Semi-Automated Reasoning About Non-Determinism in C Expressions

Léon Gondelman[1]

joint work with **Dan Frumin**[1] and **Robbert Krebbers**[2]

8 April, 2019 @ ESOP, Prague

[1] Radboud University Nijmegen [2] Delft University of Technology

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"

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

compiler	outcome	warnings
compcert	4, 7	no
clang	4, 7	yes
gcc-4.9	4, 8	no

Undefined behavior

```
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.

$$\{P\}\ \mathsf{e}\ \{Q\} \quad\Longrightarrow\quad \begin{array}{l} \quad \text{functional correctness} \\ \wedge\ \text{no sequence point violations} \\ \wedge\ \text{no other undefined behavior} \end{array}$$

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.

$$\{\mathtt{r} \mapsto \mathtt{i} * \mathtt{c} \mapsto \mathtt{j}\}$$

$$\texttt{*r} = \texttt{*r} * (\texttt{++}(\texttt{*c}));$$

$$\{v.\, v = \mathtt{i}\cdot(\mathtt{j}+1) \land \mathtt{r} \mapsto \mathtt{i}\cdot(\mathtt{j}+1) * \mathtt{c} \mapsto \mathtt{j} + 1\}$$

(Krebbers POPL'14)

Observation: view non-determinism through concurrency

Idea: use concurrent separation logic

$$\frac{\{P_1\}\,\mathtt{e}_1\,\{\Psi_1\} \qquad \{P_2\}\,\mathtt{e}_2\,\{\Psi_2\} \qquad \forall \mathtt{v}_1\,\mathtt{v}_2.\, \Psi_1\,\mathtt{v}_1 * \Psi_2\,\mathtt{v}_2 \vdash \Phi(\mathtt{w}_1\,[\![\circledcirc]\!]\,\mathtt{w}_2)}{\{P_1 * P_2\}\,\mathtt{e}_1 \circledcirc \mathtt{e}_2\,\{\Phi\}}$$

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

(Krebbers POPL'14)

Observation: view non-determinism through concurrency

Idea: use concurrent separation logic

$$\frac{\{P_1\}\ \mathtt{e}_1\ \{\Psi_1\} \qquad \{P_2\}\ \mathtt{e}_2\ \{\Psi_2\} \qquad \forall \mathtt{v}_1\, \mathtt{v}_2.\ \Psi_1\ \mathtt{v}_1 * \Psi_2\ \mathtt{v}_2 \vdash \Phi(\mathtt{w}_1\ [\![\circledcirc]\!]\ \mathtt{w}_2)}{\{P_1 * P_2\}\ \mathtt{e}_1 \circledcirc \mathtt{e}_2\ \{\Phi\}}$$

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

(Krebbers POPL'14)

Observation: view non-determinism through concurrency
Idea: use concurrent separation logic

$$\frac{\boxed{\{P_1\}\ \mathtt{e}_1\ \{\Psi_1\}} \qquad \boxed{\{P_2\}\ \mathtt{e}_2\ \{\Psi_2\}} \qquad \forall \mathtt{v}_1\, \mathtt{v}_2.\, \Psi_1\ \mathtt{v}_1 * \Psi_2\ \mathtt{v}_2 \vdash \Phi(\mathtt{w}_1\ [\![\circledcirc]\!]\ \mathtt{w}_2)}{\{P_1 * P_2\}\ \mathtt{e}_1 \circledcirc \mathtt{e}_2\ \{\Phi\}}$$

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.

$$\frac{\{P_1\}\ \mathtt{e}_1\ \{\Psi_1\} \qquad \{P_2\}\ \mathtt{e}_2\ \{\Psi_2\} \qquad \forall \mathtt{v}_1\, \mathtt{v}_2.\ \Psi_1\ \mathtt{v}_1 * \Psi_2\ \mathtt{v}_2 \vdash \Phi(\mathtt{w}_1\ [\![\circledcirc]\!]\ \mathtt{w}_2)}{\{P_1 * P_2\}\ \mathtt{e}_1 \circledcirc \mathtt{e}_2\ \{\Phi\}}$$

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.

$$\frac{\{P_1\}\,\mathtt{e}_1\,\{\Psi_1\} \qquad \{P_2\}\,\mathtt{e}_2\,\{\Psi_2\} \qquad \forall \mathtt{v}_1\,\mathtt{v}_2.\ \Psi_1\ \mathtt{v}_1 * \Psi_2\ \mathtt{v}_2 \vdash \Phi(\mathtt{w}_1\ [\![\circledcirc]\!]\ \mathtt{w}_2)}{\{P_1 * P_2\}\,\mathtt{e}_1 \circledcirc \mathtt{e}_2\,\{\Phi\}}$$

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.

$$\frac{\{P_1\}\ \mathtt{e}_1\ \{\Psi_1\} \qquad \{P_2\}\ \mathtt{e}_2\ \{\Psi_2\} \qquad \forall \mathtt{v}_1\ \mathtt{v}_2.\ \Psi_1\ \mathtt{v}_1 * \Psi_2\ \mathtt{v}_2 \vdash \Phi(\mathtt{w}_1\ [\![\circledcirc]\!]\ \mathtt{w}_2)}{\{P_1 * P_2\}\ \mathtt{e}_1 \circledcirc \mathtt{e}_2\ \{\Phi\}}$$

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.

$$\frac{\{P_1\}\,\mathtt{e}_1\,\{\Psi_1\} \qquad \{P_2\}\,\mathtt{e}_2\,\{\Psi_2\} \qquad \forall \mathtt{v}_1\,\mathtt{v}_2.\ \Psi_1\ \mathtt{v}_1 * \Psi_2\ \mathtt{v}_2 \vdash \Phi(\mathtt{w}_1\ [\![\circledcirc]\!]\ \mathtt{w}_2)}{\{P_1 * P_2\}\,\mathtt{e}_1 \circledcirc \mathtt{e}_2\,\{\Phi\}}$$

$\Longrightarrow$ 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

$$\{P\}\ \mathsf{e}\ \{Q\} \quad \triangleq \quad P \vdash \mathsf{wp}\ \mathsf{e}\ \{Q\}$$

$$\mathsf{wp}\ \mathsf{e}\ \{Q\}$$

Contribution 1:

A redesign of Krebbers's logic using

a **weakest precondition calculus.**

$\Rightarrow$ makes automation possible

Contributions

$\{P\}\ \mathsf{e}\ \{Q\} \quad \triangleq \quad P \vdash \mathsf{wp}\ \mathsf{e}\ \{Q\}$

$\mathsf{wp}\ \mathsf{e}\ \{Q\}$

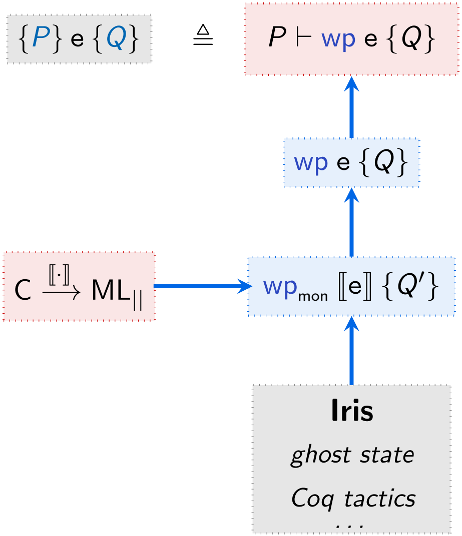
$\mathsf{C} \xrightarrow{[\![\cdot]\!]} \mathsf{ML}_{||}$

Contribution 2:

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

$\Rightarrow$ makes the semantics declarative

Contributions



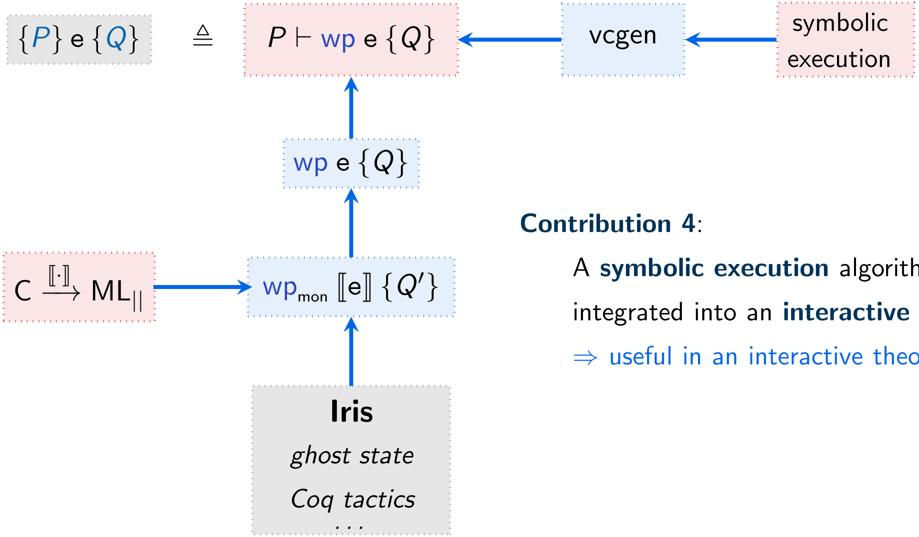
Contribution 3:

A **layered model** of our program logic

built on top of the Iris framework

$\Rightarrow$ modular and expresive logic, Coq tactics

Contributions



Contribution 4:

A **symbolic execution** algorithm

integrated into an **interactive vcgen**

$\Rightarrow$ useful in an interactive theorem prover

Contributions

$\{P\}\ \mathsf{e}\ \{Q\} \triangleq P \vdash \mathsf{wp}\ \mathsf{e}\ \{Q\}$

vcgen

symbolic execution

$\mathsf{wp}\ \mathsf{e}\ \{Q\}$

$\mathsf{C} \xrightarrow{[\![\cdot]\!]} \mathsf{ML}_{||}$

$\mathsf{wp}_{\mathsf{mon}}\ [\![\mathsf{e}]\!]\ \{Q'\}$

Iris

ghost state

Coq tactics

...

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:

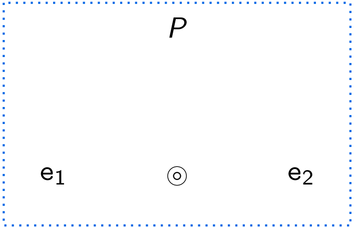
input		**output**
precondition		value
program	$\dashrightarrow$	(strongest) postcondition
		frame = resources not used

Key idea

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

input		**output**
$\mathtt{r} \mapsto \mathtt{i} * \mathtt{c} \mapsto \mathtt{j} * \mathtt{d} \mapsto \mathtt{k}$		$\mathtt{i}\cdot(\mathtt{j}+1)$
`*r = *r * (++(*c));`	$--\!\!\rightarrow$	$\mathtt{r} \mapsto \mathtt{i}\cdot(\mathtt{j}+1) * \mathtt{c} \mapsto \mathtt{j} + 1$
		$\mathtt{d} \mapsto \mathtt{k}$

Symbolic execution algorithm



Symbolic execution algorithm

$$P$$

$$\boxed{\begin{matrix} P \\ \mathsf{e}_1 \end{matrix}} \qquad \circledcirc \qquad \mathsf{e}_2$$

Symbolic execution algorithm

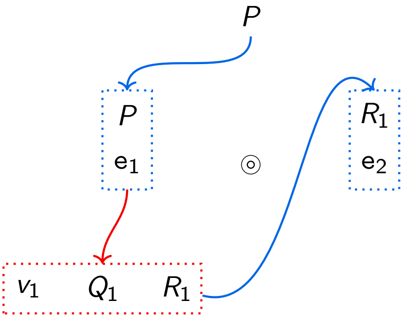
P

P

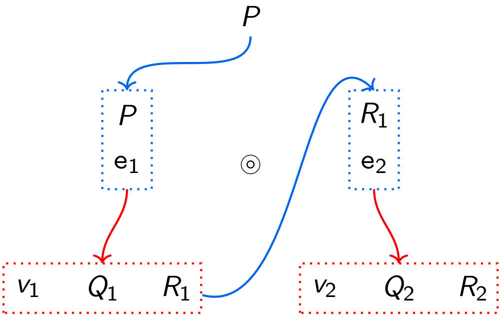
$\mathtt{e}_1 \quad \circledcirc \quad \mathtt{e}_2$

$v_1 \quad Q_1 \quad R_1$

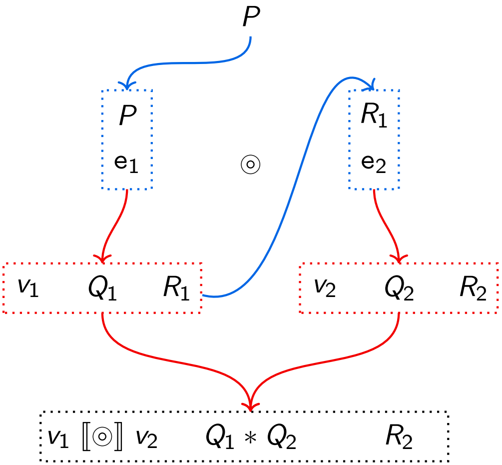
Symbolic execution algorithm



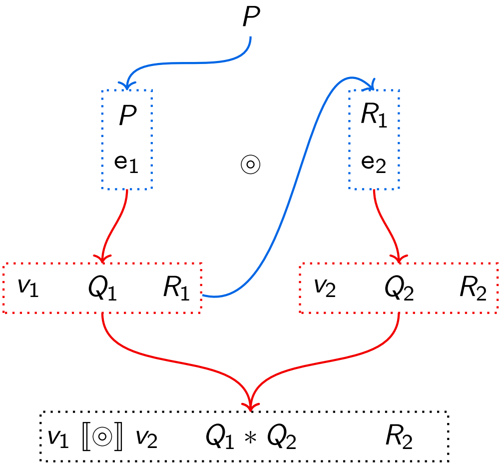
Symbolic execution algorithm



Symbolic execution algorithm



Symbolic execution algorithm



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

$$\frac{(P, \mathtt{e}) \xrightarrow{\textit{symb. exec.}} (\mathtt{w}, Q, R)}{P \vdash \mathsf{wp}\ \mathtt{e}\ \{\mathtt{v}.\ \mathtt{v} = \mathtt{w} * Q\} * R}$$

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

$$\frac{(P, \mathtt{e}_1) \xrightarrow{\textit{symb. exec.}} (\mathtt{w}_1, Q, R) \qquad R \vdash \mathsf{wp}\ \mathtt{e}_2\ \{\mathtt{w}_2.\ Q \twoheadrightarrow \Phi\ (\mathtt{w}_1\ [\![\circledcirc]\!]\ \mathtt{w}_2)\}}{P \vdash \mathsf{wp}\ (\mathtt{e}_1 \circledcirc \mathtt{e}_2)\ \{\Phi\}}$$

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

$$\frac{P_1 \vdash \mathsf{wp}\ \mathtt{e}_1\ \{\Psi_1\} \quad P_2 \vdash \mathsf{wp}\ \mathtt{e}_2\ \{\Psi_2\} \quad (\forall \mathtt{w}_1 \mathtt{w}_2.\ \Psi_1\ \mathtt{w}_1 * \Psi_2\ \mathtt{w}_2 \twoheadrightarrow \Phi(\mathtt{w}_1\ [\![\circledcirc]\!]\ \mathtt{w}_2))}{P_1 * P_2 \vdash \mathsf{wp}\ (\mathtt{e}_1 \circledcirc \mathtt{e}_2)\ \{\Phi\}}$$

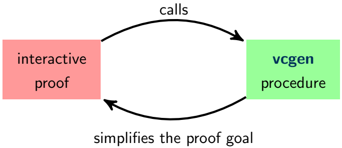
Symbolic execution algorithm that computes the frame allows to apply the program logic rules in an algorithmic manner:

$$\frac{(P, \mathtt{e}_1) \xrightarrow{\textit{symb. exec.}} (\mathtt{w}_1, Q, R) \qquad R \vdash \mathsf{wp}\ \mathtt{e}_2\ \{\mathtt{w}_2.\ Q \mathrel{-\!\!*} \Phi\ (\mathtt{w}_1\ [\![\circledcirc]\!]\ \mathtt{w}_2)\}}{P \vdash \mathsf{wp}\ (\mathtt{e}_1 \circledcirc \mathtt{e}_2)\ \{\Phi\}}$$

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.

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

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

$\exists k \leq n.$

Hr: $r \mapsto \mathrm{fact}(k)$

Hc: $c \mapsto k$

Proof.

generalize Hr Hc.

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

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

Hr: $r \mapsto \mathsf{fact}(k)$

Hc: $c \mapsto k$

IH: $\forall k. \triangleright$

$r \mapsto \mathsf{fact}(k) * c \mapsto k * k \leq n \mathrel{-\!\!*}$

$\mathsf{wp}\ (\texttt{while}(..)\{\ldots\})$

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

Proof.

generalize Hr Hc. induction.

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

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

Hr: $r \mapsto \mathsf{fact}(k)$

Hc: $c \mapsto k$

IH: $\forall k.$

$r \mapsto \mathsf{fact}(k) * c \mapsto k * k \leq n \twoheadrightarrow$

$\quad \mathsf{wp}\ (\texttt{while}(..)\{\ldots\})$

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

Proof.

generalize Hr Hc. induction. while_spec.

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

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

Hr: $r \mapsto \mathsf{fact}(k)$

Hc: $c \mapsto k$

IH: $\forall k.$

$r \mapsto \mathsf{fact}(k) * c \mapsto k * k \leq n -\!\!*$

$\mathsf{wp}\ (\texttt{while}(..)\{\ldots\})$

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

Proof.

generalize Hr Hc. induction. while_spec.

vcgen.

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

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

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

Hc: $c \mapsto k$

IH: $\forall k.$

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

$\text{wp}\ (\mathtt{while}(..)\{\ldots\})$

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

Proof.

generalize Hr Hc. induction. while_spec.

vcgen.

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

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

Hr: $r \mapsto \mathrm{fact}(k)$

Hc: $c \mapsto k$

Hk: $k < n$

IH: $\forall k.$
$r \mapsto \mathrm{fact}(k) * c \mapsto k * k \leq n \mathrel{-\!\!*}$
$\mathrm{wp}\ (\texttt{while}(..)\{\ldots\})$
$\{r \mapsto \mathrm{fact}(n) * c \mapsto n\}$

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

Goal [1/2].

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

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

Hr: $r \mapsto \mathsf{fact}(k)$

Hc: $c \mapsto k$

Hk: $k < n$

IH: $\forall k.$

$r \mapsto \mathsf{fact}(k) * c \mapsto k * k \leq n \mathrel{-\!\!*}$

$\mathsf{wp}\ (\texttt{while}(..)\{\ldots\})$

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

Proof.

generalize Hr Hc. induction. while_spec.

vcgen.

- **vcgen.**

Goal [1/2].

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

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

Hr: $r \mapsto \text{fact}(k) \cdot (k+1)$

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

Hk: $k < n$

IH: $\forall k.$

$$r \mapsto \text{fact}(k) * c \mapsto k * k \leq n -\!\!* \quad \text{wp}\,(\texttt{while}(..)\{\ldots\}) \quad \{r \mapsto \text{fact}(n) * c \mapsto n\}$$

Proof.

generalize Hr Hc. induction. while_spec.

vcgen.

- **vcgen.**

Goal [1/2].

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

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

Hr: $r \mapsto \text{fact}(k) \cdot (k+1)$

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

Hk: $k < n$

IH: $\forall k.$

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

$\text{wp}\ (\texttt{while}(..)\{\ldots\})$

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

Proof.

generalize Hr Hc. induction. while_spec.

vcgen.

- **vcgen.** apply IH.

Goal [1/2].

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

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

Hr: $r \mapsto \mathrm{fact}(k)$

Hc: $c \mapsto k$

Hk: $k = n$

IH: $\forall k.$

$$r \mapsto \mathrm{fact}(k) * c \mapsto k * k \leq n \twoheadrightarrow$$
$$\mathrm{wp}\ (\texttt{while}(..)\{\ldots\})$$
$$\{r \mapsto \mathrm{fact}(n) * c \mapsto n\}$$

Proof.

generalize Hr Hc. induction. while_spec.

vcgen.

- **vcgen.** apply IH.
- eauto.

Qed.

Goal [2/2].

$$()$$

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

Implementation (1/2)

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

$$\text{forward} : (\text{sheap} \times \text{expr}) \to (\text{val} \times \text{sheap} \times \text{sheap})$$

satisfying the following specification:

$$\frac{\text{forward}(m, \mathtt{e}) = (\mathtt{w}, m_1^o, m_1)}{[\![m]\!] \vdash \text{wp}\ \mathtt{e}\ \{\mathtt{v}.\ \mathtt{v} = \mathtt{w} * [\![m_1^o]\!]\} * [\![m_1]\!]}$$

Implementation (1/2)

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

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

Future work:

- lift the restriction for the precondition to handle pure facts
- enable interaction with external decision procedures

Implementation (2/2)

The vcgen is implemented as a total function

$$\mathsf{vcg} : (\mathsf{sheap} \times \mathsf{expr} \times (\mathsf{sheap} \to \mathsf{val} \to \mathsf{Prop})) \to \mathsf{Prop}$$

satisfying the following specification:

$$\frac{P' \vdash \mathsf{vcg}(m, \mathsf{e}, \lambda\, m'\, \mathtt{v}.\ [\![m']\!] \mathrel{-\!\!*} \Phi\ \mathtt{v})}{P' * [\![m]\!] \vdash \mathsf{wp}\ \mathsf{e}\ \{\Phi\}}$$

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 !