An Operational and Axiomatic Semantics for Non-determinism and Sequence Points in C

Robbert Krebbers

Radboud University Nijmegen

January 22, 2014 @ POPL, San Diego, USA
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.
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
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
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
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

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

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

- Non-determinism is subtle in innocent looking examples:
  
  ```c
  *p = g (x, y, z);
  Here, g may change p, so the evaluation order matters
  ```

- Interleaving of subexpressions is possible, for example
  ```c
  printf("a") + (printf("b") + printf("c"))
  ```
  may print "bac"
Examples

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

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

- Non-determinism is subtle in innocent looking examples:

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

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

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

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

- Non-determinism is subtle in innocent looking examples:

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

Here, \( g \) may change \( p \), so the evaluation order matters

- Interleaving of subexpressions is possible, for example

```c
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
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
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
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

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

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
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
Key idea

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

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

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

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**
Definition of the memory

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

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

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

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

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

Actual permissions contains:

- Locked flag to catch sequence point violations
  - Assignment: lock memory location
  - Sequence point: unlock memory locations
- A fraction $\left(0, 1\right]$ to allow sharing
- Block scope variable or allocated with malloc flag
Permission systems

Actual permissions contains:

- Locked flag to catch sequence point violations
  - Assignment: lock memory location
  - Sequence point: unlock memory locations
- A fraction \((0, 1]_{\mathbb{Q}}\) to allow sharing
- Block scope variable or allocated with `malloc` flag

A permission system \(P\) abstracts from these details.

- \(\cup, \setminus\) : \(P \to P \to P\)
- \(\perp, \subseteq\) : \(P \to P \to \text{Prop}\)
- kind : \(P \to \{\text{Free}, \text{Write}, \text{Read}, \text{Locked}\}\)
- lock, unlock : \(P \to P\)
- satisfying certain axioms
Permission systems

Actual permissions contains:

- Locked flag to catch sequence point violations
  - Assignment: lock memory location
  - Sequence point: unlock memory locations
- A fraction \((0, 1]_{\mathbb{Q}}\) to allow sharing
- Block scope variable or allocated with `malloc` flag

A permission system \(P\) abstracts from these details.

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

Actual permissions contains:

- Locked flag to catch sequence point violations
  - Assignment: lock memory location
  - Sequence point: unlock memory locations
- A fraction \((0, 1]_\mathbb{Q}\) to allow sharing
- Block scope variable or allocated with `malloc` flag

A permission system \(P\) abstracts from these details.

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

Actual permissions contains:
- Locked flag to catch sequence point violations
  - Assignment: lock memory location
  - Sequence point: unlock memory locations
- A fraction \((0, 1]_\mathbb{Q}\) to allow sharing
- Block scope variable or allocated with \texttt{malloc} flag

A permission system \(P\) abstracts from these details.
- \(\cup, \setminus : P \rightarrow P \rightarrow P\)
- \(\bot, \subseteq : P \rightarrow P \rightarrow \text{Prop}\)
- \(\text{kind} : P \rightarrow \{\text{Free, Write, Read, Locked}\}\)
- \(\text{lock, unlock} : P \rightarrow P\)
- satisfying certain axioms

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

Our language

⊠ ∈ binop ::= == | <= | + | - | * | / | % | ...

e ∈ expr ::= x_i | [ν]_Ω | e_1 := e_2 | f(\vec{e}) | load e | alloc
  | free e | e_1 ⊠ e_2 | e_1 ? e_2 : e_3 | (\tau)e

s ∈ stmt ::= e | skip | goto l | return e | block c s | s_1 ; s_2
  | l : s | while(e) s | if (e) s_1 else s_2
The language

Our language

\[⊗ ∈ \text{binop} ::= == | <= | + | - | * | / | % | \ldots\]

\[e ∈ \text{expr} ::= x_i | [v]_Ω | e_1 := e_2 | f(\vec{e}) | \text{load } e | \text{alloc}\]

\[| \text{free } e | e_1 ⊗ e_2 | e_1 ? e_2 : e_3 | (τ) e\]

\[s ∈ \text{stmt} ::= e | \text{skip} | \text{goto } l | \text{return } e | \text{block } c s | s_1 ; s_2\]

\[| l : s | \text{while}(e) s | \text{if } (e) s_1 \text{ else } s_2\]

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

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

**Head reduction for expressions** \((e, m) \xrightarrow{h} (e', m')\)  (9 rules)

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

- On sequence points: \(\Omega\) unlocked in memory
  \(([v]_{\Omega} ? e_2 : e_3, m) \xrightarrow{h} (e_2, \text{unlock } \Omega \ m)\) provided . . .
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
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
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:

\[
(P \ast Q)_{\rho} m := \exists m_1 \ m_2 . \]

\[
m = m_1 \cup m_2 \land m_1 \perp m_2 \land P_{\rho} m_1 \land Q_{\rho} m_2
\]
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:

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

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

(with \(e_1\) and \(e_2\) side-effect free)
Assertions of separation logic

Defined using a shallow embedding:

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

Maps variables to indexes in memory

Some assertions:

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

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

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

(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
The Hoare “triples”

### Statement judgment
\[ \Delta; J; R \vdash \{P\} s \{Q\} \]
- Function conditions
- Goto/return conditions
- \( Q : \text{assert} \)

### Expression judgment
\[ \Delta \vdash \{P\} e \{Q\} \]
- \( Q : \text{val} \to \text{assert} \)

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:

\[
\text{Write } \subseteq \text{ kind } \gamma \quad \{P_1\} e_1 \{Q_1\} \quad \{P_2\} e_2 \{Q_2\} \quad \forall av. ((Q_1 \ a \ast Q_2 \ v) \rightarrow ((a \mapsto \gamma \ast) \ast R \ a \ v))
\]

\[
\{P_1 \ast P_2\} e_1 := e_2 \{\lambda v. \exists a. (a \lock \gamma v) \ast R \ a \ v\}
\]
The axiomatic semantics

24 separation logic proof rules, e.g.:

For assignments:

\[
\begin{align*}
\text{Write } & \subseteq \text{ kind } \gamma \quad \{P_1\} e_1 \{Q_1\} \quad \{P_2\} e_2 \{Q_2\} \quad \forall a v . \ ((Q_1 a \ast Q_2 v) \rightarrow ((a \xmapsto{\gamma} \neg) \ast R a v)) \\
& \{P_1 \ast P_2\} e_1 := e_2 \{\lambda v . \exists a . (a \xmapsto{\text{lock } \gamma} v) \ast R a v\}
\end{align*}
\]

For dereferencing:

\[
\begin{align*}
\text{kind } \gamma & \neq \text{ Locked} \quad \{P\} \ e \ \{\lambda a . \exists v . Q a v \ast (a \xmapsto{\gamma} v)\} \\
& \{P\} \ \text{load} \ e \ \{\lambda v . \exists a . Q a v \ast (a \xmapsto{\gamma} v)\}
\end{align*}
\]
The axiomatic semantics

24 separation logic proof rules, e.g.:

For assignments:

\[
\text{Write } \subseteq \text{ kind } \gamma \quad \{P_1\} e_1 \{Q_1\} \quad \{P_2\} e_2 \{Q_2\} \quad \forall a, v. ((Q_1 a \ast Q_2 v) \rightarrow ((a \mapsto \gamma) \ast R a v))
\]

\[
\{P_1 \ast P_2\} e_1 := e_2 \{\lambda v. \exists a. (a \mapsto \gamma \longrightarrow v) \ast R a v\}
\]

For dereferencing:

\[
\text{kind } \gamma \neq \text{ Locked} \quad \{P\} e \{\lambda a. \exists v. Q a v \ast (a \mapsto \gamma \longrightarrow v)\}
\]

\[
\{P\} \text{load } e \{\lambda v. \exists a. Q a v \ast (a \mapsto \gamma \longrightarrow v)\}
\]

For the conditional:

\[
\Delta \vdash \{P\} e_1 \{\lambda v. v \neq \text{ indet} \land P’ v \triangleright\} \quad \Delta \vdash \exists v. \text{istrue } v \land P’ v \quad e_2 \{Q\} \quad \ldots
\]

\[
\Delta \vdash \{P\} e_1 ? e_2 : e_3 \{Q\}
\]
The axiomatic semantics

24 separation logic proof rules, e.g.:

For assignments:

Write \( \subseteq \text{kind } \gamma \quad \{P_1\} \ e_1 \ \{Q_1\} \quad \{P_2\} \ e_2 \ \{Q_2\} \quad \forall a \nu . ((Q_1 \ a \ * \ Q_2 \ \nu) \rightarrow ((a \stackrel{\gamma}{\rightarrow} \nu) \ * \ R \ a \ \nu)) \)

\[
\{P_1 \ * \ P_2\} \ e_1 \ := \ e_2 \ \{\nu \ . \ \exists a . (a \stackrel{\text{lock } \gamma}{\rightarrow} \nu) \ * \ R \ a \ \nu\}
\]

For dereferencing:

\[
\text{kind } \gamma \neq \text{Locked} \quad \{P\} \ e \ \{\lambda \ a . \ \exists \nu . \ Q \ a \ \nu \ * \ (a \stackrel{\gamma}{\rightarrow} \nu)\}
\]

\[
\{P\} \ \text{load} \ e \ \{\nu . \ \exists a . \ Q \ a \ \nu \ * \ (a \stackrel{\gamma}{\rightarrow} \nu)\}
\]

For the conditional:

\[
\Delta \vdash \{P\} \ e_1 \ \{\nu . \ \nu \neq \text{indet } \land \ P' \ \nu \} \quad \Delta \vdash \{\exists \nu . \ \text{istrue } \nu \ \land \ P' \ \nu\} \ e_2 \ \{Q\} \quad \ldots
\]

\[
\Delta \vdash \{P\} \ e_1 \ ? \ e_2 : \ e_3 \ \{Q\}
\]

Common separation logic and more complex rules can be derived
Formalization in Coq

- Based on [Krebbers/Wiedijk, FoSSaCS’13]
- 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 \( \Delta A \gamma e P Q : \)
\[
\text{perm\_kind } \gamma \neq \text{Locked} \rightarrow \\
\Delta \setminus A \models_e \{\{ P \} \} e \{\{ \lambda a, \exists v, Q a v \ast \text{valc} a \rightarrow \{\gamma\} \text{valc} v \}\} \rightarrow \\
\Delta \setminus A \models_e \{\{ P \} \} \text{load} e \{\{ \lambda v, \exists a, Q a v \ast \text{valc} a \rightarrow \{\gamma\} \text{valc} v \}\}.
\]
Future research

- Integration with the C type system [Krebbers, CPP’13]
- Integration with non-aliasing restrictions [Krebbers, CPP’13]
- Interpreter in Coq
- Verification condition generator in Coq
- Automation of separation logic
Questions

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