Separation Logic for Non-local Control Flow and Block Scope Variables

Robbert Krebbers
Joint work with Freek Wiedijk

Radboud University Nijmegen

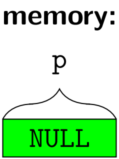
March 19, 2013 @ FoSSaCS, Rome, Italy

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?

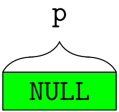
```
int *p = NULL;
l: if (p) {
  return (*p);
} else {
  int j = 10;
  p = &j;
  goto l;
}
```



What is this program supposed to do?

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

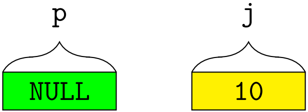
memory:



What is this program supposed to do?

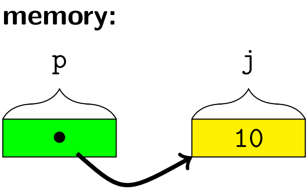
```
int *p = NULL;
l: if (p) {
  return (*p);
} else {
  int j = 10;
  p = &j;
  goto l;
}
```

memory:



What is this program supposed to do?

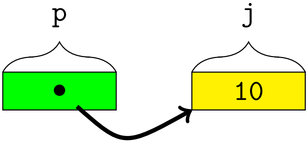
```
int *p = NULL;
l: if (p) {
  return (*p);
} else {
  int j = 10;
  p = &j;
  goto l;
}
```



What is this program supposed to do?

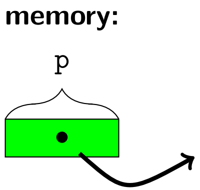
```
int *p = NULL;
l: if (p) {
  return (*p);
} else {
  int j = 10;
  p = &j;
  goto l;
}
```

memory:



What is this program supposed to do?

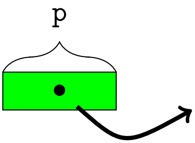
```
int *p = NULL;
l: if (p) {
  return (*p);
} else {
  int j = 10;
  p = &j;
  goto l;
}
```



What is this program supposed to do?

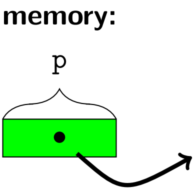
```
int *p = NULL;
l: if (p) {
  return (*p);
} else {
  int j = 10;
  p = &j;
  goto l;
}
```

memory:



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*

Undefined behavior in C

- *Undefined behavior* is shown by “wrong” C programs
- Programs may do anything on undefined behavior
- It allows compilers to omit (expensive) dynamic checks

Undefined behavior in C

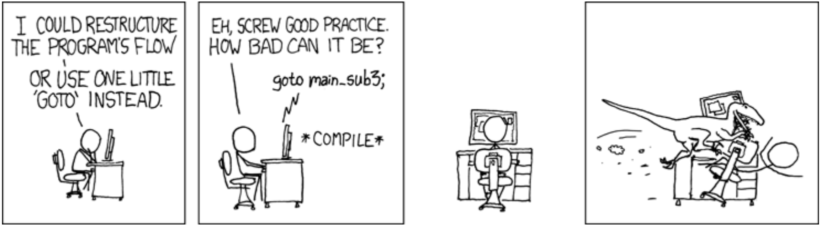
- *Undefined behavior* is shown by "wrong" C programs
- Programs may do anything on undefined behavior
- It allows compilers to omit (expensive) dynamic checks
- It cannot be checked for statically
- Not accounting for it means that
  - programs can be proven to be correct with respect to the formal semantics ...
  - whereas they may crash when compiled with an actual compiler

Undefined behavior in C

- *Undefined behavior* is shown by “wrong” C programs
- Programs may do anything on undefined behavior
- It allows compilers to omit (expensive) dynamic checks
- It cannot be checked for statically
- Not accounting for 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

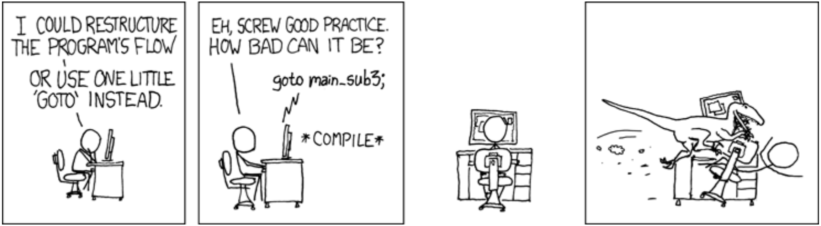
This talk: undefined behavior due to dangling pointers by non-local control flow and block scopes

Goto considered harmful?



http://xkcd.com/292/

Goto considered harmful?



`http://xkcd.com/292/`

Not necessarily:

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

Contribution

A concise small step operational, and axiomatic, semantics for goto, supporting:

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

Approach

- Execute gotos and returns in small steps
  - Not so much to search for labels, ...
  - but to naturally perform required allocations and deallocations

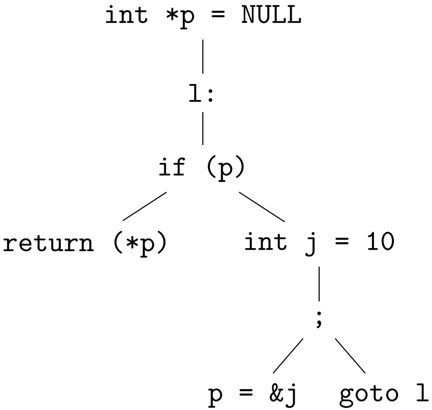
Approach

- Execute gotos and returns in small steps
  - Not so much to search for labels, ...
  - but to naturally perform required allocations and deallocations
- Traversal through the AST in the following directions:
  - $\searrow$ downwards to the next statement
  - $\nearrow$ upwards to the next statement

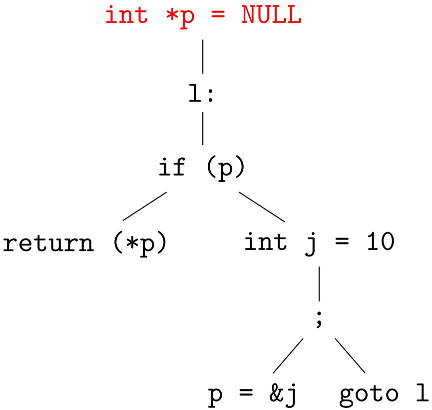
Approach

- Execute gotos and returns in small steps
  - Not so much to search for labels, ...
  - but to naturally perform required allocations and deallocations
- Traversal through the AST in the following 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 of the statement after a `return`

Example



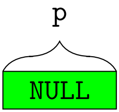
Example



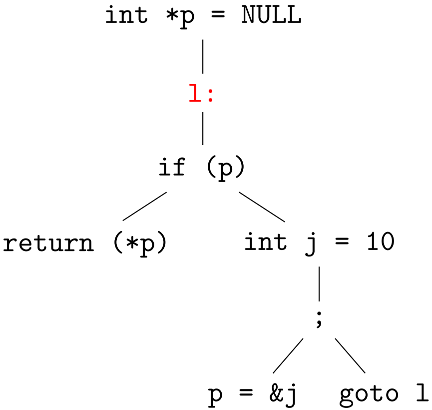
direction:

↘

memory:

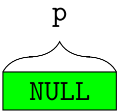


Example

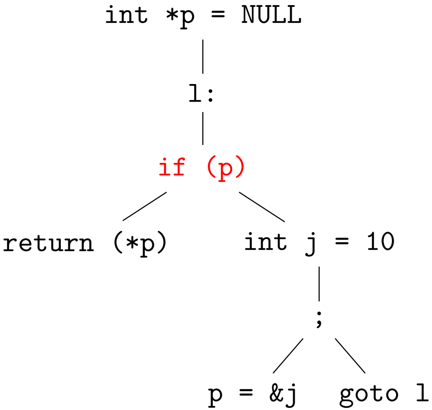


direction:

memory:

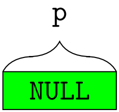


Example

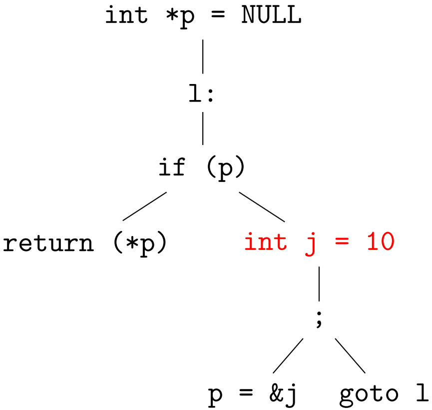


direction:

memory:

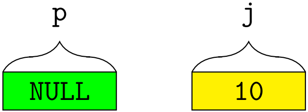


Example

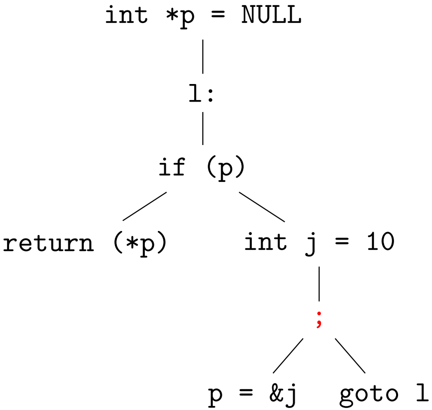


direction:

memory:

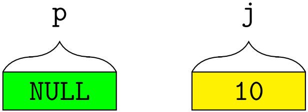


Example

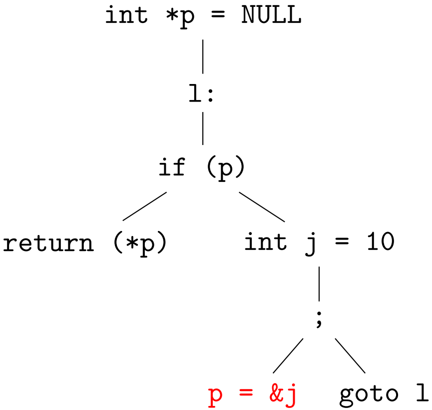


direction:

memory:

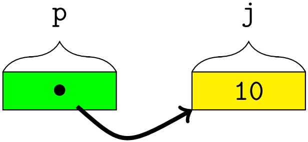


Example

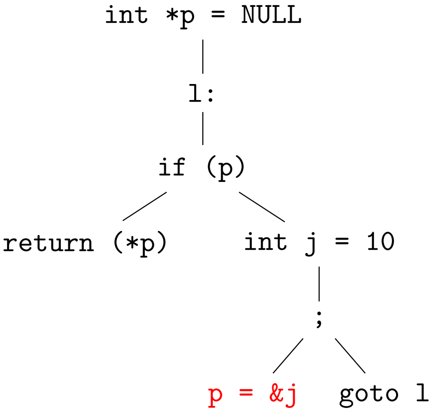


direction:

memory:



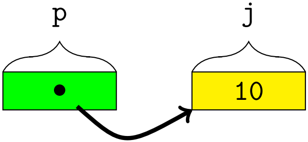
Example



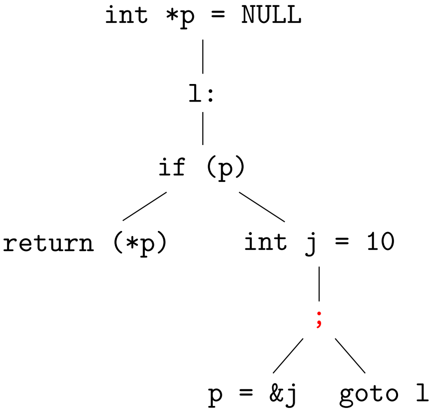
direction:

↗

memory:



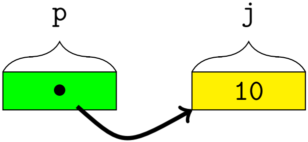
Example



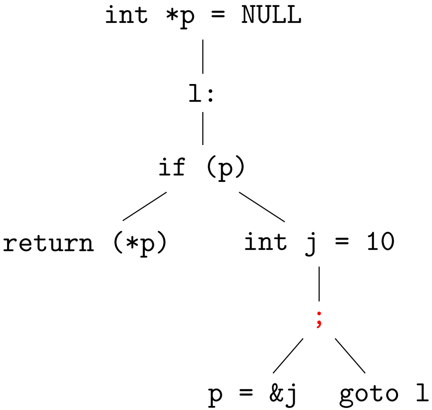
direction:

↗

memory:



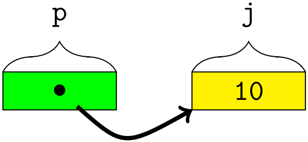
Example



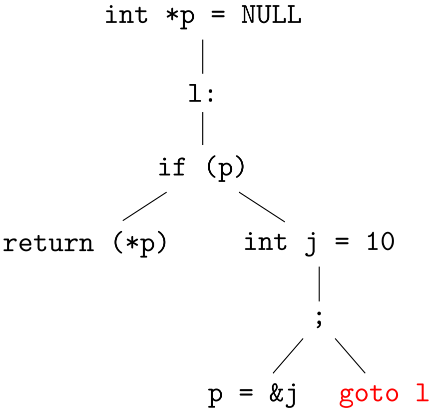
direction:

↘

memory:

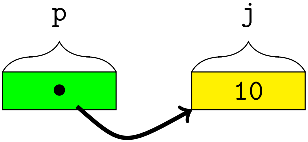


Example

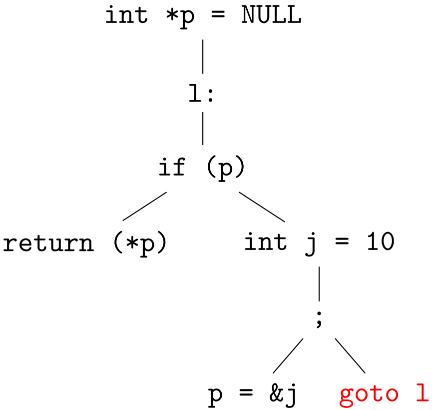


direction:

memory:



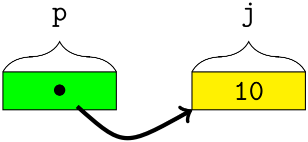
Example



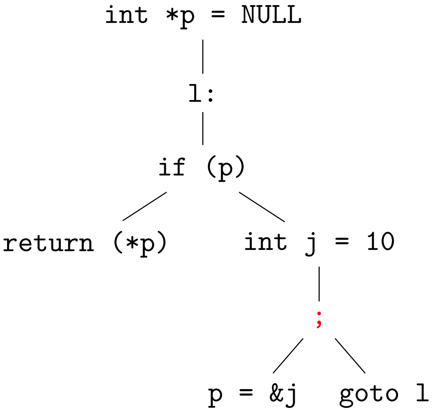
direction:

$\curvearrowright$ `l`

memory:



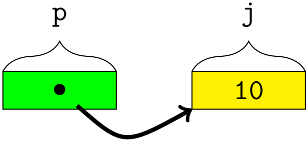
Example



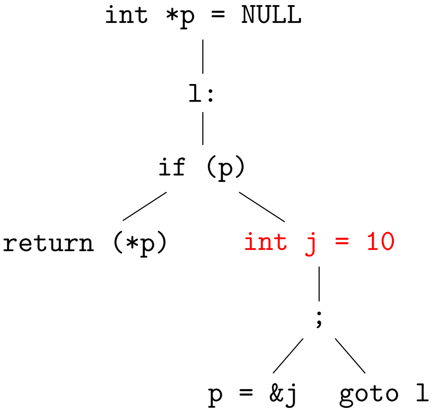
direction:

$\curvearrowright$ `l`

memory:



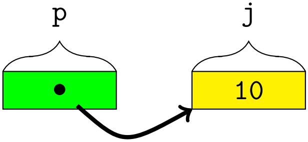
Example



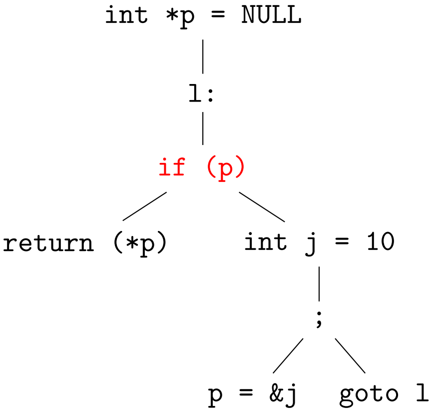
direction:

↷ l

memory:



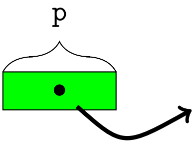
Example



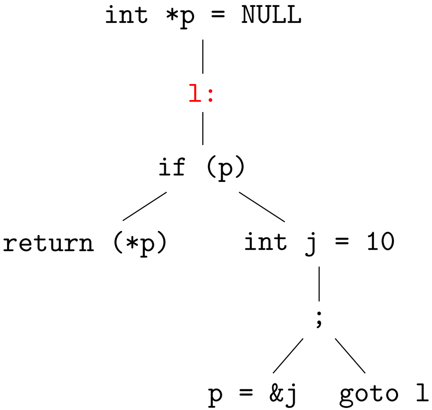
direction:

↷ l

memory:



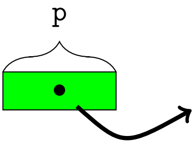
Example



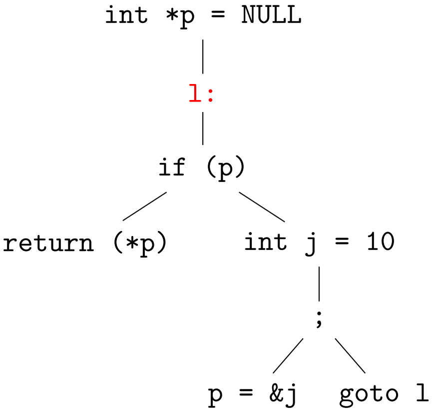
direction:

↷ l

memory:



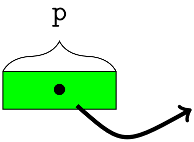
Example



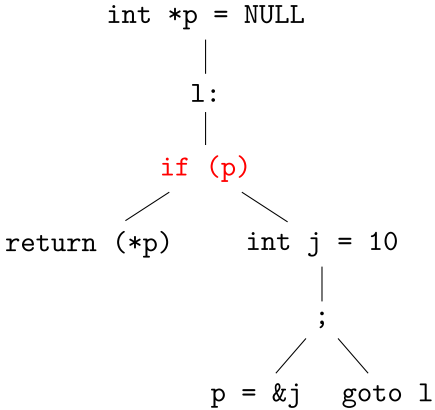
direction:

$\searrow$

memory:



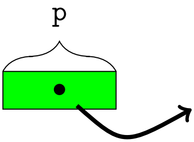
Example



direction:

↘

memory:

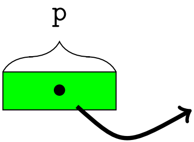


Example

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

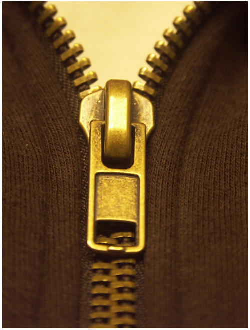
direction:

memory:



How to model the current *location* in the program

Huet's zipper



Purely functional way to store a pointer into a data structure

Statement contexts

- Statements:

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

Statement contexts

- Statements:

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

- The `block` construct is unnamed as we use De Bruijn indexes

Statement contexts

- Statements:

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

- The `block` construct is unnamed as we use De Bruijn indexes
- Singular statement contexts:

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

Statement contexts

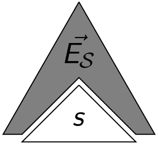
- Statements:

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

- The `block` construct is unnamed as we use De Bruijn indexes
- 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

- Make the zipper stateful to also contain *the stack* (to assign memory indexes to local variables)
- Extend the zipper dynamically on function calls

Program contexts

- Make the zipper stateful to also contain *the stack* (to assign memory indexes to local variables)
- Extend the zipper dynamically on function calls
- Program contexts k are lists of singular program contexts:

$$E ::= E_{\mathcal{S}} \mid \texttt{block}_b\,\Box \mid \ \ldots$$

where $\texttt{block}_b\,\Box$ associates a block scope variable with its corresponding memory index b

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 $f(\vec{v})$

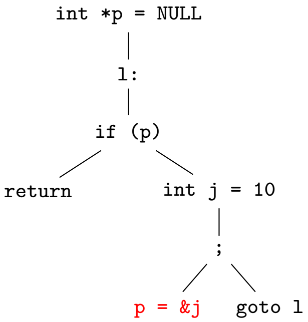
States

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
- $\overline{\text{call}}\, f\ \vec{v}$ calling a function $f(\vec{v})$
- $\overline{\text{return}}$ returning from a function

Example



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

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

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 $s = s'$.

The small step semantics

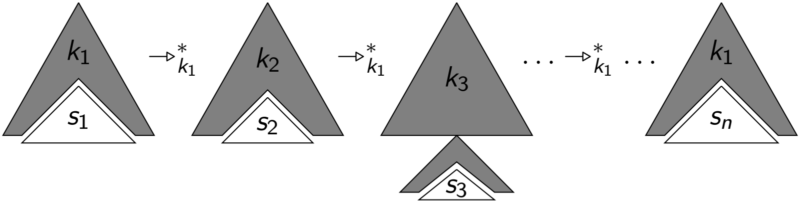
Lemma

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 $s = s'$.

In a picture: if



then $s_1 = s_n$.

Hoare sextuples

Our *Hoare sextuples* are of the shape

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

Hoare sextuples

Our *Hoare sextuples* are of the shape

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

where:

- $\{P\}\, s\, \{Q\}$ is a Hoare triple, as usual

Hoare sextuples

Our *Hoare sextuples* are of the shape

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

where:

- $\{P\}\, s\, \{Q\}$ is a Hoare triple, as usual
- Δ maps function names to their pre- and post-conditions

Hoare sextuples

Our *Hoare sextuples* are of the shape

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

where:

- $\{P\}\, s\, \{Q\}$ is a Hoare triple, as usual
- Δ maps function names to their pre- and post-conditions
- J maps labels to their jumping condition
  When executing a `goto` l, the assertion $J\, l$ has to hold

Hoare sextuples

Our *Hoare sextuples* are of the shape

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

where:

- $\{P\}\, s\, \{Q\}$ is a Hoare triple, as usual
- Δ maps function names to their pre- and post-conditions
- 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`

Hoare sextuples

Our *Hoare sextuples* are of the shape

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

where:

- $\{P\}\, s\, \{Q\}$ is a Hoare triple, as usual
- Δ maps function names to their pre- and post-conditions
- 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`

Remark: the assertions P, Q, J and R correspond to the directions $\searrow$, $\nearrow$, $\curvearrowright$ and $\upuparrows$ of traversal

Some Hoare rules

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

Composition:

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

Goto:

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

Some Hoare rules

Composition:

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

Goto:

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

Return:

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

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 block scope variable rule

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

When entering a block:

- The De Bruijn indexes are lifted: $(\_)\uparrow$
- The memory is extended: $(\_) * x_0 \mapsto \text{-}$

The block scope variable rule

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

When entering a block:

- The De Bruijn indexes are lifted: $(\_)\uparrow$
- The memory is extended: $(\_) * x_0 \mapsto \text{-}$

When leaving a block: the reverse

The block scope variable rule

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

When entering a block:

- The De Bruijn indexes are lifted: $(\_)\uparrow$
- The memory is extended: $(\_) * x_0 \mapsto \text{-}$

When leaving a block: the reverse

Important: using De Bruijn indexes avoids shadowing

Formalization in Coq

- Extremely useful for debugging

Proved in Coq

Formalization in Coq

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

Proved in Coq

Formalization in Coq

- Extremely useful for debugging
- Notations close to those on paper
- Also supports while and functions with return values
- Uses lots of automation
- 3500 lines of code

Proved in Coq

Future research

- Expressions with side effects *(recently finished)*
- Machine integers *(recently finished)*

Future research

- Expressions with side effects *(recently finished)*
- Machine integers *(recently finished)*
- The C type system *(in progress)*
- Non-aliasing restrictions *(in progress)*

Future research

- Expressions with side effects *(recently finished)*
- Machine integers *(recently finished)*
- The C type system *(in progress)*
- Non-aliasing restrictions *(in progress)*
- Verification condition generator in Coq

Future research

- Expressions with side effects *(recently finished)*
- Machine integers *(recently finished)*
- The C type system *(in progress)*
- Non-aliasing restrictions *(in progress)*
- Verification condition generator in Coq
- Correspondence with CompCert

Questions

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