Formalizing C in Coq

Robbert Krebbers

ICIS, Radboud University Nijmegen, The Netherlands

October 24, 2014 @ Princeton University, USA
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
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 \textit{x=4,y=7}, seems just left-right
- **GCC** prints \textit{x=4,y=8}, does not correspond to any order

This program violates the sequence point restriction due to two unsequenced writes to \texttt{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* (+57 more)
  Non-determinism

- **Implementation defined behavior**: like unspecified behavior, but the compiler has to document its choice
  *For example: size andendianness 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
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/Challenges** 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
- Specific choices for unspecified/impl-defined behavior
  
  For example: 32-bits ints
- Describes some undefined behavior
  
  Undefined: dereferencing NULL, . . .
  
  But still defined: integer overflow, aliasing violations, . . .
- VST separation logic proofs specific to CompCert

**Formalin** (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
The Formalin project (Krebbers & Wiedijk, 2019)

- OCaml part
- Coq part

- CH₂O core syntax
- Type judgment
- Subject red. and progress
- CH₂O operational semantics

Translation: = translation
Proof: = proof

Executable structured memory model
The Formalin project (Krebbers & Wiedijk, 2008)

Executable structured memory model

- .c file
- CIL abstract syntax
- CH₂O abstract syntax
- CH₂O core syntax
- Type soundness
- Type judgment
- Subject red. and progress
- CH₂O operational semantics

OCaml part

Coq part

= translation
= proof

Executable structured memory model
The Formalin project (Krebbers & Wiedijk, 2023)

Executable structured memory model

- CIL abstract syntax
- CH₂O abstract syntax
- CH₂O core syntax
- Stream of finite sets of states

- Type soundness
- Subject red. and progress
- CH₂O operational semantics

- OCaml part
- Coq part

Translation

Soundness and completeness

Type judgment

Subject red. and progress

= translation

= proof

Executable structured memory model
The Formalin project (Krebbers & Wiedijk, 2020)

- **OCaml part**
  - .c file
  - CIL abstract syntax
  - CH₂O abstract syntax

- **Coq part**
  - CH₂O core syntax
  - Stream of finite sets of states

- Type soundness
- Subject red. and progress
- Soundness and completeness
- Separation logic

= translation
= proof

Executable structured memory model
The Formalin project (Krebbers & Wiedijk, 2016)

Executable structured memory model

- CIL abstract syntax
- CH\textsubscript{2}O abstract syntax
- CH\textsubscript{2}O core syntax
- Stream of finite sets of states

**OCaml part**
- .c file
- CIL abstract syntax
- CH\textsubscript{2}O abstract syntax

**Coq part**

- CH\textsubscript{2}O core syntax
- Stream of finite sets of states

**Soundness and completeness**

- Type soundness
- Subject red. and progress
- Preservation and composition
- Soundness

- Refinement judgment
- Separation logic
- Type judgment

- Soundness and completeness

- = translation
- = proof

Executable structured memory model
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
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
  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;
}
```
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

17
```

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;
}
```

memory:

```
    p
```

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;
}
```

**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
Problem: goto/return/etc. affect memory
Non-local control flow and block scope variables
Operational semantics [Krebbers/Wiedijk,FoSSaCS’13]

**Problem:** goto/return/etc. affect memory

**Solution:**
- Execute gotos and returns in small steps
  - Not so much to search for labels, . . .
  - but to naturally perform required allocations and deallocations
Non-local control flow and block scope variables

Operational semantics [Krebbers/Wiedijk,FoSSaCS’13]

Problem: goto/return/etc. affect memory

Solution:
- Execute gotos and returns in small steps
  - Not so much to search for labels, ...
  - but to naturally perform required allocations and deallocations
- Traversal through the AST in the following directions:
  - ↘ downwards to the next statement
  - ↗ upwards to the next statement
  - ↼ l to a label l: after a goto l
  - ↑↑ v to the top of the statement after a return
Non-local control flow and block scope variables
Operational semantics [Krebbers/Wiedijk,FoSSaCS’13]

**Problem:** goto/return/etc. affect memory

**Solution:**
- Execute gotos and returns in small steps
  - Not so much to search for labels, . . .
  - but to naturally perform required allocations and deallocations
- Traversal through the AST in the following directions:
  - \( \searrow \) downwards to the next statement
  - \( \nearrow \) upwards to the next statement
  - \( \circ \) to a label \( l \): after a goto \( l \)
  - \( \uparrow \uparrow \) to the top of the statement after a return
- Use a zipper to keep track of the position and the stack
Non-local control flow and block scope variables

Visualization of the operational semantics on an example

```c
int *p = NULL

l:
    if (p)
        return (*p)
```

```c
int j = 17;
p = &j goto l
```
Non-local control flow and block scope variables
Visualization of the operational semantics on an example

```
i = 0
l:
if (i)
    return (*i)
j = 17
p = &j
goto l
```

Diagram:
- **direction:**
- **memory:**
  - `p`
  - `NULL`
Non-local control flow and block scope variables
Visualization of the operational semantics on an example

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

direction:
memory:

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

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

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

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

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

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

Visualization of the operational semantics on an example

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

Direction:

Memory:
```
NULL
```
Non-local control flow and block scope variables

Visualization of the operational semantics on an example

```c
int *p = NULL

l:
if (p)
    return (*p)

int j = 17;
p = &j
```

Diagram:
- **Direction:**
- **Memory:**
  - `p`: NULL
  - `j`: 17
Non-local control flow and block scope variables

Visualization of the operational semantics on an example

```
int *p = NULL

l:
  if (p)
    return (*p)

int j = 17;
p = &j
  goto l
```
Non-local control flow and block scope variables

Visualization of the operational semantics on an example

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

Visualization of the operational semantics on an example

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

Diagram:
- `direction`: Arrows indicating control flow direction.
- `memory`: Variables `p` and `j` with memory locations.
  - `p` points to `j`.
  - `j` contains the value `17`.
Non-local control flow and block scope variables

Visualization of the operational semantics on an example

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

Visualization of the operational semantics on an example

```
int *p = NULL

l:

if (p)
    return (*p)

int j = 17;
p = &j
goto l
```
Non-local control flow and block scope variables

Visualization of the operational semantics on an example

```c
int *p = NULL

l:
if (p)
return (*p)

int j = 17;
p = &j
goto l
```

```
int *p = NULL

l:
if (p)
return (*p)

int j = 17

p = &j
goto l
```
Non-local control flow and block scope variables

Visualization of the operational semantics on an example

```c
int *p = NULL

l:
if (p)
return (*p)

int j = 17;
p = &j; goto l
```

direction:

memory:

```
<table>
<thead>
<tr>
<th>p</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>17</td>
</tr>
</tbody>
</table>
```
Non-local control flow and block scope variables

Visualization of the operational semantics on an example

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

Visualization of the operational semantics on an example

```c
int *p = NULL

l:
if (p)
    return (*p)

int j = 17;
p = &j
goto l
```

```
int *p = NULL

l:
if (p)
    return (*p)

int j = 17;
p = &j
```
Non-local control flow and block scope variables

Visualization of the operational semantics on an example

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

Visualization of the operational semantics on an example

```c
int *p = NULL

l:
if (p)
    return (*p)

int j = 17;
p = &j
```

- direction: \[ \rightsquigarrow l \]
- memory: \[ p \]

...
Non-local control flow and block scope variables
Visualizaton of the operational semantics on an example

```c
int *p = NULL
l:
if (p)
  return (*p)
int j = 17
p = &j
```

- **direction:**
- **memory:**

```
;  
```
Non-local control flow and block scope variables
Visualization of the operational semantics on an example

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

🔗 ├{P}... goto main_sub3; ...{Q}
Non-local control flow and block scope variables
Axiomatic semantics [Krebbers/Wiedijk,FoSSaCS’13]

CH₂O Hoare sextuples are of the shape

$$\Delta; J; R \vdash \{P\} s \{Q\}$$
Non-local control flow and block scope variables
Axiomatic semantics [Krebbers/Wiedijk,FoSSaCS’13]

**CH$_2$O Hoare sextuples** are of the shape

\[ \Delta; J; R \vdash \{ P \} s \{ Q \} \]

where:

- \( \{ P \} s \{ Q \} \) is a Hoare triple, as usual
Non-local control flow and block scope variables
Axiomatic semantics [Kreinders/Wiedijk,FoSSaCS’13]

**CH$_2$O Hoare sextuples** are of the shape

$$\Delta; J; R \vdash \{P\} s \{Q\}$$

where:
- $\{P\} s \{Q\}$ is a Hoare triple, as usual
- $\Delta$ maps function names to their pre- and post-conditions
Non-local control flow and block scope variables
Axiomatic semantics [Krebbers/Wiedijk,FoSSaCS’13]

**CH$_2$O Hoare sextuples** are of the shape

$$\Delta; J; R \vdash \{P\} s \{Q\}$$

where:

- $\{P\} s \{Q\}$ is a Hoare triple, as usual
- $\Delta$ maps function names to their pre- and post-conditions
- $J$ 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
Axiomatic semantics [Krebbers/Wiedijk,FoSSaCS’13]

CH₂O Hoare sextuples are of the shape

\[ \Delta; J; R \vdash \{ P \} s \{ Q \} \]

where:

- \( \{ P \} s \{ Q \} \) is a Hoare triple, as usual
- \( \Delta \) maps function names to their pre- and post-conditions
- \( J \) maps labels to their jumping condition
  When executing a goto \( l \), the assertion \( J l \) has to hold
- \( R \) has to hold to execute a return

Remark: the assertions \( P, Q, J \) and \( R \) correspond to the directions \( \downarrow \), \( \uparrow \), \( \rightarrow \) and \( \uparrow \uparrow \) of traversal
Non-local control flow and block scope variables
Axiomatic semantics [Krebbers/Wiedijk, FoSSaCS ’13]

**CH₂O Hoare sextuples** are of the shape

\[ \Delta; J; R \vdash \{ P \} s \{ Q \} \]

where:

- \( \{ P \} s \{ Q \} \) is a Hoare triple, as usual
- \( \Delta \) maps function names to their pre- and post-conditions
- \( J \) maps labels to their jumping condition
  - When executing a goto \( l \), the assertion \( J l \) has to hold
- \( R \) has to hold to execute a return

Remark: the assertions \( P, Q, J \) and \( R \) correspond to the directions ↘, ↗, ↤ and ↑↑ of traversal
Non-local control flow and block scope variables

The block scope variable rule

\[
\Delta; J \uparrow^\ast x_0^\tau \mapsto -; R \uparrow^\ast x_0^\tau \mapsto - \vdash \{ P \uparrow^\ast x_0^\tau \mapsto - \} s \{ Q \uparrow^\ast x_0^\tau \mapsto - \}
\]

When entering a block:

- The De Bruijn indexes are lifted: \((\_ \uparrow \_ )\)
- The memory is extended: \((\_ ) \ast x_0^\tau \mapsto \_ )\)
Non-local control flow and block scope variables

The block scope variable rule

\[
\Delta; J \uparrow \ast x_0^\tau \rightarrow \neg; R \uparrow \ast x_0^\tau \rightarrow \neg \vdash \{ P \uparrow \ast x_0^\tau \rightarrow \neg \} s \{ Q \uparrow \ast x_0^\tau \rightarrow \neg \} \\
\Delta; J; R \vdash \{ P \} \text{local}_\tau s \{ Q \}
\]

When entering a block:

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

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

The block scope variable rule

\[
\Delta; J \uparrow \ast x_0^\tau \longmapsto -; R \uparrow \ast x_0^\tau \longmapsto - \vdash \{P \uparrow \ast x_0^\tau \longmapsto -\} s \{Q \uparrow \ast x_0^\tau \longmapsto -\}
\]

\[
\Delta; J; R \vdash \{P\} \text{local}_\tau s \{Q\}
\]

When entering a block:

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

When leaving a block: the reverse

Important: symmetry matches gotos going both in and out
Non-local control flow and block scope variables

The block scope variable rule

\[
\Delta; J \uparrow \ast x_0^\tau \leftrightarrow -; R \uparrow \ast x_0^\tau \leftrightarrow - \vdash \{ P \uparrow \ast x_0^\tau \leftrightarrow - \} s \{ Q \uparrow \ast x_0^\tau \leftrightarrow - \} \\
\Delta; J; R \vdash \{ P \} \text{local}_\tau s \{ Q \}
\]

When entering a block:

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

When leaving a block: the reverse

Important: symmetry matches gotos going both in and out

Important: 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 main() {
    *p = f();
    printf("x=%d,y=%d\n", x, y);
}
```
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();
    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 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
Separation logic [Krebbers, POPL'14]

**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 [Krebbers, POPL'14]

**Observation**: non-determinism corresponds to concurrency

**Idea**: use the separation logic rule for parallel composition

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

\[
\frac{\{P_1 \ast P_2\} e_1 \odot e_2 \{Q_1 \ast Q_2\}}{}
\]

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 \(e_1 \odot e_2\) can be executed safely in the whole memory
Non-determinism and sequence points
Separation logic [Krebbers, POPL'14]

**Observation:** non-determinism corresponds to concurrency

**Idea:** use the separation logic rule for parallel composition

\[
\{P_1\} e_1 \{Q_1\} \quad \{P_2\} e_2 \{Q_2\} \\
\{P_1 \ast P_2\} e_1 \odot e_2 \{Q_1 \ast Q_2\}
\]

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

Operational semantics [Krebbers, POPL’14]

**Head reduction for expressions** \((e, m) \rightarrow_h (e', m')\)
Non-determinism and sequence points
Operational semantics [Krebbers, POPL’14]

**Head reduction for expressions** $(e, m) \rightarrow_h (e', m')$

- On assignments: add locked address $a$ to $\Omega_1 \cup \Omega_2$
  $([a]_{\Omega_1} := [v]_{\Omega_2}, m) \rightarrow_h ([v]\{a\} \cup \Omega_1 \cup \Omega_2, \text{lock } a (m[a := v]))$

- Meanwhile: lock $a$ makes accesses to $a$ undefined

- On sequence points: unlock $\Omega$ in memory
  $([v]_{\Omega} \ ? \ e_2 : e_3, m) \rightarrow_h (e_2, \text{unlock } \Omega \ m)$ provided . . .
Non-determinism and sequence points
Operational semantics [Krebbers, POPL’14]

**Head reduction for expressions** \((e, m) \rightarrow_h (e', m')\)

- On assignments: add locked address \(a\) to \(\Omega_1 \cup \Omega_2\)
  
  \((\{a\}_{\Omega_1} := \{v\}_{\Omega_2}, m) \rightarrow_h (\{v\}_{\{a\} \cup \Omega_1 \cup \Omega_2}, \text{lock } a \ (m[a := v]))\)

- Meanwhile: lock \(a\) makes accesses to \(a\) undefined

- On sequence points: unlock \(\Omega\) in memory
  
  \((\{v\}_{\Omega} \ ? \ e_2 : e_3, m) \rightarrow_h (e_2, \text{unlock } \Omega \ m)\) provided 

Gives a local treatment of sequence points
Non-determinism and sequence points
Operational semantics [Krebbers, POPL’14]

Head reduction for expressions \((e, m) \rightarrow_h (e', m')\)
- On assignments: add locked address \(a\) to \(\Omega_1 \cup \Omega_2\)
  \( ([a]_{\Omega_1} := [v]_{\Omega_2}, m) \rightarrow_h ([v]_{\{a\} \cup \Omega_1 \cup \Omega_2}, \text{lock } a (m[a := v])) \)
- Meanwhile: \(\text{lock } a\) makes accesses to \(a\) undefined
- On sequence points: unlock \(\Omega\) in memory
  \( ([v]_{\Omega} ? e_2 : e_3, m) \rightarrow_h (e_2, \text{unlock } \Omega m) \) provided . . .

Gives a local treatment of sequence points

Small step reduction \(S(\mathcal{P}, \phi, m) \rightarrow S(\mathcal{P}', \phi', m')\)
- \((\mathcal{P}, \phi)\) gives the position/direction in the whole program
- Different \(\phi\)s for expressions, statements, function calls
- Uses evaluation contexts, \textit{i.e.} if \((e_1, m_1) \rightarrow_h (e_2, m_2)\), then
  \(S(\mathcal{P}, \mathcal{E}[e_1], m_1) \rightarrow S(\mathcal{P}, \mathcal{E}[e_2], m_2)\)
Strict aliasing restrictions
What is aliasing?

**Aliasing:** multiple pointers referring to the same object

```c
int f(int *p, int *q) {
    int x = *q; *p = 17; return x;
}
```

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

Optimizing `x` away is unsound: 17 would be returned.
Strict aliasing restrictions

What is aliasing?

**Aliasing:** multiple pointers referring to the same object

```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

![Diagram: Two pointers pointing to the same object](image)
Strict aliasing restrictions

What is aliasing?

**Aliasing:** multiple pointers referring to the same object

```c
int f(int *p, int *q) {
    int x = *q, *p = 17; return * *q;
}
```

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

Optimizing `x` away is unsound: `17` would be returned
Strict aliasing restrictions

What is aliasing?

**Aliasing:** multiple pointers referring to the same object

```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

Optimizing `x` away is unsound: 17 would be returned

**Alias analysis:** to determine whether pointers can alias
Strict aliasing restrictions
The C quiz, question 4

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 restrictions
The C quiz, question 4

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);
```

C89 allows `p` and `q` to be aliased, and thus requires `g` to return 3.
Strict aliasing restrictions

The C quiz, question 4

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);
```

**C89** allows `p` and `q` to be aliased, and thus requires `g` to return 3

**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 restrictions

How to treat pointers [Krebbers, CPP’13]

<table>
<thead>
<tr>
<th>Others (e.g. CompCert)</th>
<th>Our approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory: a finite map of cells which consist of <strong>arrays</strong> of bytes</td>
<td></td>
</tr>
<tr>
<td>Pointers: pairs ((o, i)) where (o) identifies the cell, and (i) the <strong>offset</strong> into that cell</td>
<td></td>
</tr>
<tr>
<td><strong>Too little information to capture strict aliasing restrictions</strong></td>
<td></td>
</tr>
</tbody>
</table>
Strict aliasing restrictions
How to treat pointers [Krebbers, CPP’13]

<table>
<thead>
<tr>
<th>Others (e.g. CompCert)</th>
<th>Our approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory: a finite map of cells which consist of arrays of bytes</td>
<td>A finite map of cells which consist of well-typed trees of bits</td>
</tr>
<tr>
<td>Pointers: pairs ((o, i)) where (o) identifies the cell, and (i) the offset into that cell</td>
<td></td>
</tr>
<tr>
<td>Too little information to capture strict aliasing restrictions</td>
<td></td>
</tr>
</tbody>
</table>
**Strict aliasing restrictions**

How to treat pointers [Krebbers, CPP’13]

<table>
<thead>
<tr>
<th>Others (e.g. CompCert)</th>
<th>Our approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory: a finite map of cells which consist of <strong>arrays</strong> of bytes</td>
<td>A finite map of cells which consist of well-typed <strong>trees</strong> of bits</td>
</tr>
<tr>
<td>Pointers: pairs ((o, i)) where (o) identifies the cell, and (i) the \textit{offset} into that cell</td>
<td>Pairs ((o, r)) where (o) identifies the cell, and (r) the \textit{path through the tree} in that cell</td>
</tr>
<tr>
<td></td>
<td>Too little information to capture strict aliasing restrictions</td>
</tr>
</tbody>
</table>
### Strict aliasing restrictions
How to treat pointers [Krebbers, CPP’13]

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

```
op
  .0

signed char: 10000100 01000100

p = (os : struct S, s, u, x[2], 0, 0, 1, 0, 8) signed char > void
```
Strict aliasing restrictions
The C quiz, question 5

```c
union U { int x; short y; } u = { .x = 3 }; 
short q1() {
  return u.y;
}
short q2() {
  short *p = &u.y;
  return *p;
}
```
Strict aliasing restrictions
The C quiz, question 5

```c
union U { int x; short y; } u = { .x = 3 }; short q1() {
    return u.y; // OK
}
short q2() {
    short *p = &u.y;
    return *p; // Undefined
}
```

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

- **C11** vaguely mentions a notion of “visible”
- **GCC** only if “the memory is accessed through the union type”
Strict aliasing restrictions
The C quiz, question 5

```c
union U { int x; short y; } u = { .x = 3 };  
short q1() {  
    return u.y; // OK  
}  
short q2() {  
    short *p = &u.y;  
    return *p; // Undefined  
}
```

**Type-punning:** reading a union using a pointer to another variant
  - C11 vaguely mentions a notion of “visible”
  - GCC only if “the memory is accessed through the union type”

Formalized by decorating pointers with annotations
Strict aliasing restrictions

Strict-aliasing Theorem

Theorem (Strict-aliasing)

Given:

- addresses $\Gamma; m \vdash a_1 : \sigma_1$ and $\Gamma; m \vdash a_2 : \sigma_2$
- 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*) has undefined behavior
Stric taliasing restrictions

Strict-aliasing Theorem

Theorem (Strict-aliasing)

Given:

- addresses $\Gamma; m \vdash a_1 : \sigma_1$ and $\Gamma; m \vdash a_2 : \sigma_2$
- 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) has undefined behavior

Theorem

Compilers can perform type-based alias analysis
\( \times \in \text{string} := \text{Set of strings} \)

\( k \in \text{cintrank} ::= \text{char} | \text{short} | \text{int} \)

\( | \text{long} | \text{long long} | \text{ptr} \)

\( si \in \text{signedness} ::= \text{signed} | \text{unsigned} \)

\( \tau_i \in \text{cinttype} ::= si? k \)

\( \tau \in \text{ctype} ::= \text{void} | \text{def} x | \tau_i | \tau \star \)

\( | \tau[e] | \text{struct} x | \text{union} x \)

\( | \text{enum} x | \text{typeof} e \)

\( \alpha \in \text{assign} ::= := | \odot := | := \odot \)

\( e \in \text{cexpr} ::= x | \text{const}_{\tau_i} z | \text{sizeof} \ \tau \)

\( | \tau_i \text{min} | \tau_i \text{max} | \tau_i \text{bits} \)

\( | \& e | \ast e \)

\( | e_1 \alpha e_2 \)

\( | x(\vec{e}) | \text{abort} \)

\( | \text{alloc}_{\tau} e | \text{free} e \)

\( | \odot_u e | e_1 \odot e_2 \)

\( | e_1 \&\& e_2 | e_1 \mid \mid e_2 \)

\( | e_1 ? e_2 : e_3 | (e_1, e_2) \)

\( | (\tau) l | e . x \)

\( r \in \text{crefseg} ::= [e] | .x \)

\( l \in \text{iinit} ::= e | \{ \overrightarrow{r} := \overrightarrow{l} \} \)

\( sto \in \text{cstorage} ::= \text{static} | \text{extern} | \text{auto} \)

\( s \in \text{cstmt} ::= e | \text{skip} \)

\( | \text{goto} x | \text{return} e? \)

\( | \} s \}

\( | \text{typedef} x ::= \tau ; s \)

\( | s_1 ; s_2 | x : s \)

\( | \text{while}(e) s \)

\( | \text{for}(e_1 ; e_2 ; e_3) s \)

\( | \text{do} s \text{while}(e) \)

\( | \text{if} (e) s_1 \text{else} s_2 \)

\( d \in \text{decl} ::= \text{struct} \overrightarrow{\tau \times} | \text{union} \overrightarrow{\tau \times} \)

\( | \text{typedef} \ \tau \)

\( | \text{enum} x ::= e_i? : \tau_i \)

\( | \text{global} l? : sto \ \tau \)

\( | \text{fun} (\overrightarrow{\tau x?}) s? : sto \ \tau \)

\( \Theta \in \text{decls} ::= \text{list} (\text{string} \times \text{decl}) \)
CH$_2$O Abstract C
Translation to CH$_2$O core C in Coq [Krebbers/Wiedijk, Submitted]

- Named variables to De Bruijn indices
CH$_2$O Abstract C

Translation to CH$_2$O core C in Coq [Krebbers/Wiedijk, Submitted]

- Named variables to De Bruijn indices
- Disambiguate l-values and r-values
CH₂O Abstract C
Translation to CH₂O core C in Coq [Krebbers/Wiedijk, Submitted]

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

\[ \llbracket e \rrbracket_{\Gamma, \text{getstack } P, m} = v \quad \text{iff} \quad S(P, e, m) \rightarrow^* S(P, v, m) \]
**CH₂O Abstract C**

Translation to CH₂O core C in Coq [Krebbers/Wiedijk, Submitted]

- Named variables to De Bruijn indices
- Disambiguate l-values and r-values
- Sound/complete constant expression evaluation, e.g. in $\tau[e]$
  
  $$\llbracket e \rrbracket_{\Gamma, \text{getstack } \mathcal{P}, m} = v \quad \text{iff} \quad S(\mathcal{P}, e, m) \xrightarrow{*} S(\mathcal{P}, v, m)$$

- Simplification of loops, e.g. `while(e) s` becomes
  
  catch (loop (if (e′) skip else throw 0; catch s′))
CH₂O Abstract C
Translation to CH₂O core C in Coq [Krebbers/Wiedijk, Submitted]

- Named variables to De Bruijn indices
- Disambiguate l-values and r-values
- Sound/complete constant expression evaluation, e.g. in $\tau[e]$
  $$\left[ e \right]_{\Gamma, \text{getstack } \mathcal{P}, m = v} \iff \mathcal{S} \left( \mathcal{P}, e, m \right) \rightarrow^{*} \mathcal{S} \left( \mathcal{P}, v, m \right)$$
- Simplification of loops, e.g. while($e$) $s$ becomes
  $$\text{catch (loop (if (e') skip else throw 0; catch s'))}$$
- Expansion of typedef and enum declarations
CH$_2$O Abstract C
Translation to CH$_2$O core C in Coq [Krebbers/Wiedijk, Submitted]

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

\[ \llbracket e \rrbracket_{\Gamma, \text{getstack} P, m} = v \quad \text{iff} \quad S(P, e, m) \Rightarrow^* S(P, v, m) \]

- 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
CH₂O Abstract C
Translation to CH₂O core C in Coq [Krebbers/Wiedijk, Submitted]

- Named variables to De Bruijn indices
- Disambiguate l-values and r-values
- Sound/complete constant expression evaluation, e.g. in $\tau[e]$
  
  $\llbracket e \rrbracket_{\Gamma, \text{getstack}\mathcal{P}, m = v} \iff S(\mathcal{P}, e, m) \rightarrow^* S(\mathcal{P}, v, m)$

- 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
CH$_2$O Abstract C

Translation to CH$_2$O core C in Coq [Krebbers/Wiedijk, Submitted]

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

\[ \llbracket e \rrbracket_{\Gamma, \text{getstack } P, m = v} \iff S(P, e, m) \xrightarrow{*} S(P, v, m) \]

- 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)**

If the translator succeeds, the CH$_2$O core program is well-typed.
Executable semantics

**Goal:** define $\text{exec} : \text{state} \rightarrow \mathcal{P}_{\text{fin}}(\text{state})$
Executable semantics

**Goal:** define exec : state → \( \mathcal{P}_{\text{fin}}(\text{state}) \)

**Problems:**

1. Decomposition \( \mathcal{E}[e_1] \) of expressions is non-deterministic:

\[
\mathcal{S}(\mathcal{P}, \mathcal{E}[e_1], m_1) \rightarrow \mathcal{S}(\mathcal{P}, \mathcal{E}[e_2], m_2) \quad \text{if} \quad (e_1, m_1) \rightarrow_n (e_2, m_2)
\]

2. Object identifiers \( o \) for newly allocated memory are arbitrary:

\[
\mathcal{S}(\mathcal{P}, (\downarrow, \text{local}_\tau s), m) \\
\rightarrow \mathcal{S}((\text{local}_{o:\tau} \square) :: \mathcal{P}, (\downarrow, s), \text{alloc} \ o \ \tau \ m) \quad \text{if} \quad o \notin \text{dom} \ m
\]
Executable semantics

**Goal:** define \( \text{exec} : \text{state} \rightarrow \mathcal{P}_{\text{fin}}(\text{state}) \)

**Problems:**

1. Decomposition \( \mathcal{E}[e_1] \) of expressions is non-deterministic:

\[
S(\mathcal{P}, \mathcal{E}[e_1], m_1) \rightarrow S(\mathcal{P}, \mathcal{E}[e_2], m_2) \quad \text{if} \quad (e_1, m_1) \rightarrow_h (e_2, m_2)
\]

2. Object identifiers \( o \) for newly allocated memory are arbitrary:

\[
S(\mathcal{P}, (\downarrow, \text{local}_\tau s), m) \\
\rightarrow S((\text{local}_{o:\tau} \square) :: \mathcal{P}, (\downarrow, s), \text{alloc} \ o \ \tau \ m) \quad \text{if} \quad o \notin \text{dom} \ m
\]

**Solutions:**

1. Enumerate all possible decompositions \( \mathcal{E}[e_1] \)
2. Pick a canonical object identifier fresh \( m \) for \( o \)
Executable semantics

Example of the Coq code

```coq
Definition cexec (Γ : env Ti) (δ : funenv Ti) (S : state Ti) : listset (state Ti) :=
  let 'State k φ m := S in
  match φ with
  | Stmt ↓ s =>
    match s with
    | skip => {[ State k (Stmt ↑ skip) m ]}
    | goto l => {[ State k (Stmt (↷ l) (goto l)) m ]}
    | ...
    | local{τ} s =>
      let o := fresh (dom indexset m) in (* use canonical object identifier *)
      {[ State (CLocal o τ :: k) (Stmt ↓ s) (mem_alloc Γ o false τ m) ]}
    end
  | Expr e =>
    match maybe_EVal e with
    | Some (Ω, v) => ...
    | None =>
      '(E,e') ← expr_redexes e; (* monadic programming to try each decomposition *)
      match ehexec Γ (get_stack k) e’ m with
      | Some (e2,m2) => {[ State k (Expr (subst E e2)) m2 ]}
      | None =>
        match maybe_ECall_redex e’ with
        | Some (f, Ωs, vs) =>
          {[ State (CFun E :: k) (Call f vs) (mem_unlock (∪ Ωs) m) ]}
        | _ => {[ State k (Undef (UndefExpr E e')) m ]}
      end
    end
  end
| ...
```
Executable semantics

Example of the Coq code

```coq
Definition cexec (Γ : env Ti) (δ : funenv Ti) (S : state Ti) : listset (state Ti) :=
  let 'State k φ m := S in
  match φ with
  | Stmt s =>
    match s with
    | skip => { State k (Stmt ↘ skip) m }
    | goto l => { State k (Stmt (↷ l) (goto l)) m }
    | ...
    | local{τ} s =>
      let o := fresh (dom indexset m) in (* use canonical object identifier *)
      { State (CLocal o τ :: k) (Stmt ↘ s) (mem_alloc Γ o false τ m) }
    end
  | Expr e =>
    match maybe_EVal e with
    | Some (Ω,v) => ...
    | None =>
      '(E,e′) ← expr_redexes e; (* monadic programming to try each decomposition *)
      match ehexec Γ (get_stack k) e' m with
      | Some (e2,m2) => { State k (Expr (subst E e2)) m2 }
      | None =>
        match maybe_ECall_redex e' with
        | Some (f, Ωs, vs) =>
          { State (CFun E :: k) (Call f vs) (mem_unlock (⋃ Ωs) m) }
        | _ => { State k (Undef (UndefExpr E e′)) m }
      end
    end
  end
  ...
```
Definition cexec (Γ : env Ti) (δ : funenv Ti) (S : state Ti) : listset (state Ti) :=
  let 'State k φ m := S in
  match φ with
  | Stmt s =>
    match s with
    | skip => { [ State k (Stmt ↘ skip) m ] }
    | goto l => { [ State k (Stmt ↘ l (goto l)) m ] }
    | ...
    | local{τ} s =>
      let o := fresh (dom indexset m) in (* use canonical object identifier *)
      { [ State (CLocal o τ :: k) (Stmt ↘ s) (mem_alloc Γ o false τ m) ] }
  end
  | Expr e =>
    match maybe_EVal e with
    | Some (Ω,v) => ...
    | None =>
      ' (E,e') ← expr_redexes e; (* monadic programming to try each decomposition *)
      match ehexec Γ (get_stack k) e' m with
      | Some (e2,m2) => { [ State k (Expr (subst E e2)) m2 ] }
      | None =>
        match maybe_ECall_redex e' with
        | Some (f, Ωs, vs) =>
          { [ State (CFun E :: k) (Call f vs) (mem_unlock (∪ Ωs) m) ] }
        | _ => { [ State k (Undef (UndefExpr E e')) m ] }
      end
    end
  | ...
end
Executable semantics
Soundness and completeness [Krebbers/Wiedijk, Submitted]

**Theorem (Soundness)**

If $S_2 \in \text{exec } S_1$, then $S_1 \rightarrow S_2$. 
Executable semantics

Soundness and completeness [Krebbers/Wiedijk, Submitted]

Theorem (Soundness)
If $S_2 \in \text{exec } S_1$, then $S_1 \rightarrow S_2$.

Definition (Permutation)
We let $S_1 \sim_f S_2$, if $S_2$ is obtained by renaming $S_1$ with respect to $f : \text{index} \rightarrow \text{option index}$. 
Executable semantics
Soundness and completeness [Krebbers/Wiedijk, Submitted]

Theorem (Soundness)
If $S_2 \in \text{exec } S_1$, then $S_1 \rightarrow S_2$.

Definition (Permutation)
We let $S_1 \sim_f S_2$, if $S_2$ is obtained by renaming $S_1$ with respect to $f : \text{index} \rightarrow \text{option index}$.

Theorem (Completeness)
If $S_1 \rightarrow^* S_2$, then there exists an $f$ and $S'_2$ such that:

$$
\begin{align*}
S_1 & \rightarrow \text{exec} S_2' \\
& \rightarrow^* \overset{f}{\rightarrow} S_2
\end{align*}
$$
$ cat foo.c
int main(void) {
    int x = 10, *p = &x;
    return *p + *p + *p + *p + *p + *p;
}
$ ./ch2o -t foo.c
......,........,,......+;!|?*%$$$$$$$$$$%*?|!;:+-,..
........
"" 60
Conclusion

C11 is not a simple language
Questions

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

http://xkcd.com/371/