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

```
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

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

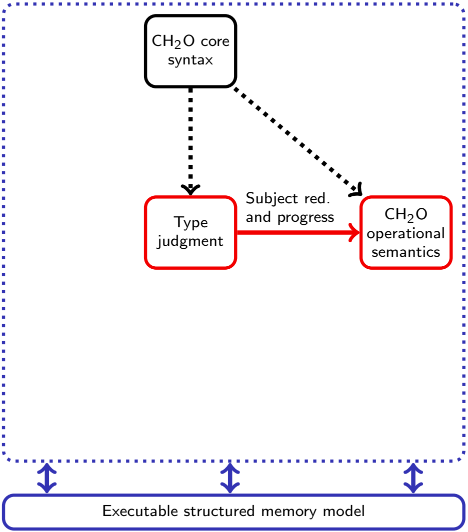
Formalin (Krebbers & Wiedijk)

- 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, 🇳🇱)

OCaml part

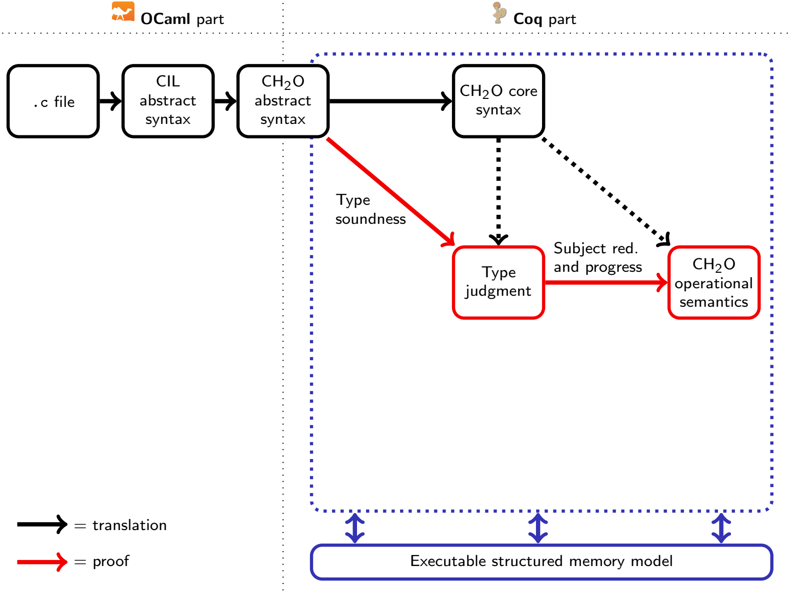
Coq part



→ = translation

→ = proof

The Formalin project (Krebbers & Wiedijk, 🇳🇱)



OCaml part

Coq part

.c file

CIL abstract syntax

CH_2O abstract syntax

CH_2O core syntax

Type soundness

Type judgment

Subject red. and progress

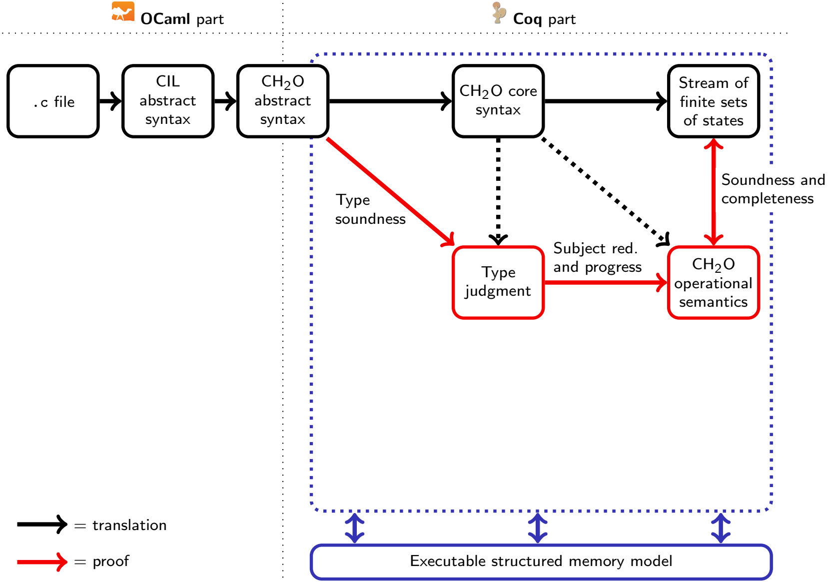
CH_2O operational semantics

Executable structured memory model

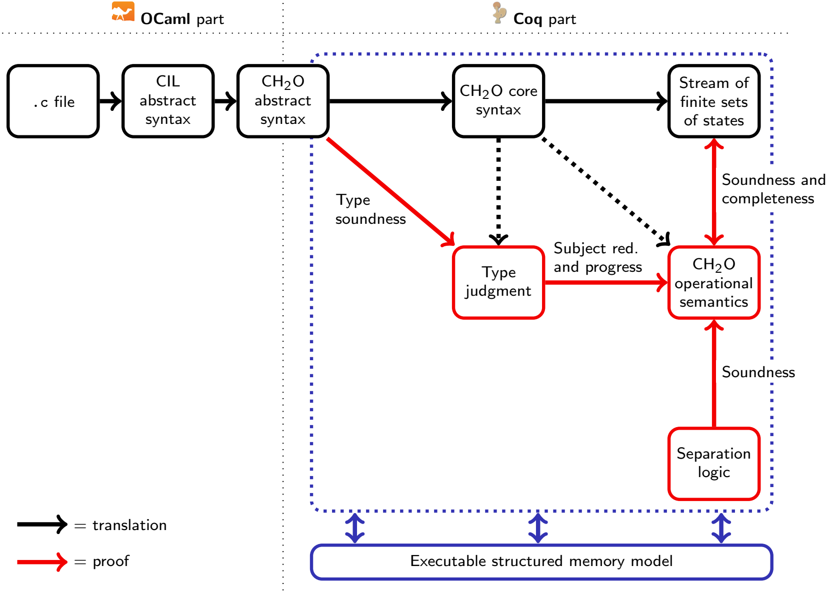
→ = translation

→ = proof

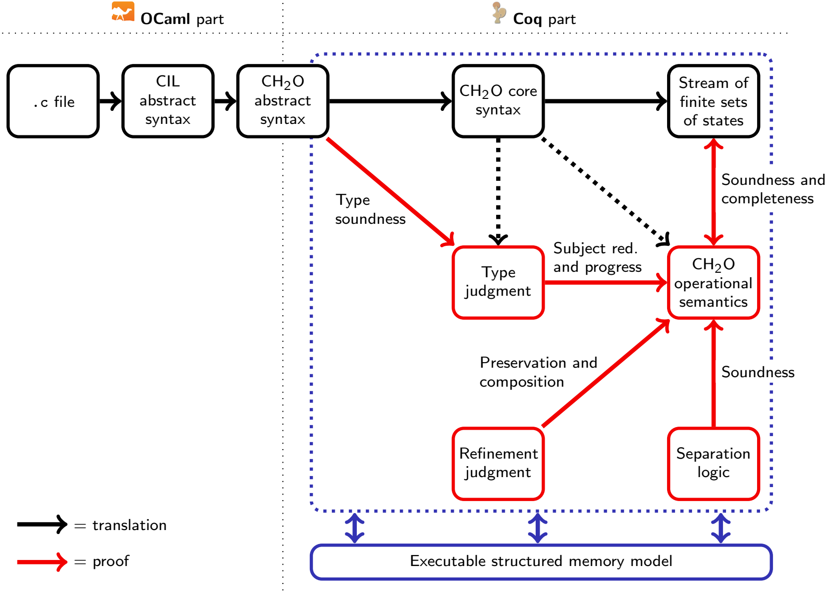
The Formalin project (Krebbers & Wiedijk, 🇳🇱)



The Formalin project (Krebbers & Wiedijk, 🇳🇱)



The Formalin project (Krebbers & Wiedijk, 🇳🇱)



OCaml part

Coq part

.c file

CIL abstract syntax

CH_2O abstract syntax

CH_2O core syntax

Stream of finite sets of states

Type soundness

Soundness and completeness

Type judgment

Subject red. and progress

CH_2O operational semantics

Preservation and composition

Soundness

Refinement judgment

Separation logic

Executable structured memory model

→ = translation

→ = proof

Non-local control flow and block scope variables

The C quiz, question 2

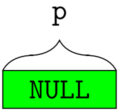
```
int *p = NULL;
l: if (p) {
  return (*p);
} else {
  int j = 17;
  p = &j;
  goto l;
}
```

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:

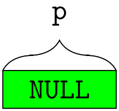


Non-local control flow and block scope variables

The C quiz, question 2

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

memory:

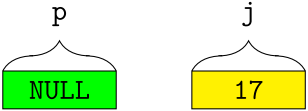


Non-local control flow and block scope variables

The C quiz, question 2

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

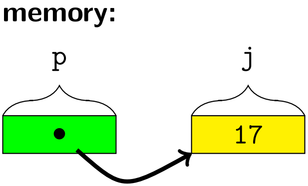
memory:



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;
}
```

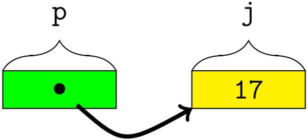


Non-local control flow and block scope variables

The C quiz, question 2

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

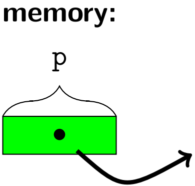
memory:



Non-local control flow and block scope variables

The C quiz, question 2

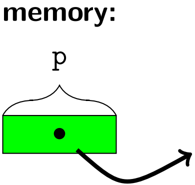
```
int *p = NULL;
l: if (p) {
  return (*p);
} else {
  int j = 17;
  p = &j;
  goto l;
}
```



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

Operational semantics [Krebbers/Wiedijk,FoSSaCS'13]

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:
    - $\searrow$ downwards to the next statement
    - $\nearrow$ upwards to the next statement
    - $\curvearrowright$ `l` to a label `l:` after a `goto l`
    - $\Uparrow v$ to the top of the statement after a `return`

Non-local control flow and block scope variables

Operational semantics [Krebbers/Wiedijk,FoSSaCS'13]

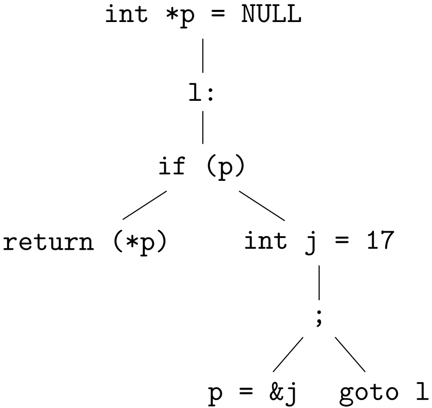
Problem: goto/return/*etc.* affect memory

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
  - $\curvearrowright$ `l` to a label `l:` after a `goto l`
  - $\Uparrow v$ 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



`int *p = NULL`

`l:`

`if (p)`

`return (*p)`

`int j = 17`

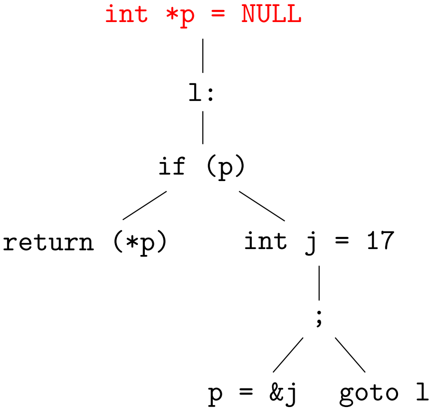
`;`

`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`

`;`

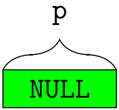
`p = &j`

`goto l`

direction:

↘

memory:

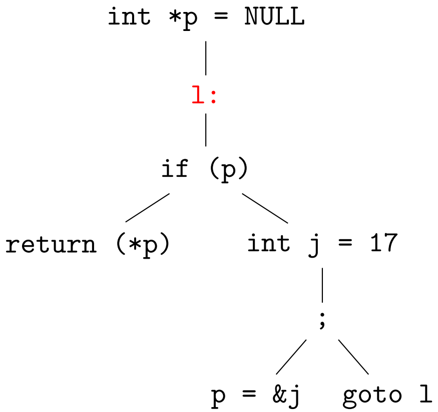


`p`

`NULL`

Non-local control flow and block scope variables

Visualization of the operational semantics on an example



`int *p = NULL`

`l:`

`if (p)`

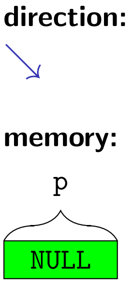
`return (*p)`

`int j = 17`

`;`

`p = &j`

`goto l`



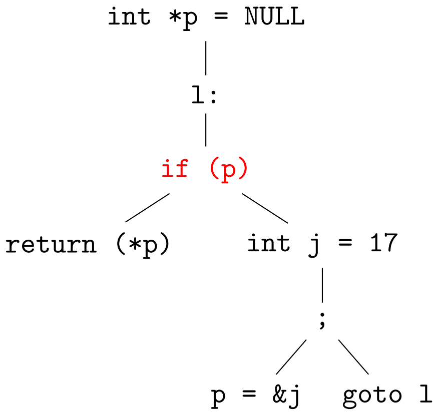
direction:

↘

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

;

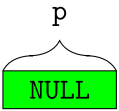
p = &j

goto l

direction:

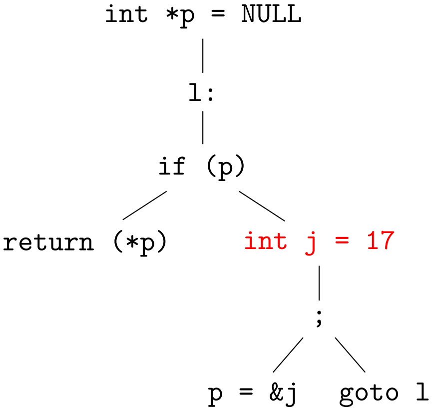
↘

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`

`;`

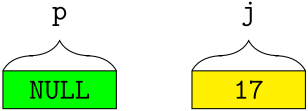
`p = &j`

`goto l`

direction:

↘

memory:



p

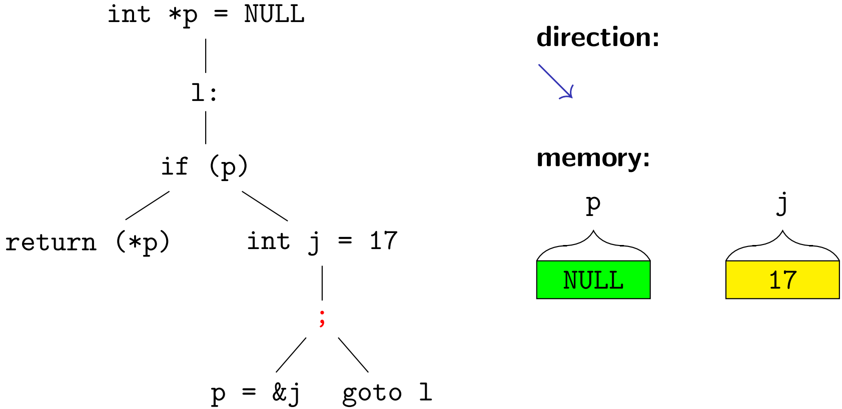
NULL

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
```

direction:

↘

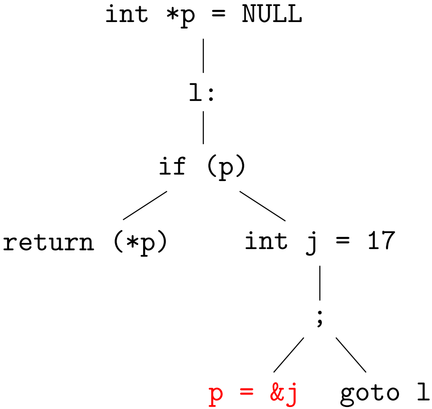
memory:

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

;

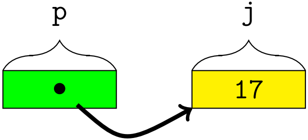
p = &j

goto l

direction:

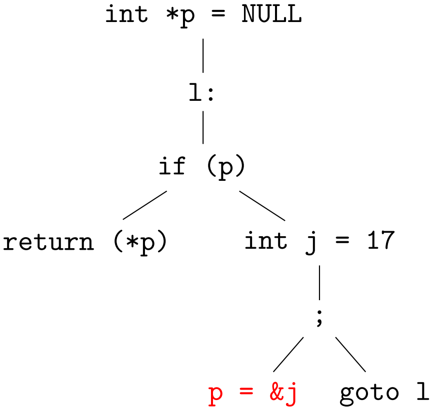
↘

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`

`;`

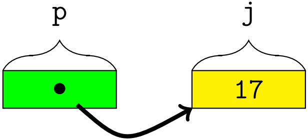
`p = &j`

`goto l`

direction:

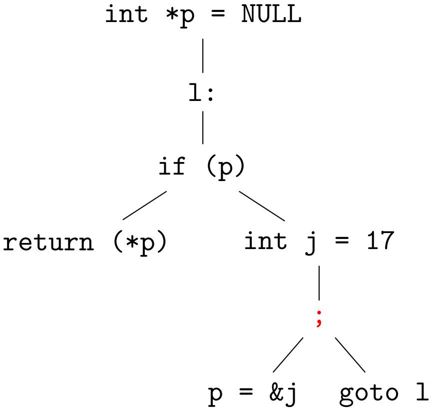
↗

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)`

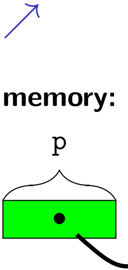
`int j = 17`

`;`

`p = &j`

`goto l`

direction:

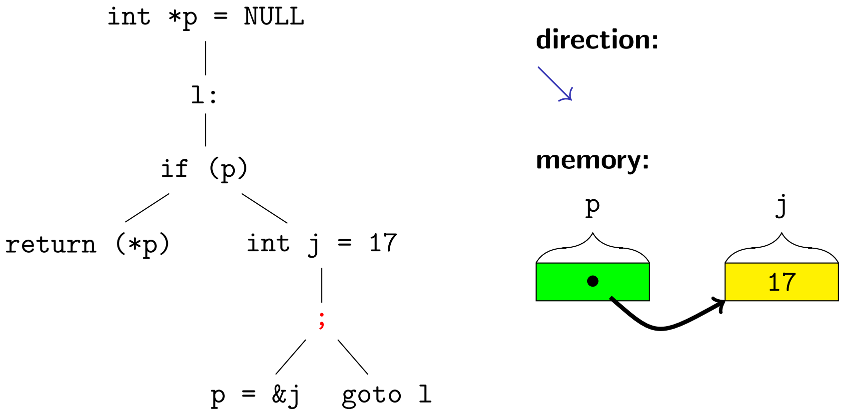


↗

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`

direction:

↘

memory:

p

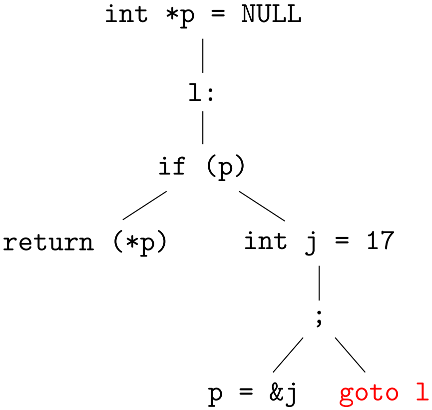
●

j

17

Non-local control flow and block scope variables

Visualization of the operational semantics on an example

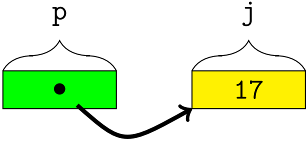


```
int *p = NULL
l:
if (p)
return (*p)
int j = 17
;
p = &j
goto l
```

direction:

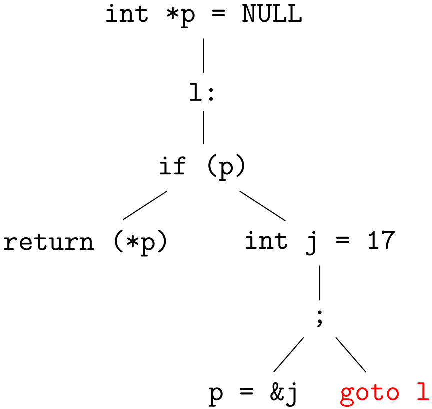
↘

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`

`;`

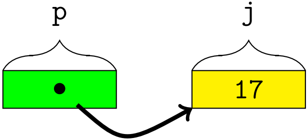
`p = &j`

`goto l`

direction:

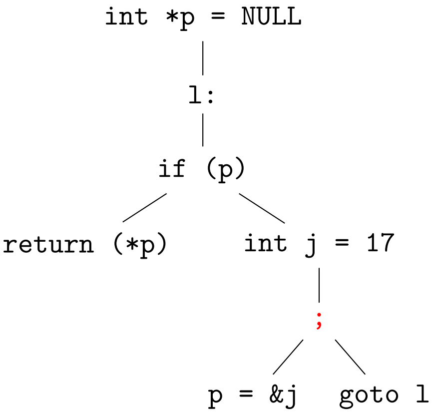
↷ `l`

memory:

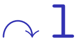


Non-local control flow and block scope variables

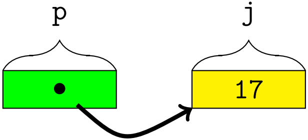
Visualization of the operational semantics on an example



direction:

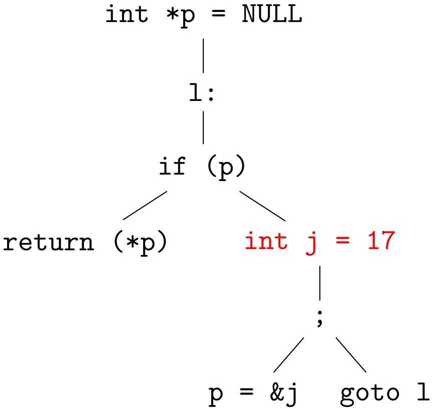


memory:



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`

`;`

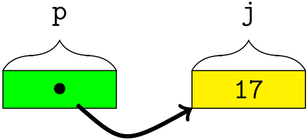
`p = &j`

`goto l`

direction:

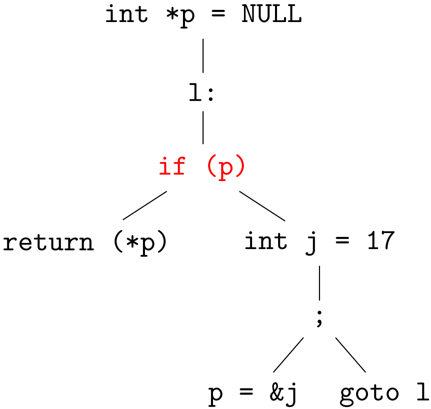
↷ `l`

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`

`;`

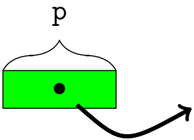
`p = &j`

`goto l`

direction:

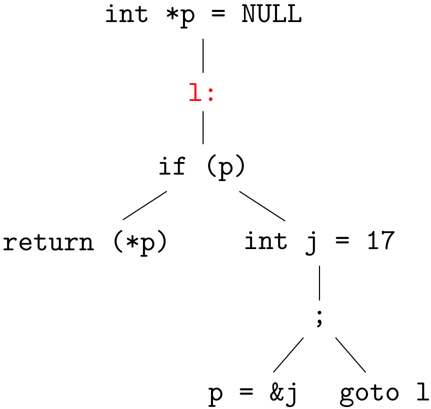
↷ `l`

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`

`;`

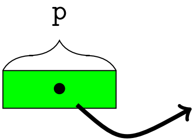
`p = &j`

`goto l`

direction:

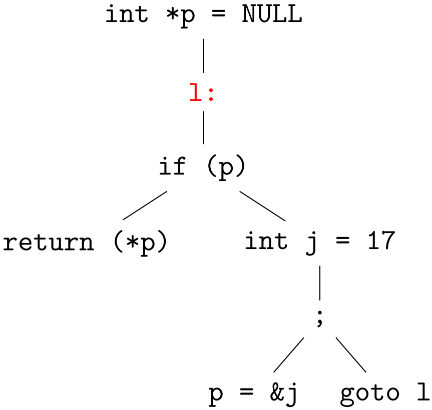
↷ `l`

memory:



Non-local control flow and block scope variables

Visualization of the operational semantics on an example

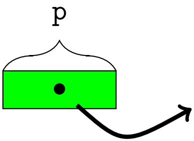


```
int *p = NULL
l:
if (p)
return (*p)
int j = 17
;
p = &j
goto l
```

direction:

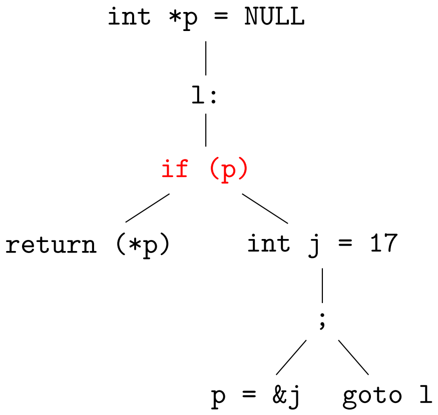
↘

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`

`;`

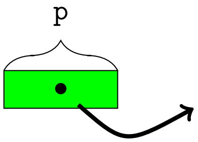
`p = &j`

`goto l`

direction:

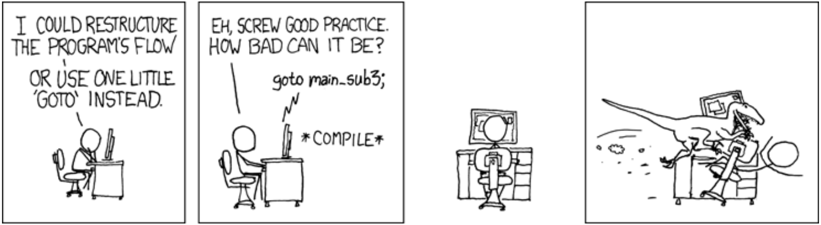
↘

memory:



Non-local control flow and block scope variables

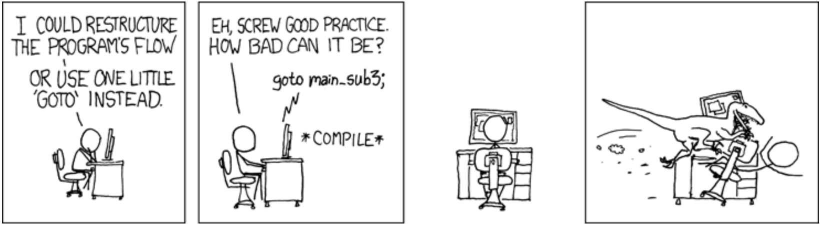
Goto considered harmful?



`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

Axiomatic semantics [Krebbers/Wiedijk,FoSSaCS'13]

CH$_2$O 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]

$\mathrm{CH_2O}$ 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 [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
- Δ maps function names to their pre- and post-conditions

Non-local control flow and block scope variables

Axiomatic semantics [Krebbers/Wiedijk,FoSSaCS'13]

CH_2O Hoare sextuples are of the shape

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

where:

- $\{P\}\, s\, \{Q\}$ is a Hoare triple, as usual
- Δ 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]

CH_2O Hoare sextuples are of the shape

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

where:

- $\{P\}\, s\, \{Q\}$ is a Hoare triple, as usual
- Δ 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

Non-local control flow and block scope variables

Axiomatic semantics [Krebbers/Wiedijk,FoSSaCS'13]

CH_2O Hoare sextuples are of the shape

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

where:

- $\{P\}\, s\, \{Q\}$ is a Hoare triple, as usual
- Δ 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 $\searrow$, $\nearrow$, $\curvearrowright$ and $\upuparrows$ of traversal

Non-local control flow and block scope variables

The block scope variable rule

$$\frac{\Delta;\, J\uparrow * x_0^\tau \mapsto -;\, R\uparrow * x_0^\tau \mapsto - \vdash \{P\uparrow * x_0^\tau \mapsto -\}\, s\, \{Q\uparrow * x_0^\tau \mapsto -\}}{\Delta;\, J;\, R \vdash \{P\}\, \mathsf{local}_\tau\, s\, \{Q\}}$$

When entering a block:

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

Non-local control flow and block scope variables

The block scope variable rule

$$\frac{\Delta;\ J\uparrow * x_0^\tau \mapsto -;\ R\uparrow * x_0^\tau \mapsto - \vdash \{P\uparrow * x_0^\tau \mapsto -\}\, s\, \{Q\uparrow * x_0^\tau \mapsto -\}}{\Delta;\ J;\ R \vdash \{P\}\, \mathsf{local}_\tau\, s\, \{Q\}}$$

When entering a block:

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

When leaving a block: the reverse

Non-local control flow and block scope variables

The block scope variable rule

$$\frac{\Delta;\ J\uparrow * x_0^\tau \mapsto -;\ R\uparrow * x_0^\tau \mapsto - \vdash \{P\uparrow * x_0^\tau \mapsto -\}\, s\, \{Q\uparrow * x_0^\tau \mapsto -\}}{\Delta;\ J;\ R \vdash \{P\}\, \mathsf{local}_\tau\, s\, \{Q\}}$$

When entering a block:

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

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

$$\frac{\Delta;\, J\uparrow * x_0^\tau \mapsto -;\, R\uparrow * x_0^\tau \mapsto - \vdash \{P\uparrow * x_0^\tau \mapsto -\}\, s\, \{Q\uparrow * x_0^\tau \mapsto -\}}{\Delta;\, J;\, R \vdash \{P\}\, \mathsf{local}_\tau\, s\, \{Q\}}$$

When entering a block:

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

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

```
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

```
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();
  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

```
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

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

Non-determinism and sequence points

Separation logic [Krebbers, POPL'14]

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 [Krebbers, POPL'14]

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

Operational semantics [Krebbers, POPL'14]

Head reduction for expressions $(e, m) \rightarrow_{\mathsf{h}} (e', m')$

Non-determinism and sequence points

Operational semantics [Krebbers, POPL'14]

Head reduction for expressions $(e, m) \twoheadrightarrow_{\mathsf{h}} (e', m')$

- On assignments: add locked address a to $\Omega_1 \cup \Omega_2$
  $([a]_{\Omega_1} := [v]_{\Omega_2}, m) \twoheadrightarrow_{\mathsf{h}} ([v]_{\{a\} \cup \Omega_1 \cup \Omega_2}, \mathsf{lock}\ a\ (m[a := v]))$
- Meanwhile: lock a makes accesses to a undefined
- On sequence points: unlock Ω in memory
  $([v]_{\Omega}\ ?\ e_2 : e_3, m) \twoheadrightarrow_{\mathsf{h}} (e_2, \mathsf{unlock}\ \Omega\ m)$ provided ...

Non-determinism and sequence points

Operational semantics [Krebbers, POPL'14]

Head reduction for expressions $(e, m) \rightarrow_{\mathsf{h}} (e', m')$

- On assignments: add locked address a to $\Omega_1 \cup \Omega_2$
  $([a]_{\Omega_1} := [v]_{\Omega_2}, m) \rightarrow_{\mathsf{h}} ([v]_{\{a\} \cup \Omega_1 \cup \Omega_2}, \mathsf{lock}\ a\ (m[a := v]))$
- Meanwhile: lock a makes accesses to a undefined
- On sequence points: unlock Ω in memory
  $([v]_{\Omega}\ ?\ e_2 : e_3, m) \rightarrow_{\mathsf{h}} (e_2, \mathsf{unlock}\ \Omega\ m)$ provided ...

Gives a local treatment of sequence points

Non-determinism and sequence points

Operational semantics [Krebbers, POPL'14]

Head reduction for expressions $(e, m) \rightarrow_{\mathsf{h}} (e', m')$

- On assignments: add locked address a to $\Omega_1 \cup \Omega_2$
  $([a]_{\Omega_1} := [v]_{\Omega_2}, m) \rightarrow_{\mathsf{h}} ([v]_{\{a\} \cup \Omega_1 \cup \Omega_2}, \mathsf{lock}\ a\ (m[a := v]))$
- Meanwhile: lock a makes accesses to a undefined
- On sequence points: unlock Ω in memory
  $([v]_{\Omega}\ ?\ e_2 : e_3, m) \rightarrow_{\mathsf{h}} (e_2, \mathsf{unlock}\ \Omega\ m)$ provided ...

Gives a local treatment of sequence points

Small step reduction $\mathbf{S}(\mathcal{P}, \phi, m) \rightarrow \mathbf{S}(\mathcal{P}', \phi', m')$

- $(\mathcal{P}, \phi)$ gives the position/direction in the *whole* program
- Different ϕs for expressions, statements, function calls
- Uses evaluation contexts, *i.e.* if $(e_1, m_1) \rightarrow_{\mathsf{h}} (e_2, m_2)$, then

$$\mathbf{S}(\mathcal{P}, \mathcal{E}[\, e_1 \,], m_1) \rightarrow \mathbf{S}(\mathcal{P}, \mathcal{E}[\, e_2 \,], m_2)$$

Strict aliasing restrictions

What is aliasing?

Aliasing: multiple pointers referring to the same object

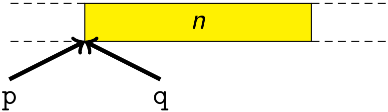
Strict aliasing restrictions

What is aliasing?

Aliasing: multiple pointers referring to the same object

```
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



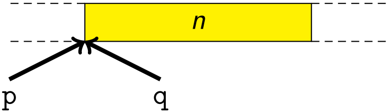
Strict aliasing restrictions

What is aliasing?

Aliasing: multiple pointers referring to the same object

```
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

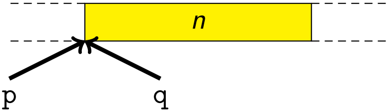
Strict aliasing restrictions

What is aliasing?

Aliasing: multiple pointers referring to the same object

```
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

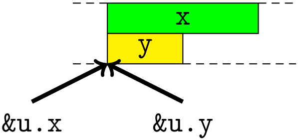
The C quiz, question 4

Consider a similar function:

```
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 restrictions

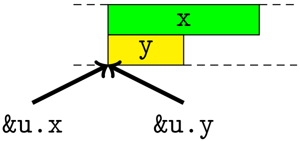
The C quiz, question 4

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);
```



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

Strict aliasing restrictions

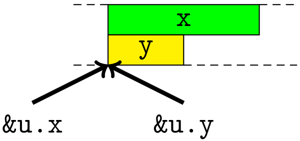
The C quiz, question 4

Consider a similar function:

```
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);
```



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]

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

Strict aliasing restrictions

How to treat pointers [Krebbers, CPP'13]

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

Strict aliasing restrictions

How to treat pointers [Krebbers, CPP'13]

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

Strict aliasing restrictions

How to treat pointers [Krebbers, CPP'13]

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

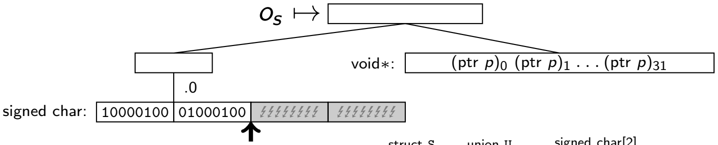
Strict aliasing restrictions

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:



$$p = (o_s : \text{struct } \mathtt{S}, \xhookrightarrow{\text{struct } \mathtt{S}} 0 \xhookrightarrow{\text{union } \mathtt{U}} \bullet\ 1 \xhookrightarrow{\text{signed char[2]}} 0, 8)_{\text{signed char}>\text{void}}$$

Strict aliasing restrictions

The C quiz, question 5

```
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

```
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

```
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
- 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*) has undefined behavior

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
- 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*) has undefined behavior

Theorem

Compilers can perform type-based alias analysis

CH$_2$O Abstract C

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

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

CH_2O Abstract C

Translation to CH_2O core C in Coq [Krebbers/Wiedijk, Submitted]

- Named variables to De Bruijn indices

CH_2O Abstract C

Translation to CH_2O core C in Coq [Krebbers/Wiedijk, Submitted]

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

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 $\tau[e]$

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

CH_2O Abstract C

Translation to CH_2O core C in Coq [Krebbers/Wiedijk, Submitted]

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

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

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

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

CH_2O Abstract C

Translation to CH_2O 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 $\tau[e]$

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

- 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

CH_2O Abstract C

Translation to CH_2O 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 $\tau[e]$

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

- 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`

$\mathrm{CH_2O}$ Abstract C

Translation to $\mathrm{CH_2O}$ 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 $\tau[e]$

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

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

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

- 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 $\tau[e]$

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

- 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)

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

Executable semantics

Goal: define $\mathsf{exec} : \mathsf{state} \to \mathcal{P}_{\mathsf{fin}}(\mathsf{state})$

Executable semantics

Goal: define $\mathsf{exec} : \mathsf{state} \to \mathcal{P}_{\mathsf{fin}}(\mathsf{state})$

Problems:

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

$$\mathbf{S}(\mathcal{P},\, \mathcal{E}[\,e_1\,],\, m_1) \multimap \mathbf{S}(\mathcal{P},\, \mathcal{E}[\,e_2\,],\, m_2) \quad \text{if} \quad (e_1, m_1) \multimap_{\mathsf{h}} (e_2, m_2)$$

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

$$\begin{aligned} &\mathbf{S}(\mathcal{P},\, (\searrow,\, \mathsf{local}_\tau\, s),\, m) \\ \multimap\ &\mathbf{S}((\mathsf{local}_{o:\tau}\, \square) :: \mathcal{P},\, (\searrow,\, s),\, \mathsf{alloc}\ o\ \tau\ m) \quad \text{if} \quad o \notin \mathsf{dom}\ m \end{aligned}$$

Executable semantics

Goal: define $\mathsf{exec} : \mathsf{state} \to \mathcal{P}_{\mathsf{fin}}(\mathsf{state})$

Problems:

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

$$\mathbf{S}(\mathcal{P},\, \mathcal{E}[\, e_1 \,],\, m_1) \rightarrowtriangle \mathbf{S}(\mathcal{P},\, \mathcal{E}[\, e_2 \,],\, m_2) \quad \text{if} \quad (e_1, m_1) \mapstochar\rightarrowtriangle_{\mathsf{h}} (e_2, m_2)$$

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

$$\begin{aligned} &\mathbf{S}(\mathcal{P},\, (\searrow,\, \mathsf{local}_{\tau}\, s),\, m) \\ \rightarrowtriangle\ &\mathbf{S}((\mathsf{local}_{o:\tau}\, \Box) :: \mathcal{P},\, (\searrow,\, s),\, \mathsf{alloc}\ o\ \tau\ m) \quad \text{if} \quad o \notin \mathsf{dom}\ m \end{aligned}$$

Solutions:

1. Enumerate all possible decompositions $\mathcal{E}[\, e_1 \,]$
2. Pick a canonical object identifier $\mathsf{fresh}\ m$ for o

Executable semantics

Example of the Coq code

```
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

```
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

```
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 $S_1 \rightharpoonup S_2$.

Executable semantics

Soundness and completeness [Krebbers/Wiedijk, Submitted]

Theorem (Soundness)

If $S_2 \in \mathrm{exec}\ S_1$, then $S_1 \rightarrowtriangle S_2$.

Definition (Permutation)

We let $S_1 \sim_f S_2$, if S_2 is obtained by renaming S_1 with respect to $f : \mathrm{index} \to \mathrm{option\ index}$.

Executable semantics

Soundness and completeness [Krebbers/Wiedijk, Submitted]

Theorem (Soundness)

If $S_2 \in \mathrm{exec}\ S_1$, then $S_1 \rightarrowtriangle S_2$.

Definition (Permutation)

We let $S_1 \sim_f S_2$, if S_2 is obtained by renaming S_1 with respect to $f : \mathrm{index} \to \mathrm{option\ index}$.

Theorem (Completeness)

If $S_1 \rightarrowtriangle^* S_2$, then there exists an f and S_2' such that:

$$\begin{tikzcd}
 & S_2' \arrow[d, dash, dashed, "f"] \\
S_1 \arrow[ur, dashed, "\text{exec}", "*"{near end}] \arrow[r, "*"{near end}] & S_2
\end{tikzcd}$$

Executable semantics

Demo

```
$ 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/`