A Separation Logic for Non-determinism and Sequence Points in C Formalized in Coq

Robbert Krebbers

Radboud University Nijmegen

May 14, 2014 @ TYPES, Paris, France

What is this program supposed to do?

```
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`
- resulting in undefined behavior
- thus both compilers are right

Undefined behavior in C

"Garbage in, garbage out" principle

- Programs with undefined behavior are not statically excluded
- Undefined behavior $\Rightarrow$ all bets are off
- Allows compilers to omit (expensive) dynamic checks

A compiler independent C semantics should account for undefined behavior

Examples

- Sequenced side-effects are allowed

```
(x = f()) ? (x = x + 1) : 0;
```

- Side-effects are useful in a `while`, `for`, `return`, ...

```
while ((x = getchar()) != EOF)
  /* do something */
```

- Non-determinism is subtle in innocent looking examples:

```
*p = g (x, y, z);
```

  Here, `g` may change `p`, so the evaluation order matters

- Interleaving of subexpressions is possible, for example

```
printf("a") + (printf("b") + printf("c"));
```

  may print “bac”

Contribution

A compiler independent small step operational, and axiomatic, semantics for non-determinism and sequence points, supporting:

- expressions with function calls, assignments, conditionals
- undefined behavior due to integer overflow
- parametrized by integer types
- dynamically allocated memory (`malloc` and `free`)
- non-local control (`return` and `goto`)
- local variables (and pointers to those)
- mutual recursion
- separation logic
- soundness proof fully checked by Coq

Key idea

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

Definition of the memory

Given a set of permissions P, we define:

$$
\begin{aligned}
m \in \mathsf{mem} &:= \mathsf{index} \to_{\mathsf{fin}} (\mathsf{val} \times P) \\
b \in \mathsf{index} &:= \mathbb{N} \\
v \in \mathsf{val} &::= \mathsf{indet} \mid \mathsf{int}_\tau\, n \mid \mathsf{ptr}\, b \mid \mathsf{NULL}
\end{aligned}
$$

Integer types: unsigned char, signed int, ...

C permissions

The C permissions contain:

- Locked flag to catch sequence point violations
  - Assignment: lock memory location
  - Sequence point: unlock memory locations
- A fraction $(0,1]_{\mathbb{Q}}$ for share accounting of read only memory
- Block scope variable or allocated with `malloc` flag

A permission system P abstracts from these details.

- $\cup \; : P \to P \to P$ and $\perp \; : P \to P \to \mathsf{Prop}$
- $\mathsf{kind} : P \to \{\mathsf{Free}, \mathsf{Write}, \mathsf{Read}, \mathsf{Locked}\}$
- $\mathsf{lock}, \mathsf{unlock} : P \to P$
- satisfying certain axioms

We lift these operations to memories $\mathsf{index} \to_{\mathsf{fin}} (\mathsf{val} \times P)$

The language

Our language

$$
\begin{aligned}
\circledcirc \in \text{binop} &::= \texttt{==} \mid \texttt{<=} \mid \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \ldots \\
e \in \text{expr} &::= x_i \mid [v]_\Omega \mid e_1 := e_2 \mid f(\vec{e}) \mid \texttt{load}\ e \mid \texttt{alloc} \\
&\quad\ \mid \texttt{free}\ e \mid e_1 \circledcirc e_2 \mid e_1\ ?\ e_2 : e_3 \mid (\tau)\, e \\
s \in \text{stmt} &::= e \mid \texttt{skip} \mid \texttt{goto}\ l \mid \texttt{return}\ e \mid \texttt{block}\ c\ s \mid s_1\ ; s_2 \\
&\quad\ \mid l : s \mid \texttt{while}(e)\ s \mid \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2
\end{aligned}
$$

Values $[v]_\Omega$ carry a set Ω of indexes (memory locations) to be unlocked at the next sequence point

Operational semantics

Head reduction for expressions $(e, m) \rightarrow_{\mathsf{h}} (e', m')$ (9 rules)

- On assignments: locked index b added to $\Omega_1 \cup \Omega_2$
  $([\mathrm{ptr}\ b]_{\Omega_1} := [v]_{\Omega_2}, m) \rightarrow_{\mathsf{h}} ([v]_{\{b\} \cup \Omega_1 \cup \Omega_2}, \mathsf{lock}\ b\ (m[b := v]))$
- On sequence points: Ω unlocked 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 $\mathbf{S}(k,\ \phi,\ m) \rightarrow \mathbf{S}(k',\ \phi',\ m')$ (33 rules)

- (k, ϕ) gives the position in the *whole* program
- Uses evaluation contexts to lift head reduction
- Different ϕs for expressions, statements, function calls

Assertions of separation logic

Defined using a shallow embedding:

$$P, Q \in \text{assert} := \underset{\substack{\uparrow \\ \boxed{\text{Maps variables to indexes in memory}}}}{\text{stack}} \to \text{mem} \to \text{Prop}$$

Some assertions:

$$\begin{aligned}
(P * Q)\,\rho\,m &:= \exists m_1\, m_2\,.\\
&\qquad m = m_1 \cup m_2 \wedge m_1 \perp m_2 \wedge P\,\rho\,m_1 \wedge Q\,\rho\,m_2\\
(e_1 \overset{\gamma}{\mapsto} e_2)\,\rho\,m &:= \exists b\, v\,.\, [\![\, e_1 \,]\!]_{\rho,m} = \mathrm{ptr}\; b\, \wedge\\
&\qquad [\![\, e_2 \,]\!]_{\rho,m} = v \wedge m = \{(b, (v, \gamma))\}\\
(P \triangleright)\,\rho\,m &:= P\,\rho\,(\mathrm{unlock}\,(\mathrm{locks}\; m) m)
\end{aligned}$$

(with e_1 and e_2 side-effect free)

The Hoare "triples"

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

Function conditions | Goto/return conditions | Q : assert

Expression judgment $\Delta \vdash \{P\}\, e\, \{Q\}$

Q : val $\to$ assert

In the semantics: if P holds beforehand, then

- e does not crash
- $Q\, v$ holds afterwards when terminating with v
- with framing memories that can change at each step

The axiomatic semantics

24 separation logic proof rules, *e.g.*:

For assignments:

$$\frac{\mathsf{Write} \subseteq \mathsf{kind}\ \gamma \quad \{P_1\}\, e_1\, \{Q_1\} \quad \{P_2\}\, e_2\, \{Q_2\} \quad \forall a v\, .\, ((Q_1\, a * Q_2\, v) \to ((a \xmapsto{\gamma} -) * R\, a\, v))}{\{P_1 * P_2\}\, e_1 := e_2\, \{\lambda v\, .\, \exists a\, .\, (a \xmapsto{\mathsf{lock}\ \gamma} v) * R\, a\, v\}}$$

For dereferencing:

$$\frac{\mathsf{kind}\ \gamma \neq \mathsf{Locked} \quad \{P\}\, e\, \{\lambda a\, .\, \exists v\, .\, Q\, a\, v * (a \xmapsto{\gamma} v)\}}{\{P\}\, \mathtt{load}\ e\, \{\lambda v\, .\, \exists a\, .\, Q\, a\, v * (a \xmapsto{\gamma} v)\}}$$

For the conditional:

$$\frac{\{P\}\, e_1\, \{\lambda v\, .\, v \neq \mathsf{indet} \wedge P'\, v \triangleright\} \quad \{\exists v\, .\, \mathsf{istrue}\ v \wedge P'\, v\}\, e_2\, \{Q\} \quad \ldots}{\{P\}\, e_1\ ?\ e_2 : e_3\, \{Q\}}$$

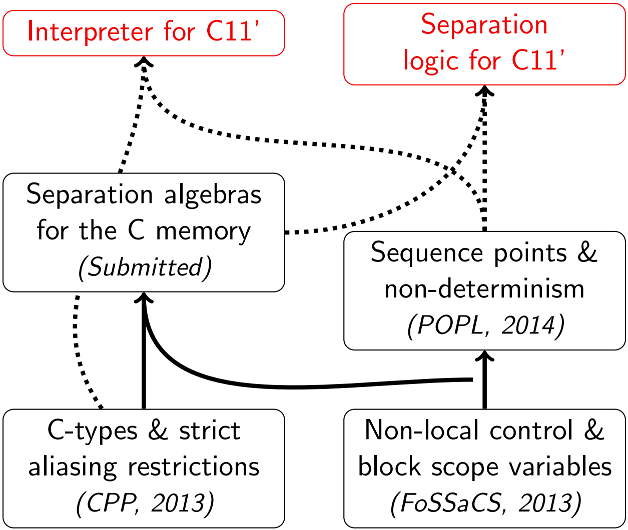
Common separation logic and more complex rules can be derived

Formalization in Coq

- Extremely useful for debugging
- Notations close to those on paper
- Various extensions of the separation logic
- Uses lots of automation
- 10 000 lines of code

```
Lemma ax_load Δ A γ e P Q :
  perm_kind γ ≠ Locked →
  Δ\ A ⊨ₑ {{ P }} e {{ λ a, ∃ v, Q a v ∗ valc a ↦{γ} valc v }} →
  Δ\ A ⊨ₑ {{ P }} load e {{ λ v, ∃ a, Q a v ∗ valc a ↦{γ} valc v }}.
```

The bigger picture / Future work



Questions

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

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