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?

```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`
- 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”
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

\[
\begin{align*}
\{P_1\} e_1 \{Q_1\} & \quad \{P_2\} e_2 \{Q_2\} \\
\{P_1 \ast P_2\} e_1 \odot e_2 \{Q_1 \ast Q_2\}
\end{align*}
\]

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
Definition of the memory

Given a set of permissions $P$, we define:

$$m \in \text{mem} := \text{index} \rightarrow_{\text{fin}} (\text{val} \times P)$$

$$b \in \text{index} := \mathbb{N}$$

$$v \in \text{val} ::= \text{indet} \mid \text{int}_\tau n \mid \text{ptr} b \mid \text{NULL}$$

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.

- \(\cup : P \rightarrow P \rightarrow P\) and \(\bot : P \rightarrow P \rightarrow \text{Prop}\)
- \(\text{kind} : P \rightarrow \{\text{Free, Write, Read, Locked}\}\)
- \(\text{lock, unlock} : P \rightarrow P\)
- satisfying certain axioms

We lift these operations to memories index \(\rightarrow_{\text{fin}} (\text{val} \times P)\)
The language

Our language

\[ \circ \in \text{binop ::= } \mathbin{=} | \leq | + | - | * | / | \% | \ldots \]
\[ e \in \text{expr ::= } x_i | \left[ v \right]_{\Omega} | e_1 := e_2 | f(\overline{e}) | \text{load } e | \text{alloc} \]
\[ | \text{free } e | e_1 \odot e_2 | e_1 ? e_2 : e_3 | (\tau) e \]
\[ s \in \text{stmt ::= } e | \text{skip} | \text{goto } l | \text{return } e | \text{block } c s | s_1 ; s_2 \]
\[ | l : s | \text{while}(e) s | \text{if } (e) s_1 \text{ else } s_2 \]

Values \([v]_{\Omega}\) carry a set \(\Omega\) of indexes (memory locations) to be unlocked at the next sequence point.
Operational semantics

**Head reduction for expressions** \((e, m) \rightarrow_h (e', m')\) (9 rules)

- On assignments: locked index \(b\) added to \(\Omega_1 \cup \Omega_2\)
  \[[\text{ptr } b]_{\Omega_1} := [v]_{\Omega_2}, m\) \(\rightarrow_h ([v]\{b\} \cup \Omega_1 \cup \Omega_2, \text{lock } b (m[b:=v]))\)

- On sequence points: \(\Omega\) unlocked 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** \(S(k, \phi, m) \rightarrow S(k', \phi', m')\) (33 rules)

- \((k, \phi)\) gives the position in the whole program
- Uses evaluation contexts to lift head reduction
- Different \(\phi\)s for expressions, statements, function calls
Assertions of separation logic

Defined using a shallow embedding:

\[ P, Q \in \text{assert} := \text{stack} \rightarrow \text{mem} \rightarrow \text{Prop} \]

Maps variables to indexes in memory

Some assertions:

\[ (P \ast Q) \rho m := \exists m_1 m_2 . \]
\[ m = m_1 \cup m_2 \land m_1 \perp m_2 \land P \rho m_1 \land Q \rho m_2 \]

\[ (e_1 \mapsto e_2) \rho m := \exists b v . \llbracket e_1 \rrbracket_{\rho,m} = \text{ptr } b \land \]
\[ \llbracket e_2 \rrbracket_{\rho,m} = v \land m = \{(b, (v, \gamma))\} \]

\[ (P \triangleright) \rho m := P \rho (\text{unlock} (\text{locks } m)m) \]

(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** $\Delta \vdash \{ P \} e \{ Q \}$

**$Q : val \rightarrow assert$**

In the semantics: if $P$ holds beforehand, then
- $e$ does not crash
- $Q \nu$ holds afterwards when terminating with $\nu$
- with framing memories that can change at each step
The axiomatic semantics

24 separation logic proof rules, e.g.:

For assignments:

Write $\subseteq$ kind $\gamma$ \hspace{1em} $\{ P_1 \} e_1 \{ Q_1 \}$ \hspace{1em} $\{ P_2 \} e_2 \{ Q_2 \}$ \hspace{1em} $\forall a v . ((Q_1 a * Q_2 v) \rightarrow ((a \mapsto \lnot) * R a v))$

\begin{align*}
\{ P_1 * P_2 \} e_1 & := e_2 \{ \lambda v . \exists a . (a \mapsto v) * R a v \}
\end{align*}

For dereferencing:

\begin{align*}
\text{kind } \gamma \neq \text{Locked} & \quad \{ P \} e \{ \lambda a . \exists v . Q a v * (a \mapsto v) \}
\Rightarrow \\
\{ P \} \text{load } e \{ \lambda v . \exists a . Q a v * (a \mapsto v) \}
\end{align*}

For the conditional:

\begin{align*}
\{ P \} e_1 \{ \lambda v . v \neq \text{indet} \wedge P' v \} \Rightarrow \{ \exists v . \text{istrue } v \wedge P' v \} e_2 \{ Q \} \quad \ldots
\Rightarrow \\
\{ P \} e_1 ? e_2 : e_3 \{ Q \}
\end{align*}

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 \( \Delta \ A \ \gamma \ e \ P \ Q : \)

\[
\text{perm_kind} \ \gamma \neq \text{Locked} \rightarrow \\
\Delta \setminus A \models_e \{ \{ P \} \} e \{ \{ \lambda a, \exists v, Q a v \ast \text{valc} a \mapsto \{ \gamma \} \ \text{valc} v \} \} \rightarrow \\
\Delta \setminus A \models_e \{ \{ P \} \} \text{load} e \{ \{ \lambda v, \exists a, Q a v \ast \text{valc} a \mapsto \{ \gamma \} \ \text{valc} v \} \}.
\]
The bigger picture / Future work

Interpreter for C11’

Separation algebras for the C memory (Submitted)

C-types & strict aliasing restrictions (CPP, 2013)

Sequence points & non-determinism (POPL, 2014)

Separation logic for C11’

Non-local control & block scope variables (FoSSaCS, 2013)
Questions

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

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