Semi-Automated Reasoning About Non-Determinism in C Expressions

Léon Gondelman[1]

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

15 April, 2019 @ Gallium, Paris

[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

```c
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 goal

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.

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

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.

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

Previous work:

Krebbers' program logic (POPL'14)

(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

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

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 ⇒ no sequence point violations

(Krebbers POPL'14)

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

$$(\texttt{*l} = \texttt{*k} + 10) + (\texttt{*r} = \texttt{*k} + 10) \quad \checkmark$$

$\Longrightarrow$ Use fractional permissions: $\texttt{k} \xmapsto{q_1+q_2} \texttt{v} \dashv\vdash \texttt{k} \xmapsto{q_1} \texttt{v} * \texttt{k} \xmapsto{q_2} \texttt{v}$

(Krebbers POPL'14)

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

$$\texttt{*l} = (\texttt{*l} = 3) \quad ✗$$

$\Longrightarrow$ Decorate permissions with a lockable flag $\xi \in \{L, U\}$

(Krebbers POPL'14)

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

$$\frac{\{P_1\}\,\mathtt{e}_1\,\{\Psi_1\} \quad \{P_2\}\,\mathtt{e}_2\,\{\Psi_2\} \quad (\forall \mathtt{l}\,\mathtt{w}.\,\Psi_1\,\mathtt{l} * \Psi_2\,\mathtt{w} \mathrel{-\!\!*} \exists \mathtt{v}.\,\mathtt{l} \overset{1}{\mapsto}_U \mathtt{v} * (\mathtt{l} \overset{1}{\mapsto}_L \mathtt{w} \mathrel{-\!\!*} \Phi\,\mathtt{w}))}{\{P_1 * P_2\}\,(\mathtt{e}_1 = \mathtt{e}_2)\,\{\Phi\}}$$

$\Longrightarrow$ Decorate permissions with a lockable flag $\xi \in \{L, U\}$

(Krebbers POPL'14)

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**

$$\texttt{*l} = \texttt{4}\,;\texttt{*l} \qquad\qquad \texttt{f()} + \texttt{g()}$$

$\Longrightarrow$ Define **unlocking modality** $\mathbb{U}$ such that $\texttt{l} \overset{q}{\mapsto}_L \texttt{v} \vdash \mathbb{U}(\texttt{l} \overset{q}{\mapsto}_U \texttt{v})$

(Krebbers POPL'14)

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**

$$\frac{\{P\}\, \mathsf{e}_1\, \{\mathbb{U}(\Psi_1)\} \quad \{\Psi_1\}\, \mathsf{e}_2\, \{\Phi\}}{\{P\}\, (\mathsf{e}_1\, ;\, \mathsf{e}_2)\, \{\Phi\}}$$

$\Longrightarrow$ Define **unlocking modality** $\mathbb{U}$ such that $\mathtt{l} \overset{q}{\mapsto}_L \mathtt{v} \vdash \mathbb{U}(\mathtt{l} \overset{q}{\mapsto}_U \mathtt{v})$

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.

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

Limitations of Krebbers' 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' 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' 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 work:

Redesign Krebbers's program logic and turn it into a semi-automated procedure

Contributions

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

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

Contribution 1:

A redesign of Krebbers's logic using

a **weakest precondition calculus.**

⇒ decouples the program from the precondition

⇒ makes automation possible

Contributions

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

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

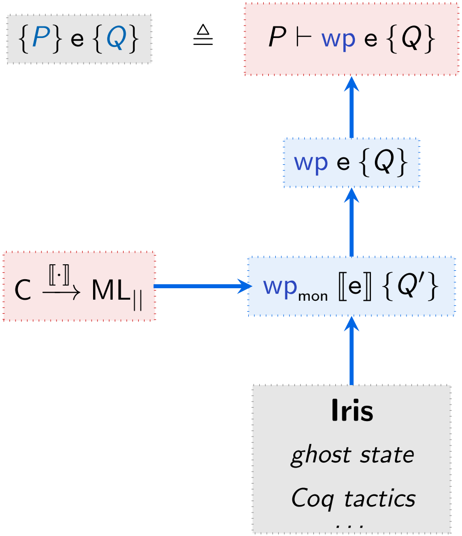
$\mathsf{C} \xrightarrow{[\![\cdot]\!]} \mathsf{ML}_{||}$

Contribution 2:

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

$\Rightarrow$ makes the semantics declarative

$\Rightarrow$ *reader monad* $M(A) \triangleq \texttt{mset}\ Ptr \to \texttt{mutex} \to A$



Contribution 3:

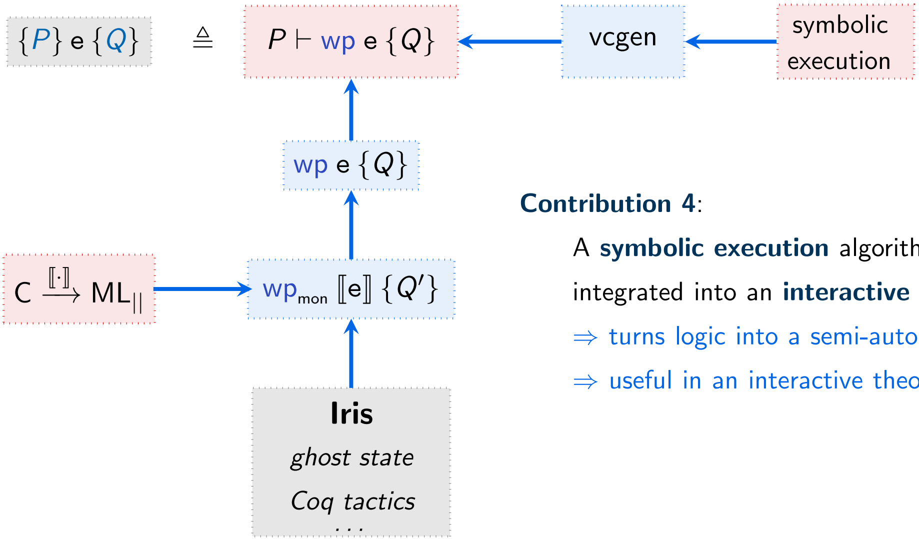
A **layered model** of our program logic

built on top of the Iris framework

⇒ makes logic more modular and expresive

⇒ support from Iris Proof Mode and Coq tactics

Contributions



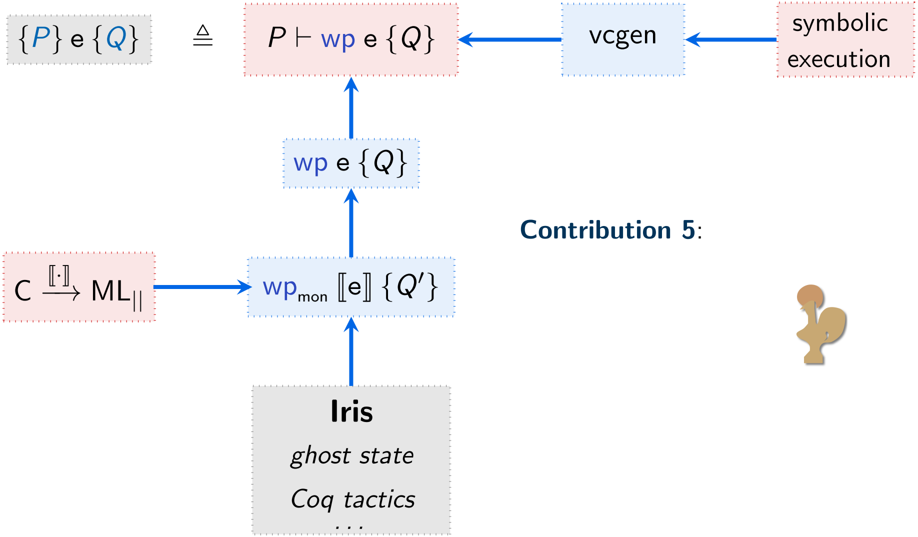
Contribution 4:

A **symbolic execution** algorithm integrated into an **interactive vcgen**

$\Rightarrow$ turns logic into a semi-automated procedure

$\Rightarrow$ useful in an interactive theorem prover

Contributions



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:

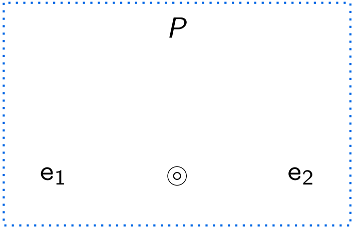
input		**output**
precondition		value
program	$\dashrightarrow$	(strongest) postcondition
		frame = resources not used

Key idea

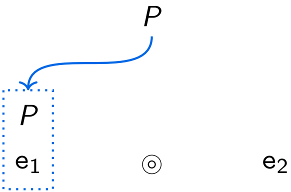
Turn the program logic into an algorithm procedure using a novel symbolic execution algorithm:

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));`	$\dashrightarrow$	$\mathtt{r} \mapsto \mathtt{i} \cdot (\mathtt{j}+1) * \mathtt{c} \mapsto \mathtt{j} + 1$
		$\mathtt{d} \mapsto \mathtt{k}$

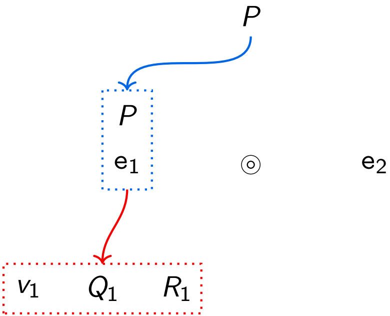
Symbolic execution algorithm



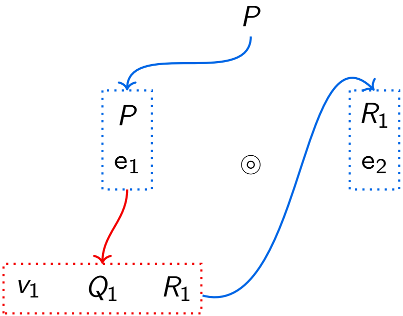
Symbolic execution algorithm



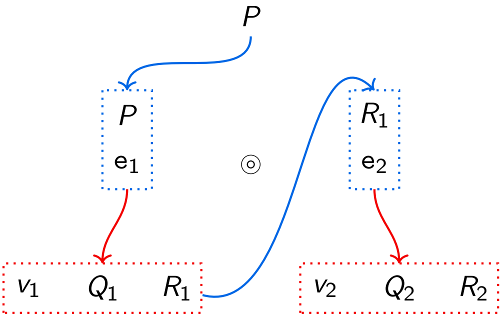
Symbolic execution algorithm



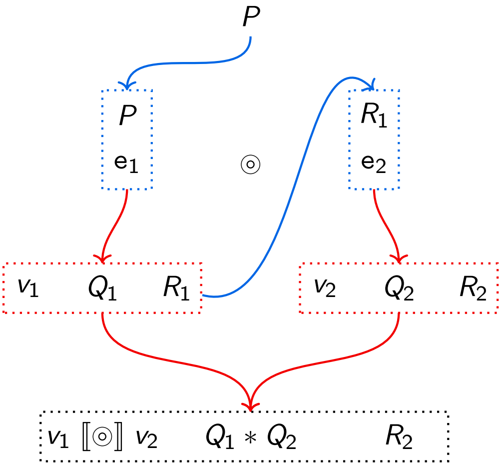
Symbolic execution algorithm



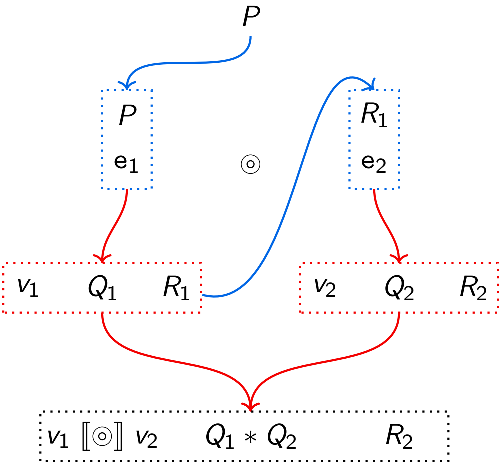
Symbolic execution algorithm



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:

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

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

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

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

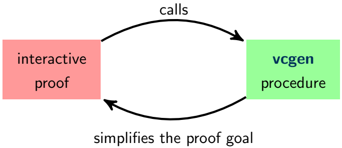
$$\frac{(P, \mathtt{e}_1) \xrightarrow{\textit{symb. exec.}} (\mathtt{w}_1, Q, R) \qquad R \vdash \mathsf{wp}\ \mathtt{e}_2\ \{\mathtt{w}_2.\ Q \mathrel{-\!\!*} \Phi\ (\mathtt{w}_1\ [\![\circledcirc]\!]\ \mathtt{w}_2)\}}{P \vdash \mathsf{wp}\ (\mathtt{e}_1 \circledcirc \mathtt{e}_2)\ \{\Phi\}}$$

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, . . .)

Vcgen

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

$\exists k \leq n.$

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

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: $\forall k.$

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

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

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

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: $\forall k.$

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

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

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

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

Hc: $c \mapsto k$

IH: $\forall k.$

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

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

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

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

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.

Goal [1/2].

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

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

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

Hc: $c \mapsto k$

Hk: $k < n$

IH: $\forall k.$

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

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

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

Proof.

generalize Hr Hc. induction. while_spec.

vcgen.

- **vcgen.**

Goal [1/2].

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

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

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

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

Hk: $k < n$

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

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

Goal [1/2].

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

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

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

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

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.** apply IH.

Goal [1/2].

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

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

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

Hc: $c \mapsto k$

Hk: $k = n$

IH: $\forall k.$

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

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

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

Proof.

generalize Hr Hc. induction. while_spec.

vcgen.

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

Qed.

Goal [2/2].

()

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

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:

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

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:

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

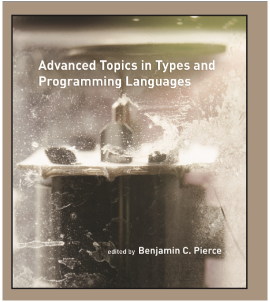
One piece of related work

$$\frac{\Gamma_1 \vdash \mathtt{t_1} : \mathtt{q}\ \mathtt{T_{11}} \rightarrow \mathtt{T_{12}} \qquad \Gamma_2 \vdash \mathtt{t_2} : \mathtt{T_{11}}}{\Gamma_1 \circ \Gamma_2 \vdash \mathtt{t_1}\ \mathtt{t_2} : \mathtt{T_{12}}} \quad \text{(T-APP)}$$

Non-deterministic typing rule

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

Algorithmic type checking



(Ch.1. Substructural Type Systems)

"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)

$$[\![\texttt{e1 = e2}]\!] \stackrel{def}{=} \textbf{let}\ (p, v) = [\![\texttt{e1}]\!] \mid\mid_{\mathsf{HL}} [\![\texttt{e2}]\!]\ \textbf{in}$$
$$p :=_{\mathsf{HL}} v \ ; \ v$$

$$[\![\texttt{e1 + e2}]\!] \stackrel{def}{=} \textbf{let}\ (v_1, v_2) = [\![\texttt{e1}]\!] \mid\mid_{\mathsf{HL}} [\![\texttt{e2}]\!]\ \textbf{in}$$
$$v_1 +_{\mathsf{HL}} v_2$$

---

the **non-determinism** is embodied by using parallel composition $\mid\mid_{\mathsf{HL}}$

$$[\![\texttt{e1 = e2}]\!] \stackrel{def}{=} \texttt{let}\ (p, v) = [\![\texttt{e1}]\!] \mid\mid_{\text{HL}} [\![\texttt{e2}]\!]\ \texttt{in}$$
$$\texttt{if}\ \text{mem}\ p\ \text{env}\ \texttt{then}\ \texttt{error}(\texttt{"Undefined behaviour"})$$
$$\texttt{else}\ \text{add}\ p\ \text{env}\ ;$$
$$p :=_{\text{HL}} v\ ;\ v$$

$$[\![;]\!] \stackrel{def}{=} \text{env} :=_{\text{HL}} \varnothing$$

the **sequence point conditions** are checked using a set of pointers env

translation scheme (3/4)

$$
\begin{aligned}
[\![\texttt{e1 = e2}]\!] \stackrel{def}{=}\ & \texttt{let}\ (p, v) = [\![\texttt{e1}]\!] \,||_{\mathsf{HL}}\, [\![\texttt{e2}]\!]\ \texttt{in} \\
& \texttt{acquire}\ \mathsf{lock}; \\
& \quad \texttt{if}\ \mathsf{mem}\ p\ \mathsf{env}\ \ \texttt{then}\ \texttt{error}(\texttt{"Undefined behaviour"}) \\
& \quad \texttt{else}\ \mathsf{add}\ p\ \mathsf{env}\ ; \\
& \quad p :=_{\mathsf{HL}} v\ ; \\
& \texttt{release}\ \mathsf{lock}\ ; \\
& v
\end{aligned}
$$

---

the **atomicity** of updates is enforced by using a global 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:

$$
\begin{aligned}
[\![\texttt{f(e1)}]\!] \stackrel{def}{=}\ & \texttt{let}\ v = [\![\texttt{e1}]\!]\ \texttt{in} \\
& \texttt{acquire}\ \text{lock}; \\
& \texttt{let}\ r = \text{f}\ \ v\ \ \texttt{in} \\
& \texttt{release}\ \text{lock}; \\
& r
\end{aligned}
$$

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:

$$
\begin{aligned}
[\![\texttt{f(e1)}]\!] \stackrel{def}{=}\ & \texttt{fun}\ \text{lock} \Rightarrow \\
& \texttt{let}\ v = [\![\texttt{e1}]\!]\ \texttt{in} \\
& \texttt{acquire}\ \text{lock}; \\
& \texttt{let}\ lock' = \texttt{newmutex}()\ \texttt{in} \\
& \texttt{let}\ r = \text{f}\ \ v\ \text{lock}'\ \texttt{in} \\
& \texttt{release}\ \text{lock}; r
\end{aligned}
$$

Vcg rule for add

$$
\begin{aligned}
\mathsf{vcg}\,(m, \mathtt{e}_1 + \mathtt{e}_2, \mathcal{K}) \;&\overset{def}{=}\\
&\quad \mathtt{match}\ \mathsf{forward}\,(m, \mathtt{e}_1)\ \mathtt{with}\\
&\quad \mid \mathtt{Some}\ (v_1, m_o, m_f) \to \mathsf{vcg}\,(m_f, \mathtt{e}_2, \lambda\, m'\, \mathtt{v}_2.\, \mathcal{K}\ (m' \sqcup m^o)\,(\mathtt{v}_1 + \mathtt{v}_2))\\
&\quad \mid \mathtt{None} \to\\
&\qquad \mathtt{match}\ \mathsf{forward}\,(m, \mathtt{e}_2)\ \mathtt{with}\\
&\qquad \mid \mathtt{Some}\ (v_2, m_o, m_f) \to \mathsf{vcg}\,(m_f, \mathtt{e}_1, \lambda\, m'\, \mathtt{v}_1.\, \mathcal{K}\ (m' \sqcup m^o)\,(\mathtt{v}_1 + \mathtt{v}_2))\\
&\qquad \mid \mathtt{None} \to [\![m]\!] \mathrel{-\!\!*} \mathsf{wp}\ (\mathtt{e}_1 + \mathtt{e}_2)\,\{\lambda\, v, \exists m'.\ [\![m']\!] * \mathcal{K}\ m'\ \mathtt{v}\}
\end{aligned}
$$

(Krebbers POPL'14)

Fractional lockable permissions enforce the sequence point restriction:

$$\frac{\{P\}\ \mathtt{e}\ \left\{\mathtt{l}.\ \exists \mathtt{w}\, q.\ \mathtt{l} \overset{q}{\mapsto}_U \mathtt{w} * (\mathtt{l} \overset{q}{\mapsto}_U \mathtt{w} \twoheadrightarrow\!\!* \ \Phi\ \mathtt{w})\right\}}{\{P\}\ (\mathtt{*e})\ \{\Phi\}}$$

$$\frac{\{P_1\}\ \mathtt{e}_1\ \{\Psi_1\} \quad \{P_2\}\ \mathtt{e}_2\ \{\Psi_2\} \quad (\forall \mathtt{l}\, \mathtt{w}.\ \Psi_1\ \mathtt{l} * \Psi_2\ \mathtt{w} \twoheadrightarrow\!\!* \ \exists \mathtt{v}.\ \mathtt{l} \overset{1}{\mapsto}_U \mathtt{v} * (\mathtt{l} \overset{1}{\mapsto}_L \mathtt{w} \twoheadrightarrow\!\!* \ \Phi\ \mathtt{w}))}{\{P_1 * P_2\}\ (\mathtt{e}_1 = \mathtt{e}_2)\ \{\Phi\}}$$

$\Rightarrow$ Allows to prove $\left\{\mathtt{l} \overset{q}{\mapsto}_U \mathtt{v}\right\} \mathtt{*l} + \mathtt{*l} \left\{\lambda w.(w = v + v) * \mathtt{l} \overset{q}{\mapsto}_U \mathtt{v}\right\}$

$\Rightarrow$ Rules out programs with undefined behavior like $\mathtt{*l} = (\mathtt{*l} = 3)$

(Krebbers POPL'14)

Fractional lockable permissions enforce the sequence point restriction:

$$\frac{\{P\}\ \mathtt{e}\ \left\{\mathtt{l}.\ \exists \mathtt{w}\, q.\ \mathtt{l} \overset{q}{\mapsto}_U \mathtt{w} * (\mathtt{l} \overset{q}{\mapsto}_U \mathtt{w} \twoheadrightarrow \Phi\ \mathtt{w})\right\}}{\{P\}\ (\mathtt{*e})\ \{\Phi\}}$$

$$\frac{\{P_1\}\ \mathtt{e}_1\ \{\Psi_1\} \quad \{P_2\}\ \mathtt{e}_2\ \{\Psi_2\} \quad (\forall \mathtt{l}\, \mathtt{w}.\ \Psi_1\ \mathtt{l} * \Psi_2\ \mathtt{w} \twoheadrightarrow \exists \mathtt{v}.\ \mathtt{l} \overset{1}{\mapsto}_U \mathtt{v} * (\mathtt{l} \overset{1}{\mapsto}_L \mathtt{w} \twoheadrightarrow \Phi\ \mathtt{w}))}{\{P_1 * P_2\}\ (\mathtt{e}_1 = \mathtt{e}_2)\ \{\Phi\}}$$

⇒ Allows to prove $\left\{\mathtt{l} \overset{q}{\mapsto}_U \mathtt{v}\right\} \mathtt{*l} + \mathtt{*l} \left\{\lambda w.(w = v + v) * \mathtt{l} \overset{q}{\mapsto}_U \mathtt{v}\right\}$

⇒ Rules out programs with undefined behavior like $\mathtt{*l} = (\mathtt{*l} = 3)$

(Krebbers POPL'14)

Fractional lockable permissions enforce the sequence point restriction:

$$\frac{\{P\}\ \mathtt{e}\ \left\{\mathtt{l}.\ \exists \mathtt{w}\ q.\ \mathtt{l} \overset{q}{\mapsto}_U \mathtt{w} * (\mathtt{l} \overset{q}{\mapsto}_U \mathtt{w} \mathrel{-\!\!*} \Phi\ \mathtt{w})\right\}}{\{P\}\ (\mathtt{*e})\ \{\Phi\}}$$

$$\frac{\{P_1\}\ \mathtt{e}_1\ \{\Psi_1\} \quad \{P_2\}\ \mathtt{e}_2\ \{\Psi_2\} \quad (\forall \mathtt{l}\ \mathtt{w}.\ \Psi_1\ \mathtt{l} * \Psi_2\ \mathtt{w} \mathrel{-\!\!*} \exists \mathtt{v}.\ \mathtt{l} \overset{1}{\mapsto}_U \mathtt{v} * (\mathtt{l} \overset{1}{\mapsto}_L \mathtt{w} \mathrel{-\!\!*} \Phi\ \mathtt{w}))}{\{P_1 * P_2\}\ (\mathtt{e}_1 = \mathtt{e}_2)\ \{\Phi\}}$$

⇒ Allows to prove $\left\{\mathtt{l} \overset{q}{\mapsto}_U \mathtt{v}\right\}\ \mathtt{*l} + \mathtt{*l}\ \left\{\lambda w.(w = v + v) * \mathtt{l} \overset{q}{\mapsto}_U \mathtt{v}\right\}$

⇒ Rules out programs with undefined behavior like $\mathtt{*l} = (\mathtt{*l} = 3)$

(Krebbers POPL'14)

Fractional lockable permissions enforce the sequence point restriction:

$$\frac{\{P\}\ \mathtt{e}\ \left\{\mathtt{l}.\ \exists \mathtt{w}\, q.\ \mathtt{l} \overset{q}{\mapsto}_U \mathtt{w} * (\mathtt{l} \overset{q}{\mapsto}_U \mathtt{w} -\!\!* \Phi\ \mathtt{w})\right\}}{\{P\}\ (\texttt{*}\mathtt{e})\ \{\Phi\}}$$

$$\frac{\{P_1\}\ \mathtt{e}_1\ \{\Psi_1\} \quad \{P_2\}\ \mathtt{e}_2\ \{\Psi_2\} \quad (\forall \mathtt{l}\, \mathtt{w}.\ \Psi_1\ \mathtt{l} * \Psi_2\ \mathtt{w} -\!\!* \exists \mathtt{v}.\ \mathtt{l} \overset{1}{\mapsto}_U \mathtt{v} * (\mathtt{l} \overset{1}{\mapsto}_L \mathtt{w} -\!\!* \Phi\ \mathtt{w}))}{\{P_1 * P_2\}\ (\mathtt{e}_1 = \mathtt{e}_2)\ \{\Phi\}}$$

$\Rightarrow$ Allows to prove $\left\{\mathtt{l} \overset{q}{\mapsto}_U \mathtt{v}\right\} \texttt{*}\mathtt{l} + \texttt{*}\mathtt{l} \left\{\lambda w.(w = v + v) * \mathtt{l} \overset{q}{\mapsto}_U \mathtt{v}\right\}$

$\Rightarrow$ Rules out programs with undefined behavior like $\texttt{*}\mathtt{l} = (\texttt{*}\mathtt{l} = 3)$

Unlocking modality

remark: we want to access locked pointers later again

$$\texttt{*l} = \texttt{4} \,;\, \texttt{*l}$$

we use the unlocking modality $\mathbb{U}$ that unlocks
all locked locations at the sequence point ;

$$\frac{\mathsf{wp}\ \mathtt{e}_1\ \{\_.\ \mathbb{U}(\mathsf{wp}\ \mathtt{e}_2\ \{\Phi\})\}}{\mathsf{wp}\ (\mathtt{e}_1\,;\,\mathtt{e}_2)\ \{\Phi\}} \qquad \frac{\mathtt{l} \overset{q}{\mapsto}_L \mathtt{v}}{\mathbb{U}(\mathtt{l} \overset{q}{\mapsto}_U \mathtt{v})} \qquad \frac{P \mathrel{-\!\!*} Q}{\mathbb{U}P \mathrel{-\!\!*} \mathbb{U}Q}$$

Example

$\texttt{l} \mapsto \texttt{v1} * \texttt{k} \mapsto \texttt{v2} * \texttt{r} \mapsto \texttt{v3}$

- -

$\texttt{*l} = \texttt{*k} + 10$

postcondition: $\top$

frame: $\top$

Example

$\mathtt{l} \mapsto \mathtt{v1} * \cancel{\mathtt{k} \mapsto \mathtt{v2}} * \mathtt{r} \mapsto \mathtt{v3}$

\- -

$\mathtt{*l} = \mathtt{v2} + 10$

postcondition: $\mathtt{k} \xmapsto{0.5} \mathtt{v2}$

frame: $\mathtt{k} \xmapsto{0.5} \mathtt{v2}$

Example

$\mathtt{l} \mapsto \mathtt{v1} * \sout{\mathtt{k} \mapsto \mathtt{v2}} * \mathtt{r} \mapsto \mathtt{v3}$

- -

$\mathtt{*l} = \mathtt{v2} + 10$

postcondition: $\mathtt{k} \overset{0.5}{\longmapsto} \mathtt{v2}$

frame: $\mathtt{k} \overset{0.5}{\longmapsto} \mathtt{v2}$

Example

$\cancel{\texttt{l} \mapsto \texttt{v1}} * \cancel{\texttt{k} \mapsto \texttt{v2}} * \texttt{r} \mapsto \texttt{v3}$

- -

$\texttt{v2} + 10$

postcondition: $\texttt{k} \xmapsto{0.5} \texttt{v2} * \texttt{l} \mapsto_L (\texttt{v2} + 10)$

frame: $\texttt{k} \xmapsto{0.5} \texttt{v2}$

Example

$\sout{\texttt{l} \mapsto \texttt{v1}} * \sout{\texttt{k} \mapsto \texttt{v2}} * \sout{\texttt{r} \mapsto \texttt{v3}}$

- -

$\texttt{v2} + 10$

postcondition: $\texttt{k} \overset{0.5}{\longmapsto} \texttt{v2} * \texttt{l} \mapsto_L (\texttt{v2} + 10)$

frame: $\texttt{k} \overset{0.5}{\longmapsto} \texttt{v2} * \texttt{r} \mapsto \texttt{v3}$

Example (continued)

$\texttt{l} \mapsto \texttt{v1} * \texttt{k} \mapsto \texttt{v2} * \texttt{r} \mapsto \texttt{v3}$

- -

$(\texttt{*l} = \texttt{*k} + 10) + (\texttt{*r} = \texttt{*k} + 10)$

postcondition: $\top$

frame: $\top$

After executing the LHS

$\sout{\mathtt{l} \mapsto \mathtt{v1}} * \sout{\mathtt{k} \mapsto \mathtt{v2}} * \sout{\mathtt{r} \mapsto \mathtt{v3}}$

- -

$(\mathtt{v2} + \mathtt{10}) + (\mathtt{*r} = \mathtt{*k} + \mathtt{10})$

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

frame: $\mathtt{k} \xrightarrow{0.5} \mathtt{v2} * \mathtt{r} \mapsto \mathtt{v3}$

Before executing the RHS

$\cancel{\texttt{l} \mapsto \texttt{v1}} * \texttt{k} \overset{0.5}{\longmapsto} \texttt{v2} * \texttt{r} \mapsto \texttt{v3}$

- -

$(\texttt{v2} + 10) + (\texttt{*r} = \texttt{*k} + 10)$

postcondition: $\texttt{k} \overset{0.5}{\longmapsto} \texttt{v2} * \texttt{l} \mapsto_L (\texttt{v2} + 10)$

frame: $\cancel{\texttt{k} \overset{0.5}{\longmapsto} \texttt{v2}} * \cancel{\texttt{r} \mapsto \texttt{v3}}$

Executing the RHS

$$\cancel{\mathtt{l} \mapsto \mathtt{v1}} * \cancel{\mathtt{k} \xmapsto{0.5} \mathtt{v2}} * \mathtt{r} \mapsto \mathtt{v3}$$

- -

$$(\mathtt{v2} + 10) + (\mathtt{*r} = \mathtt{v2} + 10)$$

postcondition: $\mathtt{k} \xmapsto{3/4} \mathtt{v2} * \mathtt{l} \mapsto_L (\mathtt{v2} + 10)$

frame: $\mathtt{k} \xmapsto{1/4} \mathtt{v2} * \cancel{\mathtt{r} \mapsto \mathtt{v3}}$

Final result

$\cancel{\texttt{l} \mapsto \texttt{v1}} * \cancel{\texttt{k} \overset{0.5}{\longmapsto}} \texttt{v2} * \cancel{\texttt{r} \mapsto \texttt{v3}}$

- -

$(\texttt{v2} + 10) + (\texttt{v2} + 10)$

postcondition: $\texttt{k} \overset{3/4}{\longmapsto} \texttt{v2} * \texttt{l} \mapsto_L (\texttt{v2} + 10) * \texttt{r} \mapsto_L (\texttt{v2} + 10)$

frame: $\texttt{k} \overset{1/4}{\longmapsto} \texttt{v2} * \cancel{\texttt{r} \mapsto \texttt{v3})}$