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

Robbert Krebbers
Joint work with Freek Wiedijk

Radboud University Nijmegen

February 4, 2013 @ Gallium, INRIA Rocquencourt, France

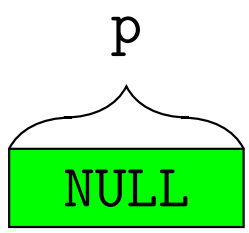
What is this program supposed to do?

```
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;
}
```

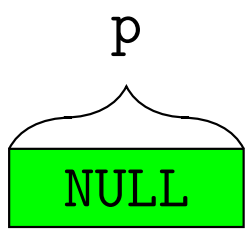
memory:



What is this program supposed to do?

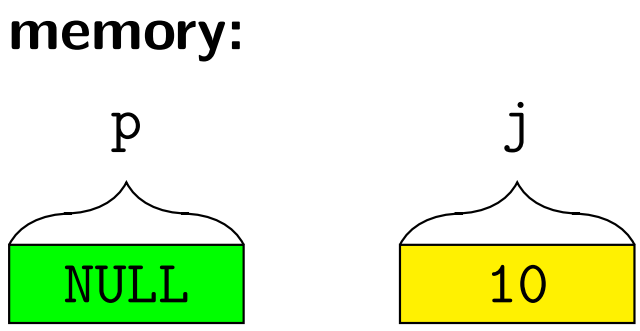
```
int *p = NULL;
l: if (p) {
  return (*p);
} else {
  int j = 10;
  p = &j;
  goto l;
}
```

memory:



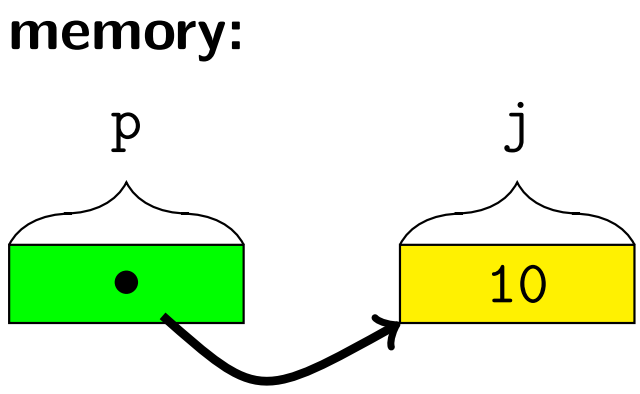
What is this program supposed to do?

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



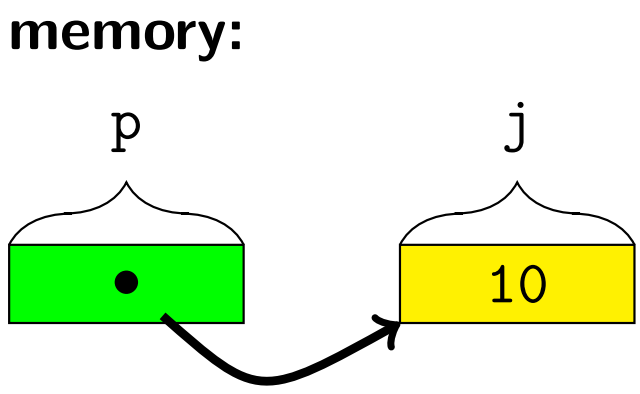
What is this program supposed to do?

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



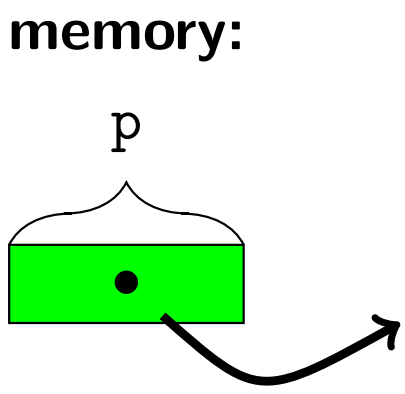
What is this program supposed to do?

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



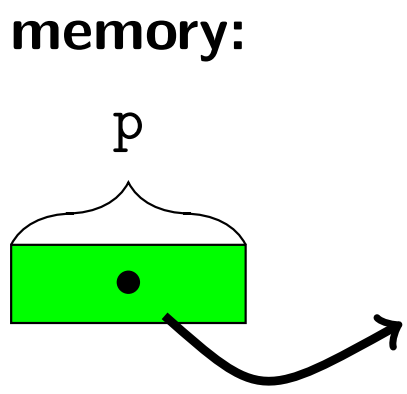
What is this program supposed to do?

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



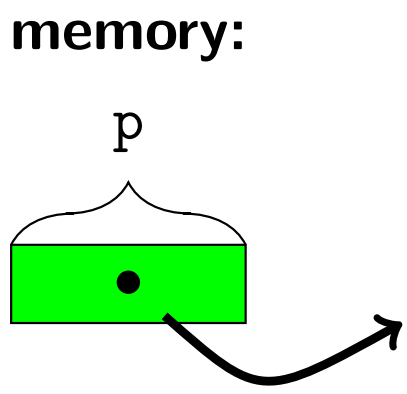
What is this program supposed to do?

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



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

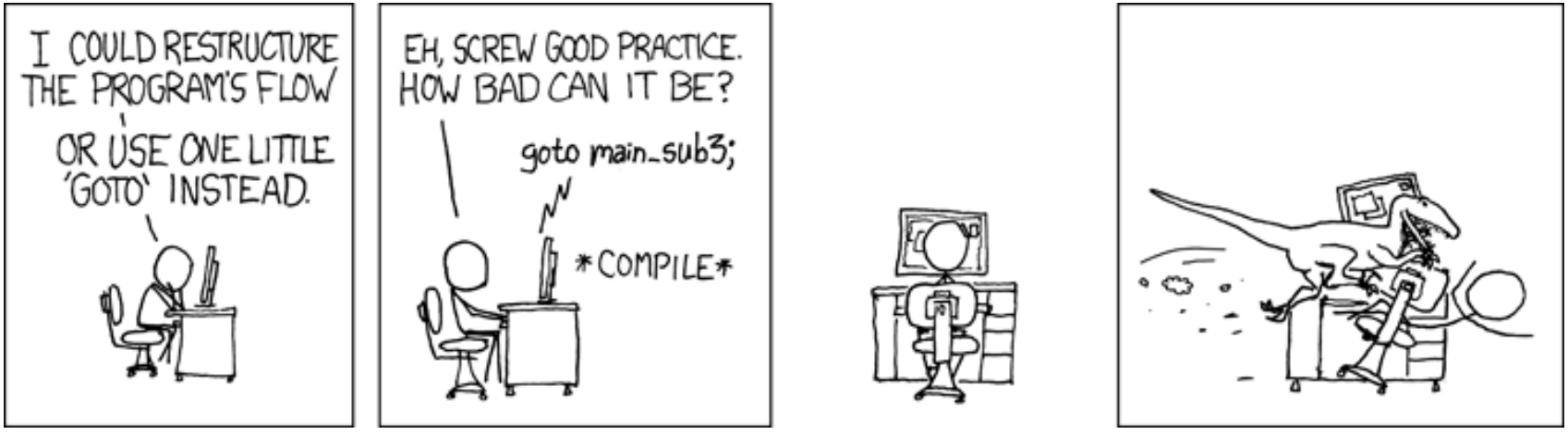
Why is catching undefined behavior important

- 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

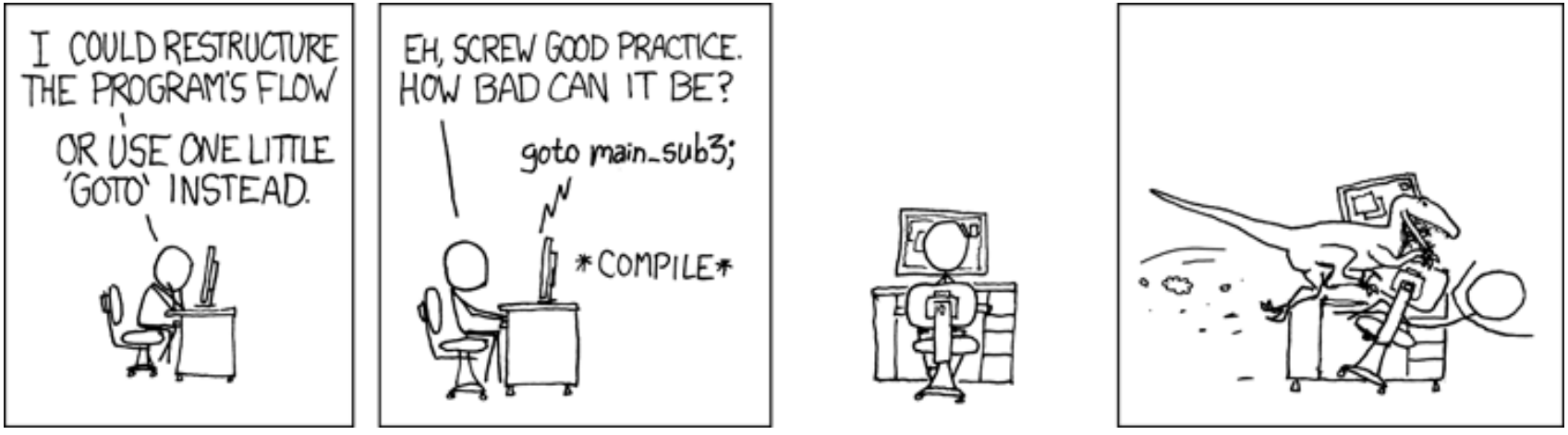
- C allows a program to do **anything** on undefined behavior
- It cannot be checked statically
- Not catching it means that
  - programs can be proven to be correct with respect to the formal semantics . . .
  - whereas they may crash when compiled with an actual compiler

Goto considered harmful?



http://xkcd.com/292/

Goto considered harmful?



http://xkcd.com/292/

Not necessarily:

 $\vdash \{P\} \ldots$ `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
$ find ./ -name \*.c -exec cat '{}' \; | wc -l
11152601
```

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`

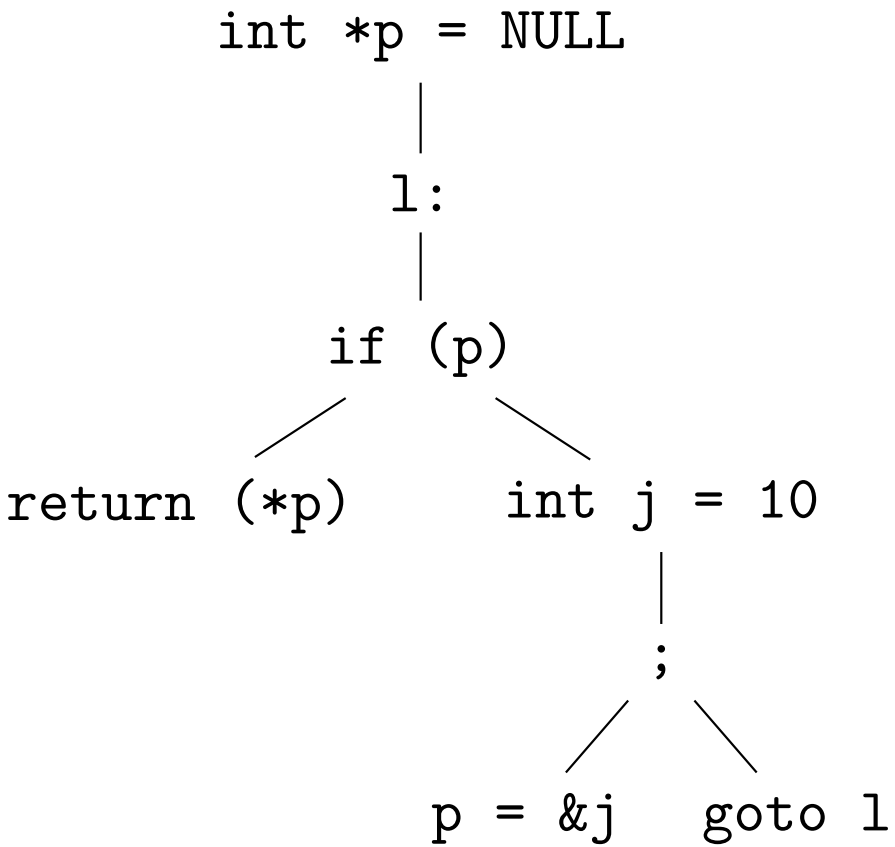
Approach

- 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`
  - $\Uparrow$ to the top after a `return`

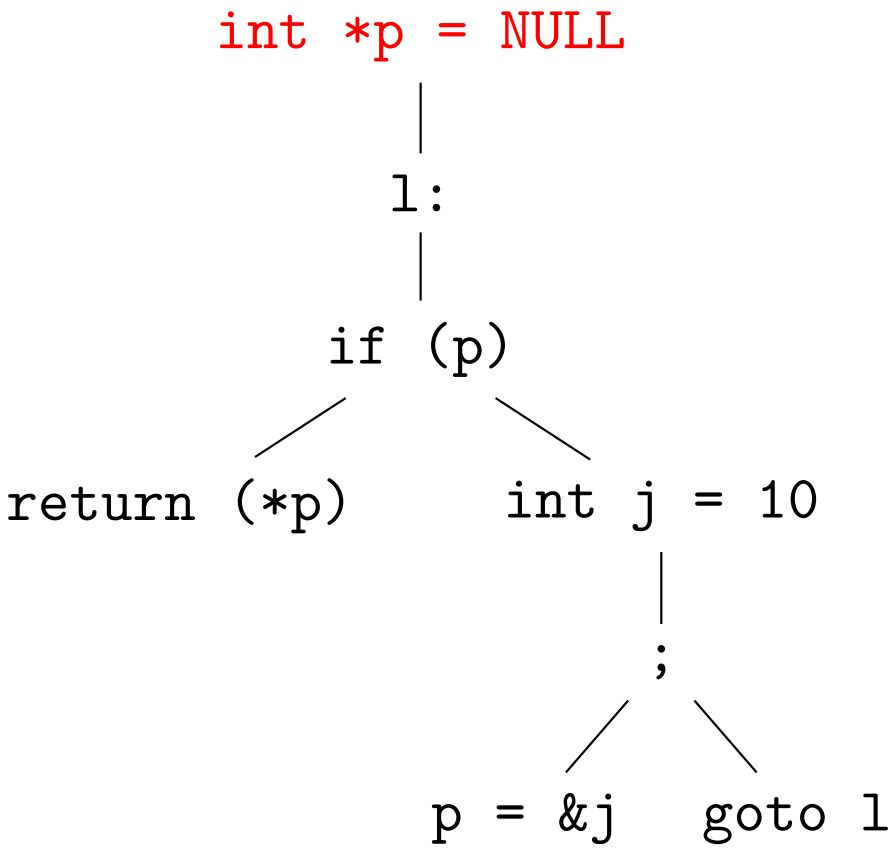
Approach

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

Example

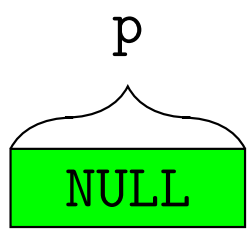


Example

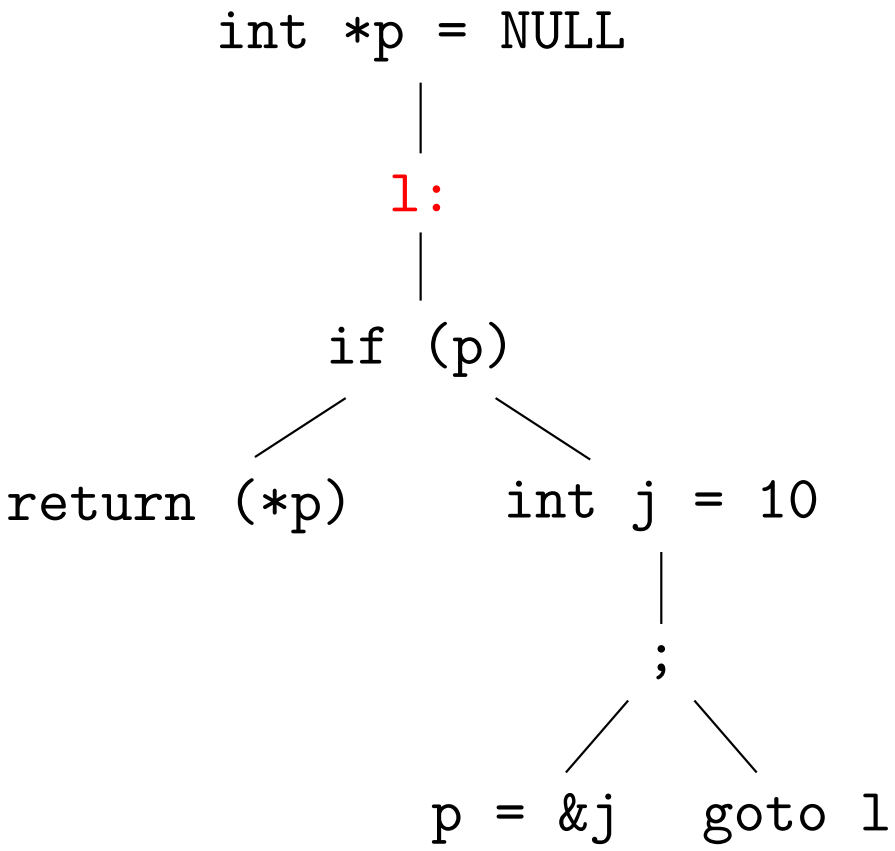


direction:

memory:

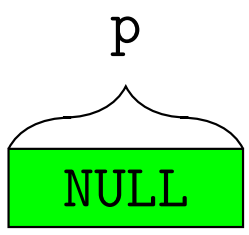


Example

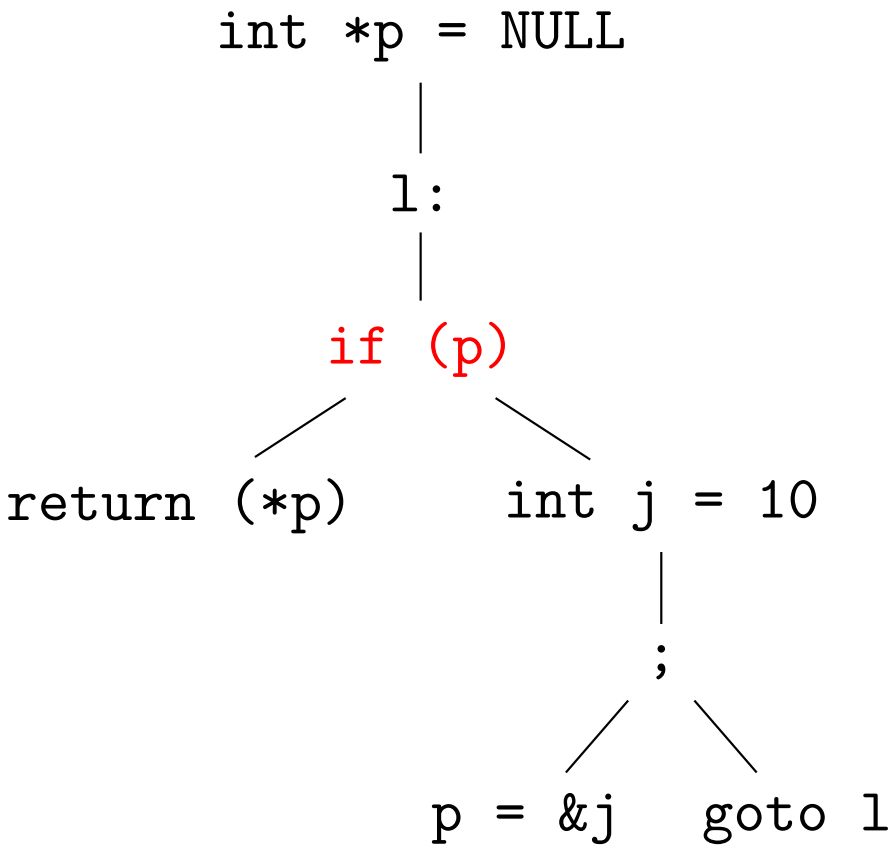


direction:

memory:

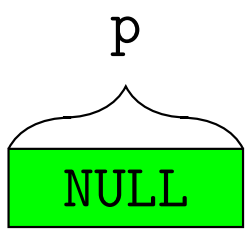


Example

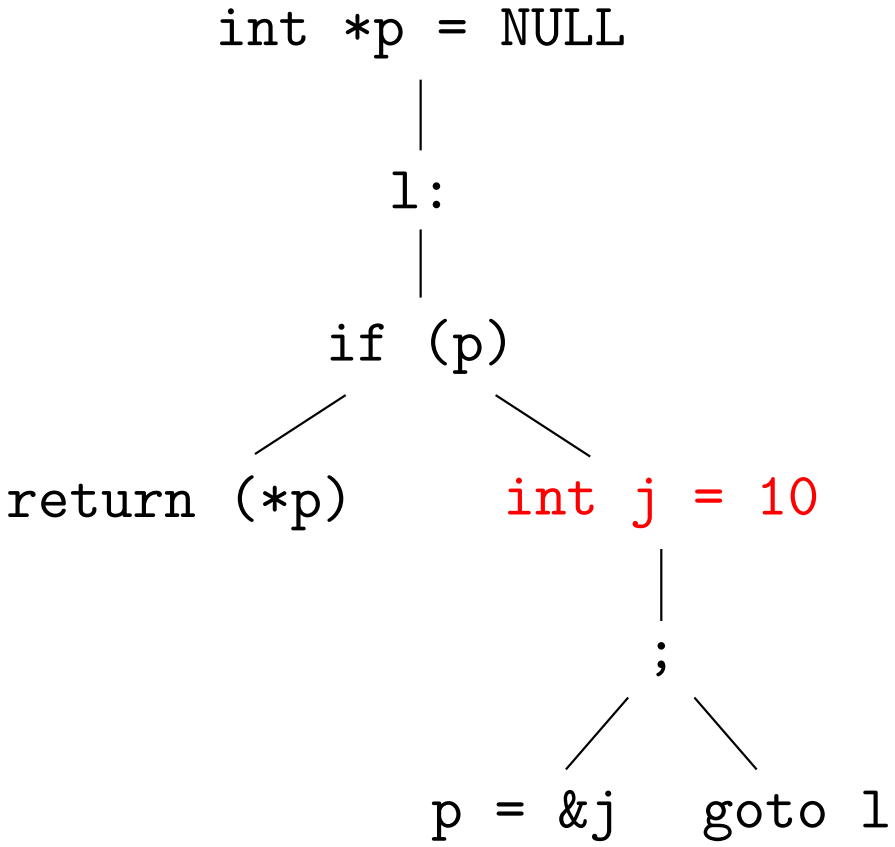


direction:

memory:

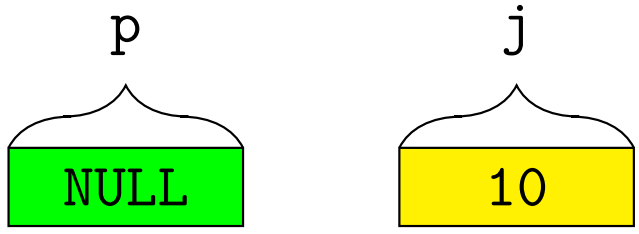


Example

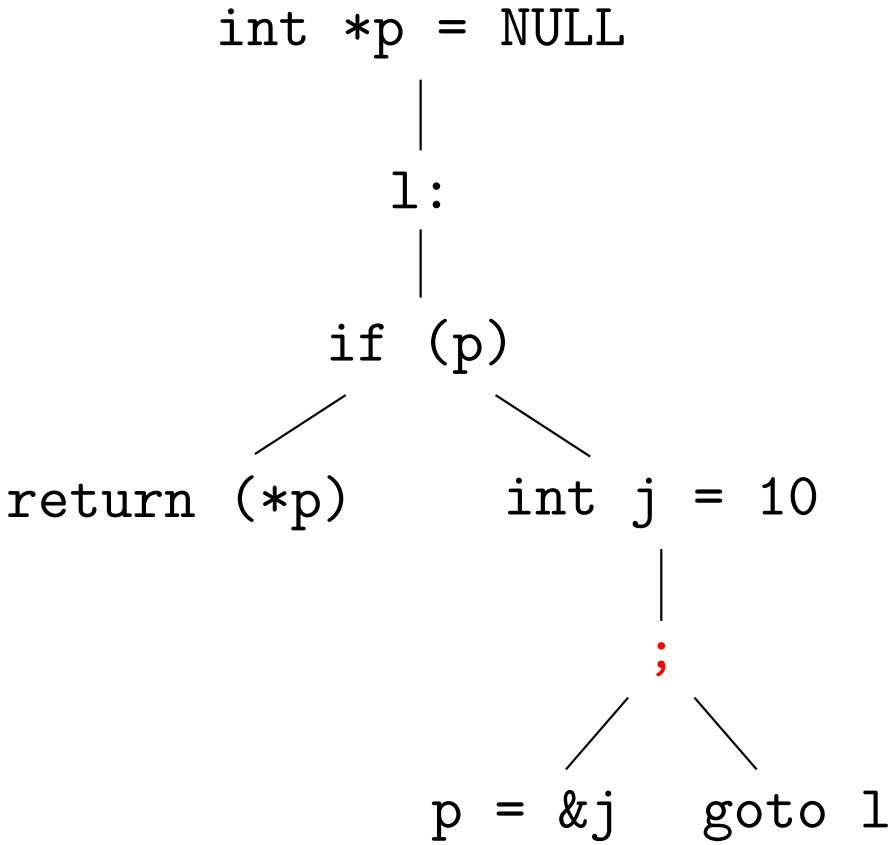


direction:

memory:

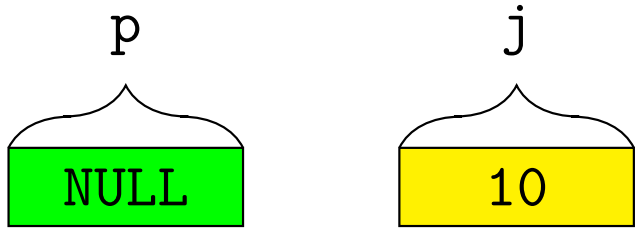


Example

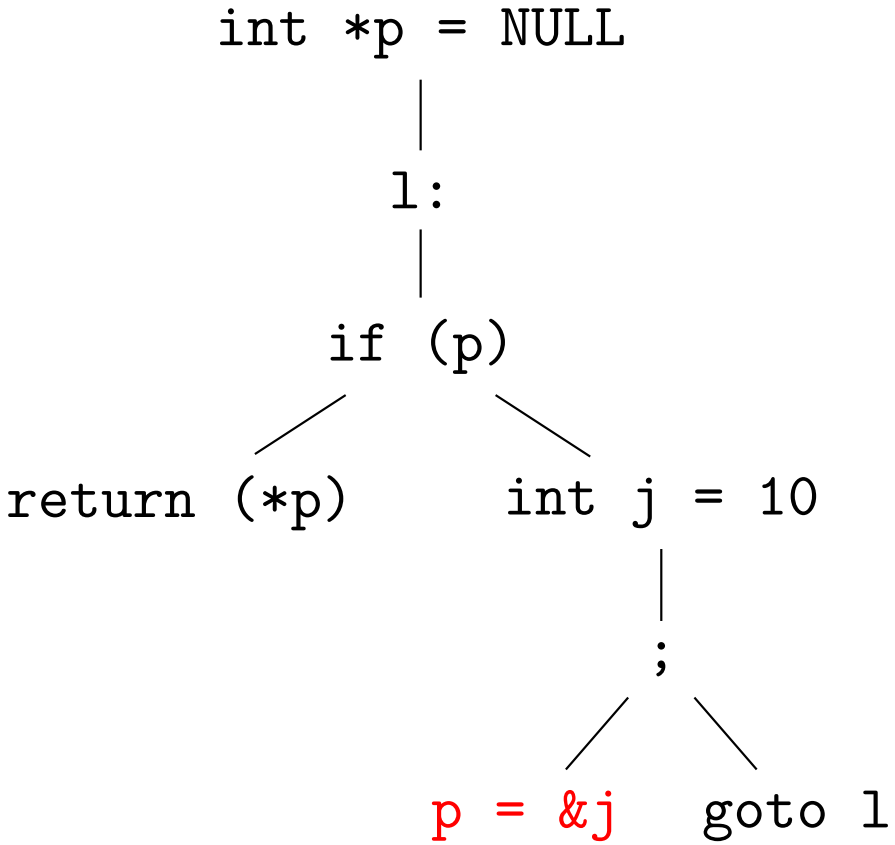


direction:

memory:

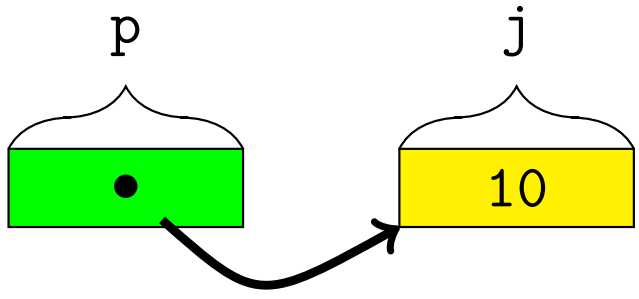


Example

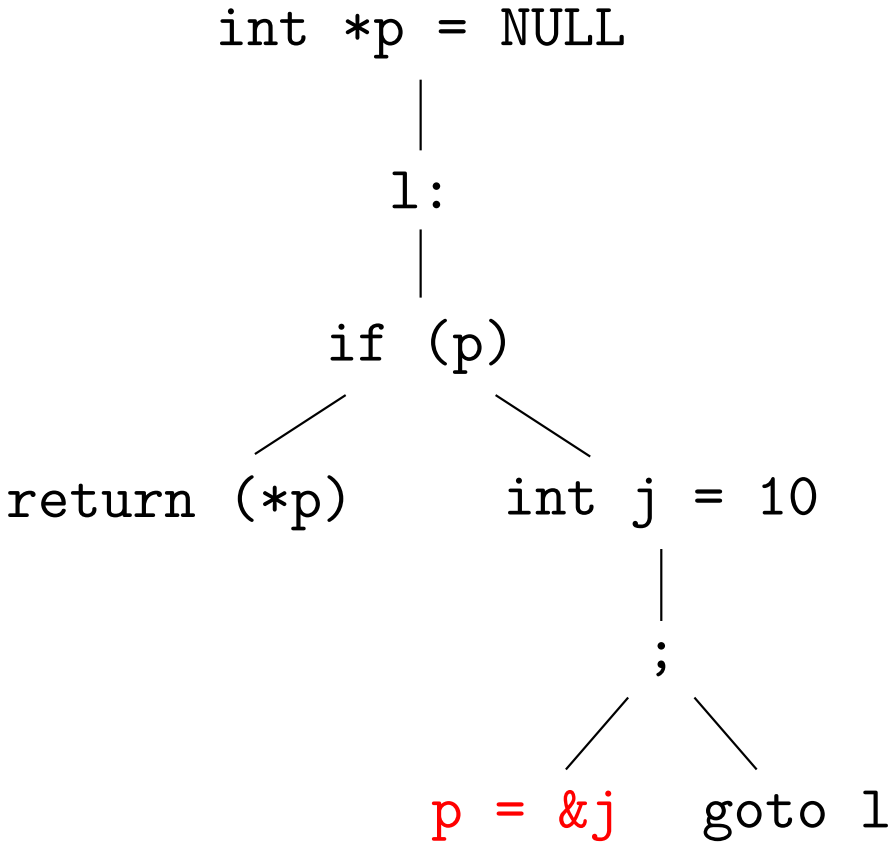


direction:

memory:

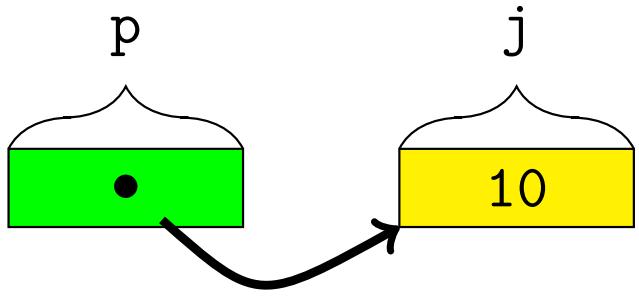


Example

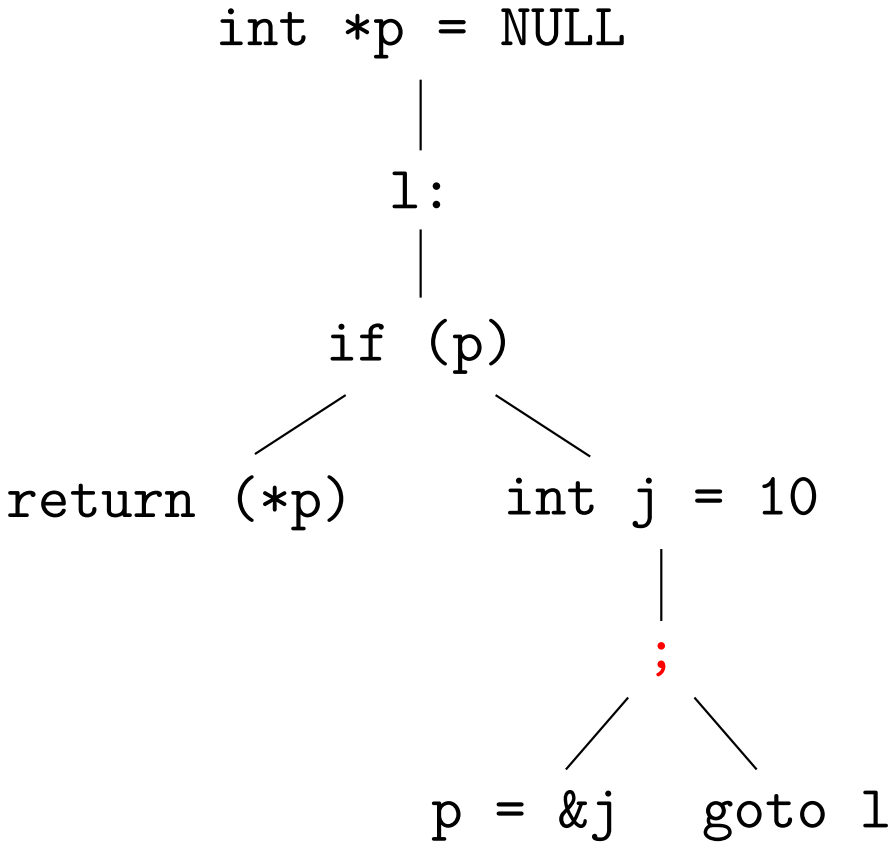


direction:

memory:

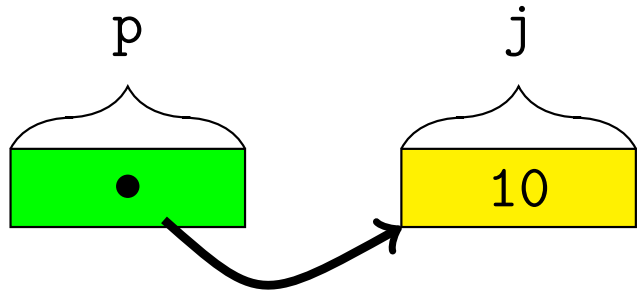


Example

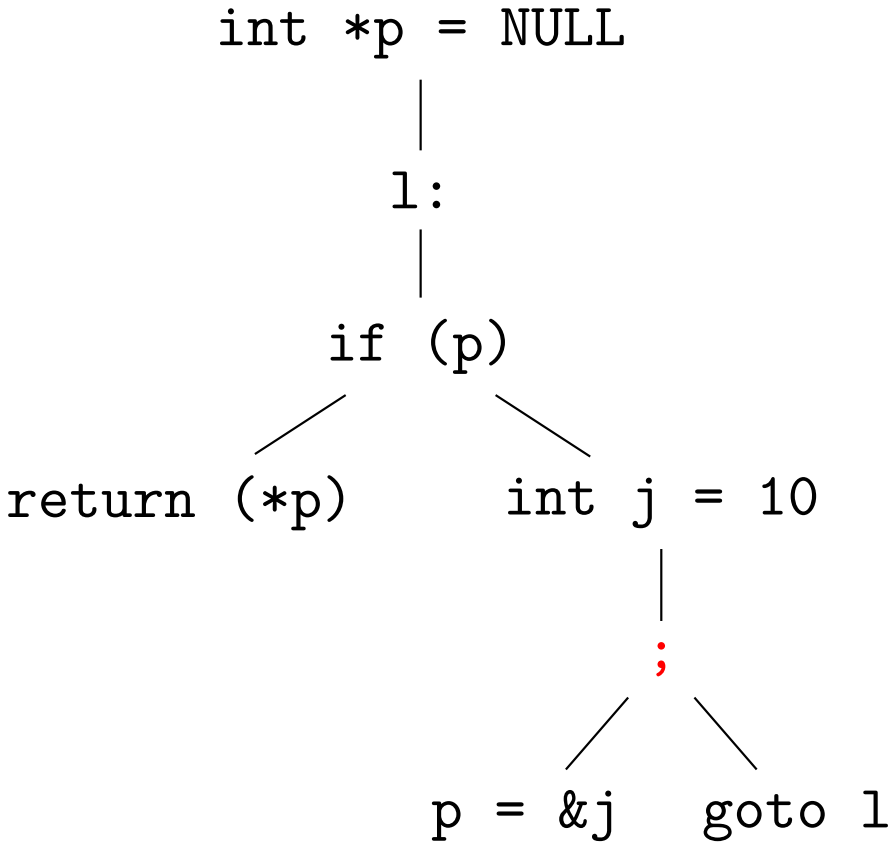


direction:

memory:

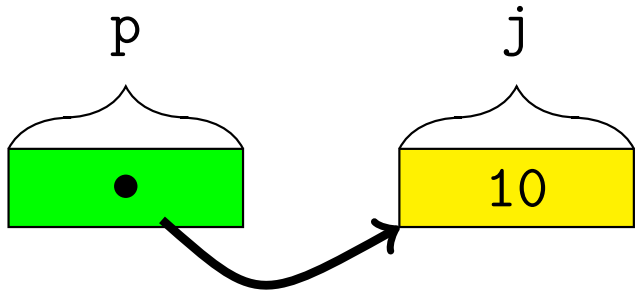


Example

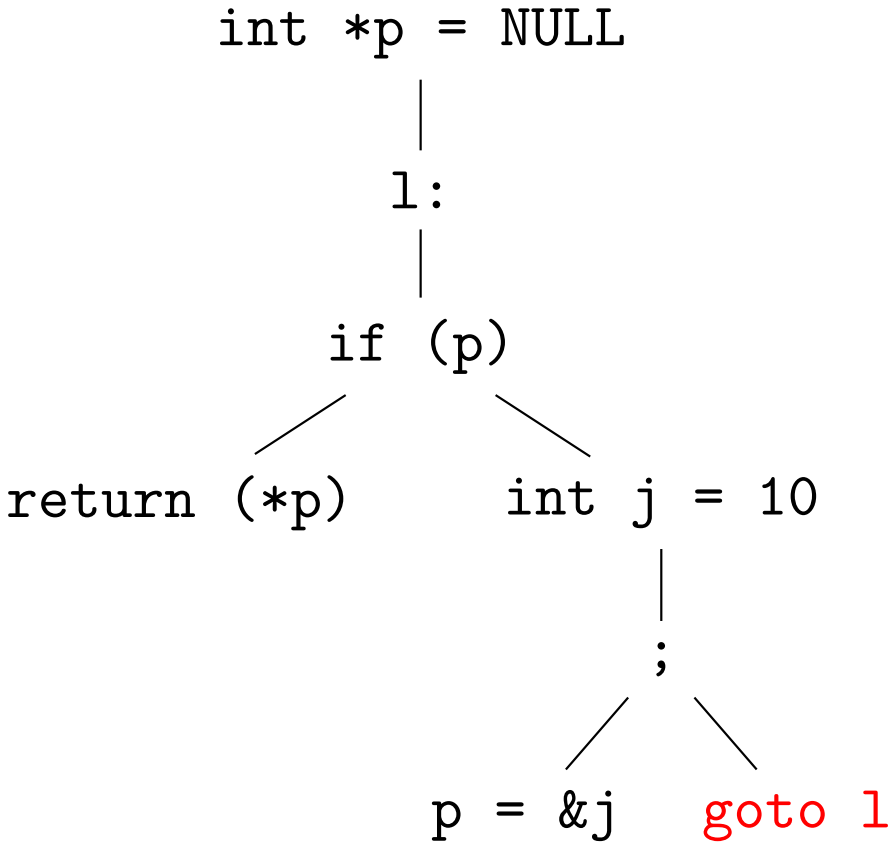


direction:

memory:

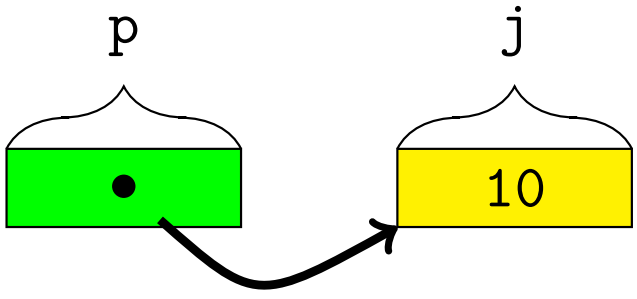


Example

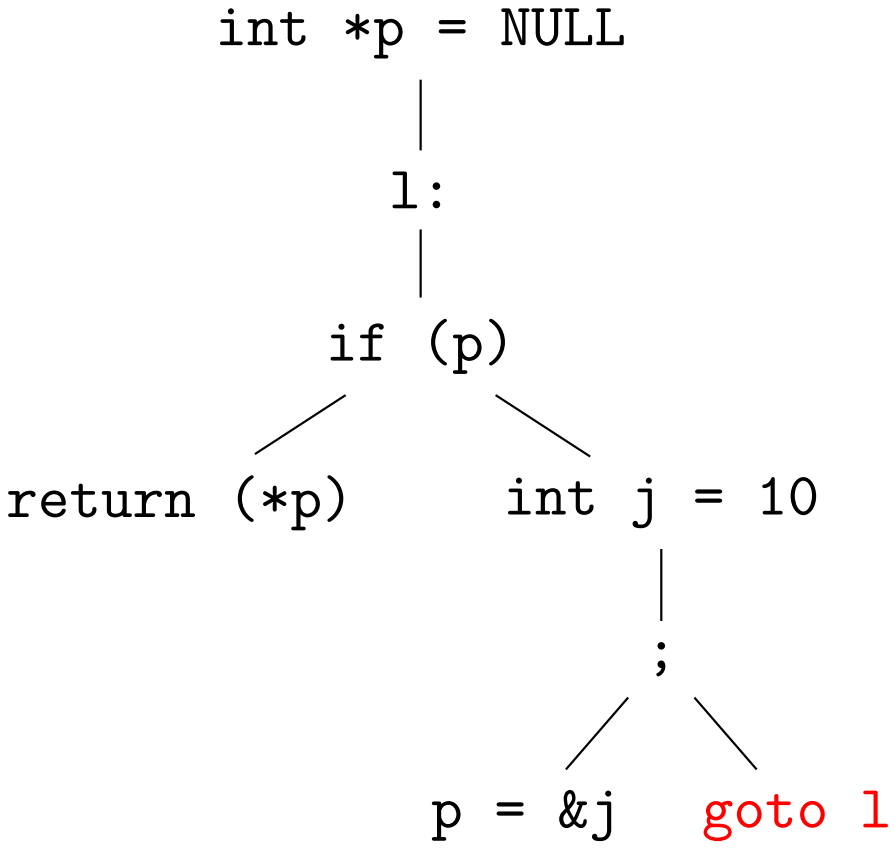


direction:

memory:

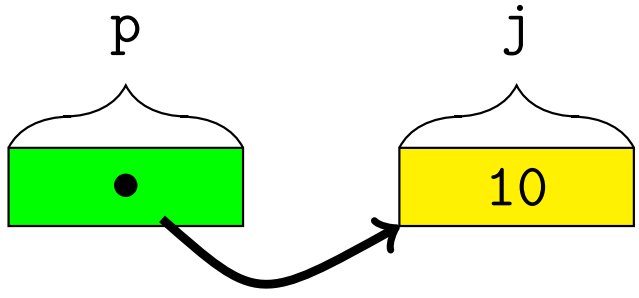


Example

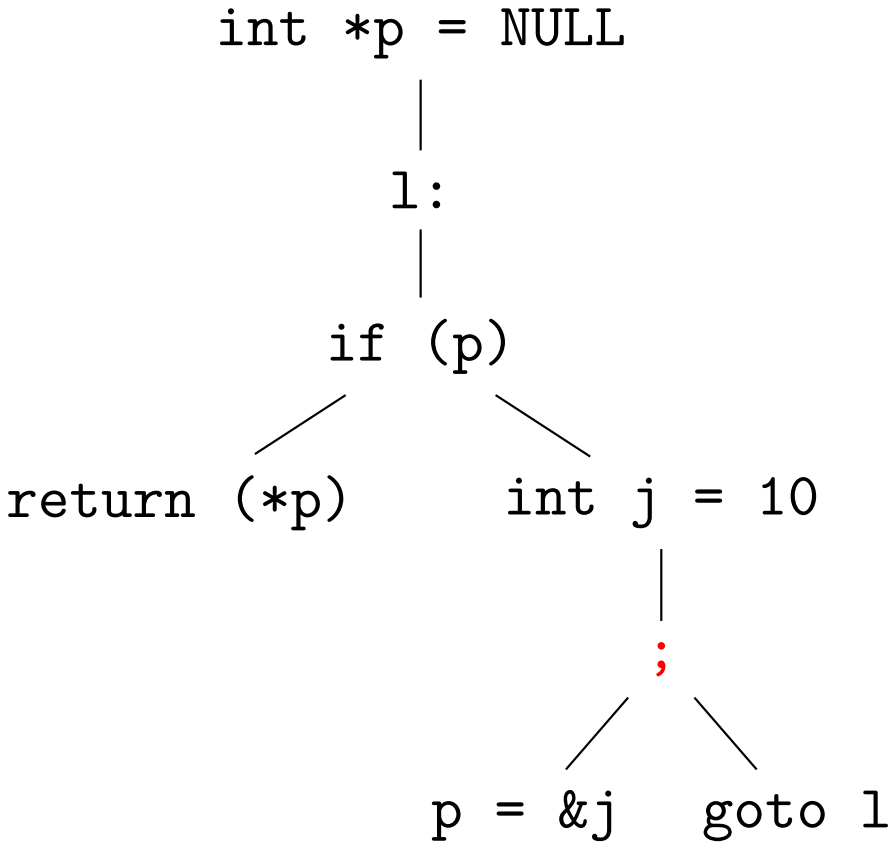


direction:

memory:

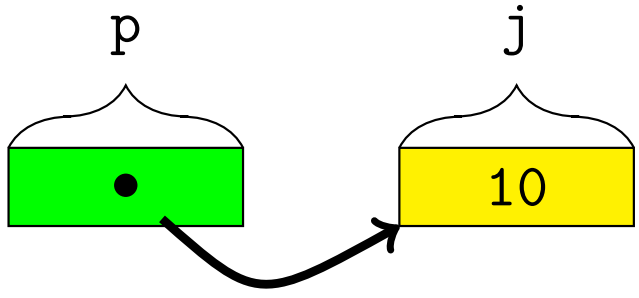


Example

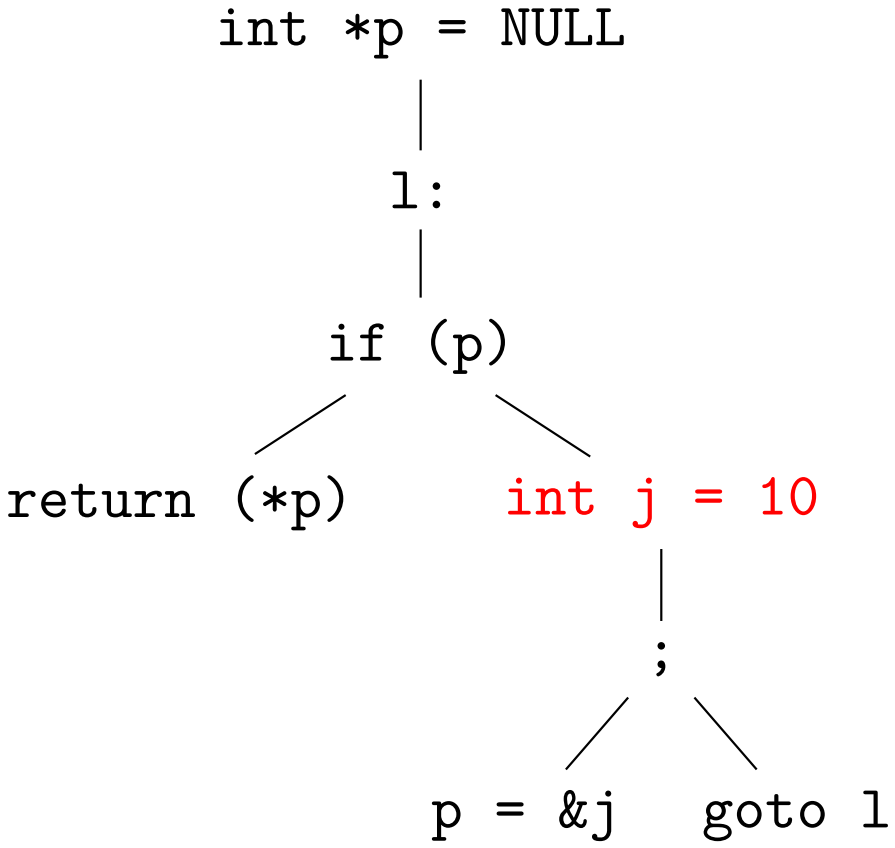


direction:

memory:

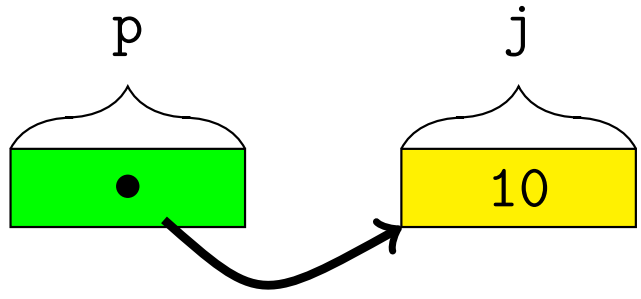


Example

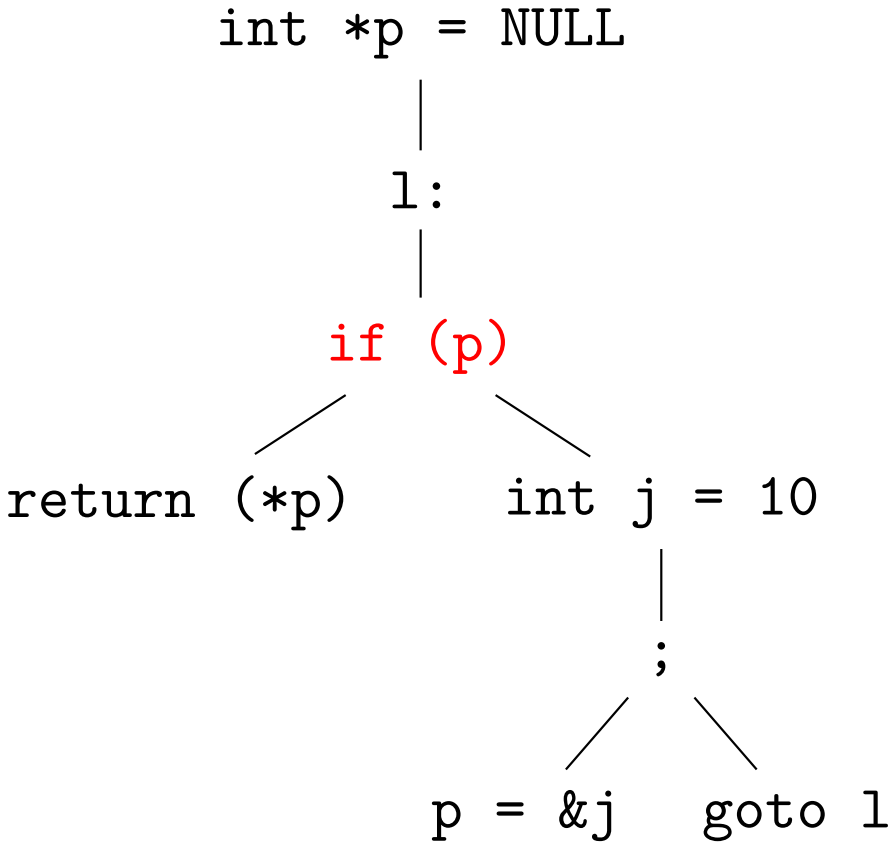


direction:

memory:

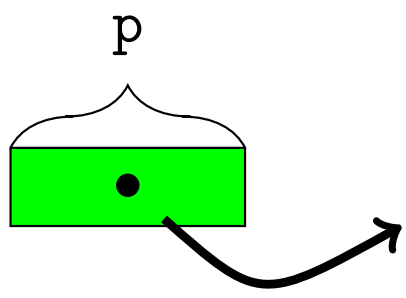


Example

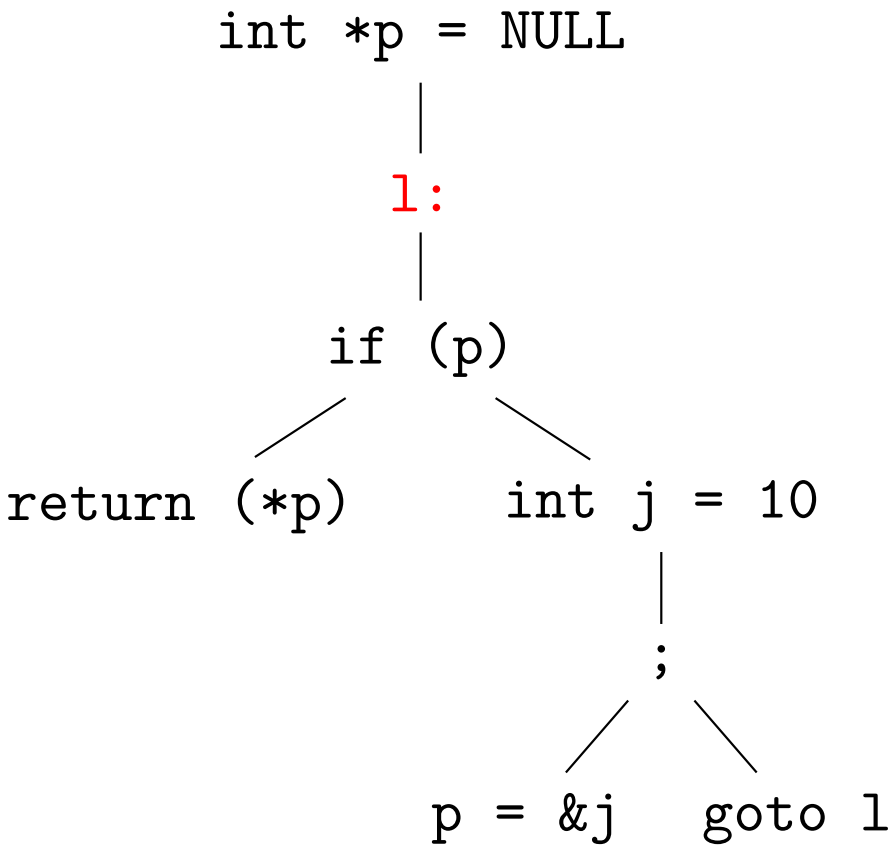


direction:

memory:

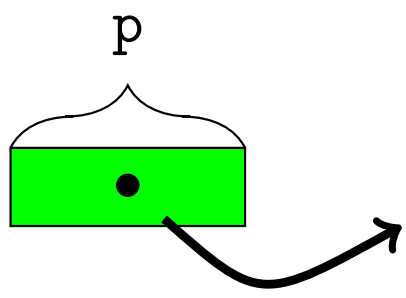


Example

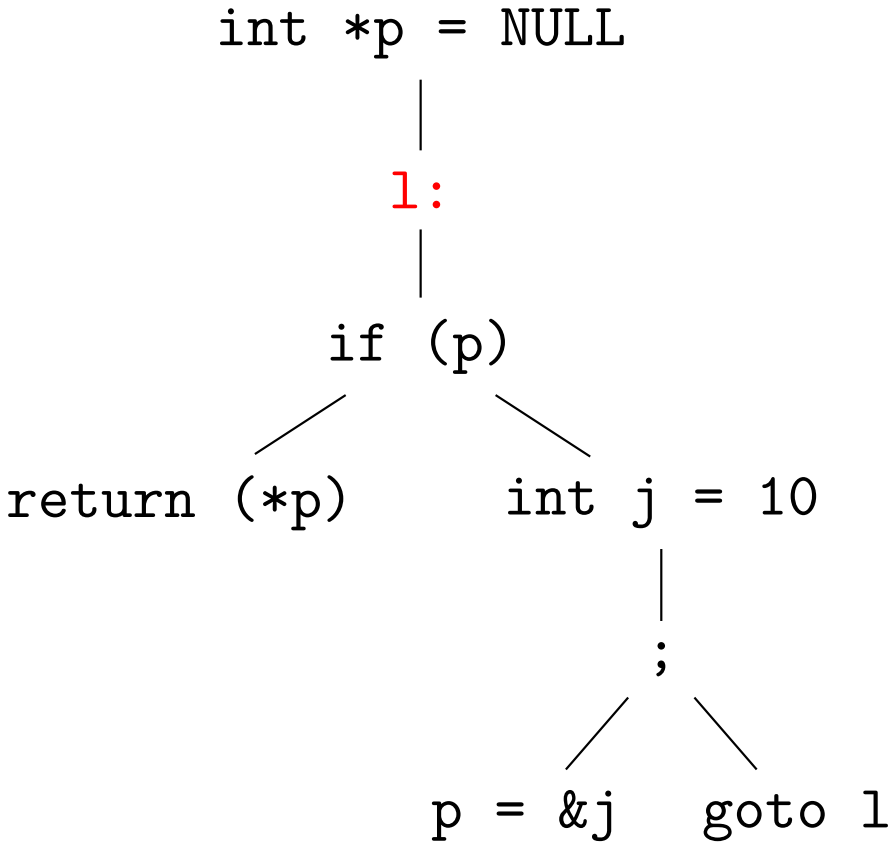


direction:

memory:

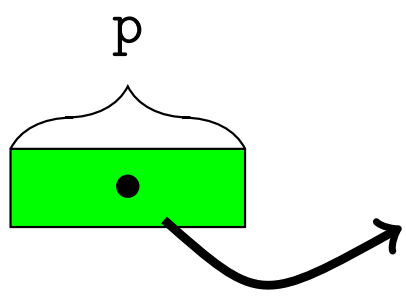


Example

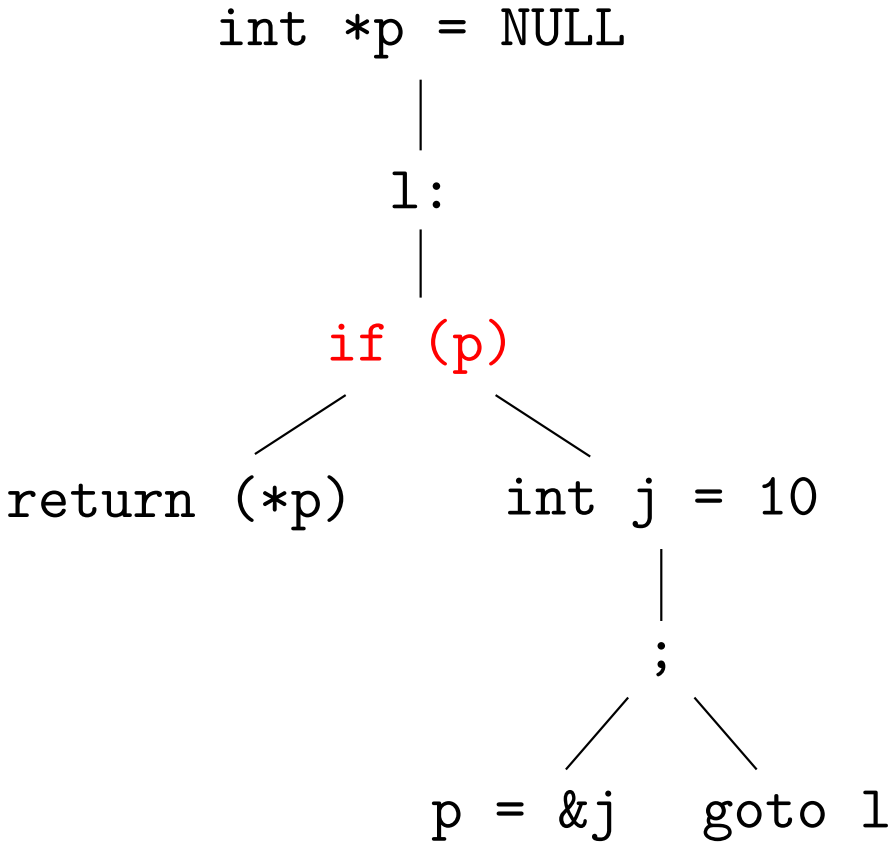


direction:

memory:

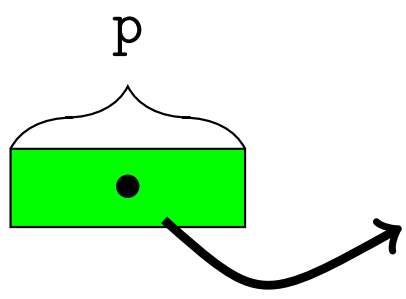


Example

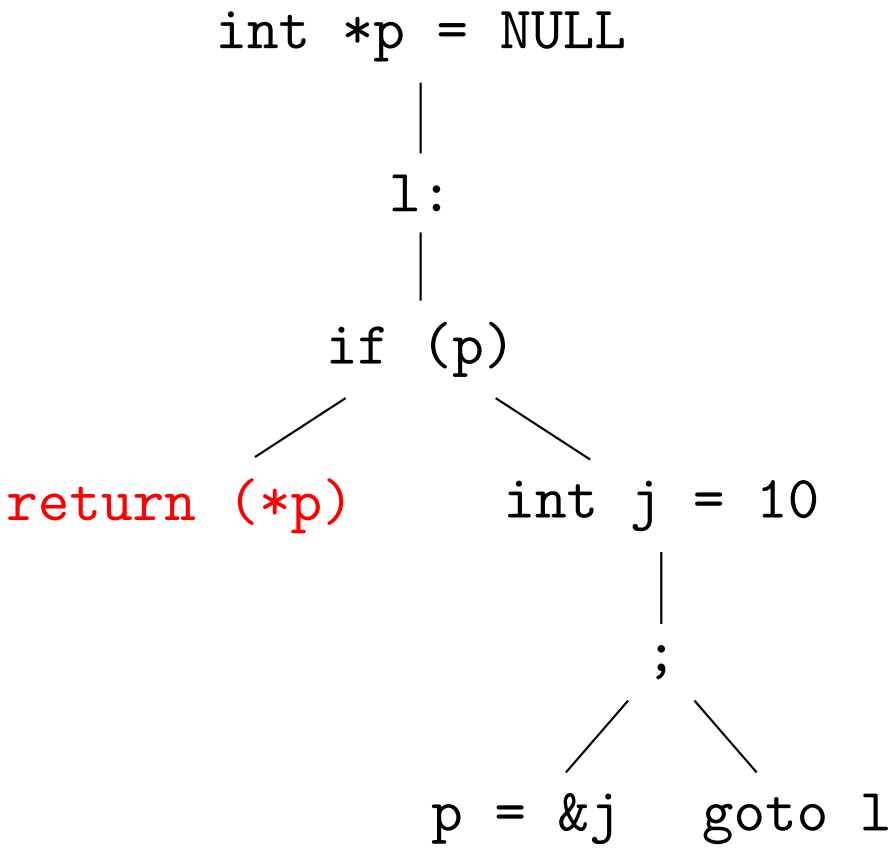


direction:

memory:

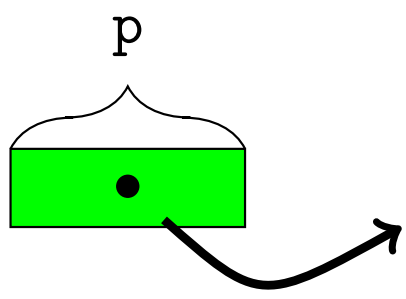


Example



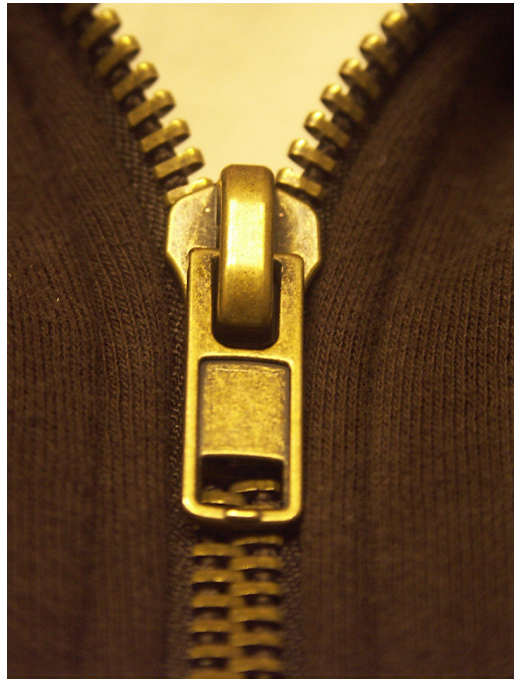
direction:

memory:



How to model the current *location* in the program

Huet's zipper



Purely functional way to store a pointer into a data structure

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:

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

- Singular statement contexts:

$$E_S ::= \square ; s_2 \mid s_1 ; \square \mid \texttt{if}\ (e)\ \square\ s_2 \mid \texttt{if}\ (e)\ s_1\ \square \mid l : \square$$

Statement contexts

- Statements:

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

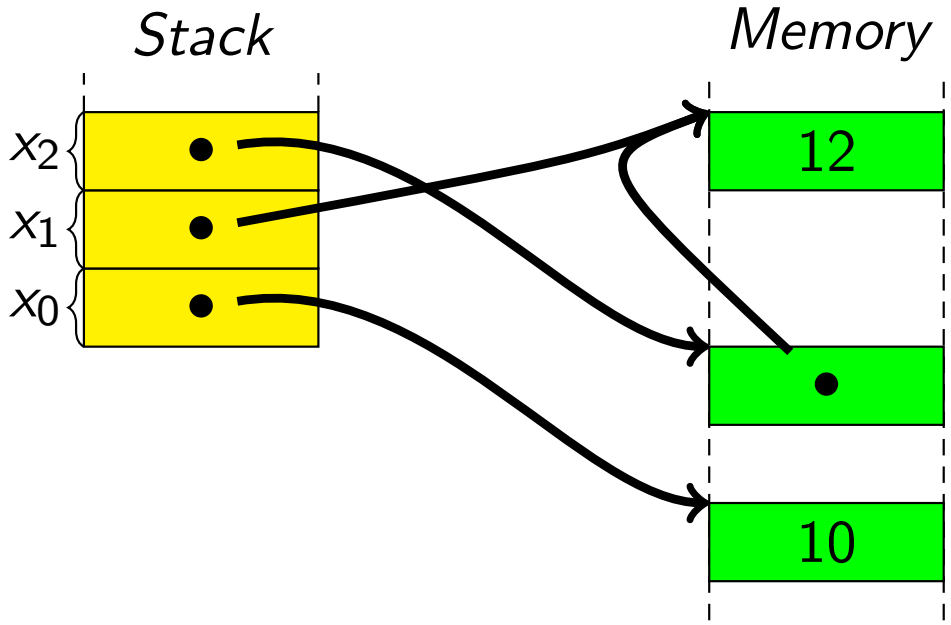
- Singular statement contexts:

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

- 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

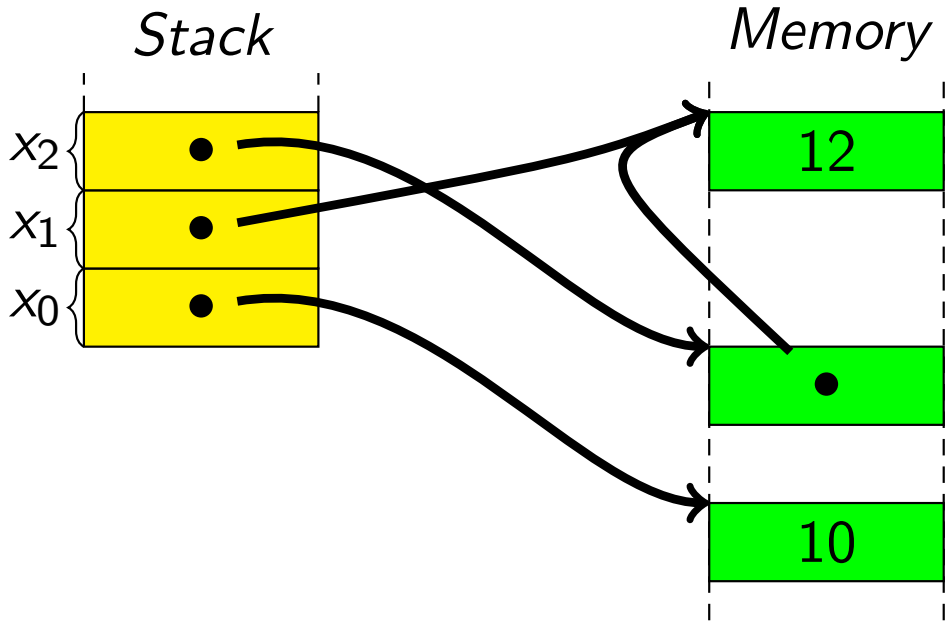
Stacks

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



Stacks

- 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

Program contexts (1)

Singular program contexts:

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

where:

Program contexts (1)

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

Program contexts (1)

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 (1)

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 (1)

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

Program contexts (2)

- Program contexts contain the stack:

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

Program contexts (2)

- Program contexts contain the stack:

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

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

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 $f(\vec{v})$

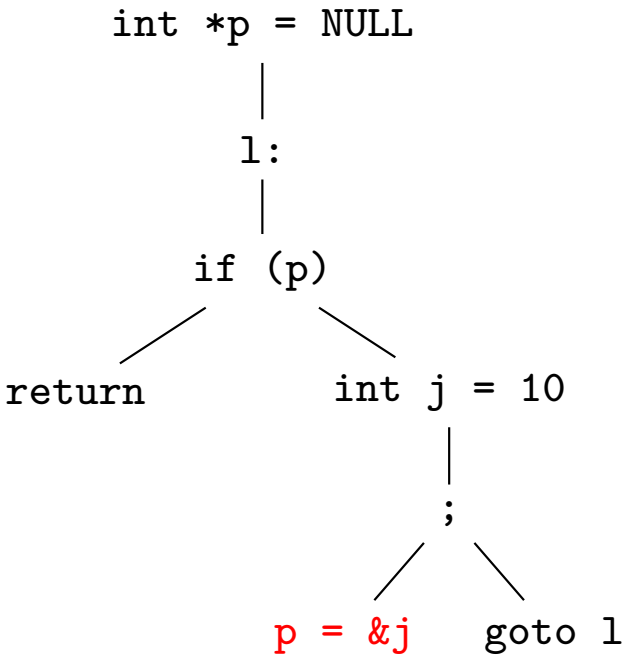
States

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
- $\overline{\text{call}}\, f\ \vec{v}$ 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

Some rules:

- $\mathbf{S}(k, (\searrow, e_1 := e_2), m) \rightarrow \mathbf{S}(k, (\nearrow, e_1 := e_2), m[a := v])$ for a and v s.t. $[\![e_1]\!]_{k,m} = \texttt{ptr}\ a$, $[\![e_2]\!]_{k,m} = v$ and $m\, a \neq \bot$.

The small step semantics

Some rules:

- $\mathbf{S}(k, (\searrow, e_1 := e_2), m) \rightarrowtriangle \mathbf{S}(k, (\nearrow, e_1 := e_2), m[a := v])$ for a and v s.t. $[\![e_1]\!]_{k,m} = \texttt{ptr}\ a$, $[\![e_2]\!]_{k,m} = v$ and $m\, a \neq \bot$.
- $\mathbf{S}(k, (\searrow, s_1 ; s_2), m) \rightarrowtriangle \mathbf{S}((\square ; s_2) :: k, (\searrow, s_1), m)$

The small step semantics

Some rules:

- $\mathbf{S}(k, (\searrow, e_1 := e_2), m) \rightarrowtriangle \mathbf{S}(k, (\nearrow, e_1 := e_2), m[a := v])$ for a and v s.t. $[\![e_1]\!]_{k,m} = \texttt{ptr}\ a$, $[\![e_2]\!]_{k,m} = v$ and $m\, a \neq \bot$.
- $\mathbf{S}(k, (\searrow, s_1 ; s_2), m) \rightarrowtriangle \mathbf{S}((\square ; s_2) :: k, (\searrow, s_1), m)$
- $\mathbf{S}((\square ; s_2) :: k, (\nearrow, s_1), m) \rightarrowtriangle \mathbf{S}((s_1 ; \square) :: k, (\searrow, s_2), m)$

The small step semantics

Some rules:

- $\mathbf{S}(k, (\searrow, e_1 := e_2), m) \rightarrowtriangle \mathbf{S}(k, (\nearrow, e_1 := e_2), m[a := v])$
  for a and v s.t. $[\![e_1]\!]_{k,m} = \texttt{ptr}\ a$, $[\![e_2]\!]_{k,m} = v$ and $m\, a \neq \bot$.
- $\mathbf{S}(k, (\searrow, s_1 ; s_2), m) \rightarrowtriangle \mathbf{S}((\square ; s_2) :: k, (\searrow, s_1), m)$
- $\mathbf{S}((\square ; s_2) :: k, (\nearrow, s_1), m) \rightarrowtriangle \mathbf{S}((s_1 ; \square) :: k, (\searrow, s_2), m)$
- $\mathbf{S}((s_1 ; \square) :: k, (\nearrow, s_2), m) \rightarrowtriangle \mathbf{S}(k, (\nearrow, s_1 ; s_2), m)$

The small step semantics

Some rules:

- $\mathbf{S}(k, (\searrow, e_1 := e_2), m) \rightarrowtriangle \mathbf{S}(k, (\nearrow, e_1 := e_2), m[a := v])$ for a and v s.t. $[\![e_1]\!]_{k,m} = \texttt{ptr}\ a$, $[\![e_2]\!]_{k,m} = v$ and $m\, a \neq \bot$.
- $\mathbf{S}(k, (\searrow, s_1 ; s_2), m) \rightarrowtriangle \mathbf{S}((\square ; s_2) :: k, (\searrow, s_1), m)$
- $\mathbf{S}((\square ; s_2) :: k, (\nearrow, s_1), m) \rightarrowtriangle \mathbf{S}((s_1 ; \square) :: k, (\searrow, s_2), m)$
- $\mathbf{S}((s_1 ; \square) :: k, (\nearrow, s_2), m) \rightarrowtriangle \mathbf{S}(k, (\nearrow, s_1 ; s_2), m)$
- $\mathbf{S}(k, (\searrow, \texttt{goto}\ l), m) \rightarrowtriangle \mathbf{S}(k, (\curvearrowright l, \texttt{goto}\ l), m)$

The small step semantics

Some rules:

- $\mathbf{S}(k, (\searrow, e_1 := e_2), m) \rightarrowtriangle \mathbf{S}(k, (\nearrow, e_1 := e_2), m[a := v])$ for a and v s.t. $[\![e_1]\!]_{k,m} = \mathtt{ptr}\ a$, $[\![e_2]\!]_{k,m} = v$ and $m\, a \neq \bot$.
- $\mathbf{S}(k, (\searrow, s_1 ; s_2), m) \rightarrowtriangle \mathbf{S}((\square ; s_2) :: k, (\searrow, s_1), m)$
- $\mathbf{S}((\square ; s_2) :: k, (\nearrow, s_1), m) \rightarrowtriangle \mathbf{S}((s_1 ; \square) :: k, (\searrow, s_2), m)$
- $\mathbf{S}((s_1 ; \square) :: k, (\nearrow, s_2), m) \rightarrowtriangle \mathbf{S}(k, (\nearrow, s_1 ; s_2), m)$
- $\mathbf{S}(k, (\searrow, \mathtt{goto}\ l), m) \rightarrowtriangle \mathbf{S}(k, (\curvearrowright l, \mathtt{goto}\ l), m)$
- $\mathbf{S}(k, (\curvearrowright l, l : s), m) \rightarrowtriangle \mathbf{S}((l : \square) :: k, (\searrow, s), m)$

The small step semantics

Some rules:

- $\mathbf{S}(k, (\searrow, e_1 := e_2), m) \rightarrowtriangle \mathbf{S}(k, (\nearrow, e_1 := e_2), m[a := v])$ for a and v s.t. $[\![e_1]\!]_{k,m} = \mathtt{ptr}\ a$, $[\![e_2]\!]_{k,m} = v$ and $m\, a \neq \bot$.
- $\mathbf{S}(k, (\searrow, s_1 ; s_2), m) \rightarrowtriangle \mathbf{S}((\square ; s_2) :: k, (\searrow, s_1), m)$
- $\mathbf{S}((\square ; s_2) :: k, (\nearrow, s_1), m) \rightarrowtriangle \mathbf{S}((s_1 ; \square) :: k, (\searrow, s_2), m)$
- $\mathbf{S}((s_1 ; \square) :: k, (\nearrow, s_2), m) \rightarrowtriangle \mathbf{S}(k, (\nearrow, s_1 ; s_2), m)$
- $\mathbf{S}(k, (\searrow, \mathtt{goto}\ l), m) \rightarrowtriangle \mathbf{S}(k, (\curvearrowright l, \mathtt{goto}\ l), m)$
- $\mathbf{S}(k, (\curvearrowright l, l : s), m) \rightarrowtriangle \mathbf{S}((l : \square) :: k, (\searrow, s), m)$
- $\mathbf{S}(k, (\searrow, f(\vec{e})), m) \rightarrowtriangle \mathbf{S}(\mathrm{call}\ f\ \vec{e} :: k, \overline{\mathrm{call}}\ f\ \vec{v}, m)$ 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

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

then $s = s'$.

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

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 $\Uparrow$

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 $\Uparrow$
- We thus treat the sextuple as

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

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

Some Hoare rules

Weakening:

$$\frac{\begin{array}{c}(\forall l \in \mathsf{labels}\ s\,.\ J'\,l \to J\,l) \quad (\forall l \notin \mathsf{labels}\ s\,.\ J\,l \to J'\,l) \quad R \to R' \\ P' \to P \quad \Delta;\ J;\ R \vdash \{P\}\,s\,\{Q\} \quad Q \to Q'\end{array}}{\Delta;\ J';\ R' \vdash \{P'\}\,s\,\{Q'\}}$$

Some Hoare rules

Weakening:

$$\frac{(\forall l \in \text{labels}\ s\,.\,J'\,l \to J\,l) \quad (\forall l \notin \text{labels}\ s\,.\,J\,l \to J'\,l) \quad R \to R' \quad P' \to P \quad \Delta;\ J;\ R \vdash \{P\}\,s\,\{Q\} \quad Q \to Q'}{\Delta;\ J';\ R' \vdash \{P'\}\,s\,\{Q'\}}$$

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:

$$\frac{(\forall l \in \text{labels}\ s\,.\,J'\,l \to J\,l) \quad (\forall l \notin \text{labels}\ s\,.\,J\,l \to J'\,l) \quad R \to R' \quad P' \to P \quad \Delta;\ J;\ R \vdash \{P\}\,s\,\{Q\} \quad Q \to Q'}{\Delta;\ J';\ R' \vdash \{P'\}\,s\,\{Q'\}}$$

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:

$$\frac{(\forall l \in \text{labels}\ s\,.\,J'\,l \to J\,l) \quad (\forall l \notin \text{labels}\ s\,.\,J\,l \to J'\,l) \quad R \to R' \quad P' \to P \quad \Delta;\ J;\ R \vdash \{P\}\,s\,\{Q\} \quad Q \to Q'}{\Delta;\ J';\ R' \vdash \{P'\}\,s\,\{Q'\}}$$

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:

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

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

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

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

Example

```
void swap(int *p, int *q) {
  int z = *p; *p = *q; *q = z;
}
```

Example

$$\{x_0 \mapsto p * x_1 \mapsto q * p \mapsto y * q \mapsto z\}$$

$$\texttt{block}\,(\quad x_0 := \texttt{load}\,(\texttt{load}\,x_1)\,;\quad \texttt{load}\,x_1 := \texttt{load}\,(\texttt{load}\,x_2)\,;\quad \texttt{load}\,x_2 := \texttt{load}\,x_0\quad)$$

$$\{x_0 \mapsto p * x_1 \mapsto q * p \mapsto z * q \mapsto y\}$$

Example

$$\{x_0 \mapsto p * x_1 \mapsto q * p \mapsto y * q \mapsto z\}$$

$$\texttt{block}\,(\,$$

$$\{x_0 \mapsto \text{-} * x_1 \mapsto p * x_2 \mapsto q * p \mapsto y * q \mapsto z\}$$

$$x_0 := \texttt{load}\,(\texttt{load}\,x_1)\,;$$

$$\texttt{load}\,x_1 := \texttt{load}\,(\texttt{load}\,x_2)\,;$$

$$\texttt{load}\,x_2 := \texttt{load}\,x_0$$

$$)$$

$$\{x_0 \mapsto p * x_1 \mapsto q * p \mapsto z * q \mapsto y\}$$

Example

$\{x_0 \mapsto p * x_1 \mapsto q * p \mapsto y * q \mapsto z\}$

$\texttt{block}\,($

$\{x_0 \mapsto \text{-} * x_1 \mapsto p * x_2 \mapsto q * p \mapsto y * q \mapsto z\}$

$x_0 := \texttt{load}\,(\texttt{load}\,x_1)\,;$

$\{x_0 \mapsto y * x_1 \mapsto p * x_2 \mapsto q * p \mapsto y * q \mapsto z\}$

$\texttt{load}\,x_1 := \texttt{load}\,(\texttt{load}\,x_2)\,;$

$\texttt{load}\,x_2 := \texttt{load}\,x_0$

$)$

$\{x_0 \mapsto p * x_1 \mapsto q * p \mapsto z * q \mapsto y\}$

Example

$\{x_0 \mapsto p * x_1 \mapsto q * p \mapsto y * q \mapsto z\}$

$\texttt{block}\,($

$\{x_0 \mapsto \text{-} * x_1 \mapsto p * x_2 \mapsto q * p \mapsto y * q \mapsto z\}$

$\quad x_0 := \texttt{load}\,(\texttt{load}\,x_1)\,;$

$\{x_0 \mapsto y * x_1 \mapsto p * x_2 \mapsto q * p \mapsto y * q \mapsto z\}$

$\quad \texttt{load}\,x_1 := \texttt{load}\,(\texttt{load}\,x_2)\,;$

$\{x_0 \mapsto y * x_1 \mapsto p * x_2 \mapsto q * p \mapsto z * q \mapsto z\}$

$\quad \texttt{load}\,x_2 := \texttt{load}\,x_0$

$)$

$\{x_0 \mapsto p * x_1 \mapsto q * p \mapsto z * q \mapsto y\}$

Example

$\{x_0 \mapsto p * x_1 \mapsto q * p \mapsto y * q \mapsto z\}$

$\texttt{block}\,($

$\{x_0 \mapsto \text{-} * x_1 \mapsto p * x_2 \mapsto q * p \mapsto y * q \mapsto z\}$

$x_0 := \texttt{load}\,(\texttt{load}\,x_1)\,;$

$\{x_0 \mapsto y * x_1 \mapsto p * x_2 \mapsto q * p \mapsto y * q \mapsto z\}$

$\texttt{load}\,x_1 := \texttt{load}\,(\texttt{load}\,x_2)\,;$

$\{x_0 \mapsto y * x_1 \mapsto p * x_2 \mapsto q * p \mapsto z * q \mapsto z\}$

$\texttt{load}\,x_2 := \texttt{load}\,x_0$

$\{x_0 \mapsto y * x_1 \mapsto p * x_2 \mapsto q * p \mapsto z * q \mapsto y\}$

$)$

$\{x_0 \mapsto p * x_1 \mapsto q * p \mapsto z * q \mapsto y\}$

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
- 3500 lines of code

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/`