Separation Logic for Non-local Control Flow

Robbert Krebbers
Joint work with Freek Wiedijk

Radboud University Nijmegen

November 20, 2012 @ The Brouwer Seminar
int *p = NULL;
l: if (p) {
    return (*p);
} else {
    int j = 10;
p = &j;
goto l;
}
What is this program supposed to do?

```c
int *p = NULL;
l: if (p) {
    return (*p);
} else {
    int j = 10;
    p = &j;
    goto l;
}
```

It exhibits undefined behavior, thus it may do anything.
What is this program supposed to do?

```c
int *p = NULL;
l: if (p) {
    return (*p);
} else {
    int j = 10;
    p = &j;
    goto l;
}
```

It exhibits undefined behavior, thus it may do anything.
What is this program supposed to do?

```c
int *p = NULL;
l: if (p) {
    return (*p);
} else {
    int j = 10;
p = &j;
goto l;
}
```

It exhibits undefined behavior, thus it may do anything.
What is this program supposed to do?

```c
int *p = NULL;
l: if (p) {
    return (*p);
} else {
    int j = 10;
    p = &j;
    goto l;
}
```

It exhibits undefined behavior, thus it may do anything.
What is this program supposed to do?

```c
int *p = NULL;
l: if (p) {
    return (*p);
} else {
    int j = 10;
    p = &j;
    goto l;
}
```

memory:

- `p` (green)
- `j` (yellow)

It exhibits undefined behavior, thus it may do anything.
What is this program supposed to do?

```c
int *p = NULL;
l: if (p) {
    return (*p);
} else {
    int j = 10;
p = &j;
goto l;
}
```

It exhibits undefined behavior, thus it may do anything
What is this program supposed to do?

```c
int *p = NULL;
l: if (p) {
    return (*p);
} else {
    int j = 10;
    p = &j;
    goto l;
}
```

It exhibits undefined behavior, thus it may do anything.
int *p = NULL;
l: if (p) {
    return (*p);
} else {
    int j = 10;
p = &j;
goto l;
}

It exhibits undefined behavior, thus it may do anything
Why is catching undefined behavior important

- C allows a program to do **anything** on undefined behavior
Why is catching undefined behavior important

- C allows a program to do anything on undefined behavior
- It cannot be checked statically
Why is catching undefined behavior important

- C allows a program to do **anything** on undefined behavior
- It cannot be checked statically
- Not catching it means that
  - programs can be proven to be correct with respect to the formal semantics . . .
Why is catching undefined behavior important

- C allows a program to do **anything** on undefined behavior
- It cannot be checked statically
- Not catching it means that
  - programs can be proven to be correct with respect to the formal semantics . . .
  - whereas they may crash when compiled with an actual compiler
Goto considered harmful?

http://xkcd.com/292/
Goto considered harmful?

Not necessarily:

\[ \vdash \{P\} \ldots \text{goto main_sub3}; \ldots \{Q\} \]
Why goto

Goto can be useful

- Breaking from multiple nested loops

```java
for (int i = 0; i++; i < n) {
    // do something here

    for (int j = i; j++; j < m) {
        // do some work here

        goto outer;
    }
}
outer:;
```
Why goto

Goto can be useful

- Breaking from multiple nested loops
- Systematically cleaning up resources after errors

```c
if (!openDataFile())
    goto quit;
if (!getDataFromFile())
    goto closeFileAndQuit;
if (!allocateSomeResources)
    goto freeResourcesAndQuit;

// do actual work here

freeResourcesAndQuit:  // free resources
closeFileAndQuit:     // close file
quit:                  // quit!
```
Why goto

Goto can be useful

- Breaking from multiple nested loops
- Systematically cleaning up resources after errors
- To increase performance
Why goto

Goto can be useful
  ▶ Breaking from multiple nested loops
  ▶ Systematically cleaning up resources after errors
  ▶ To increase performance

Goto is used in practice
  ▶ The Linux kernel contains about \( \sim 100.000 \) uses of goto

```
$ grep -w -r goto ~/src/linux-3.5.5 --include \*.c | wc -l
101026
```
This talk

An elegant small step semantics, and axiomatic semantics for goto, supporting:

- local variables (and pointers to those),
- mutual recursion,
- separation logic,
- soundness proof fully checked by Coq
Approach

- Small step semantics by traversal through the program
Approach

- Small step semantics by traversal through the program
- Traversal in four directions:
  - downwards to the next statement
  - upwards to the next statement
Approach

- Small step semantics by traversal through the program
- Traversal in four directions:
  - \( \downarrow \) downwards to the next statement
  - \( \uparrow \) upwards to the next statement
  - \( \rightarrow \) to a label \( l \): after a `goto l`
  - \( \uparrow \uparrow \) to the top after a `return`
Approach

- Small step semantics by traversal through the program
- Traversal in four directions:
  - \(\searrow\) downwards to the next statement
  - \(\nearrow\) upwards to the next statement
  - \(\rightsquigarrow l\) to a label \(l\): after a \texttt{goto} \(l\)
  - \(\uparrow\uparrow\) to the top after a \texttt{return}
Approach

- Small step semantics by traversal through the program
- Traversal in four directions:
  - \(\searrow\) downwards to the next statement
  - \(\nearrow\) upwards to the next statement
  - \(\leadsto l\) to a label \(l\): after a `goto l`
  - \(\uparrow\uparrow\) to the top after a `return`
- Gotos and returns are also executed in small steps
Example

```c
int *p = NULL

l:
if (p)
    return (*p)

int j = 10
p = &j; goto l
```
Example

```c
int *p = NULL

l:
if (p)
    return (*p)

int j = 10
p = &j
```

memory:
- p
  - NULL

direction:
Example

```c
int *p = NULL

l:
if (p)
return (*p)

int j = 10;
p = &j

goto l
```

**Direction:**

- **Memory:**
  - `p`: NULL
Example

```c
int *p = NULL

l:
if (p)
    return (*p)

int j = 10;
p = &j
```

Direction:

Memory:

```
int *p = NULL
int j = 10
```

NULL
Example

```c
int *p = NULL
l:
if (p)
    return (*p)
int j = 10;
p = &j
```
Example

```c
int *p = NULL

l:
if (p)
    return (*p)

int j = 10;
p = &j

```

**memory:**
- `p`: `NULL`
- `j`: `10`

**direction:**
- `l` to `p`
Example

```c
int *p = NULL
    l:
    if (p)
        return (*p)
    int j = 10
    goto l
p = &j
```
Example

```c
int *p = NULL

l:
if (p)
    return (*p)

int j = 10;
p = &j

goto l
```

direction:

memory:

```
P

j

10
```
Example

```c
int *p = NULL

l:
if (p)
    return (*p)

int j = 10;
p = &j
goto l
```

Diagram:
- `p`: Pointer
- `j`: Integer value 10
- `direction`: Arrow indicating the direction of memory access
- `memory`: Diagram showing the memory allocation and access with a green box for `p` and a yellow box for `j`
Example

```c
int *p = NULL

l:
if (p)
    return (*p)

int j = 10
p = &j
```

memory:
- p
- j

direction:
- From p to j

- From j to p
Example

```c
int *p = NULL

l:
if (p)
    return (*p)

int j = 10;
p = &j

```

```plaintext
memory:

```
```

direction:
```
```
Example

```c
int *p = NULL

l:
if (p)
    return (*p)

int j = 10;
p = &j
```

- **Direction:**
  - l

- **Memory:**
  - p
  - j
  - 10

The code snippet demonstrates a simple program with a null pointer `p` initialized. The `if` statement checks if the pointer is not null. If it is not null, the program returns the value of the pointer. The pointer `p` is then set to the address of the integer `j`, which is initialized to 10. The program then jumps back to the label `l`.
Example

```c
int *p = NULL
l:
    if (p)
        return (*p)
    goto l
int j = 10
p = &j
```

direction:

```
-> l
```

memory:

```
p
```

```
j
```

```
10
```
Example

```c
int *p = NULL

l:
if (p)
    return (*p)

int j = 10;
p = &j
```

**memory:**
- `p` points to `j`
- `j` is initialized to `10`

**direction:**
- Move from `p` to `j`
Example

```c
int *p = NULL
if (p)
    return (*p)
int j = 10;
p = &j
goto l
```
Example

```c
int *p = NULL

l:
if (p)
    return (*p)

int j = 10
p = &j goto l
```

direction:

memory:

```
int j = 10
p = &j
```
Example

```c
int *p = NULL

l:

if (p)
    return (*p)

int j = 10;
p = &j
    goto l
```

direction:

memory:

```
p
```
Example

```c
int *p = NULL
l:
if (p)
    return (*p)
```

```c
int j = 10
p = &j
goto l
```
Example

```c
int *p = NULL
l:
if (p)
    return (*p)
int j = 10
p = &j
goto l
```
How to model the current *location* in the program

Huet’s zipper

Purely functional way to store a pointer into a data structure
Program contexts (1)

Statements:

\[
s ::= \text{block } s \mid e_l := e_r \mid f(\bar{e}) \mid \text{skip} \mid \text{goto } l \mid s_1; s_2 \mid \text{if } (e) s_1 s_2 \mid \text{return}
\]
Program contexts (1)

Statements:

\[ s ::= \text{block } s \mid e_l := e_r \mid f(\bar{e}) \mid \text{skip} \mid \text{goto } l \mid s_1; s_2 \mid \text{if } (e) s_1 s_2 \mid \text{return} \]

Singular statement contexts:

\[ ES ::= \square; s_2 \mid s_1; \square \mid \text{if } (e) \square s_2 \mid \text{if } (e) s_1 \square \mid l:\square \]
Program contexts (1)

- **Statements:**
  
  \[
  s ::= \text{block } s \mid e_l := e_r \mid f(\vec{e}) \mid \text{skip} \mid \text{goto } l \mid s_1; s_2 \mid \text{if } (e) s_1 s_2 \mid \text{return}
  \]

- **Singular statement contexts:**

  \[
  E_S ::= \Box; s_2 \mid s_1; \Box \mid \text{if } (e) \Box s_2 \mid \text{if } (e) s_1 \Box \mid l: \Box
  \]

- **A pair** \((\vec{E}_S, s)\) **forms a zipper for statements**, where
  - \(\vec{E}_S\) is a statement turned inside-out
  - \(s\) is the focused substatement
Program contexts (2)

Singular program contexts:

\[ E ::= E_S \mid \text{block}_b \square \mid \text{call} f \vec{e} \mid \text{params} \vec{b} \]

where:

- \text{block}_b \square associates a block scope variable with its corresponding memory index
- \text{call} f \vec{e} contains the location of the caller so that it can be restored when \text{f} returns
- \text{params} \vec{b} contains the memory indexes of the function parameters
Program contexts (2)

Singular program contexts:

\[ E ::= E_s \mid \text{block}_b \square \mid \text{call } f \vec{e} \mid \text{params } \vec{b} \]

where:

- \text{block}_b \square \text{ associates a block scope variable with its corresponding memory index } b

- \text{call } f \vec{e} \text{ contains the location of the caller so that it can be restored when } f \text{ returns}

- \text{params } \vec{b} \text{ contains the memory indexes of the function parameters}
Program contexts (2)

Singular program contexts:

\[ E ::= E_S \mid \text{block}_b \square \mid \text{call } f \vec{e} \mid \text{params } \vec{b} \]

where:

- \text{block}_b \square \text{associates a block scope variable with its corresponding memory index } b
- \text{call } f \vec{e} \text{ contains the location of the caller so that it can be restored when } f \text{ returns}
Program contexts (2)

Singular program contexts:

\[ E ::= E_S \mid \text{block}_b \square \mid \text{call } f \vec{e} \mid \text{params } \vec{b} \]

where:

- block\(_b \square\) associates a block scope variable with its corresponding memory index \(b\)
- call \(f \vec{e}\) contains the location of the caller so that it can be restored when \(f\) returns
- params \(\vec{b}\) contains the memory indexes of the function parameters
Program contexts (2)

Singular program contexts:

\[ E ::= E_S \mid \text{block}_b \square \mid \text{call } f \vec{e} \mid \text{params } \vec{b} \]

where:

- \( \text{block}_b \square \) associates a block scope variable with its corresponding memory index \( b \)
- \( \text{call } f \vec{e} \) contains the location of the caller so that it can be restored when \( f \) returns
- \( \text{params } \vec{b} \) contains the memory indexes of the function parameters

Program contexts \( k \) are lists of singular program contexts
Stacks (1)

- A *stack* is a list of memory indexes
- A variable $x_i$ refers to the $i$th element on the stack
Stacks (1)

- A stack is a list of memory indexes
- A variable $x_i$ refers to the $i$th element on the stack

Uniform way of dealing with pointers to local variables
Program contexts contain the stack:

\[
\begin{align*}
\text{getstack } (E_S :: k) & := \text{getstack } k \\
\text{getstack } (\text{block}_b \; \square :: k) & := b :: \text{getstack } k \\
\text{getstack } (\text{call } f \; \vec{e} :: k) & := [] \\
\text{getstack } (\text{params } \vec{b} :: k) & := \vec{b} +\!+ \text{getstack } k
\end{align*}
\]
Program contexts contain the stack:

\[
\begin{align*}
g \text{getstack} \ (E_S :: k) & := \text{getstack} \ k \\
g \text{getstack} \ (\text{block}_b \ □ :: k) & := b :: \text{getstack} \ k \\
g \text{getstack} \ (\text{call} \ f \ \vec{e} :: k) & := [] \\
g \text{getstack} \ (\text{params} \ \vec{b} :: k) & := \vec{b} \ ++ \ \text{getstack} \ k
\end{align*}
\]

Remark: not \( \text{getstack} \ (\text{call} \ f \ \vec{e} :: k) := \text{getstack} \ k \)
A state $S(k, \phi, m)$ consists of a program context $k$, focus $\phi$, and memory $m$. 
States

A state $S(k, \phi, m)$ consists of a program context $k$, focus $\phi$, and memory $m$.

We consider the following focuses:

- $(d, s)$ execution of a statement $s$ in direction $d$
States

A state $S(k, \phi, m)$ consists of a program context $k$, focus $\phi$, and memory $m$

We consider the following focuses:

- $(d, s)$ execution of a statement $s$ in direction $d$
- $\text{call } f \overrightarrow{v}$ calling a function $f(\overrightarrow{v})$
States

A state $S(k, \phi, m)$ consists of a program context $k$, focus $\phi$, and memory $m$

We consider the following focuses:

- $(d, s)$ execution of a statement $s$ in direction $d$
- $\text{call } f \vec{v}$ calling a function $f(\vec{v})$
- $\text{return}$ returning from a function
Example

```
int *p = NULL
         |
         l:
          |
         if (p)
          |
        return
         int j = 10
             |
             |
         p = &j   goto l
```

The corresponding state is $S(k, \phi, m)$, where:

- $k = [\square; \text{goto } l, x_0 := \text{int } 10; \square, \text{block}_b \square, \text{if (load } x_0) \text{return } \square, l: \square, x_0 := \text{NULL }; \square, \text{block}_b \square]$

- $\phi = (\uparrow, x_1 := x_0)$

- $m = \{b_p \mapsto \text{ptr } b_j, b_j \mapsto 10\}$
The small step semantics

Some rules:

\[ S(k, (\downarrow, e_l := e_r), m) \rightarrow S(k, (\uparrow, e_l := e_r), m[a := v]) \]
provided that \[ \llbracket e_1 \rrbracket_{k,m} = \text{ptr } a, \llbracket e_2 \rrbracket_{k,m} = v, \text{ and } m[a] \neq \perp \]
The small step semantics

Some rules:

- \( S(k, (\downarrow, e_l := e_r), m) \rightarrow S(k, (\uparrow, e_l := e_r), m[a := v]) \)
  provided that \([e_1]_{k,m} = \text{ptr } a\), \([e_2]_{k,m} = v\), and \(m[a] \neq \bot\)

- \( S(k, (\downarrow, s_1 ; s_2), m) \rightarrow S((\square ; s_2) :: k, (\downarrow, s_1), m) \)
The small step semantics

Some rules:

- \( S(k, (\downarrow, e_l := e_r), m) \rightarrow S(k, (\uparrow, e_l := e_r), m[a := v]) \)
  provided that \([e_1]_{k,m} = \text{ptr } a, [e_2]_{k,m} = v\), and \( m[a] \neq \bot \)

- \( S(k, (\downarrow, s_1 ; s_2), m) \rightarrow S((\square ; s_2) :: k, (\downarrow, s_1), m) \)

- \( S((\square ; s_2) :: k, (\uparrow, s_1), m) \rightarrow S((s_1 ; \square) :: k, (\downarrow, s_2), m) \)
The small step semantics

Some rules:

- \( S(k, (\leftarrow, e_l := e_r), m) \rightarrow S(k, (\rightarrow, e_l := e_r), m[a := v]) \)
  provided that \( [e_1]_{k,m} = \text{ptr } a \), \( [e_2]_{k,m} = v \), and \( ma \neq \perp \)
- \( S(k, (\leftarrow, s_1 ; s_2), m) \rightarrow S((\square ; s_2) :: k, (\leftarrow, s_1), m) \)
- \( S((\square ; s_2) :: k, (\rightarrow, s_1), m) \rightarrow S((s_1 ; \square) :: k, (\leftarrow, s_2), m) \)
- \( S((s_1 ; \square) :: k, (\rightarrow, s_2), m) \rightarrow S(k, (\rightarrow, s_1 ; s_2), m) \)
The small step semantics

Some rules:

1. \( S(k, (\downarrow, e_I := e_r), m) \rightarrow S(k, (\uparrow, e_I := e_r), m[a := v]) \)
   provided that \([e_1]_{k,m} = \text{ptr } a\), \([e_2]_{k,m} = v\), and \(m[a] \neq \bot\)

2. \( S(k, (\downarrow, s_1 ; s_2), m) \rightarrow S((\square ; s_2) :: k, (\downarrow, s_1), m)\)

3. \( S((\square ; s_2) :: k, (\uparrow, s_1), m) \rightarrow S((s_1 ; \square) :: k, (\downarrow, s_2), m)\)

4. \( S((s_1 ; \square) :: k, (\uparrow, s_2), m) \rightarrow S(k, (\uparrow, s_1 ; s_2), m)\)

5. \( S(k, (\downarrow, \text{goto } l), m) \rightarrow S(k, (\nearrow l, \text{goto } l), m)\)
The small step semantics

Some rules:

- $S(k, \langle \downarrow, e_l := e_r \rangle, m) \rightarrow S(k, \langle \uparrow, e_l := e_r \rangle, m[a := v])$
  provided that $[e_1]_{k,m} = \text{ptr } a$, $[e_2]_{k,m} = v$, and $ma \neq \bot$

- $S(k, \langle \downarrow, s_1 ; s_2 \rangle, m) \rightarrow S((\square ; s_2) :: k, \langle \downarrow, s_1 \rangle, m)$

- $S((\square ; s_2) :: k, \langle \uparrow, s_1 \rangle, m) \rightarrow S((s_1 ; \square) :: k, \langle \downarrow, s_2 \rangle, m)$

- $S((s_1 ; \square) :: k, \langle \uparrow, s_2 \rangle, m) \rightarrow S(k, \langle \uparrow, s_1 ; s_2 \rangle, m)$

- $S(k, \langle \downarrow, \text{goto } l \rangle, m) \rightarrow S(k, \langle \swarrow l, \text{goto } l \rangle, m)$

- $S(k, \langle \swarrow l, l : s \rangle, m) \rightarrow S((l : \square) :: k, \langle \downarrow, s \rangle, m)$
The small step semantics

Some rules:

- $S(k, (\leftarrow, e_l := e_r), m) \rightarrow S(k, (\rightarrow, e_l := e_r), m[a := v])$
  provided that $\llbracket e_1 \rrbracket_{k,m} = \text{ptr } a$, $\llbracket e_2 \rrbracket_{k,m} = v$, and $m a \neq \bot$

- $S(k, (\leftarrow, s_1; s_2), m) \rightarrow S((\square; s_2) :: k, (\leftarrow, s_1), m)$

- $S((\square; s_2) :: k, (\rightarrow, s_1), m) \rightarrow S((s_1; \square) :: k, (\leftarrow, s_2), m)$

- $S((s_1; \square) :: k, (\rightarrow, s_2), m) \rightarrow S(k, (\rightarrow, s_1; s_2), m)$

- $S(k, (\leftarrow, \text{goto } l), m) \rightarrow S(k, (\Rightarrow l, \text{goto } l), m)$

- $S(k, (\Rightarrow l, l : s), m) \rightarrow S((l : \square) :: k, (\leftarrow, s), m)$

- $S(k, (\leftarrow, f(\bar{e})), m) \rightarrow S(\text{call } f \bar{e} :: k, \underline{\text{call } f \bar{v}}, m)$
  provided that $\llbracket e_i \rrbracket_{k,m} = v_i$ for each $i$
The small step semantics

Lemma

The small step semantics behaves as traversing through a zipper. That is, if

\[ S(k, (d, s), m) \rightarrow^* k S(k, (d', s'), m') \]

then \( s = s' \).
Program contexts versus continuations

- Program contexts also contain the part of the program that has already been executed

Continuation

Program context
Program contexts versus continuations

- Program contexts also contain the part of the program that has already been executed.

\[
\begin{align*}
\text{Continuation} & \quad \text{Program context} \\
\end{align*}
\]

- Looping constructs do not have to duplicate code.

\[
S(k, (\vartriangleright, \text{while}(e) s), m) \rightarrow S((\text{while}(e) \Box) :: k, (\vartriangleright, s), m)
\]

provided that \( [e]_{k,m} = v \) and istrue \( v \).
Program contexts versus continuations

- Program contexts also contain the part of the program that has already been executed

Continuation

Program context

- Looping constructs do not have to duplicate code

\[ S(k, (\langle, \text{while}(e) \, s), m) \rightarrow S((\text{while}(e) \, \square) :: k, (\langle, s), m) \]

provided that \[ \llbracket e \rrbracket_{k,m} = v \] and istrue \( v \)

- Program contexts implicitly contain the stack
Traditional *Hoare triples* are of the shape

\[ \{P\} \; s \; \{Q\} \]

Intuitive meaning:

- If \( P \) holds for the state before execution of \( s \),
- and execution of \( s \) terminates,
- then \( Q \) will hold afterwards
Traditional *Hoare triples* are of the shape

\[ \{ P \} \ s \ \{ Q \} \]

Intuitive meaning:

- If \( P \) holds for the state before execution of \( s \),
- and execution of \( s \) terminates,
- then \( Q \) will hold afterwards

We use a *shallow embedding*, i.e. assertions \( P, Q \) are predicates on the stack and memory.
Our *Hoare sextuples* are of the shape

\[ \Delta; J; R \vdash \{ P \} s \{ Q \} \]
Extended Hoare ‘triples’ (1)

Our Hoare sextuples are of the shape

\[ \Delta; J; R \vdash \{P\} s \{Q\} \]

where:
- \(\Delta\) maps function names to their pre- and postconditions
Extended Hoare ‘triples’ (1)

Our Hoare sextuples are of the shape

\[ \Delta; J; R \vdash \{ P \} s \{ Q \} \]

where:

- \( \Delta \) maps function names to their pre- and postconditions
- \( J \) maps labels to their jumping condition
  - When executing a goto \( l \), the assertion \( J l \) has to hold
Our *Hoare sextuples* are of the shape

\[ \Delta; J; R \vdash \{P\} s \{Q\} \]

where:
- \(\Delta\) maps function names to their pre- and postconditions
- \(J\) maps labels to their jumping condition
  - When executing a `goto l`, the assertion \(J l\) has to hold
- \(R\) has to hold to execute a `return`
Extended Hoare ‘triples’ (2)

Our Hoare sextuples are of the shape

\[
\Delta; J; R \vdash \{P\} s \{Q\}
\]

Observations:
▶ The assertions \(P\), \(Q\), \(J\) and \(R\) correspond to the four directions \(\searrow\), \(\nearrow\), \(\swarrow\) and \(\uparrow\uparrow\)
Our Hoare sextuples are of the shape

\[ \Delta; J; R \vdash \{ P \} s \{ Q \} \]

Observations:

- The assertions \( P \), \( Q \), \( J \) and \( R \) correspond to the four directions \( \searrow \), \( \nearrow \), \( \swarrow \) and \( \uparrow \)
- We thus treat the sextuple as

\[ \Delta; \bar{P} \vdash s \]

where \( \bar{P}_\searrow = P \), \( \bar{P}_\nearrow = Q \), \( \bar{P}_\swarrow (\bowtie I) = J I \) and \( \bar{P}_\uparrow \uparrow = R \)
Some Hoare rules

Weakening:

\[ P \rightarrow P' \quad \Delta; \ J; \ R \vdash \{ P' \} s \{ Q' \} \quad Q' \rightarrow Q \]

\[ \Delta; \ J; \ R \vdash \{ P \} s \{ Q \} \]
Some Hoare rules

Weakening:

\[ P \rightarrow P' \quad \Delta; J; R \vdash \{ P' \} s \{ Q' \} \quad Q' \rightarrow Q \]

\[ \Delta; J; R \vdash \{ P \} s \{ Q \} \]

Composition:

\[ \Delta; J; R \vdash \{ P \} s_1 \{ P' \} \quad \Delta; J; R \vdash \{ P' \} s_2 \{ Q \} \]

\[ \Delta; J; R \vdash \{ P \} s_1 ; s_2 \{ Q \} \]
Some Hoare rules

Weakening:

\[
P \rightarrow P' \quad \Delta; J; R \vdash \{P'\} s\{Q'\} \quad Q' \rightarrow Q \]

\[
\frac{\Delta; J; R \vdash \{P\} s\{Q\}}{\Delta; J; R \vdash \{P\} s\{Q\}}
\]

Composition:

\[
\Delta; J; R \vdash \{P\} s_1 \{P'\} \quad \Delta; J; R \vdash \{P'\} s_2 \{Q\}
\]

\[
\frac{\Delta; J; R \vdash \{P\} s_1 s_2 \{Q\}}{\Delta; J; R \vdash \{P\} s_1 s_2 \{Q\}}
\]

Non-local control:

\[
\Delta; J; R \vdash \{R\} \text{return}\{Q\}
\]
Some Hoare rules

Weakening:

\[ P \rightarrow P' \quad \Delta; J; R \vdash \{ P' \} s \{ Q' \} \quad Q' \rightarrow Q \]

\[ \Delta; J; R \vdash \{ P \} s \{ Q \} \]

Composition:

\[ \Delta; J; R \vdash \{ P \} s_1 \{ P' \} \quad \Delta; J; R \vdash \{ P' \} s_2 \{ Q \} \]

\[ \Delta; J; R \vdash \{ P \} s_1 ; s_2 \{ Q \} \]

Non-local control:

\[ \Delta; J; R \vdash \{ R \} \text{return} \{ Q \} \]

\[ \Delta; J; R \vdash \{ J l \} \text{goto} \, l \{ Q \} \]

\[ \Delta; J; R \vdash \{ J l \} \, l : s \{ Q \} \]

\[ \Delta; J; R \vdash \{ J l \} s \{ Q \} \]
Separation logic

\[ \text{emp} := \lambda \rho \ m . \ m = \emptyset \]
Separation logic

\[
\text{emp} := \lambda \rho \ m. \ m = \emptyset
\]

\[
P \ast Q := \lambda \rho \ m. \ \exists m_1 \ m_2 .
\]

\[
m = m_1 \cup m_2 \land m_1 \perp m_2 \land P \rho m_1 \land Q \rho m_2
\]
Separation logic

\[ \text{emp} := \lambda \rho m . m = \emptyset \]

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

\[ e_1 \mapsto e_2 := \exists v_1 v_2 . e_1 \Downarrow v_1 \land e_2 \Downarrow v_2 \land m = \{(v_1, v_2)\} \]
The frame rule

Used for local reasoning

\[
\begin{align*}
\Delta; J; R &\vdash \{ P \} s \{ Q \} \\
\Delta; J \ast A; R \ast A &\vdash \{ P \ast A \} s \{ Q \ast A \}
\end{align*}
\]
The frame rule

Used for local reasoning

\[ \Delta; J; R \vdash \{ P \} s \{ Q \} \]

\[ \Delta; J \ast A; R \ast A \vdash \{ P \ast A \} s \{ Q \ast A \} \]

Using our alternative notation:

\[ \Delta; \bar{P} \vdash s \]

\[ \Delta; \bar{P} \ast A \vdash s \]
The block scope variable rule

The assertion $A \uparrow$ lifts the DeBruijn indexes in $A$

\[
\Delta; J \uparrow \ast x_0 \mapsto \; ; R \uparrow \ast x_0 \mapsto \; \vdash \{P \uparrow \ast x_0 \mapsto \} s \{Q \uparrow \ast x_0 \mapsto \}
\]

\[
\Delta; J; R \vdash \{P\} \text{ block } s \{Q\}
\]
The block scope variable rule

The assertion $A \uparrow$ lifts the DeBruijn indexes in $A$

\[
\Delta; J \uparrow * x_0 \mapsto -; R \uparrow * x_0 \mapsto - \vdash \{P \uparrow * x_0 \mapsto -\} s \{Q \uparrow * x_0 \mapsto -\}
\]

\[
\Delta; J; R \vdash \{P\} \text{block } s \{Q\}
\]

Using our alternative notation

\[
\Delta; \bar{P} \uparrow * x_0 \mapsto - \vdash s
\]

\[
\Delta; \bar{P} \vdash \text{block } s
\]
Soundness of the axiomatic semantics

- Define $\Delta; J; R \vDash \{P\} s \{Q\}$ in terms of operational semantics
Soundness of the axiomatic semantics

- Define $\Delta; J; R \models \{P\} s \{Q\}$ in terms of operational semantics
- Prove $\Delta; J; R \vdash \{P\} s \{Q\}$ implies $\Delta; J; R \models \{P\} s \{Q\}$
Soundness of the axiomatic semantics

- Define $\Delta; J; R \models \{P\} s \{Q\}$ in terms of operational semantics
- Prove $\Delta; J; R \vdash \{P\} s \{Q\}$ implies $\Delta; J; R \models \{P\} s \{Q\}$
- Tricky definition of $\Delta; J; R \models \{P\} s \{Q\}$ because
  - The frame rule
  - Undefined behavior
  - Non-local control
  - Mutual recursion
Formalization in Coq

- Extremely useful for debugging
Formalization in Coq

- Extremely useful for debugging
- Notations close to those on paper
Formalization in Coq

- Extremely useful for debugging
- Notations close to those on paper
- Also supports while and functions with return values
Formalization in Coq

- Extremely useful for debugging
- Notations close to those on paper
- Also supports while and functions with return values
- Axiomatic semantics as derived lemmas instead of inference system

6500 lines, of which half on the actual formalization
Formalization in Coq

- Extremely useful for debugging
- Notations close to those on paper
- Also supports while and functions with return values
- Axiomatic semantics as derived lemmas instead of inference system
- Uses lots of automation

6500 lines, of which half on the actual formalization
Formalization in Coq

- Extremely useful for debugging
- Notations close to those on paper
- Also supports while and functions with return values
- Axiomatic semantics as derived lemmas instead of inference system
- Uses lots of automation
- 6500 lines, of which half on the actual formalization
Future research

- Expressions with side effects
- The C type system
- Non-aliasing restrictions
- Verification condition generator in Coq
- Correspondence with CompCert
Questions

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