Non-determinism in C expressions

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

According to the C standard, the order of evaluation is **unspecified**, e.g., compilers are free to choose their evaluation strategy

...so we would expect as the outcome either "4, 7" or "3, 7"
```c
int main() {
    int x;
    int y = (x = 3) + (x = 4);
    printf("%d, %d\n", x, y);
}
```

However, a small experiment with existing compilers gives

<table>
<thead>
<tr>
<th>compiler</th>
<th>outcome</th>
<th>warnings</th>
</tr>
</thead>
<tbody>
<tr>
<td>compcert</td>
<td>4, 7</td>
<td>no</td>
</tr>
<tr>
<td>clang</td>
<td>4, 7</td>
<td>yes</td>
</tr>
<tr>
<td>gcc-4.9</td>
<td>4, 8</td>
<td>no</td>
</tr>
</tbody>
</table>
int main() {
    int x;
    int y = (x = 3) + (x = 4);
    printf("%d, %d\n", x, y);
}

According to the C standard, this program violates the sequence point restriction due to two unsequenced writes of the same variable x.

A sequence point violation results in the undefined behavior i.e., the program is allowed do anything it is even allowed to crash.
The problem: sequence point violations may cause a C program to crash or to have arbitrary results.

The goal: we need a framework that, besides the functional correctness, ensures the absence of undefined behavior for any evaluation order.

\[
\{ P \} e \{ Q \} \implies \text{functional correctness} \land \text{no sequence point violations} \land \text{no other undefined behavior}
\]
The problem: sequence point violations may cause a C program to crash or to have arbitrary results.

The goal: we need a framework that, besides the functional correctness, ensures the absence of undefined behavior for any evaluation order.

\[
\begin{align*}
\{ r \mapsto i \times c \mapsto j \} \\
* r &= * r \times (++(*c)); \\
\{ v. v = i \times (j+1) \land r \mapsto i \times (j+1) \times c \mapsto j + 1 \}
\end{align*}
\]
Observation: view non-determinism through concurrency

Idea: use concurrent separation logic

\[
\begin{array}{c}
\{P_1\}e_1 \{\Psi_1\} \quad \{P_2\}e_2 \{\Psi_2\} \\
\forall v_1 v_2. \Psi_1 v_1 * \Psi_2 v_2 \vdash \Phi(w_1 [\odot] w_2) \\
\{P_1 * P_2\} e_1 \odot e_2 \{\Phi\}
\end{array}
\]

With the rules of this logic we can
- split the memory resources into two disjoint parts
- independently prove that each subexpression executes safely in its own part

Disjointedness $\Rightarrow$ no sequence point violations
Observation: view non-determinism through concurrency

Idea: use concurrent separation logic

\[
\{ P_1 \} e_1 \{ \Psi_1 \} \quad \{ P_2 \} e_2 \{ \Psi_2 \} \quad \forall v_1 \ v_2. \ \Psi_1 \ v_1 \ast \Psi_2 \ v_2 \vdash \Phi(w_1 \ [\odot] \ w_2)
\]

\[
\{ P_1 \ast P_2 \} e_1 \odot e_2 \{ \Phi \}
\]

With the rules of this logic we can

- split the memory resources into two disjoint parts
- independently prove that each subexpression executes safely in its own part

Disjointedness ⇒ no sequence point violations
**Observation**: view non-determinism through **concurrency**

**Idea**: use concurrent separation logic

\[
\{ P_1 \} e_1 \{ \Psi_1 \} \quad \{ P_2 \} e_2 \{ \Psi_2 \} \quad \forall v_1 v_2. \, \Psi_1 \, v_1 \ast \Psi_2 \, v_2 \vdash \Phi(w_1 \sqcup w_2)
\]

\[
\{ P_1 \ast P_2 \} e_1 \ominus e_2 \{ \Phi \}
\]

With the rules of this logic we can

- split the memory resources **into two disjoint parts**
- independently prove that each subexpression **executes safely in its own part**

Disjointedness $\Rightarrow$ **no sequence point violations**
Limitations of Krebbers’s program logic

1. The program logic is difficult to extend with new features.
2. The proof process is tedious and has no support for automation:
   - we have to subdivide resources manually all the time
   - and to infer the intermediate postconditions.

\[
\begin{align*}
\{P_1\} e_1 \{\psi_1\} & \quad \{P_2\} e_2 \{\psi_2\} & \quad \forall v_1 v_2. \, \psi_1 v_1 * \psi_2 v_2 \vdash \phi(w_1 [\odot] w_2) \\
\{P_1 * P_2\} e_1 \odot e_2 \{\phi\}
\end{align*}
\]
Limitations of Krebbers’s program logic

1. The program logic is difficult to extend with new features.
2. The proof process is tedious and has no support for automation:
   - we have to subdivide resources manually all the time
   - and to infer the intermediate postconditions.

\[
\{ P_1 \} e_1 \{ \Psi_1 \} \quad \{ P_2 \} e_2 \{ \Psi_2 \} \quad \forall v_1, v_2. \, \Psi_1 \, v_1 * \Psi_2 \, v_2 \vdash \Phi(w_1 \Box w_2)
\]

\[
\{ P_1 * P_2 \} e_1 \diamond e_2 \{ \Phi \}
\]
Limitations of Krebbers’s program logic

1. The program logic is difficult to extend with new features.
2. The proof process is tedious and has no support for automation:
   - we have to subdivide resources manually all the time
   - and to infer the intermediate postconditions.

\[
\begin{align*}
\{P_1\} e_1 \{\Psi_1\} \quad \{P_2\} e_2 \{\Psi_2\} \quad \forall v_1, v_2. \Psi_1 v_1 \ast \Psi_2 v_2 \vdash \Phi(w_1 \circ \circ \w_2) \\
\{P_1 \ast P_2\} e_1 \circ e_2 \{\Phi\}
\end{align*}
\]
Limitations of Krebbers’s program logic

1. The program logic is difficult to extend with new features.
2. The proof process is tedious and has no support for automation:
   - we have to subdivide resources manually all the time
   - and to infer the intermediate postconditions.

\[
\begin{align*}
\{P_1\} e_1 \{\psi_1\} & \quad \{P_2\} e_2 \{\psi_2\} & \quad \forall v_1 v_2. \psi_1 v_1 * \psi_2 v_2 \vdash \phi(w_1 \circledast w_2) \\
\{P_1 * P_2\} e_1 \odot e_2 \{\phi\}
\end{align*}
\]

⇒ Such rules cannot be applied in an algorithmic fashion.
This paper: Redesign Krebbers’s program logic and turn it into a semi-automated procedure
Contributions

Contribution 1:

A redesign of Krebbers’s logic using a **weakest precondition calculus**.

⇒ makes automation possible
Contribution 2:

A **monadic semantics** of C non-determinism **by translation** into a concurrent ML language.

⇒ makes the semantics declarative
Contributions

Contribution 1: A redesign of Krebbers's logic using a weakest precondition calculus. ⇒ makes automation possible

Contribution 2: A monadic semantics of C non-determinism by translation into a concurrent ML language. ⇒ makes the semantics declarative

Contribution 3: A layered model of our program logic built on top of the Iris framework ⇒ modular and expressive logic, Coq tactics

Contribution 4: A symbolic execution algorithm integrated into an interactive vcgen ⇒ useful in an interactive theorem prover

Contribution 5: 8
Contribution 1:
A redesign of Krebbers’s logic using a weakest precondition calculus.

Contribution 2:
A monadic semantics of C non-determinism by translation into a concurrent ML language.

Contribution 3:
A layered model of our program logic built on top of the Iris framework.

Contribution 4:
A symbolic execution algorithm integrated into an interactive vcgen.
⇒ useful in an interactive theorem prover

Contribution 5:
Contribution 1: A redesign of Krebbers's logic using a weakest precondition calculus.⇒ makes automation possible

Contribution 2: A monadic semantics of C non-determinism by translation into a concurrent ML language.⇒ makes the semantics declarative

Contribution 3: A layered model of our program logic built on top of the Iris framework⇒ modular and expressive logic, Coq tactics

Contribution 4: A symbolic execution algorithm integrated into an interactive vcgen⇒ useful in an interactive theorem prover

Contribution 5:
This talk:
Symbolic execution algorithm and vcgen
Key idea

Turn the program logic into an algorithm procedure using a novel *symbolic execution* algorithm:

<table>
<thead>
<tr>
<th><strong>input</strong></th>
<th><strong>output</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>precondition</td>
<td>value</td>
</tr>
<tr>
<td>program</td>
<td>(strongest) postcondition</td>
</tr>
<tr>
<td></td>
<td><em>frame</em> = resources not used</td>
</tr>
</tbody>
</table>
Turn the program logic into an algorithm procedure using a novel symbolic execution algorithm:

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r \mapsto i \cdot c \mapsto j \cdot d \mapsto k )</td>
<td>( i \cdot (j+1) )</td>
</tr>
<tr>
<td>( *r = *r \cdot (++(*c)); )</td>
<td>( r \mapsto i \cdot (j+1) \cdot c \mapsto j + 1 )</td>
</tr>
<tr>
<td></td>
<td>( d \mapsto k )</td>
</tr>
</tbody>
</table>
Symbolic execution algorithm

The evaluation order in the symbolic execution algorithm does not matter: 

\((P, e_1) \xrightarrow{symb.\,exec.} (w, Q, R)\) 

\[P \vdash wp e_1 \{v_1 \cdot v_2 = w \cdot Q_1 \cdot Q_2 \} \cdot R_1 \]
Symbolic execution algorithm

The evaluation order in the symbolic execution algorithm does not matter: 

\((P, e_1) \xrightarrow{symb\_exec} (w, Q, R)\)

\[P \vdash w \left\{ v_1 = w \ast Q \right\} \ast R\]
Symbolic execution algorithm

The evaluation order in the symbolic execution algorithm does not matter:

\[(P, e_1) \xrightarrow{\text{symb}.\, \text{exec.}} (w, Q_1, R_1)\]

\[P \vdash_{\text{wp}} e_2 \{ v_1, v_2 = w \ast Q_1 \ast R_1 \}\]
Symbolic execution algorithm

The evaluation order in the symbolic execution algorithm does not matter:

\[(P, e_1) \xrightarrow{\text{symb. exec.}} (w, Q_1, R_1)\]

\[P \not\vdash w \ni e_2 \{v_1. v_1 = w \ast Q_1\} \ast R_1\]
Symbolic execution algorithm

The evaluation order in the symbolic execution algorithm does not matter:

\[(P, e_1) \rightarrow \text{symb. exec.} \rightarrow (w, Q_1, R_1)\]

\[P \vdash w\{v_1.v_1 = w \ast Q_2\}\ast R_2\]

Diagram:

- \(P\) to \(Q_1\) through \(e_1\)
- \(Q_1\) to \(R_1\)
- \(R_1\) to \(Q_2\)
- \(Q_2\) to \(R_2\)
- \(P\) to \(v_1\) along a circle
The evaluation order in the symbolic execution algorithm does not matter:

\[(P, e_1) \xrightarrow{symb.\,exec.} (w, Q_1, R_1)\]

\(P \vdash wp e \{ v_1.v_1 = w \ast Q_2 \} \ast R_2\)
Symbolic execution algorithm

The evaluation order in the symbolic execution algorithm does not matter:

\[
(P, e) \xrightarrow{symb. \ exec.} (w, Q, R)
\]

\[
P \vdash wp \ e \{v. v = w \ast Q\} \ast R
\]
Towards automation

Symbolic execution algorithm that computes the frame allows to apply the program logic rules in an algorithmic manner:

\[
(P, e_1) \xrightarrow{\text{symb. exec.}} (w_1, Q, R) \quad R \vdash \text{wp } e_2 \{w_2. \ Q \rightarrow \Phi (w_1 \Rrightarrow w_2)\}
\]

\[
P \vdash \text{wp } (e_1 \circ e_2) \{\Phi\}
\]

Compare this with applying the rule that does not use symbolic execution:

\[
P_1 \vdash \text{wp } e_1 \{\Psi_1\} \quad P_2 \vdash \text{wp } e_2 \{\Psi_2\} \quad (\forall w_1, w_2. \ \Psi_1 \ w_1 \ast \Psi_2 \ w_2 \rightarrow \Phi (w_1 \Rrightarrow w_2))
\]

\[
P_1 \ast P_2 \vdash \text{wp } (e_1 \circ e_2) \{\Phi\}
\]
Towards automation

Symbolic execution algorithm that computes the frame allows to apply the program logic rules in an algorithmic manner:

\[ (P, e_1) \xrightarrow{symb.\,\,exec.} (w_1, Q, R) \quad R \vdash wp\, e_2 \{ w_2. Q \not= \Phi (w_1 [\odot] w_2) \} \]

However, the algorithm itself may fail for several reasons:

- the program is not of the right shape (loop, function call, . . .)
- the precondition is not of the right shape (needed resource is missing, . . .)
Key idea: design an interactive verification condition generator (vcgen).

Vcgen automates the proof as long as the symbolic executor does not fail. When the symbolic executor fails, vcgen does not fail itself, but

- returns to the user a partially solved goal
- from which it can be called back after the user helped out.
Hr: \( r \mapsto 1 \)

Hc: \( c \mapsto 0 \)

\[
\text{while}(*c < n)\{
\quad *r = *r \ast (++(*c));
\}
\]

Post-condition: \( r \mapsto \text{fact}(n) \ast c \mapsto n \)
\exists k \leq n.

Hr: \ r \mapsto \text{fact}(k)

Hc: \ c \mapsto k

\[
\begin{align*}
\text{while}(*c < n)\{ \\
\quad *r = *r \ast (++(*c)); \\
\}
\end{align*}
\]

Post-condition: \ r \mapsto \text{fact}(n) \ast c \mapsto n
Hr: \( r \mapsto \text{fact}(k) \)

Hc: \( c \mapsto k \)

IH: \( \forall k. \, k \leq n \)

\( r \mapsto \text{fact}(k) \times c \mapsto k \times k \leq n \)

\( \text{wp} (\text{while}(\ldots)\{\ldots\}) \)

\( \{ r \mapsto \text{fact}(n) \times c \mapsto n \} \)

Proof.

generalize Hr Hc. induction.

Post-condition: \( r \mapsto \text{fact}(n) \times c \mapsto n \)
Hr:  \( r \mapsto \text{fact}(k) \)

Hc:  \( c \mapsto k \)

IH:  \( \forall k. \)

\[
\begin{align*}
    r & \mapsto \text{fact}(k) \ast c \mapsto k \ast k \leq n \\
    \text{wp} \left( \text{while}(..)[...][..] \right) \\
    \{ r \mapsto \text{fact}(n) \ast c \mapsto n \}
\end{align*}
\]

Proof.

generalize Hr Hc. induction. while_spec.

\[
\begin{align*}
\text{if} \ (*c < n) \{ \\
    *r & = *r \ast (++(*c)) \\
    \text{while}(*c < n)\{ \\
        *r & = *r \ast (++(*c)) \\
    \} \\
\}
\end{align*}
\]

Post-condition:  \( r \mapsto \text{fact}(n) \ast c \mapsto n \)
Hr: \( r \mapsto \text{fact}(k) \)

Hc: \( c \mapsto k \)

IH: \( \forall k. \)
\[
r \mapsto \text{fact}(k) \land c \mapsto k \land k \leq n \implies
\]
wp (while(.){...})
\[
\{ r \mapsto \text{fact}(n) \land c \mapsto n \}
\]

Proof.
- generalize Hr Hc.
- induction.
- while_spec.
- vcgen.

Post-condition: \( r \mapsto \text{fact}(n) \land c \mapsto n \)
Hr: \( r \mapsto \text{fact}(k) \)

Hc: \( c \mapsto k \)

IH: \( \forall k. \)

\[
\begin{align*}
  r &\mapsto \text{fact}(k) \star c \mapsto k \star k \leq n \rightarrow \\
  \wp (\text{while}(\ldots)\{\ldots\}) \\
  \{ r \mapsto \text{fact}(n) \star c \mapsto n \}
\end{align*}
\]

Proof.

generalize Hr Hc. induction. while_spec. 

vcgen.

if (*c < n) {

  \( *r = *r \star (++(*c)) \);

  while(*c < n) {

    \( *r = *r \star (++(*c)) \);

  }

}

Post-condition: \( r \mapsto \text{fact}(n) \star c \mapsto n \)
Hr: \( r \mapsto \text{fact}(k) \)

Hc: \( c \mapsto k \)

Hk: \( k < n \)

IH: \( \forall k. \begin{align*}
    r & \mapsto \text{fact}(k) \ast c \mapsto k \ast k \leq n \rightarrow \\
    \text{wp} \begin{scope}[very small]
        \text{while}(..)\{\ldots\}
    \end{scope}
    \{ r \mapsto \text{fact}(n) \ast c \mapsto n \}
\end{align*} \)

Goal [1/2].

\[ *r = *r \ast (++(*c)); \]

\[ \text{while}(*c < n)\begin{scope}[very small]
    \{ *r = *r \ast (++(*c)); \}
\end{scope} \]


Proof.

generalize Hr Hc. induction. while_spec.

\texttt{vcgen}.

Post-condition: \( r \mapsto \text{fact}(n) \ast c \mapsto n \)
Hr: \( r \mapsto \text{fact}(k) \)

Hc: \( c \mapsto k \)

Hk: \( k < n \)

IH: \( \forall k. \)
\[
    r \mapsto \text{fact}(k) \quad c \mapsto k \quad k \leq n \quad \ast
\]

wp (while(\( \ast c < n \))\{ \ast r = \ast r \ast (\ast c); \}
\[
    \ast r = \ast r \ast (\ast c); \}
\}

Goal [1/2].

\[
\begin{align*}
    \ast r &= \ast r \ast (\ast c); \\
    \text{while}(\ast c < n)\{ \\
        \ast r &= \ast r \ast (\ast c); \\
    \}
\end{align*}
\]

Post-condition: \( r \mapsto \text{fact}(n) \ast c \mapsto n \)
Hr: $r \mapsto \text{fact}(k) \cdot (k + 1)$

Hc: $c \mapsto (k + 1)$

Hk: $k < n$

IH: $\forall k.
  r \mapsto \text{fact}(k) \cdot c \mapsto k \cdot k \leq n \rightarrow
  \wp (\text{while}(..)\{\ldots\})
  \{ r \mapsto \text{fact}(n) \cdot c \mapsto n \}$

Proof.
  generalize Hr Hc. induction. while_spec.
  vcgen.
  - vcgen.

Goal [1/2].

Post-condition: $r \mapsto \text{fact}(n) \cdot c \mapsto n$
Hr: $r \mapsto \text{fact}(k) \cdot (k + 1)$

Hc: $c \mapsto (k + 1)$

Hk: $k < n$

IH: $\forall k. \ \ r \mapsto \text{fact}(k) \ast c \mapsto k \ast k \leq n \rightarrow$

\[
\text{wp}\left(\text{while}(..\{\ldots\})\right)\]
\[\{r \mapsto \text{fact}(n) \ast c \mapsto n\}\]

Goal [1/2].

Post-condition: $r \mapsto \text{fact}(n) \ast c \mapsto n$

Proof.

generalize Hr Hc. induction. while_spec.

vcgen.

- vcgen. apply IH.
Hr: $r \mapsto \text{fact}(k)$

Hc: $c \mapsto k$

Hk: $k = n$

IH: $\forall k. \ r \mapsto \text{fact}(k) \ * \ c \mapsto k \ * \ k \leq n \ *$

wp (while(..){...})

\{ r \mapsto \text{fact}(n) \ * \ c \mapsto n \}$

Goal [2/2].

Post-condition: $r \mapsto \text{fact}(n) \ * \ c \mapsto n$

Proof.

generalize Hr Hc. induction. while_spec.

vcgen.

- vcgen. apply IH.

- eauto.

Qed.
We implemented the symbolic execution algorithm as a partial function which we restrict to symbolic heaps:

\[
\text{forward} : (\text{sheap} \times \text{expr}) \rightarrow (\text{val} \times \text{sheap} \times \text{sheap})
\]

satisfying the following specification:

\[
\text{forward}(m, e) = (w, m^0_1, m_1)
\]

\[
\begin{array}{c}
\llbracket m \rrbracket \vdash \text{wp } e \{ v. v = w \ast \llbracket m^0_1 \rrbracket \} \ast \llbracket m_1 \rrbracket
\end{array}
\]
We implemented the symbolic execution algorithm as a partial function which we restrict to symbolic heaps:

\[
\text{forward} : (\text{sheap} \times \text{expr}) \rightarrow (\text{val} \times \text{sheap} \times \text{sheap})
\]

**Future work:**
- lift the restriction for the precondition to handle pure facts
- enable interaction with external decision procedures
The vcgen is implemented as a total function

\[ \text{vcg} : (\text{sheap} \times \text{expr} \times (\text{sheap} \rightarrow \text{val} \rightarrow \text{Prop})) \rightarrow \text{Prop} \]

satisfying the following specification:

\[
\begin{align*}
  P' & \vdash \text{vcg}(m, e, \lambda m'. v. [m'] \rightarrow \Phi \ v) \\
  P' \ast [m] & \vdash \text{wp} e \ {\{\Phi\}}
\end{align*}
\]
Conclusion

Other contributions and related topics not covered in this talk:
- monadic definitional semantics of a subset of C
- multi-layered design of weakest precondition calculus on top of Iris
- proof by reflection as a part of development of automated procedures

The main message for today:

Symbolic execution with frames is a key to enable semi-automated reasoning about C non-determinism in an interactive theorem prover.
Thank you!