Semi-Automated Reasoning About Non-Determinism in C Expressions

Léon Gondelman

joint work with Dan Frumin and Robbert Krebbers

15 April, 2019 @ Gallium, Paris

1 Radboud University Nijmegen 2 Delft University of Technology
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"
Unexpectedly

```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 to 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.

\[
\begin{align*}
\{ r \rightarrow i \ast c \rightarrow j \} \\
* r &= * r \ast (++(* c)) ; \\
\{ v. v = i \cdot (j + 1) \land r \rightarrow i \cdot (j + 1) \ast c \rightarrow j + 1 \}
\end{align*}
\]
Previous work:
Krebbers’ program logic (POPL’14)
**Observation**: view non-determinism through concurrency

**Idea**: use concurrent separation logic

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

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

\[
\{ P_1 \} e_1 \{ \psi_1 \} \quad \{ P_2 \} e_2 \{ \psi_2 \} \quad \forall v_1 v_2. \; \psi_1 v_1 \ast \psi_2 v_2 \vdash \Phi(w_1 \bowtie w_2)
\]

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

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
**Observation**: view non-determinism through concurrency

**Idea**: use concurrent separation logic

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

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
Krebbers’ logic addresses other aspects of sequence point restrictions in C:

- **sharing of resources** between subexpressions
- additional enforcement for nested assignments
- sequence points and function calls

\[(\ast l = \ast k + 10) + (\ast r = \ast k + 10)\]  

\[\Rightarrow \text{Use fractional permissions: } k \xrightarrow{q_1+q_2} v \parallel k \xrightarrow{q_1} v \ast k \xrightarrow{q_2} v\]
Krebbers’ logic addresses other aspects of sequence point restrictions in C:

- sharing of resources between subexpressions
- additional enforcement for **nested assignments**
- sequence points and function calls

\[ *1 = (*1 = 3) \]

⇒ Decorate permissions with a **lockable flag** \( \xi \in \{L, U\} \)
Kreberos’ logic addresses other aspects of sequence point restrictions in C:

- sharing of resources between subexpressions
- additional enforcement for **nested assignments**
- sequence points and function calls

\[
\begin{align*}
\{P_1\} \ e_1 \ \{\Psi_1\} \quad \{P_2\} \ e_2 \ \{\Psi_2\} \quad (\forall l \ . \ \Psi_1 \ l \ast \Psi_2 \ w \ast \exists v . \ l \mapsto_U v \ast (l \mapsto_L w \ast \Phi \ w))
\end{align*}
\]

\[
\{P_1 \ast P_2\} \ (e_1 = e_2) \ \{\Phi\}
\]

\[\implies \]

Decorate permissions with a **lockable flag** \(\xi \in \{L, U\}\)
Krebbers’ logic addresses other aspects of sequence point restrictions in C:

- sharing of resources between subexpressions
- additional enforcement for nested assignments
- sequence points and function calls

\[ *1 = 4; *l \quad f() + g() \]

\[ \Longrightarrow \text{ Define unlocking modality } \mathbb{U} \text{ such that } 1 \xrightarrow{q} v \vdash \mathbb{U}(1 \xrightarrow{q} v) \]
Krebbers’ logic addresses other aspects of sequence point restrictions in C:

- sharing of resources between subexpressions
- additional enforcement for nested assignments
- sequence points and function calls

\[
\begin{align*}
\{P\} \ e_1 \ {\{U(\Psi_1)\}} \ & \ {\Psi_1}\ e_2\ {\Phi} \\
\{P\} (e_1; e_2) \ {\Phi} \\
\end{align*}
\]

\[\implies\] Define **unlocking modality** \(U\) such that \(1 \xrightarrow{q} L v \vdash U(1 \xrightarrow{q} U v)\)
Limitations of Krebbers’ 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.

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

\[
\{ P_1 \} e_1 \{ \Psi_1 \} \quad \{ P_2 \} e_2 \{ \Psi_2 \} \quad \forall v_1 v_2. \; \Psi_1 v_1 * \Psi_2 v_2 \vdash \Phi(w_1 \llbracket \circ \rrbracket w_2) \\
\{ P_1 * P_2 \} e_1 \odot e_2 \{ \Phi \} 
\]
Limitations of Krebbers’ 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.

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

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

\[\Rightarrow \text{Such rules cannot be applied in an algorithmic fashion.}\]
This work:
Redesign Krebbers’s program logic and turn it into a semi-automated procedure
Contributions

Contribution 1:

A redesign of Krebbers’s logic using a \textit{weakest precondition calculus}.

$\Rightarrow$ decouples the program from the precondition

$\Rightarrow$ makes automation possible
Contributions

Contribution 1:
A redesign of Krebbers’s logic using a weakest precondition calculus. \( \Rightarrow \) decouples the program from the precondition \( \Rightarrow \) makes automation possible

Contribution 2:
A monadic semantics of C non-determinism by translation into a concurrent ML language. \( \Rightarrow \) makes the semantics declarative
\[ M(A) \triangleq \text{mset Ptr} \rightarrow \text{mutex} \rightarrow A \]

Contribution 3:
A layered model of our program logic built on top of the Iris framework \( \Rightarrow \) makes logic more modular and expressive \( \Rightarrow \) support from Iris Proof Mode and Coq tactics

Contribution 4:
A symbolic execution algorithm integrated into an interactive vcgen \( \Rightarrow \) turns logic into a semi-automated procedure \( \Rightarrow \) useful in an interactive theorem prover

Contribution 5:
Contributions

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

Contribution 2: A monadic semantics of C non-determinism by translation into a concurrent ML language. ⇒ makes the semantics declarative ⇒ reader monad $M(A) \equiv \text{mset} \text{Ptr} \rightarrow \text{mutex} \rightarrow A$

Contribution 3: A layered model of our program logic built on top of the Iris framework ⇒ makes logic more modular and expressive ⇒ support from Iris Proof Mode and Coq tactics

Contribution 4: A symbolic execution algorithm integrated into an interactive vcgen ⇒ turns logic into a semi-automated procedure ⇒ useful in an interactive theorem prover

Contribution 5: A ghost state Coq tactics...
Contributions

Contribution 1:
A redesign of Krebbers’s logic using a weakest precondition calculus.
⇒ decouples the program from the precondition
⇒ makes automation possible

Contribution 2:
A monadic semantics of C non-determinism by translation into a concurrent ML language.
⇒ makes the semantics declarative
⇒ reader monad $M(A) \equiv \text{mset} \text{Ptr} \rightarrow \text{mutex} \rightarrow A$

Contribution 3:
A layered model of our program logic built on top of the Iris framework
⇒ makes logic more modular and expressive
⇒ support from Iris Proof Mode and Coq tactics

Contribution 4:
A symbolic execution algorithm integrated into an interactive vcgen
⇒ turns logic into a semi-automated procedure
⇒ useful in an interactive theorem prover

Contribution 5:
12
Contribution 1: A redesign of Krebbers's logic using a weakest precondition calculus. ⇒ decouples the program from the precondition ⇒ makes automation possible

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

Contribution 3: A layered model of our program logic built on top of the Iris framework ⇒ makes logic more modular and expressive ⇒ support from Iris Proof Mode and Coq tactics

Contribution 4: A symbolic execution algorithm integrated into an interactive vcgen ⇒ turns logic into a semi-automated procedure ⇒ useful in an interactive theorem prover

Contribution 5:
This talk:
Symbolic execution algorithm and vcgen
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>frame = resources not used</td>
</tr>
</tbody>
</table>
Turn the program logic into an algorithm procedure using a novel **symbolic execution** algorithm:

**input**

\[ r \mapsto i \cdot c \mapsto j \cdot d \mapsto k \]

\[ *r = *r \cdot (++(*c)); \quad \rightarrow \]

**output**

\[ i \cdot (j+1) \]

\[ r \mapsto i \cdot (j+1) \cdot c \mapsto j + 1 \]

\[ d \mapsto k \]
The evaluation order in the symbolic execution algorithm does not matter: 

\[(P, e_1, e_2) \xrightarrow{\text{symb} \cdot \text{exec}} (w, Q, R)\]

\[P \vdash wp e_1 \{ v, v = w \cdot Q \} \ast R\]
Symbolic execution algorithm

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

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

\(P \vdash w \left\{ v. v = w \ast Q \right\} \ast R\)
Symbolic execution algorithm

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

\[(P, e_1) \rightarrow (w, Q_1, R_1)\]

\[P \vdash w \left\{ v_1 = w \ast Q_1 \right\} \ast R_1\]
Symbolic execution algorithm

The evaluation order in the symbolic execution algorithm does not matter: $(P, e_1) \xrightarrow{\text{symb. exec.}} (w, Q, R)$

$P \vdash w \leftarrow e_1 \{ v_1 \cdot v_2 = w \cdot Q_1 \} \ast R_1$
Symbolic execution algorithm

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

\[
\begin{align*}
P &\xrightarrow{e_1} P \\
&\quad \odot \\
&\xrightarrow{e_2} R_1 \\
&\quad \odot \\
&\xrightarrow{Q_1} v_1 \quad Q_1 \quad R_1 \\
&\quad \odot \\
&\xrightarrow{Q_2} v_2 \quad Q_2 \quad R_2 \\
&\quad \odot \\
&\xrightarrow{Q_1} P \vdash \wp \{ v_1, v_2 = \ast Q_1 \} \ast R_1 \\
\end{align*}
\]
Symbolic execution algorithm

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

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

\[P \vdash w e \{ \nu_1 = w \cdot Q_1 \} \cdot R_1\]
The evaluation order in the symbolic execution algorithm does not matter:

\[
(P, e) \xrightarrow{\text{symb. exec.}} (w, Q, R) \\
P \models \text{wp } e \{ v. v = w \ast Q \} \ast R
\]
Towards automation

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

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

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

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

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

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

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

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

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: $c \mapsto 0$

Proof.

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

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

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

Hc: \( c \mapsto k \)

Proof.

generalize Hr Hc.

```
while(*c < n){
  *r = *r * (++(*c));
}
```

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

Hc: \( c \mapsto k \)

IH: \( \forall k. \quad r \mapsto \text{fact}(k) * c \mapsto k * k \leq n \not\star \)

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

Proof.

generalize Hr Hc. induction.

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

Hc: \( c \mapsto k \)

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

Proof.

generalize Hr Hc. induction. while_spec.

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

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

Hc: \( c \mapsto k \)

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

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

\[
\{ r \mapsto \text{fact}(n) \land c \mapsto n \}
\]

Proof.

generalize Hr Hc. induction. while_spec.

\texttt{vcgen}.

if \( (*c < n) \) {

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

\[
\text{while}(*c < n)\{
\]

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

\}

}

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

Hc: $c \mapsto k$

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

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

Proof.

generalize Hr Hc. induction. while_spec.

vcgen.

if (*c < n) {
  *r = *r * (++(*c));
  while(*c < n){
    *r = *r * (++(*c));
  }
}

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

Hc: \( c \mapsto k \)

Hk: \( k < n \)

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

Goal [1/2].

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

while(*c < n)\{ \)
\( *r = *r \cdot (++(*c)); \)
\} \)

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

Hc: \( c \mapsto k \)

Hk: \( k < n \)

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

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

Goal [1/2].

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

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: \( \forall k. \)
\[
\begin{align*}
& \quad r \mapsto \text{fact}(k) \cdot c \mapsto k \cdot k \leq n \rightarrow \\
& \quad \text{wp} (\text{while}(...)\{...\}) \\
& \quad \{ r \mapsto \text{fact}(n) \cdot c \mapsto n \}
\end{align*}
\]

Goal [1/2].

while(*c < n) {
  *r = *r * (++(*c));
}

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

Proof.

generalize Hr Hc.
induction.
while_spec.
vcgen.

- vcgen.
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(while(.,{}))
    \{ r \mapsto \text{fact}(n) \cdot c \mapsto n \}

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

Goal [1/2].

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

Hc: \[ c \mapsto k \]

Hk: \[ k = n \]

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

\[ \text{wp} (\text{while}(..)\{\ldots\}) \]

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

-----------------------------

Proof.

generalize Hr Hc. induction. while_spec.

vcgen.

- vcgen. apply IH.

- eauto.

Qed.

-----------------------------

Goal [2/2].

()
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:

\[
\text{forward} (m, e) = (w, m_1^0, m_1) \\
\left[ m \right] \vdash \text{wp e} \left\{ v. \ v = w \ast \left[ m_1^0 \right] \right\} \ast \left[ m_1 \right]
\]
We implemented the symbolic execution algorithm as a partial function which we restrict to symbolic heaps:

\[
\text{forward : (sheap × expr) → (val × sheap × sheap)}
\]

**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:

\[
P' \vdash \text{vcg}(m, e, \lambda m'. \mathbf{[m'] \rightarrow \Phi v})
\]

\[
P' \star \mathbf{[m]} \vdash \text{wp } e \{ \Phi \}
\]
One piece of related work

\[
\frac{\Gamma_1 \vdash t_1 : q \ T_{11} \rightarrow T_{12} \quad \Gamma_2 \vdash t_2 : T_{11}}{\Gamma_1 \cdot \Gamma_2 \vdash t_1 \ t_2 : T_{12}} \quad \text{(T-APP)}
\]

Non-deterministic typing rule

\[
\frac{\Gamma_1 \vdash t_1 : q \ T_{11} \rightarrow T_{12}; \Gamma_2 \vdash t_2 : T_{11}; \Gamma_3}{\Gamma_1 \vdash t_1 \ t_2 : T_{12}; \Gamma_3} \quad \text{(A-APP)}
\]

Algorithmic type checking

“The central idea is that rather than splitting the context into parts before checking a complex expression composed of several subexpressions, we can pass the entire context as an input to the first subexpression and have it return the unused portion as an output.” (p.12)
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!
appendix
translation scheme (1/4)

\[
[e_1 = e_2] \overset{def}{=} \text{let } (p, v) = [e_1] \parallel_{HL} [e_2] \text{ in } p :=_{HL} v ; v
\]

\[
[e_1 + e_2] \overset{def}{=} \text{let } (v_1, v_2) = [e_1] \parallel_{HL} [e_2] \text{ in } v_1 +_{HL} v_2
\]

the **non-determinism** is embodied by using parallel composition \( \parallel_{HL} \)
the sequence point conditions are checked using a set of pointers env
\[ [e_1 = e_2] \overset{def}{=} \text{let} (p, v) = [e_1] ||_{HL} [e_2] \text{ in}
\]

acquire lock;

if mem p env then error("Undefined behaviour")
else add p env;

\[ p \overset{HL}{=} v ; \]

release lock;

\[ v \]

the \textit{atomicity} of updates is enforced by using a global \textit{lock}
the execution of function call is atomic from the caller’s point of view:

$$f() + g()$$

all the instructions in one of the function are executed prior to the execution of the other function

consequently, each call should be compiled using the lock:

```plaintext
[f(e1)] \overset{\text{def}}{=} \begin{align*}
\text{let } \nu &= [e1] \text{ in} \\
\text{acquire } &\text{lock;} \\
\text{let } r &= f \nu \text{ in} \\
\text{release } &\text{lock}; \\
r
\end{align*}
```
the execution of function call is atomic from the caller’s point of view:

\[ f() + g() \]

but the function \( f \) might call some other function (or call itself) consequently, each call should be compiled, using a new lock:

\[
[f(e1)] \overset{\text{def}}{=} \text{fun lock} \Rightarrow \\
\quad \text{let } v = [e1] \text{ in} \\
\quad \text{acquire lock; } \\
\quad \text{let } lock' = \text{newmutex()} \text{ in} \\
\quad \text{let } r = f \; v \; lock' \text{ in} \\
\quad \text{release lock; } r
\]
Vcg rule for add

\[
\text{vcg}(m,e_1 + e_2, \mathcal{K}) \overset{\text{def}}{=} \\
\text{match forward}(m, e_1) \text{ with} \\
| \text{Some } (v_1, m_o, m_f) \rightarrow \text{vcg}(m_f, e_2, \lambda m' v_2. \mathcal{K} (m' \sqcup m^{\circ}) (v_1 + v_2)) \\
| \text{None } \rightarrow \\
\text{match forward}(m, e_2) \text{ with} \\
| \text{Some } (v_2, m_o, m_f) \rightarrow \text{vcg}(m_f, e_1, \lambda m' v_1. \mathcal{K} (m' \sqcup m^{\circ}) (v_1 + v_2)) \\
| \text{None } \rightarrow \llbracket m \rrbracket \ast \text{wp}(e_1 + e_2) \{ \lambda v, \exists m'. \llbracket m' \rrbracket \ast \mathcal{K} m' v \} 
\]
Fractional lockable permissions enforce the sequence point restriction:

\[
\{P\} \ e \left\{ l. \exists w. q. \ 1 \xrightarrow{q} U w \ast ( \ 1 \xrightarrow{q} U w \ast \Phi w) \right\} \\
\{P\} (*e) \{\Phi\}
\]

\[
\{P_1\} \ e_1 \{\Psi_1\} \quad \{P_2\} \ e_2 \{\Psi_2\} \quad (\forall l. \Psi_1 l \ast \Psi_2 w \ast \exists v. \ 1 \xrightarrow{u} v \ast ( \ 1 \xrightarrow{l} w \ast \Phi w)) \\
\{P_1 \ast P_2\} (e_1 = e_2) \{\Phi\}
\]

⇒ Allows to prove \(\{l \xrightarrow{q} U v\} \ast l + \ast l \{\lambda w. (w = v + v) \ast l \xrightarrow{q} U v\}\)

⇒ Rules out programs with undefined behavior like \(*l = (*l = 3)\)
Fractional lockable permissions enforce the sequence point restriction:

\[
\{ P \} \ e \left\{ \begin{array}{l}
1. \exists w q. \ 1 \xrightarrow{q} U w \ * (1 \xrightarrow{q} U w \ \rightarrow \ \Phi w) \\
\end{array} \right\}
\]

\[
\{ P \} (* e) \{ \Phi \}
\]

\[
\begin{array}{l}
\{ P_1 \} \ e_1 \{ \Psi_1 \} \quad \{ P_2 \} \ e_2 \{ \Psi_2 \} \\
(\forall w. \ \Psi_1 \ l * \ \Psi_2 \ w \ \rightarrow \ \exists v. \ l \xrightarrow{1} U v \ * (1 \xrightarrow{1} L w \ \rightarrow \ \Phi w))
\end{array}
\]

\[
\{ P_1 * P_2 \} (e_1 = e_2) \{ \Phi \}
\]

\[\Rightarrow\] Allows to prove \( \{ 1 \xrightarrow{q} U v \} * l + * l \{ \lambda w. (w = v + v) * l \xrightarrow{q} U v \} \)

\[\Rightarrow\] Rules out programs with undefined behavior like \( * l = (* l = 3) \)
Fractional lockable permissions enforce the sequence point restriction:

\[
\{P\} \ e \ \left\{ l. \exists w. q. \ l \xrightarrow{q} U \ w \right\} \ (l \xrightarrow{q} U \ w \rightarrow \Phi \ w) \\
\{P\} (\ast e) \ \{\Phi\}
\]

\[
\{P_1\} \ e_1 \ \{\Psi_1\} \ \{P_2\} \ e_2 \ \{\Psi_2\} \ (\forall w. \ \Psi_1 \ l \ast \Psi_2 \ w \ast \ \exists v. \ l \xrightarrow{U} v \ast (l \xrightarrow{L} w \ast \Phi \ w)) \\
\{P_1 \ast P_2\} \ (e_1 = e_2) \ \{\Phi\}
\]

⇒ Allows to prove \(\{ l \xrightarrow{q} U \ v\} \ast l + \ast l \ \{\lambda w. (w = v + v) \ast l \xrightarrow{q} U \ v\}\)

⇒ Rules out programs with undefined behavior like \(\ast l \ast l \ (\ast l = 3)\)
Fractional lockable permissions enforce the sequence point restriction:

\[
\{ P \} \ e \left\{ \begin{array}{l}
1. \exists w. q. \ 1 \xrightarrow{q} U w \quad \ast \ (1 \xrightarrow{q} U w \quad \ast \ \Phi w)
\end{array} \right\}
\]

\[
\{ P \} (\ast e) \{ \Phi \}
\]

\[
\{ P_1 \} \ e_1 \{ \Psi_1 \} \quad \{ P_2 \} \ e_2 \{ \Psi_2 \} \quad (\forall l. \ \Psi_1 \ l \ast \Psi_2 \ w \ast \exists v. \ l \xrightarrow{1} U v \ast (1 \xrightarrow{1} L w \ast \Phi w))
\]

\[
\{ P_1 \ast P_2 \} (e_1 = e_2) \{ \Phi \}
\]

⇒ Allows to prove \( \{ 1 \xrightarrow{q} U v \} \ast l + \ast l \{ \lambda w. (w = v + v) \ast 1 \xrightarrow{q} U v \} \)

⇒ Rules out programs with undefined behavior like \( \ast l = (\ast l = 3) \)
Unlocking modality

**remark:** we want to access locked pointers later again

\[ *1 = 4 ; *1 \]

we use the **unlocking modality** \( \mathbb{U} \) that unlocks all locked locations at the sequence point:

\[
\frac{\text{wp } e_1 \{ \ldots \mathbb{U}(\text{wp } e_2 \{ \Phi \}) \}}{\text{wp } (e_1 ; e_2) \{ \Phi \}} \quad \frac{1 \overset{q}{\mapsto}_L v}{\mathbb{U}(1 \overset{q}{\mapsto}_U v)} \quad \frac{P \rightsquigarrow Q}{\mathbb{U}P \rightsquigarrow \mathbb{U}Q}
\]
Example

\[ l \mapsto v_1 \quad k \mapsto v_2 \quad r \mapsto v_3 \]

\[ *l = *k + 10 \]

postcondition: \[ \top \]
frame: \[ \top \]
$$1 \mapsto v_1 \cdot k \mapsto v_2 \cdot r \mapsto v_3$$

$$\ast 1 = v_2 + 10$$

**postcondition:**

$$k \mapsto v_2$$

**frame:**

$$k \overset{0.5}{\mapsto} v_2$$
Example

\[ l \mapsto v_1 \ * \ k \mapsto v_2 \ * \ r \mapsto v_3 \]

\[ *l = v_2 + 10 \]

**postcondition:** \[ k \rightarrow v_2 \]

**frame:** \[ k \rightarrow v_2 \]
Example

$1 \mapsto v_1 \cdot k \mapsto v_2 \cdot r \mapsto v_3$

$v_2 + 10$

postcondition: $k \overset{0.5}{\mapsto} v_2 \cdot l \overset{L}{\mapsto} (v_2 + 10)$

frame: $k \overset{0.5}{\mapsto} v_2$
Example

\[ v_1 \cdot k \cdot v_2 \cdot r \cdot v_3 \]

\[ v_2 + 10 \]

**postcondition:** \[ k^{0.5} \cdot v_2 \cdot 1 \mapsto_L (v_2 + 10) \]

**frame:** \[ k^{0.5} \cdot v_2 \cdot r \mapsto v_3 \]
l \mapsto v1 \cdot k \mapsto v2 \cdot r \mapsto v3

\text{postcondition: } \top
\text{frame: } \top

(\ast l = \ast k + 10) + (\ast r = \ast k + 10)
After executing the LHS

\[ l \mapsto v_1 \quad * \quad k \mapsto v_2 \quad * \quad r \mapsto v_3 \]

\[
(v_2 + 10) + (*r = *k + 10)
\]

**postcondition:**

\[ k \overset{0.5}{\mapsto} v_2 \quad * \quad l \overset{L}{\mapsto} (v_2 + 10) \]

**frame:**

\[ k \overset{0.5}{\mapsto} v_2 \quad * \quad r \mapsto v_3 \]
Before executing the RHS

$1 \rightarrow v1 \ast k \xrightarrow{0.5} v2 \ast r \rightarrow v3$

$\ (v2 + 10) + (\ast r = \ast k + 10)$

**postcondition:** $k \xrightarrow{0.5} v2 \ast l \rightarrow_L (v2 + 10)$

**frame:**

$k \xrightarrow{0.5} v2 \ast r \rightarrow v3$
Executing the RHS

\begin{align*}
1 & \mapsto v_1 \quad \frac{1}{2} & \mapsto v_2 \quad r & \mapsto v_3 \\
(v_2 + 10) + (*r = v_2 + 10)
\end{align*}

**postcondition:**

\begin{align*}
\frac{3}{4} & \mapsto v_2 \quad l & \mapsto (v_2 + 10) \\
\frac{1}{4} & \mapsto v_2 \quad r & \mapsto v_3
\end{align*}
Final result

\[
\begin{align*}
1 & \mapsto v1 \ast k \overset{0.5}{\mapsto} v2 \ast r \mapsto v3 \\
\text{(postcondition:)} & \quad k \overset{3/4}{\mapsto} v2 \ast 1 \mapsto_L (v2 + 10) \ast r \mapsto_L (v2 + 10) \\
\text{(frame:)} & \quad k \overset{1/4}{\mapsto} v2 \ast r \mapsto v3
\end{align*}
\]