Separation Logic for Non-local Control Flow

Robbert Krebbers
Joint work with Freek Wiedijk

Radboud University Nijmegen

November 20, 2012 @ The Brouwer Seminar

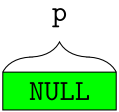
What is this program supposed to do?

```
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;
}
```

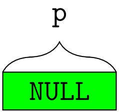
memory:



What is this program supposed to do?

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

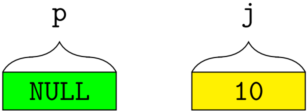
memory:



What is this program supposed to do?

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

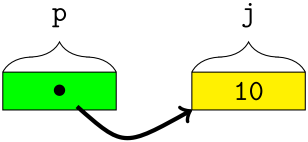
memory:



What is this program supposed to do?

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

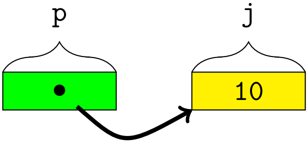
memory:



What is this program supposed to do?

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

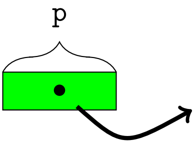
memory:



What is this program supposed to do?

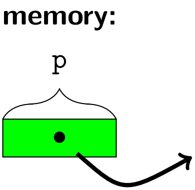
```
int *p = NULL;
l: if (p) {
  return (*p);
} else {
  int j = 10;
  p = &j;
  goto l;
}
```

memory:



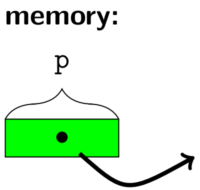
What is this program supposed to do?

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



What is this program supposed to do?

```
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

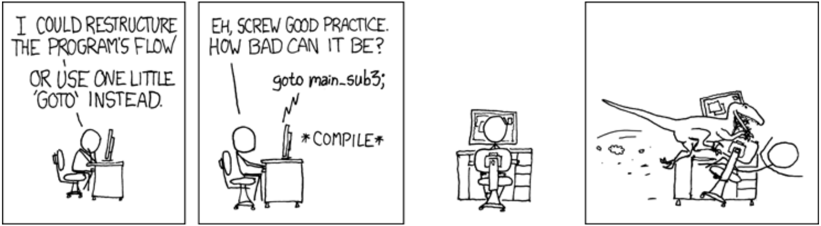
Why is catching undefined behavior important

- 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:

$$\vdash \{P\}\ldots\ \texttt{goto main\_sub3;}\ \ldots\{Q\}$$

Why goto

Goto can be useful

- Breaking from multiple nested loops

```
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

```
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:
  - $\searrow$ downwards to the next statement
  - $\nearrow$ upwards to the next statement

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
  - $\curvearrowright$ `l` to a label `l:` after a `goto l`

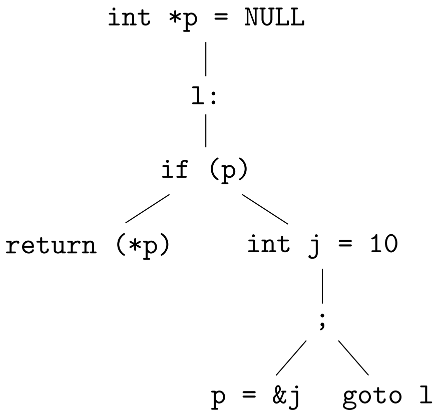
Approach

- Small step semantics by traversal through the program
- Traversal in four directions:
  - $\searrow$ downwards to the next statement
  - $\nearrow$ upwards to the next statement
  - $\curvearrowright$ `l` to a label `l:` after a `goto l`
  - $\upuparrows$ to the top after a `return`

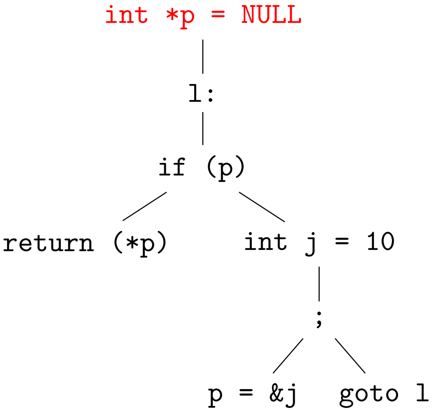
Approach

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

Example

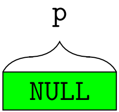


Example

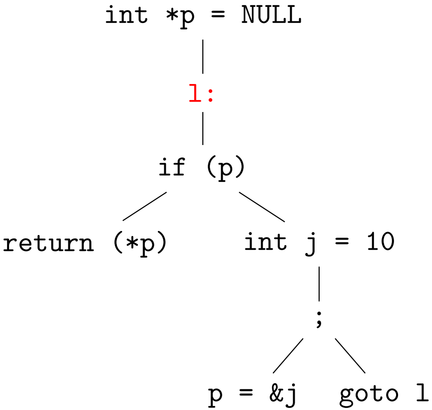


direction:

memory:

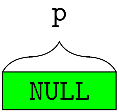


Example

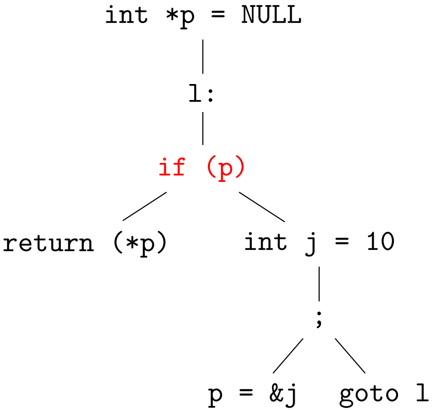


direction:

memory:

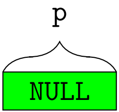


Example

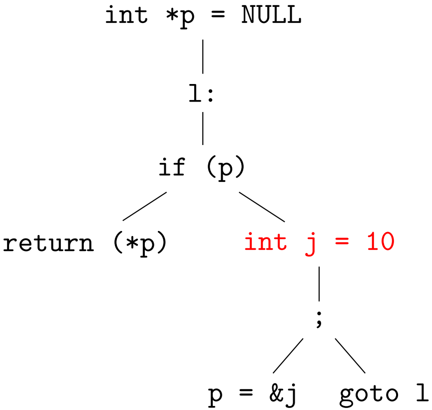


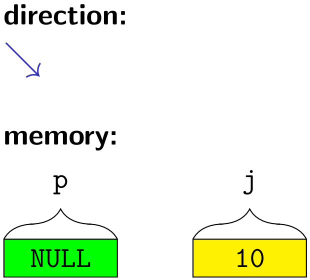
direction:

memory:

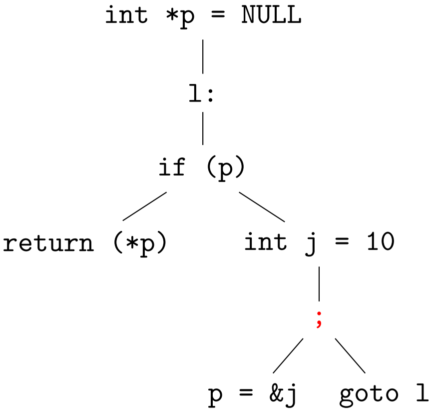


Example


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

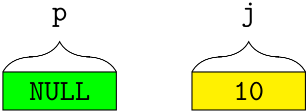

direction:
memory:
p
NULL
j
10

Example

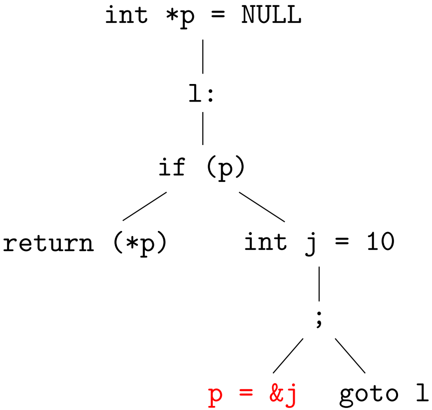


direction:

memory:

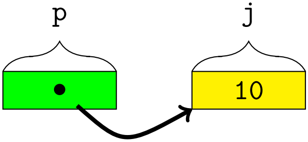


Example

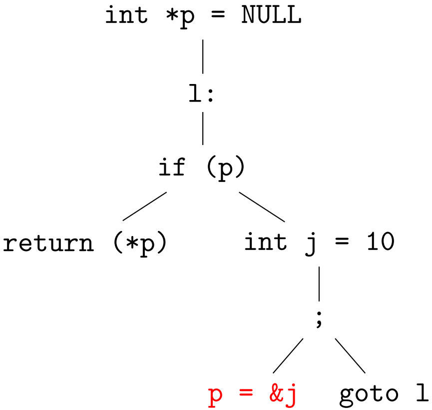


direction:

memory:

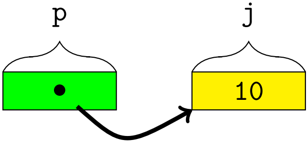


Example

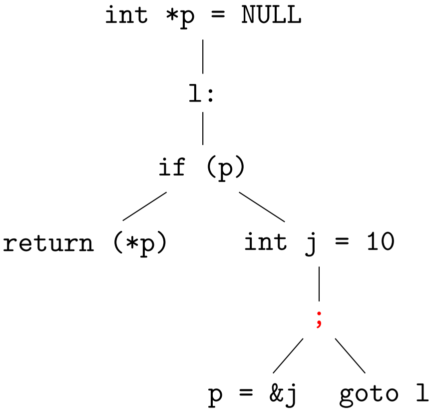


direction:

memory:

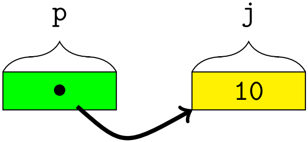


Example

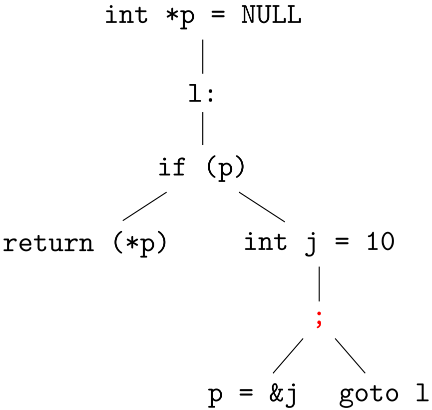


direction:

memory:

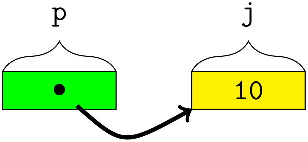


Example

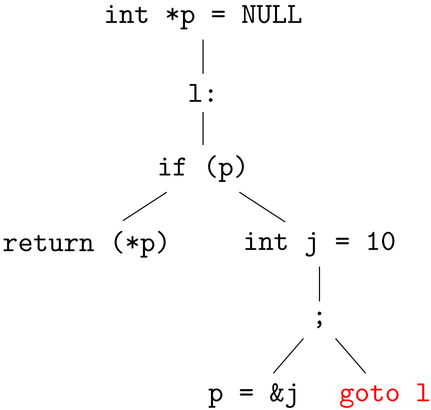


direction:

memory:

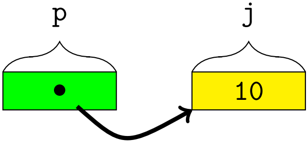


Example

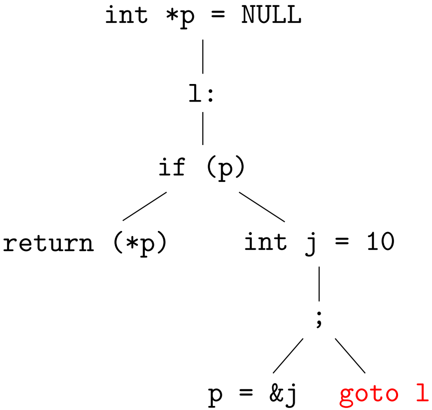


direction:

memory:



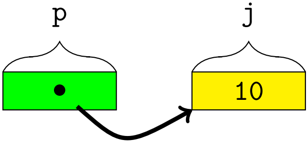
Example



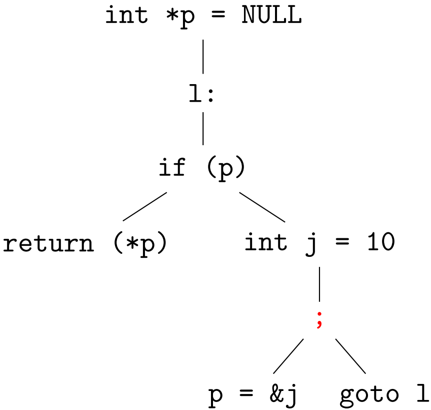
direction:

↷ l

memory:



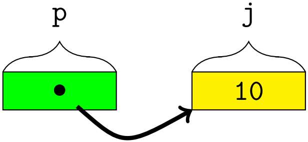
Example



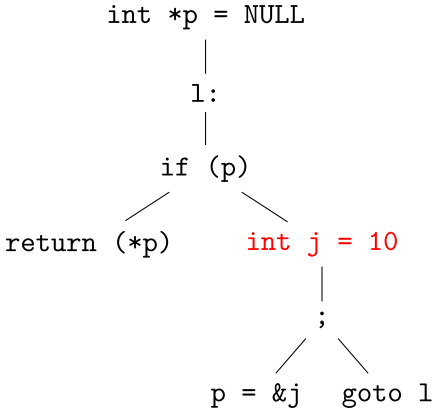
direction:

$\curvearrowright$ `l`

memory:



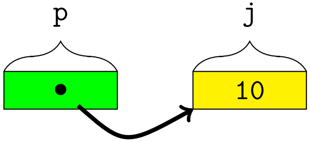
Example



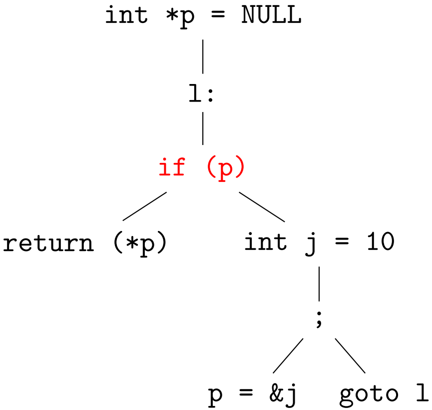
direction:

↷ l

memory:



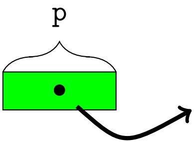
Example



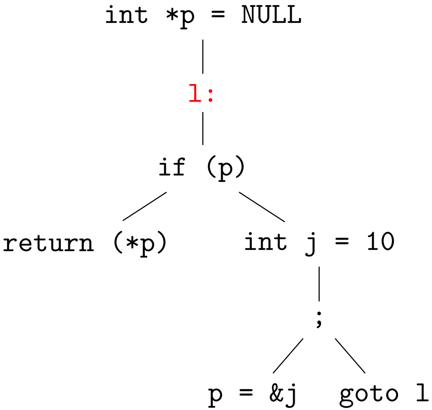
direction:

↷ l

memory:



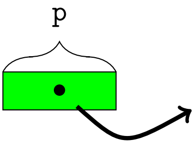
Example



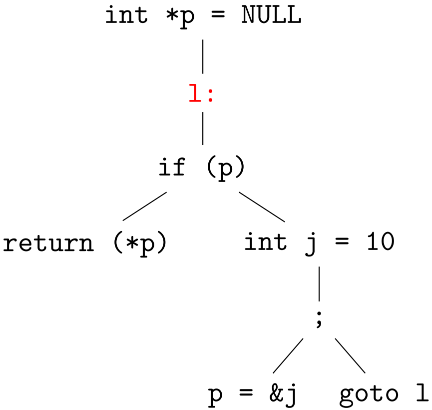
direction:

$\curvearrowright$ l

memory:

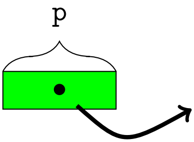


Example

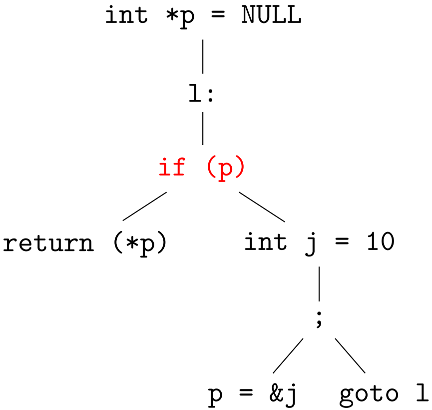


direction:

memory:

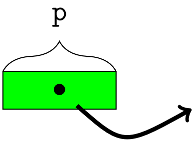


Example

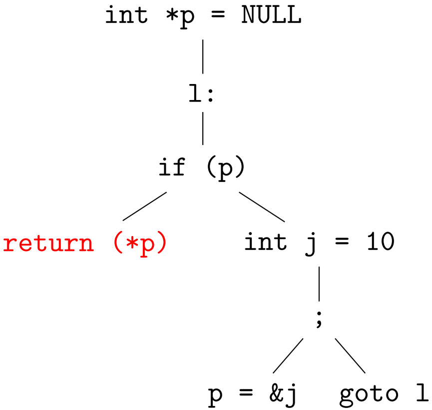


direction:

memory:

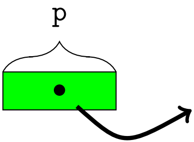


Example



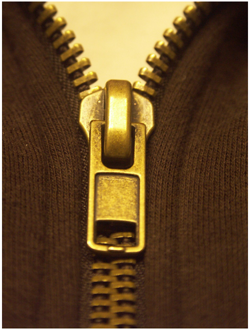
direction:

memory:



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:

$$\begin{aligned} s ::= {} & \texttt{block}\ s \mid e_l := e_r \mid f(\vec{e}) \mid \texttt{skip} \mid \texttt{goto}\ l \mid \\ & s_1\,;s_2 \mid \texttt{if}\ (e)\ s_1\ s_2 \mid \texttt{return} \end{aligned}$$

Program contexts (1)

- Statements:

$$s ::= \texttt{block}\ s \mid e_l := e_r \mid f(\vec{e}) \mid \texttt{skip} \mid \texttt{goto}\ l \mid$$
$$s_1 \,;\, s_2 \mid \texttt{if}\ (e)\ s_1\ s_2 \mid \texttt{return}$$

- Singular statement contexts:

$$E_{\mathcal{S}} ::= \square \,;\, s_2 \mid s_1 \,;\, \square \mid \texttt{if}\ (e)\ \square\ s_2 \mid \texttt{if}\ (e)\ s_1\ \square \mid l : \square$$

Program contexts (1)

- Statements:

$$s ::= \texttt{block}\ s \mid e_l := e_r \mid f(\vec{e}) \mid \texttt{skip} \mid \texttt{goto}\ l \mid \\ s_1 \,;\, s_2 \mid \texttt{if}\ (e)\ s_1\ s_2 \mid \texttt{return}$$

- Singular statement contexts:

$$E_{\mathcal{S}} ::= \Box \,;\, s_2 \mid s_1 \,;\, \Box \mid \texttt{if}\ (e)\ \Box\ s_2 \mid \texttt{if}\ (e)\ s_1\ \Box \mid l : \Box$$

- A pair $(\vec{E_{\mathcal{S}}}, s)$ forms a zipper for statements, where
  - $\vec{E_{\mathcal{S}}}$ is a statement turned inside-out
  - s is the focused substatement

Program contexts (2)

Singular program contexts:

$$E ::= E_{\mathcal{S}} \mid \texttt{block}_b \, \square \mid \text{call } f \ \vec{e} \mid \text{params } \vec{b}$$

where:

Program contexts (2)

Singular program contexts:

$$E ::= E_{\mathcal{S}} \mid \mathtt{block}_b \, \square \mid \mathsf{call}\ f\ \vec{e} \mid \mathsf{params}\ \vec{b}$$

where:

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

Program contexts (2)

Singular program contexts:

$$E ::= E_{\mathcal{S}} \mid \texttt{block}_b \, \square \mid \text{call } f \ \vec{e} \mid \text{params } \vec{b}$$

where:

- $\texttt{block}_b \, \square$ 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

Program contexts (2)

Singular program contexts:

$$E ::= E_{\mathcal{S}} \mid \texttt{block}_b \, \square \mid \text{call } f \ \vec{e} \mid \text{params } \vec{b}$$

where:

- $\texttt{block}_b \, \square$ 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:

$$E ::= E_{\mathcal{S}} \mid \texttt{block}_b\,\Box \mid \text{call } f\ \vec{e} \mid \text{params } \vec{b}$$

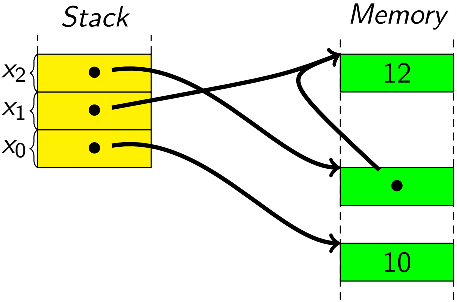
where:

- $\texttt{block}_b\,\Box$ 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 k are lists of singular program contexts

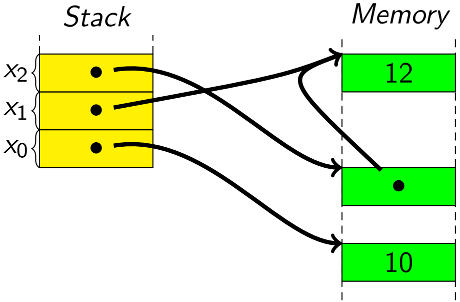
Stacks (1)

- A *stack* is a list of memory indexes
- A variable x_i refers to the ith element on the stack



Stacks (1)

- A *stack* is a list of memory indexes
- A variable x_i refers to the ith element on the stack



- Uniform way of dealing with pointers to local variables

Stacks (2)

- Program contexts contain the stack:

$$\begin{aligned}
\text{getstack}\ (E_{\mathcal{S}} :: k) &:= \text{getstack}\ k \\
\text{getstack}\ (\texttt{block}_b\,\Box :: k) &:= b :: \text{getstack}\ k \\
\text{getstack}\ (\text{call}\ f\ \vec{e} :: k) &:= [\,] \\
\text{getstack}\ (\text{params}\ \vec{b} :: k) &:= \vec{b} +\!\!+ \text{getstack}\ k
\end{aligned}$$

Stacks (2)

- Program contexts contain the stack:

$$
\begin{aligned}
\mathrm{getstack}\ (E_{\mathcal{S}} :: k) &:= \mathrm{getstack}\ k \\
\mathrm{getstack}\ (\texttt{block}_b\ \Box :: k) &:= b :: \mathrm{getstack}\ k \\
\mathrm{getstack}\ (\mathrm{call}\ f\ \vec{e} :: k) &:= [\,] \\
\mathrm{getstack}\ (\mathrm{params}\ \vec{b} :: k) &:= \vec{b} +\!\!+ \mathrm{getstack}\ k
\end{aligned}
$$

- Remark: not $\mathrm{getstack}\ (\mathrm{call}\ f\ \vec{e} :: k) := \mathrm{getstack}\ k$

States

A state $\mathbf{S}(k, \phi, m)$ consists of a program context k, *focus* ϕ, and memory m

States

A state $\mathbf{S}(k, \phi, m)$ consists of a program context k, *focus* ϕ, and memory m

We consider the following focuses:

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

States

A state $\mathbf{S}(k, \phi, m)$ consists of a program context k, *focus* ϕ, and memory m

We consider the following focuses:

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

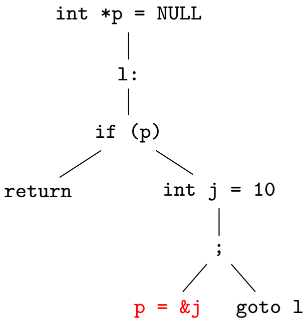
States

A state $\mathbf{S}(k, \phi, m)$ consists of a program context k, *focus* ϕ, and memory m

We consider the following focuses:

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

Example



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

- $k = [$
  $\square\,;\mathtt{goto}\ l,$
  $x_0 := \mathtt{int}\ 10\,;\square,$
  $\mathtt{block}_{b_j}\ \square,$
  $\mathtt{if}\ (\mathtt{load}\ x_0)\ \mathtt{return}\ \square,$
  $l : \square,$
  $x_0 := \mathtt{NULL}\,;\square,$
  $\mathtt{block}_{b_p}\ \square$
  $]$
- $\phi = (\nearrow,\, x_1 := x_0)$
- $m = \{b_p \mapsto \mathtt{ptr}\ b_j,\, b_j \mapsto 10\}$

The small step semantics

Some rules:

- $\mathbf{S}(k, (\searrow, e_l := e_r), m) \rightharpoondown \mathbf{S}(k, (\nearrow, e_l := e_r), m[a := v])$ provided that $[\![\, e_1 \,]\!]_{k,m} = \mathtt{ptr}\ a$, $[\![\, e_2 \,]\!]_{k,m} = v$, and $m\,a \neq \bot$

The small step semantics

Some rules:

- $\mathbf{S}(k, (\searrow, e_l := e_r), m) \rightharpoondown \mathbf{S}(k, (\nearrow, e_l := e_r), m[a := v])$ provided that $[\![e_1]\!]_{k,m} = \texttt{ptr}\ a$, $[\![e_2]\!]_{k,m} = v$, and $m\,a \neq \bot$
- $\mathbf{S}(k, (\searrow, s_1\,; s_2), m) \rightharpoondown \mathbf{S}((\square\,; s_2) :: k, (\searrow, s_1), m)$

The small step semantics

Some rules:

- $\mathbf{S}(k,\ (\searrow,\ e_l := e_r),\ m) \multimap \mathbf{S}(k,\ (\nearrow,\ e_l := e_r),\ m[a := v])$
  provided that $[\![\, e_1 \,]\!]_{k,m} = \mathtt{ptr}\ a$, $[\![\, e_2 \,]\!]_{k,m} = v$, and $m\,a \neq \bot$
- $\mathbf{S}(k,\ (\searrow,\ s_1\,;s_2),\ m) \multimap \mathbf{S}((\Box\,;s_2) :: k,\ (\searrow,\ s_1),\ m)$
- $\mathbf{S}((\Box\,;s_2) :: k,\ (\nearrow,\ s_1),\ m) \multimap \mathbf{S}((s_1\,;\Box) :: k,\ (\searrow,\ s_2),\ m)$

The small step semantics

Some rules:

- $\mathbf{S}(k,\ (\searrow,\ e_l := e_r),\ m) \rightharpoonup \mathbf{S}(k,\ (\nearrow,\ e_l := e_r),\ m[a := v])$
  provided that $[\![\ e_1\ ]\!]_{k,m} = \mathtt{ptr}\ a$, $[\![\ e_2\ ]\!]_{k,m} = v$, and $m\,a \neq \bot$
- $\mathbf{S}(k,\ (\searrow,\ s_1\,;s_2),\ m) \rightharpoonup \mathbf{S}((\square\,;s_2) :: k,\ (\searrow,\ s_1),\ m)$
- $\mathbf{S}((\square\,;s_2) :: k,\ (\nearrow,\ s_1),\ m) \rightharpoonup \mathbf{S}((s_1\,;\square) :: k,\ (\searrow,\ s_2),\ m)$
- $\mathbf{S}((s_1\,;\square) :: k,\ (\nearrow,\ s_2),\ m) \rightharpoonup \mathbf{S}(k,\ (\nearrow,\ s_1\,;s_2),\ m)$

The small step semantics

Some rules:

- $\mathbf{S}(k,\ (\searrow,\ e_l := e_r),\ m) \rightharpoonup \mathbf{S}(k,\ (\nearrow,\ e_l := e_r),\ m[a := v])$ provided that $[\![\, e_1 \,]\!]_{k,m} = \texttt{ptr}\ a$, $[\![\, e_2 \,]\!]_{k,m} = v$, and $m\,a \neq \bot$
- $\mathbf{S}(k,\ (\searrow,\ s_1\,;s_2),\ m) \rightharpoonup \mathbf{S}((\square\,;s_2) :: k,\ (\searrow,\ s_1),\ m)$
- $\mathbf{S}((\square\,;s_2) :: k,\ (\nearrow,\ s_1),\ m) \rightharpoonup \mathbf{S}((s_1\,;\square) :: k,\ (\searrow,\ s_2),\ m)$
- $\mathbf{S}((s_1\,;\square) :: k,\ (\nearrow,\ s_2),\ m) \rightharpoonup \mathbf{S}(k,\ (\nearrow,\ s_1\,;s_2),\ m)$
- $\mathbf{S}(k,\ (\searrow,\ \texttt{goto}\ l),\ m) \rightharpoonup \mathbf{S}(k,\ (\curvearrowright l,\ \texttt{goto}\ l),\ m)$

The small step semantics

Some rules:

- $\mathbf{S}(k,\ (\searrow,\ e_l := e_r),\ m) \rightharpoonup \mathbf{S}(k,\ (\nearrow,\ e_l := e_r),\ m[a := v])$ provided that $[\![\, e_1 \,]\!]_{k,m} = \mathtt{ptr}\ a$, $[\![\, e_2 \,]\!]_{k,m} = v$, and $m\,a \neq \bot$
- $\mathbf{S}(k,\ (\searrow,\ s_1\,;s_2),\ m) \rightharpoonup \mathbf{S}((\Box\,;s_2) :: k,\ (\searrow,\ s_1),\ m)$
- $\mathbf{S}((\Box\,;s_2) :: k,\ (\nearrow,\ s_1),\ m) \rightharpoonup \mathbf{S}((s_1\,;\Box) :: k,\ (\searrow,\ s_2),\ m)$
- $\mathbf{S}((s_1\,;\Box) :: k,\ (\nearrow,\ s_2),\ m) \rightharpoonup \mathbf{S}(k,\ (\nearrow,\ s_1\,;s_2),\ m)$
- $\mathbf{S}(k,\ (\searrow,\ \mathtt{goto}\ l),\ m) \rightharpoonup \mathbf{S}(k,\ (\curvearrowright l,\ \mathtt{goto}\ l),\ m)$
- $\mathbf{S}(k,\ (\curvearrowright l,\ l : s),\ m) \rightharpoonup \mathbf{S}((l : \Box) :: k,\ (\searrow,\ s),\ m)$

The small step semantics

Some rules:

- $\mathbf{S}(k,\ (\searrow,\ e_l := e_r),\ m) \rightharpoonup \mathbf{S}(k,\ (\nearrow,\ e_l := e_r),\ m[a := v])$
  provided that $[\![\ e_1\ ]\!]_{k,m} = \mathtt{ptr}\ a$, $[\![\ e_2\ ]\!]_{k,m} = v$, and $m\,a \neq \bot$
- $\mathbf{S}(k,\ (\searrow,\ s_1\,;s_2),\ m) \rightharpoonup \mathbf{S}((\square\,;s_2) :: k,\ (\searrow,\ s_1),\ m)$
- $\mathbf{S}((\square\,;s_2) :: k,\ (\nearrow,\ s_1),\ m) \rightharpoonup \mathbf{S}((s_1\,;\square) :: k,\ (\searrow,\ s_2),\ m)$
- $\mathbf{S}((s_1\,;\square) :: k,\ (\nearrow,\ s_2),\ m) \rightharpoonup \mathbf{S}(k,\ (\nearrow,\ s_1\,;s_2),\ m)$
- $\mathbf{S}(k,\ (\searrow,\ \mathtt{goto}\ l),\ m) \rightharpoonup \mathbf{S}(k,\ (\curvearrowright l,\ \mathtt{goto}\ l),\ m)$
- $\mathbf{S}(k,\ (\curvearrowright l,\ l:s),\ m) \rightharpoonup \mathbf{S}((l:\square) :: k,\ (\searrow,\ s),\ m)$
- $\mathbf{S}(k,\ (\searrow,\ f(\vec{e})),\ m) \rightharpoonup \mathbf{S}(\mathsf{call}\ f\ \vec{e} :: k,\ \overline{\mathsf{call}}\ f\ \vec{v},\ m)$
  provided that $[\![\ e_i\ ]\!]_{k,m} = v_i$ for each i

The small step semantics

Lemma

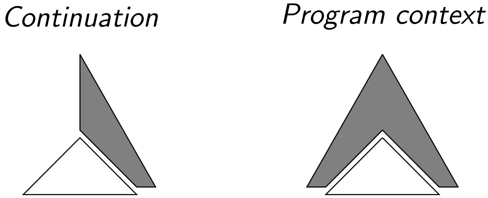
The small step semantics behaves as traversing through a zipper. That is, if

$$\mathbf{S}(k,\ (d,\ s),\ m) \rightharpoonup^*_k \mathbf{S}(k,\ (d',\ s'),\ m')$$

then $s = s'$.

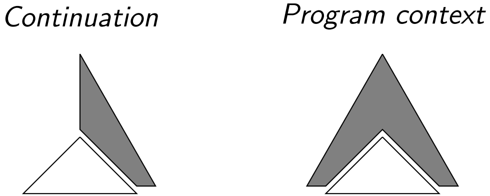
Program contexts versus continuations

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



Program contexts versus continuations

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



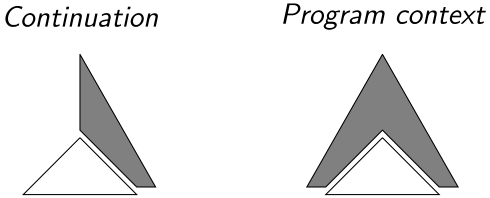
- Looping constructs do not have to duplicate code

$\mathbf{S}(k, (\searrow, \texttt{while}(e)\ s), m) \rightharpoonup \mathbf{S}((\texttt{while}(e)\square) :: k, (\searrow, s), m)$

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



- Looping constructs do not have to duplicate code

$$\mathbf{S}(k, (\searrow, \texttt{while}(e)\ s), m) \rightharpoonup \mathbf{S}((\texttt{while}(e)\square) :: k, (\searrow, s), m)$$

provided that $[\![\, e \,]\!]_{k,m} = v$ and istrue v

- Program contexts implicitly contain the stack

Hoare triples

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

Hoare triples

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

Extended Hoare 'triples' (1)

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:

- Δ 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:

- Δ 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

Extended Hoare ‘triples’ (1)

Our *Hoare sextuples* are of the shape

$$\Delta;\ J;\ R \vdash \{P\}\, s\, \{Q\}$$

where:

- Δ 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 $\searrow$, $\nearrow$, $\curvearrowright$ and $\upuparrows$

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 $\searrow$, $\nearrow$, $\curvearrowright$ and $\upuparrows$
- We thus treat the sextuple as

$$\Delta;\, \bar{P} \vdash s$$

where $\bar{P}\searrow\, = P$, $\bar{P}\nearrow\, = Q$, $\bar{P}\,(\curvearrowright l) = J\, l$ and $\bar{P}\upuparrows\, = R$

Some Hoare rules

Weakening:

$$\frac{P \rightarrow P' \qquad \Delta;\ J;\ R \vdash \{P'\}\, s\, \{Q'\} \qquad Q' \rightarrow Q}{\Delta;\ J;\ R \vdash \{P\}\, s\, \{Q\}}$$

Some Hoare rules

Weakening:

$$\frac{P \to P' \qquad \Delta;\ J;\ R \vdash \{P'\}\, s\, \{Q'\} \qquad Q' \to Q}{\Delta;\ J;\ R \vdash \{P\}\, s\, \{Q\}}$$

Composition:

$$\frac{\Delta;\ J;\ R \vdash \{P\}\, s_1\, \{P'\} \qquad \Delta;\ J;\ R \vdash \{P'\}\, s_2\, \{Q\}}{\Delta;\ J;\ R \vdash \{P\}\, s_1\, ;\, s_2\, \{Q\}}$$

Some Hoare rules

Weakening:

$$\frac{P \to P' \qquad \Delta;\ J;\ R \vdash \{P'\}\, s\, \{Q'\} \qquad Q' \to Q}{\Delta;\ J;\ R \vdash \{P\}\, s\, \{Q\}}$$

Composition:

$$\frac{\Delta;\ J;\ R \vdash \{P\}\, s_1\, \{P'\} \qquad \Delta;\ J;\ R \vdash \{P'\}\, s_2\, \{Q\}}{\Delta;\ J;\ R \vdash \{P\}\, s_1\, ;\, s_2\, \{Q\}}$$

Non-local control:

$$\frac{}{\Delta;\ J;\ R \vdash \{R\}\, \texttt{return}\, \{Q\}}$$

Some Hoare rules

Weakening:

$$\frac{P \to P' \qquad \Delta;\ J;\ R \vdash \{P'\}\, s\, \{Q'\} \qquad Q' \to Q}{\Delta;\ J;\ R \vdash \{P\}\, s\, \{Q\}}$$

Composition:

$$\frac{\Delta;\ J;\ R \vdash \{P\}\, s_1\, \{P'\} \qquad \Delta;\ J;\ R \vdash \{P'\}\, s_2\, \{Q\}}{\Delta;\ J;\ R \vdash \{P\}\, s_1\,;s_2\, \{Q\}}$$

Non-local control:

$$\frac{}{\Delta;\ J;\ R \vdash \{R\}\, \texttt{return}\, \{Q\}}$$

$$\frac{}{\Delta;\ J;\ R \vdash \{J\ l\}\, \texttt{goto}\ l\, \{Q\}} \qquad \frac{\Delta;\ J;\ R \vdash \{J\ l\}\, s\, \{Q\}}{\Delta;\ J;\ R \vdash \{J\ l\}\, l : s\, \{Q\}}$$

Separation logic

$$\mathsf{emp} := \lambda\rho\, m\, .\, m = \emptyset$$

Separation logic

$$
\begin{aligned}
\mathsf{emp} &:= \lambda \rho\, m\,.\, m = \emptyset \\
P * Q &:= \lambda \rho\, m\,.\, \exists m_1\, m_2\,. \\
&\qquad\qquad m = m_1 \cup m_2 \wedge m_1 \perp m_2 \wedge P\, \rho\, m_1 \wedge Q\, \rho\, m_2
\end{aligned}
$$

Separation logic

$$
\begin{aligned}
\mathsf{emp} &:= \lambda \rho\, m \,.\, m = \emptyset \\
P * Q &:= \lambda \rho\, m \,.\, \exists m_1\, m_2 \,. \\
&\qquad\qquad m = m_1 \cup m_2 \wedge m_1 \perp m_2 \wedge P\, \rho\, m_1 \wedge Q\, \rho\, m_2 \\
e_1 \mapsto e_2 &:= \exists v_1\, v_2 \,.\, e_1 \Downarrow v_1 \wedge e_2 \Downarrow v_2 \wedge m = \{(v_1,\ v_2)\}
\end{aligned}
$$

The frame rule

Used for local reasoning

$$\frac{\Delta;\ J;\ R \vdash \{P\}\, s\, \{Q\}}{\Delta;\ J * A;\ R * A \vdash \{P * A\}\, s\, \{Q * A\}}$$

The frame rule

Used for local reasoning

$$\frac{\Delta;\ J;\ R \vdash \{P\}\, s\, \{Q\}}{\Delta;\ J * A;\ R * A \vdash \{P * A\}\, s\, \{Q * A\}}$$

Using our alternative notation:

$$\frac{\Delta;\ \bar{P} \vdash s}{\Delta;\ \bar{P} * A \vdash s}$$

The block scope variable rule

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

$$\frac{\Delta;\, J\!\uparrow * x_0 \mapsto \text{-};\, R\!\uparrow * x_0 \mapsto \text{-} \vdash \{P\!\uparrow * x_0 \mapsto \text{-}\}\, s\, \{Q\!\uparrow * x_0 \mapsto \text{-}\}}{\Delta;\, J;\, R \vdash \{P\}\, \texttt{block}\, s\, \{Q\}}$$

The block scope variable rule

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

$$\frac{\Delta;\ J\uparrow * x_0 \mapsto \text{-};\ R\uparrow * x_0 \mapsto \text{-} \vdash \{P\uparrow * x_0 \mapsto \text{-}\}\, s\, \{Q\uparrow * x_0 \mapsto \text{-}\}}{\Delta;\ J;\ R \vdash \{P\}\,\texttt{block}\, s\, \{Q\}}$$

Using our alternative notation

$$\frac{\Delta;\ \bar{P}\uparrow * x_0 \mapsto \text{-} \vdash s}{\Delta;\ \bar{P} \vdash \texttt{block}\ s}$$

Soundness of the axiomatic semantics

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

Soundness of the axiomatic semantics

- Define $\Delta;\ J;\ R \vDash \{P\}\, s\, \{Q\}$ in terms of operational semantics
- Prove $\Delta;\ J;\ R \vdash \{P\}\, s\, \{Q\}$ implies $\Delta;\ J;\ R \vDash \{P\}\, s\, \{Q\}$

Soundness of the axiomatic semantics

- Define $\Delta;\, J;\, R \vDash \{P\}\, s\, \{Q\}$ in terms of operational semantics
- Prove $\Delta;\, J;\, R \vdash \{P\}\, s\, \{Q\}$ implies $\Delta;\, J;\, R \vDash \{P\}\, s\, \{Q\}$
- Tricky definition of $\Delta;\, J;\, R \vDash \{P\}\, s\, \{Q\}$ 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

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

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/`