant text in C11 identical to the same text in C99

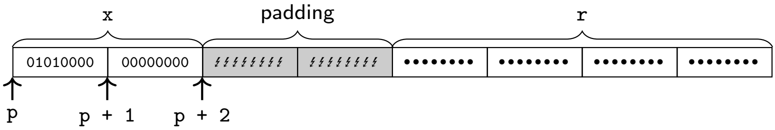
Why do we care about indeterminate values?

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



Why do we care about indeterminate values?

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



Byte-wise copy:

```
struct S s2;
for (size_t i = 0; i < sizeof(struct S); i++)
  ((unsigned char*)&s2)[i] = ((unsigned char*)&s1)[i];
```

Defect Report # 451 [Krebbers & Wiedijk 2013]

Question (2013-08-30):

> *Can an uninitialized variable with automatic storage duration [. . .] change its value without direct action of the program?*

Answer (2014-04-07):

> *an uninitialized value under the conditions described can appear to change its value.*
> *[. . .]*
> *This viewpoint reaffirms the C99 DR260 position.*
> *[. . .]*
> *The committee agrees that this area would benefit from a new definition of something akin to a "wobbly" value and that this should be considered in any subsequent revision of this standard.*

Resolution in CH_2O

Special indeterminate "wobbly" bit:

```
Inductive bit :=
  | BIndet : bit
  | BBit : bool → bit
  | BPtr : ptr_bit → bit.
```

- Indeterminate bits can be copied as `unsigned char`
- Operations on values with indeterminate bits (cast, addition, if-then-else, ...) give undefined behavior

Resolution in CH_2O

Special indeterminate "wobbly" bit:

```
Inductive bit :=
  | BIndet : bit
  | BBit : bool → bit
  | BPtr : ptr_bit → bit.
```

- Indeterminate bits can be copied as `unsigned char`
- Operations on values with indeterminate bits (cast, addition, if-then-else, ...) give undefined behavior

Possibly too much undefined behavior, but that is sound for program verification

Part 2
Separation logic for C

Non-determinism and sequence points

```
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

```
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

Brief introduction to separation logic [Reynolds *et al.*]

Hoare triple $\{P\}\, s\, \{Q\}$: if P holds beforehand, then:

- s does not crash
- Q holds afterwards when terminating with v

Brief introduction to separation logic [Reynolds *et al.*]

Hoare triple $\{P\}\, s\, \{Q\}$: if P holds beforehand, then:

- s does not crash
- Q holds afterwards when terminating with v

Separating conjunction $P * Q$: subdivide the memory into disjoint parts P and Q

Points-to predicate $a \mapsto v$: the memory consists of only a value v at address a

Example: $\{\mathtt{x} \mapsto 0 * \mathtt{y} \mapsto 0\}\, \mathtt{x:=10;\ y:=12}\, \{\mathtt{x} \mapsto 10 * \mathtt{y} \mapsto 12\}$

Brief introduction to separation logic [Reynolds *et al.*]

Hoare triple $\{P\}\, s\, \{Q\}$: if P holds beforehand, then:

- s does not crash
- Q holds afterwards when terminating with v

Separating conjunction $P * Q$: subdivide the memory into disjoint parts P and Q

Points-to predicate $a \mapsto v$: the memory consists of only a value v at address a

Example: $\{\mathtt{x} \mapsto 0 * \mathtt{y} \mapsto 0\}\, \mathtt{x:=10;\ y:=12}\, \{\mathtt{x} \mapsto 10 * \mathtt{y} \mapsto 12\}$

Frame rule: for local reasoning

$$\frac{\{P\}\, s\, \{Q\}}{\{P * R\}\, s\, \{Q * R\}}$$

Separation logic for C expressions

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

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

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

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)

Hoare "triples"

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

Goto/return/switch conditions

Type environments

Hoare "triples"

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

Goto/return/switch conditions

Type environments

Expression judgment: $\vdash_{\Gamma,\delta} \{P\}\, e\, \{Q\}$

Hoare "triples"

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

Goto/return/switch conditions — Type environments — Q : assert

Expression judgment: $\vdash_{\Gamma,\delta} \{P\}\, e\, \{Q\}$

Q : val $\to$ assert

If P holds beforehand, then

- e does not crash
- $Q\, v$ holds afterwards when terminating with v

Some actual rules

Binary operators:

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

Some actual rules

Binary operators:

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

Simple assignments:

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

Some actual rules

Binary operators:

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

Simple assignments:

$$\frac{\vdash_{\Gamma,\delta} \{P_1\}\, e_1\, \{Q_1\} \qquad \vdash_{\Gamma,\delta} \{P_2\}\, e_2\, \{Q_2\} \qquad \text{Writable} \subseteq \text{kind } \gamma \qquad \forall p\, v \,.\, \Big(Q_1\, p * Q_2\, v \models_{\Gamma,\delta} \exists v' \,.\, (\tau) v \Downarrow v' \wedge ((p \overset{\gamma}{\underset{\mu}{\mapsto}} - : \tau) * ((p \overset{\text{lock } \gamma}{\underset{\mu}{\longmapsto}} |\, v' \,|_\circ : \tau) \mathrel{-\!\!*} Q'\, v'))\Big)}{\vdash_{\Gamma,\delta} \{P_1 * P_2\}\, e_1 := e_2\, \{Q'\}}$$

Comma:

$$\frac{\vdash_{\Gamma,\delta} \{P\}\, e_1\, \{\lambda_- \,.\, P'\, \Diamond\} \qquad \vdash_{\Gamma,\delta} \{P'\}\, e_2\, \{Q\}}{\vdash_{\Gamma,\delta} \{P\}\, (e_1, e_2)\, \{Q\}}$$

Part 3

Conclusions & Future work

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

More features

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

Symbolic execution for separation logic for expressions

Expression judgment: $A \vdash_{\Gamma,\delta} \{P\}\, e\, \{Q\}$

Invariant

Symbolic execution:

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

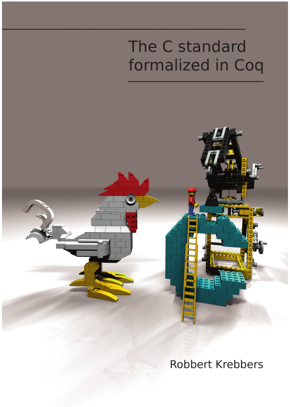
$$\frac{A_1 * A_2 \vdash_{\Gamma,\delta} \{P\}\, e\, \{Q\}}{A_1 \vdash_{\Gamma,\delta} \{A_2 * P\}\, e\, \{A_2 * Q\}}$$

- Invariant can be freely shared, but must be maintained by each atomic expression

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



PhD thesis & Coq sources:

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