Separation algebras for C verification in Coq

Robbert Krebbers

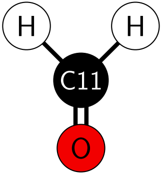
ICIS, Radboud University Nijmegen, The Netherlands

July 18, 2014 @ VSTTE, Vienna, Austria

Context of this talk

Formalin (Krebbers & Wiedijk)

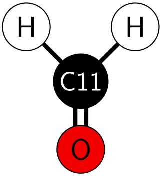
- Compiler independent C semantics in Coq
- Take underspecification by C11 seriously
- Operational semantics
- Executable semantics
- Typing and type checker
- Separation logic



Context of this talk

Formalin (Krebbers & Wiedijk)

- Compiler independent C semantics in Coq
- Take underspecification by C11 seriously
- Operational semantics
- Executable semantics
- Typing and type checker
- Separation logic $\Rightarrow$ topic of this talk



Why compiler (in)dependence matters

```c
int main() {
  int x;
  int y = (x = 3) + (x = 4);
  printf("%d %d\n", x, y);
}
```

Why compiler (in)dependence matters

```
int main() {
  int x;
  int y = (x = 3) + (x = 4);
  printf("%d %d\n", x, y);
}
```

Let us try some compilers

- Clang prints 4 7, seems just left-right

Why compiler (in)dependence matters

```
int main() {
  int x;
  int y = (x = 3) + (x = 4);
  printf("%d %d\n", x, y);
}
```

Let us try some compilers

- Clang prints `4 7`, seems just left-right
- GCC prints `4 8`, does not correspond to any evaluation order

Why compiler (in)dependence matters

```c
int main() {
  int x;
  int y = (x = 3) + (x = 4);
  printf("%d %d\n", x, y);
}
```

Let us try some compilers

- Clang prints 4 7, seems just left-right
- GCC prints 4 8, does not correspond to any evaluation order

This program violates the sequence point restriction

- due to two unsequenced writes to x
- undefined behavior: garbage in, garbage out $\Rightarrow$ all bets are off
- thus both compilers are right

Why compiler (in)dependence matters

```
int main() {
  int x;
  int y = (x = 3) + (x = 4);
  printf("%d %d\n", x, y);
}
```

Let us try some compilers

- Clang prints `4 7`, seems just left-right
- GCC prints `4 8`, does not correspond to any evaluation order

This program violates the sequence point restriction

- due to two unsequenced writes to `x`
- undefined behavior: garbage in, garbage out $\Rightarrow$ all bets are off
- thus both compilers are right

Formalin should account for all undefined behavior

Separation logic for C [Krebbers, POPL'14]

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

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

Separation logic for C [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

Separation logic for C [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 $\Rightarrow$ no sequence point violation

Connectives of separation logic

The connectives of separation logic are defined as:

$$\begin{aligned} \mathsf{emp} &:= \lambda m \,.\, m = \emptyset \\ P * Q &:= \lambda m \,.\, \exists m_1\, m_2 \,.\, m = m_1 \cup m_2 \wedge P\, m_1 \wedge Q\, m_2 \end{aligned}$$

Connectives of separation logic

The connectives of separation logic are defined as:

$$\mathrm{emp} := \lambda m \,.\, m = \emptyset$$
$$P * Q := \lambda m \,.\, \exists m_1\, m_2 \,.\, m = m_1 \cup m_2 \wedge P\, m_1 \wedge Q\, m_2$$

Definition of ● is non-trivial:

- Complex memory based on structured trees
- Fractional permissions for share-accounting
  For example needed in `x + x`
- Existence permissions for pointer arithmetic
  For example needed in `*(p + 1) = (*p = 1)`
- Locked permissions for sequence point restriction

Connectives of separation logic

The connectives of separation logic are defined as:

$$\mathsf{emp} := \lambda m \,.\, m = \emptyset$$
$$P * Q := \lambda m \,.\, \exists m_1\, m_2 \,.\, m = m_1 \cup m_2 \wedge P\, m_1 \wedge Q\, m_2$$

Definition of ● is non-trivial:

- Complex memory based on structured trees
- Fractional permissions for share-accounting
  For example needed in `x + x`
- Existence permissions for pointer arithmetic
  For example needed in `*(p + 1) = (*p = 1)`
- Locked permissions for sequence point restriction

Use separation algebras [Calcagno *et al.*, LICS'07] to abstractly describe the permissions and memory

Tweaked version of separation algebras in Coq

Def: A simple separation algebra consists of a set A, with:

- An element $\emptyset : A$
- A predicate valid $: A \to \mathsf{Prop}$
- Binary relations $\perp, \subseteq\ : A \to A \to \mathsf{Prop}$
- Binary operations $\cup, \setminus\ : A \to A \to A$

Tweaked version of separation algebras in Coq

Def: A simple separation algebra consists of a set A, with:

- An element $\emptyset : A$
- A predicate valid : $A \to \text{Prop}$
- Binary relations $\perp$, $\subseteq$: $A \to A \to \text{Prop}$
- Binary operations $\cup$, $\setminus$: $A \to A \to A$

Total instead of partial

Tweaked version of separation algebras in Coq

Def: A simple separation algebra consists of a set A, with:

- An element $\emptyset : A$
- A predicate valid $: A \to \mathrm{Prop}$
- Binary relations $\perp, \subseteq \; : A \to A \to \mathrm{Prop}$
- Binary operations $\cup, \setminus \; : A \to A \to A$

Total instead of partial

Disjointness

Tweaked version of separation algebras in Coq

Def: A simple separation algebra consists of a set A, with:

- An element $\emptyset : A$
- A predicate valid $: A \to \mathrm{Prop}$
- Binary relations $\perp, \subseteq : A \to A \to \mathrm{Prop}$
- Binary operations $\cup, \setminus : A \to A \to A$

To avoid subset types

Total instead of partial

Disjointness

Tweaked version of separation algebras in Coq

Def: A simple separation algebra consists of a set A, with:

- An element $\emptyset : A$
- A predicate valid : $A \to \mathsf{Prop}$
- Binary relations $\perp$, $\subseteq$: $A \to A \to \mathsf{Prop}$
- Binary operations $\cup$, $\setminus$: $A \to A \to A$

To avoid subset types | Total instead of partial | Disjointness

Satisfying the following laws:

1. If $x \perp y$, then $y \perp x$ and $x \cup y = y \cup x$
2. If valid x, then $\emptyset \perp x$ and $\emptyset \cup x = x$
3. Associative, non-empty, cancellative, positive, ...

Example: fractional separation algebra

Fractional permissions $[0,1]_{\mathbb{Q}}$ [Boyland, SAS'09]

No access 0 —— Read-only —— 1 Exclusive access

Rational numbers make it possible to split Read-only permissions

Example: fractional separation algebra

Fractional permissions $[0,1]_{\mathbb{Q}}$ [Boyland, SAS'09]

Rational numbers make it possible to split Read-only permissions

Def: The simple fractional separation algebra $\mathbb{Q}$ is defined as:

$$
\begin{aligned}
\text{valid } x &:= 0 \le x \le 1 & \emptyset &:= 0 \\
x \perp y &:= 0 \le x, y \wedge x + y \le 1 & x \cup y &:= x + y \\
x \subseteq y &:= 0 \le x \le y \le 1 & x \setminus y &:= x - y
\end{aligned}
$$

Organization of permissions

Separation logic: $\cup$ main connective (from separation algebra)
Operational semantics: need to know what is allowed

Organization of permissions

Separation logic: $\cup$ main connective (from separation algebra)
Operational semantics: need to know what is allowed

Def: Lattice of permission kinds $(\mathsf{pkind}, \subseteq_{\mathsf{k}})$

$$\begin{tikzcd}[column sep=small]
 & \text{Freeable} \arrow[d, dash] & \\
 & \text{Writable} \arrow[dl, dash] \arrow[dr, dash] & \\
\text{Readable} \arrow[dr, dash] & & \text{Locked} \arrow[dl, dash] \\
 & \text{Existing} \arrow[d, dash] & \\
 & \bot &
\end{tikzcd}$$

- Freeable: reading, writing, deallocation

Organization of permissions

Separation logic: $\cup$ main connective (from separation algebra)
Operational semantics: need to know what is allowed

Def: Lattice of permission kinds $(\mathsf{pkind}, \subseteq_k)$

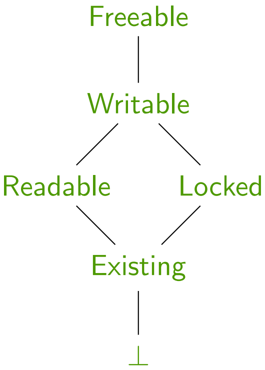
$$\begin{tikzcd}
& \text{Freeable} \arrow[d, dash] & \\
& \text{Writable} \arrow[dl, dash] \arrow[dr, dash] & \\
\text{Readable} \arrow[dr, dash] & & \text{Locked} \arrow[dl, dash] \\
& \text{Existing} \arrow[d, dash] & \\
& \bot &
\end{tikzcd}$$

- Freeable: reading, writing, deallocation
- Writable: reading, writing

Organization of permissions

Separation logic: $\cup$ main connective (from separation algebra)
Operational semantics: need to know what is allowed

Def: Lattice of permission kinds $(\mathrm{pkind}, \subseteq_k)$

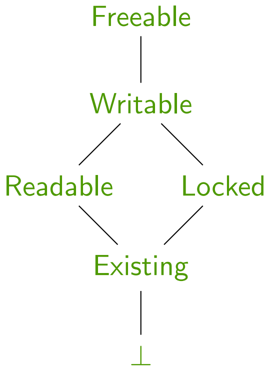


- Freeable: reading, writing, deallocation
- Writable: reading, writing
- Readable: reading

Organization of permissions

Separation logic: $\cup$ main connective (from separation algebra)
Operational semantics: need to know what is allowed

Def: Lattice of permission kinds $(\mathsf{pkind}, \subseteq_{\mathsf{k}})$

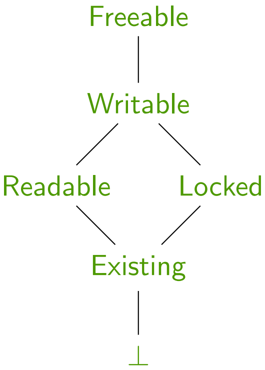


- Freeable: reading, writing, deallocation
- Writable: reading, writing
- Readable: reading
- Existing: *existence permissions*, only pointer arithmetic

Organization of permissions

Separation logic: $\cup$ main connective (from separation algebra)
Operational semantics: need to know what is allowed

Def: Lattice of permission kinds $(\mathsf{pkind}, \subseteq_\mathsf{k})$



- Freeable: reading, writing, deallocation
- Writable: reading, writing
- Readable: reading
- Existing: *existence permissions*, only pointer arithmetic
- Locked: temporarily locked until next sequence point
  Example: `(x = 3) + (*p = 4);`
  Undefined behavior if `&x == p`

Organization of permissions

Separation logic: $\cup$ main connective (from separation algebra)
Operational semantics: need to know what is allowed

Def: Lattice of permission kinds $(\mathsf{pkind}, \subseteq_{\mathsf{k}})$

$$\begin{tikzcd}[column sep=small]
& \text{Freeable} \arrow[d, dash] & \\
& \text{Writable} \arrow[dl, dash] \arrow[dr, dash] & \\
\text{Readable} \arrow[dr, dash] & & \text{Locked} \arrow[dl, dash] \\
& \text{Existing} \arrow[d, dash] & \\
& \bot &
\end{tikzcd}$$

- Freeable: reading, writing, deallocation
- Writable: reading, writing
- Readable: reading
- Existing: *existence permissions*, only pointer arithmetic
- Locked: temporarily locked until next sequence point
  Example: `(x = 3) + (*p = 4);`
  Undefined behavior if `&x == p`
- $\bot$: no operations allowed
  Example: `free(p); return (p-p);`

Interaction with permission kinds

Def: A C permissions system is a separation algebra A with functions $\mathrm{kind} : A \to \mathrm{pkind}$, $\mathrm{lock}, \mathrm{unlock} \quad : A \to A$ satisfying:

$$\mathrm{unlock}\ (\mathrm{lock}\ x) = x \qquad \text{provided that } \mathrm{Writable} \subseteq_{\mathrm{k}} \mathrm{kind}\ x$$

$$\mathrm{kind}\ (\mathrm{lock}\ x) = \mathrm{Locked} \qquad \text{provided that } \mathrm{Writable} \subseteq_{\mathrm{k}} \mathrm{kind}\ x$$

Interaction with permission kinds

Def: A C permissions system is a separation algebra A with functions $\mathsf{kind} : A \to \mathsf{pkind}$, $\mathsf{lock}, \mathsf{unlock}, \frac{1}{2} : A \to A$ satisfying:

$$
\begin{aligned}
\mathsf{unlock}\,(\mathsf{lock}\,x) &= x && \text{provided that } \mathsf{Writable} \subseteq_{\mathsf{k}} \mathsf{kind}\,x \\
\mathsf{kind}\,(\mathsf{lock}\,x) &= \mathsf{Locked} && \text{provided that } \mathsf{Writable} \subseteq_{\mathsf{k}} \mathsf{kind}\,x \\
\mathsf{kind}\,(\tfrac{1}{2}x) &= \begin{cases} \mathsf{Readable} & \text{if } \mathsf{Writable} \subseteq_{\mathsf{k}} \mathsf{kind}\,x \\ \mathsf{kind}\,x & \text{otherwise} \end{cases}
\end{aligned}
$$

Example: use $\frac{1}{2}$ in `x + x`

Interaction with permission kinds

Def: A C permissions system is a separation algebra A with functions $\mathsf{kind} : A \to \mathsf{pkind}$, $\mathsf{lock}, \mathsf{unlock}, \frac{1}{2} : A \to A$ and $\mathsf{token} : A$ satisfying:

$$
\begin{aligned}
\mathsf{unlock}\,(\mathsf{lock}\,x) &= x && \text{provided that } \mathsf{Writable} \subseteq_{\mathsf{k}} \mathsf{kind}\,x \\
\mathsf{kind}\,(\mathsf{lock}\,x) &= \mathsf{Locked} && \text{provided that } \mathsf{Writable} \subseteq_{\mathsf{k}} \mathsf{kind}\,x \\
\mathsf{kind}\,(\tfrac{1}{2}x) &= \begin{cases} \mathsf{Readable} & \text{if } \mathsf{Writable} \subseteq_{\mathsf{k}} \mathsf{kind}\,x \\ \mathsf{kind}\,x & \text{otherwise} \end{cases} \\
\mathsf{kind}\,\mathsf{token} &= \mathsf{Existing} \\
\mathsf{kind}\,(x \setminus \mathsf{token}) &= \begin{cases} \mathsf{Writable} & \text{if } \mathsf{kind}\,x = \mathsf{Freeable} \\ \mathsf{kind}\,x & \text{if } \mathsf{Existing} \subseteq_{\mathsf{k}} \mathsf{kind}\,x \end{cases}
\end{aligned}
$$

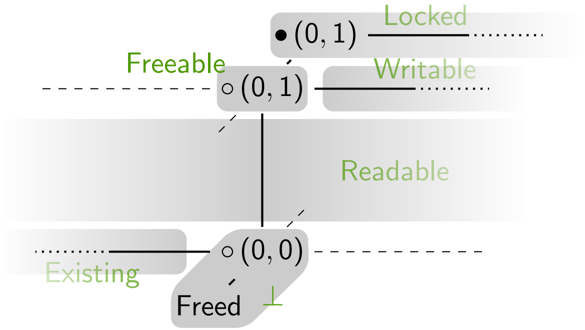
Example: use $\frac{1}{2}$ in `x + x`
Example: use $\_ \setminus \mathsf{token}$ in `*(p + 1) = (*p = 1)`

Implementation of permissions

Def: C permissions are defined as

$$\mathsf{perm} := \mathcal{F}(\mathcal{L}(\mathcal{C}(\mathbb{Q}))) = \{\mathsf{Freed}\} + \{\circ, \bullet\} \times \mathbb{Q} \times \mathbb{Q}$$

with:



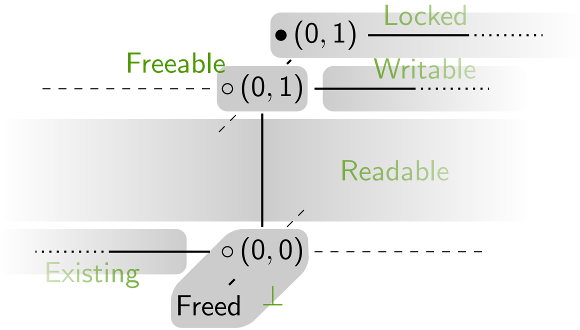
Implementation of permissions

Def: C permissions are defined as

$$\mathsf{perm} := \mathcal{F}(\mathcal{L}(\mathcal{C}(\mathbb{Q}))) = \{\mathsf{Freed}\} + \{\circ, \bullet\} \times \mathbb{Q} \times \mathbb{Q}$$

Fractional SA

with:



Implementation of permissions

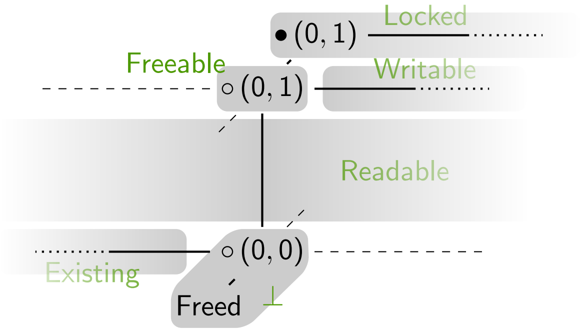
Def: C permissions are defined as

$$\mathsf{perm} := \mathcal{F}(\mathcal{L}(\mathcal{C}(\mathbb{Q}))) = \{\mathsf{Freed}\} + \{\circ, \bullet\} \times \mathbb{Q} \times \mathbb{Q}$$

Countable SA

Fractional SA

with:



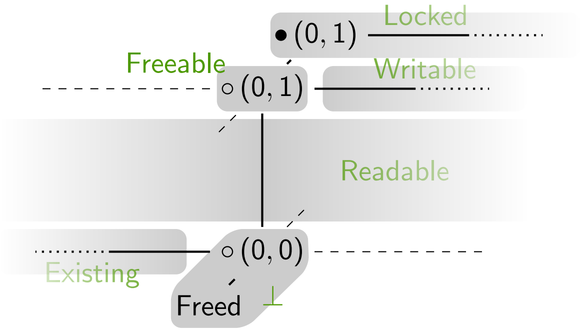
Implementation of permissions

Def: C permissions are defined as

$$\mathrm{perm} := \mathcal{F}(\mathcal{L}(\mathcal{C}(\mathbb{Q}))) = \{\mathrm{Freed}\} + \{\circ, \bullet\} \times \mathbb{Q} \times \mathbb{Q}$$

Lockable SA | Countable SA | Fractional SA

with:



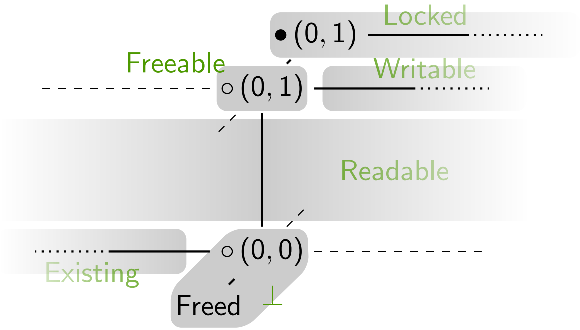
Implementation of permissions

Def: C permissions are defined as

$$\text{perm} := \mathcal{F}(\mathcal{L}(\mathcal{C}(\mathbb{Q}))) = \{\text{Freed}\} + \{\circ, \bullet\} \times \mathbb{Q} \times \mathbb{Q}$$

Freeable SA | Lockable SA | Countable SA | Fractional SA

with:



The C memory

Extremely complex:

- Pointer arithmetic
- Difficult interaction between low and high level
  - Types
  - Object representations
- Byte-wise operations on all objects
- Non-aliasing restrictions
- Permissions

Aliasing

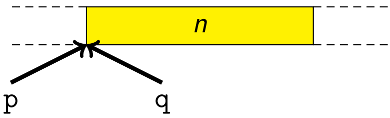
Aliasing: multiple pointers referring to the same object

Aliasing

Aliasing: multiple pointers referring to the same object

```
int f(int *p, int *q) {
  int x = *p; *q = 314; return x;
}
```

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

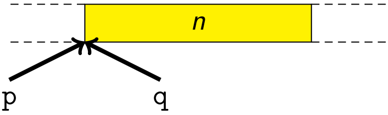


Aliasing

Aliasing: multiple pointers referring to the same object

```
int f(int *p, int *q) {
  int x = *p; *q = 314; return x *p;
}
```

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



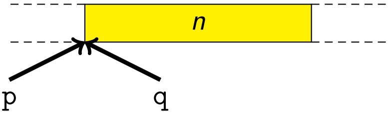
Optimizing `x` away is unsound: 314 would be returned

Aliasing

Aliasing: multiple pointers referring to the same object

```
int f(int *p, int *q) {
  int x = *p; *q = 314; return x *p;
}
```

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



Optimizing `x` away is unsound: 314 would be returned

Alias analysis: to determine whether pointers can alias

Aliasing with different types

Consider a similar function:

```
int h(int *p, float *q) {
  int x = *p; *q = 3.14; return x;
}
```

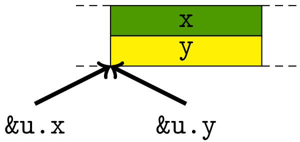
Aliasing with different types

Consider a similar function:

```
int h(int *p, float *q) {
  int x = *p; *q = 3.14; return x;
}
```

It can still be called with aliased pointers:

```
union { int x; float y; } u;
u.x = 271;
return h(&u.x, &u.y);
```



C89 allows p and q to be aliased, and thus requires it to return 271

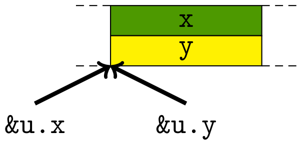
Aliasing with different types

Consider a similar function:

```
int h(int *p, float *q) {
  int x = *p; *q = 3.14; return x;
}
```

It can still be called with aliased pointers:

```
union { int x; float y; } u;
u.x = 271;
return h(&u.x, &u.y);
```



C89 allows p and q to be aliased, and thus requires it to return 271

C99/C11 allows **type-based alias analysis**:

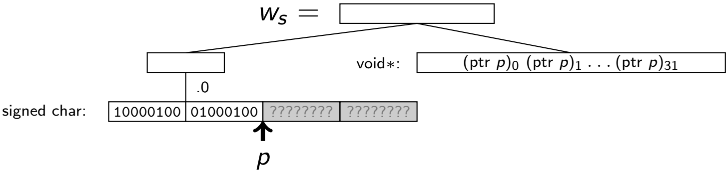
- A compiler can assume that p and q do not alias
- Reads/writes with "the wrong type" yield undefined behavior

The C memory as structured forest [Krebbers, CPP'13]

Consider:

```
struct T {
  union U {
    signed char x[2]; int y;
  } u;
  void *p;
} s = { { .x = {33,34} }, s.u.x + 2 }
```

As a picture:

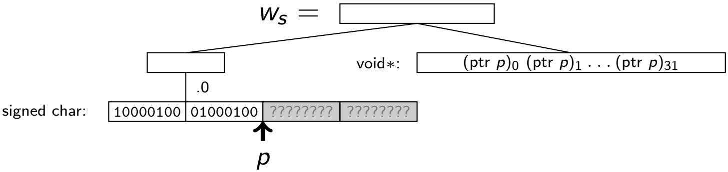


The C memory as structured forest [Krebbers, CPP'13]

Consider:

```
struct T {
  union U {
    signed char x[2]; int y;
  } u;
  void *p;
} s = { { .x = {33,34} }, s.u.x + 2 }
```

As a picture:



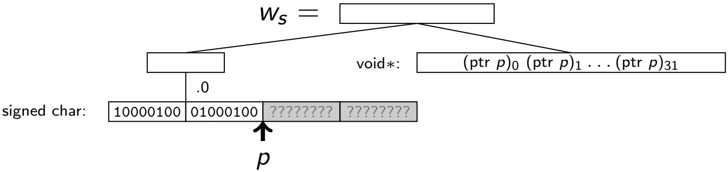
Captures aliasing restrictions of C11

The C memory as structured forest [Krebbers, CPP'13]

Consider:

```
struct T {
  union U {
    signed char x[2]; int y;
  } u;
  void *p;
} s = { { .x = {33,34} }, s.u.x + 2 }
```

As a picture:



Captures aliasing restrictions of C11

Generalization of [Krebbers, CPP'13] is a separation algebra

The C memory compositionally

Def: The C memory is defined as:

$$\mathsf{mem} := \mathsf{cmap}\,(\mathcal{T}_{?:\mathsf{bit}}(\mathcal{F}(\mathcal{L}(\mathcal{C}(\mathbb{Q})))))$$

The C memory compositionally

Def: The C memory is defined as:

$$\mathsf{mem} := \mathsf{cmap}\left(\mathcal{T}_{?:\mathsf{bit}}(\mathcal{F}(\mathcal{L}(\mathcal{C}(\mathbb{Q}))))\right)$$

Permissions

The C memory compositionally

Def: The C memory is defined as:

$$\mathsf{mem} := \mathsf{cmap}\,(\mathcal{T}_{\mathsf{?:bit}}(\mathcal{F}(\mathcal{L}(\mathcal{C}(\mathbb{Q})))))$$

(Bit) tagged SA

Permissions

The C memory compositionally

Def: The C memory is defined as:

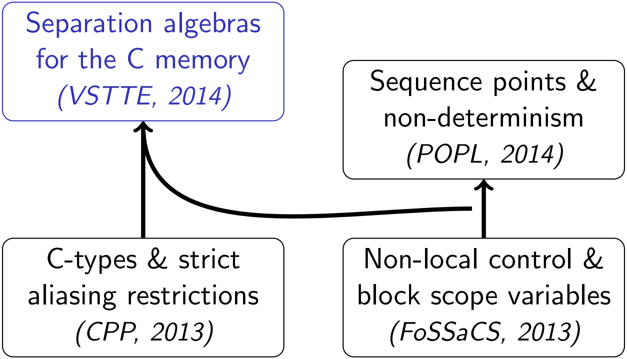
$$\mathsf{mem} := \mathsf{cmap}\,(\mathcal{T}_{?:\mathsf{bit}}(\mathcal{F}(\mathcal{L}(\mathcal{C}(\mathbb{Q})))))$$

Structured memory SA
Generalization of [Krebbers, CPP'13]

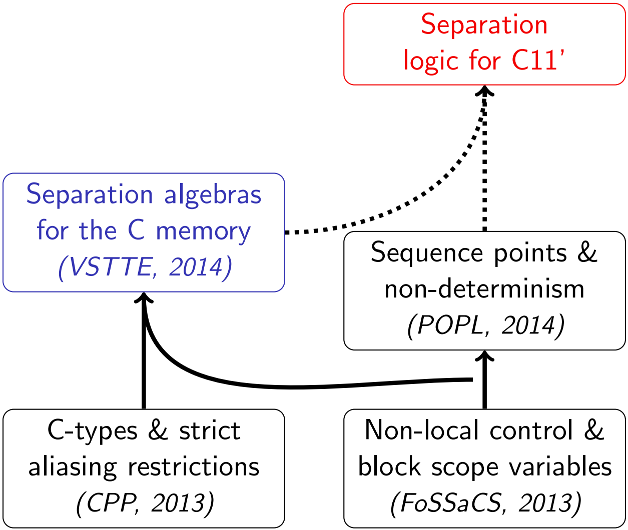
(Bit) tagged SA

Permissions

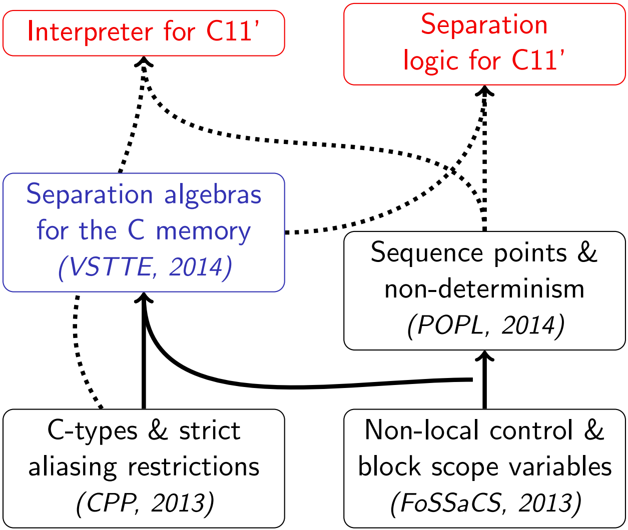
The bigger picture / Future work



The bigger picture / Future work



The bigger picture / Future work



Questions

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

(`http://xkcd.com/371/`)