An Axiomatic Basis for Computer Programming

Paper by:	“Tony” Hoare
Presentation by:	Valentin Robert @Ptival
Presented at:	Papers We Love San Diego
On:	Aug. 3rd 2017

Context of the paper (1969)

FORTRAN	LISP
ALGOL	BASIC

- Soon to be created:

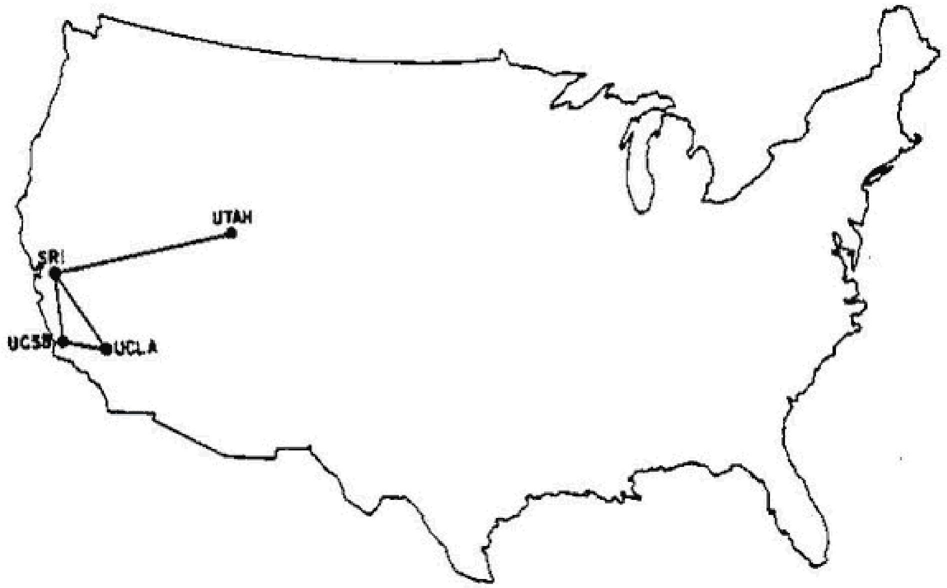
  C Prolog

  ML

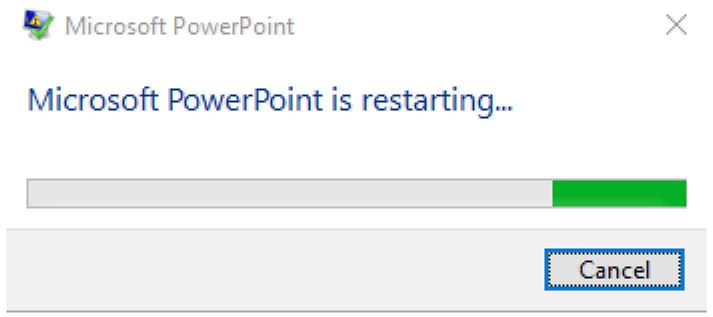
- Soon to be born:

  Linus Torvalds

The entire Internet in 1969



The problem


Microsoft PowerPoint

Microsoft PowerPoint is restarting...

Tony Hoare’s starting point

- Program execution is an exact science,

- so we can reason about programs **deductively**,

- therefore, we should elucidate the **axioms** and **rules** of reasoning about programs.

Where to start!?

- Coming up with axioms for computer arithmetic is already challenging:

`x + y = y + x` Sure…

`x + 1 > x` …overflows?

`(x + y) – y = x` …floating point?

What about entire programs?

- We would like to know whether certain **assertions** about variables of the program are true at certain **times** of the execution.

- Those assertions will not necessarily ascribe concrete values to variables, but rather describe **relations** between variables of the program.
  - i.e. “x must be equal to y + 1”

Swap

```
swap(x, y) {
$x = x_0 \quad \wedge \quad y = y_0$
    y := x + y
$x = x_0 \quad \wedge \quad y = y_0 + x_0$
    x := y − x
$x = y_0 \quad \wedge \quad y = y_0 + x_0$
    y := y − x
$x = y_0 \quad \wedge \quad y = x_0$
}
```

Floyd-Hoare triple

- The main idea of this paper is to define a notation to capture this relation between:

```
What we know before              {P}
some_instruction()                S
What we know after               {Q}
```

Floyd-Hoare triple[1][2]

A **Floyd-Hoare triple** is an assertion noted:

```
{ P } S { Q }
```

whose meaning is:

- from any starting state such that P holds (**pre-condition**)
- if running the program S terminates
- then the final state is such that Q holds (**post-condition**)

[1] Floyd, Robert W. "Assigning meanings to programs." *Program Verification*. Springer Netherlands, 1993. 65-81.

[2] Hoare, Charles Antony Richard. "An axiomatic basis for computer programming." *Communications of the ACM* 12.10 (1969): 576-580.

Note: the paper uses the older convention “P {S} Q”

Floyd-Hoare triple (examples)

```
{ x = 41 } x := x + 1 { x = 42 }

{ True   } x := 0     { x = 0  }

{ b = true ∧ a = 21 }
if b
  then x := a
  else x := − a
{ x = −21 }
```

Floyd-Hoare triple (weird examples)

- Some triples are **invalid**:

```
{ x = 41 } x := x + 1 { x = 23 }
```

- Some triples are **imprecise**:

```
{ x = 41 } x := x + 1 { x > 23 }
```

- How do we know which ones are **valid**?
- How do we know which ones are **precise**?

Axiom of assignment (example)

```
{              ????           }
x := (6 * y) - 2
{          x = 23 ∧ b = true }
```

Under what precondition will the postcondition be true?

Axiom of assignment (example)

```
{ (6 * y) - 2 = 23 ∧ b = true }
x := (6 * y) - 2
{             x = 23 ∧ b = true }
```

Surely, if the value on the right-hand-side satisfies the condition, then the variable on the left-hand-side satisfies the condition after it is assigned.

Axiom of assignment (example)

```
{ P((6 * y) - 2) }
x := (6 * y) - 2
{ P(x) }
```

Surely, if the value on the right-hand-side satisfies the condition, then the variable on the left-hand-side satisfies the condition once it is assigned.

Notation warning

Axiom

$$\frac{}{\{P\}S\{Q\}}$$

Rule

$$\frac{\{P_1\}S_1\{Q_1\} \quad \{P_2\}S_2\{Q_2\}}{\{P\}S\{Q\}}$$

Axiom of assignment

$$\frac{}{\{\ P[x \leftarrow e]\ \}\ \mathtt{x\ :=\ e}\ \{\ P\ \}}$$

- This says:
  - a property P of x will hold after the assignment
  - as long as the property already holds for e instead of x before the assignment

Swap

```
swap(x, y) {
      y     = y₀ ∧           x  = x₀
    y := x + y
      y - x = y₀ ∧           x  = x₀
    x := y - x
          x = y₀ ∧ y - x        = x₀
    y := y - x
          x = y₀ ∧            y = x₀
}
```

Compositional reasoning

- We would like a set of **axioms** to capture **precisely** the effect of each **instruction** on the facts we know to be true,

- and a set of **rules** to derive the meaning of complex programs as a combination of the meaning of its instructions

Rule of composition

$$\frac{\{P\}\ S_1\ \{R\} \qquad \{R\}\ S_2\ \{Q\}}{\{P\}\ S_1\ ;\ S_2\ \{Q\}}$$

Rule of composition

not very compositional...

$$\frac{\{P\}\ S_1\ \{R\} \qquad \{R\}\ S_2\ \{Q\}}{\{P\}\ S_1\ ;\ S_2\ \{Q\}}$$

Rules of consequence

$$\frac{\{P\}\ S\ \{R\} \qquad R \Rightarrow Q}{\{P\}\ S\ \{Q\}}$$

weakens the post-condition

$$\frac{P \Rightarrow R \qquad \{R\}\ S\ \{Q\}}{\{P\}\ S\ \{Q\}}$$

strengthens the pre-condition

Composition + Consequence

$$\{P\}\ S_1\ \{A\} \quad \boxed{A \Rightarrow B} \quad \{B\}\ S_2\ \{Q\}$$

[redacted]

$$\{P\}\ S_1\ ;\ S_2\ \{Q\}$$

REASONING BACKWARD AND FORWARD

Hoare's axiom of assignment

$$\frac{}{\{\ Q[x \leftarrow e]\ \}\quad x := e \quad \{\ Q\ \}}$$

This axiom goes backward:

- given a post-condition Q
- it gives us the pre-condition

How do we go forward?

```
{ P }
x := e
{ ??? }
```

Floyd's axiom of assignment

$$\{\ P\ \}$$

$$x := e$$

$$\{\ \exists x_0.\ x = e[x \leftarrow x_0] \land P[x \leftarrow x_0]\ \}$$

This axiom goes forward:

- given a pre-condition P
- it gives us the post-condition

Floyd’s axiom of assignment

```
{ x = 40 ∧ y = 2 ∧ b = 0 }
x := x + y
{ … }
```

Floyd's axiom of assignment

```
{ x = 40 ∧ y = 2 ∧ b = 0 }
x := x + y
{ ∃x₀.
x = (x + y)[x←x₀]
∧ (x = 40 ∧ y = 2 ∧ b = 0)[x←x₀]
}
```

Floyd’s axiom of assignment

```
{ x = 40 ∧ y = 2 ∧ b = 0 }
x := x + y
{ ∃x₀.
x = x₀ + y
∧ (x₀ = 40 ∧ y = 2 ∧ b = 0)
}
```

Floyd’s axiom of assignment

```
{ x = 40 ∧ y = 2 ∧ b = 0 }
x := x + y
{ x = 40 + y ∧ y = 2 ∧ b = 0 }
```

Two modes of reasoning

- **Forward** reasoning

  I know some facts at the beginning of execution…

  …I can compute the set of known facts at the end of execution

- **Backward** reasoning

  I want some facts to be true at the end of execution…

  …I can compute the necessary conditions for the beginning of execution

Automating the reasoning

- A couple years later, Dijkstra will discover that:
  - given a post-condition, there exists a **weakest pre-condition**, i.e. a minimal set of conditions to ensure the post-condition
  - given a pre-condition, there exists a **strongest post-condition**, i.e. a maximal amount of facts that are ensured by the pre-condition
  - and those can often be derived **automatically**!

Automating the reasoning

- Hoare’s axiom of assignment was the weakest pre-condition for assignment!
- Floyd’s axiom of assignment was the strongest post-condition for assignment!
- Those capture the notion of **preciseness** we cared about earlier

A LOOPY PROBLEM

A loopy problem

```
{ $P_1(x) \wedge P_2(y)$ }
while !done do
    x := f(x)
    done := done?(x)
{ ??? }
```

- What do we know after the loop ends?

Rule of iteration

$$\frac{\{\ I \land C\ \}\ \mathbf{S}\ \{\ I\ \}}{\{\ I\ \}\ \mathbf{while}\ \mathbf{C}\ \mathbf{do}\ \mathbf{S}\ \{\ I \land \neg C\ \}}$$

- This rule introduces the notion of a **loop invariant**: a property I that is preserved by the loop body

Loop invariant

```
I:   m ⩾ 0 ∧ ∃ n ⩾ 0, a = n*b + m

m := a
 I
while (m ⩾ b) {
 I ∧ m ⩾ b
  m := m − b
 I
}
 I ∧ ¬(m ⩾ b)
```

Let’s boogie!

- http://rise4fun.com/Boogie

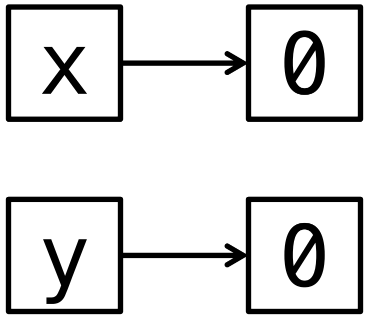
Boogie is an automated verifier from Microsoft Research based on Floyd-Hoare automatic verification

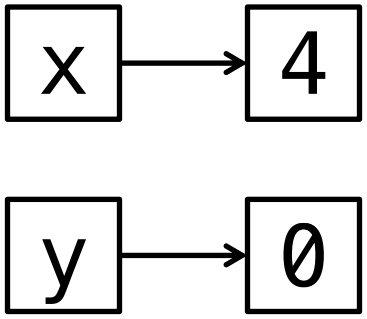
+ lot of cool work on “guessing” loop invariants!

ANOTHER HEAP OF PROBLEMS

Pointers are tricky!

$\{x \mapsto 0 \wedge y \mapsto 0\}$ `[x] := 4` $\{x \mapsto 4 \wedge y \mapsto 0\}$





Aliasing

$$\{x \mapsto 0 \land y \mapsto 0\}\ [x] := 4\ \{x \mapsto 4 \land y \mapsto 0\}$$

$$\begin{tikzcd}
\boxed{x} \arrow[r] & \boxed{0} \\
\boxed{y} \arrow[ru] &
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
\boxed{x} \arrow[r] & \boxed{4} \\
\boxed{y} \arrow[ru] &
\end{tikzcd}$$

We say that x and y are aliases.

No big deal

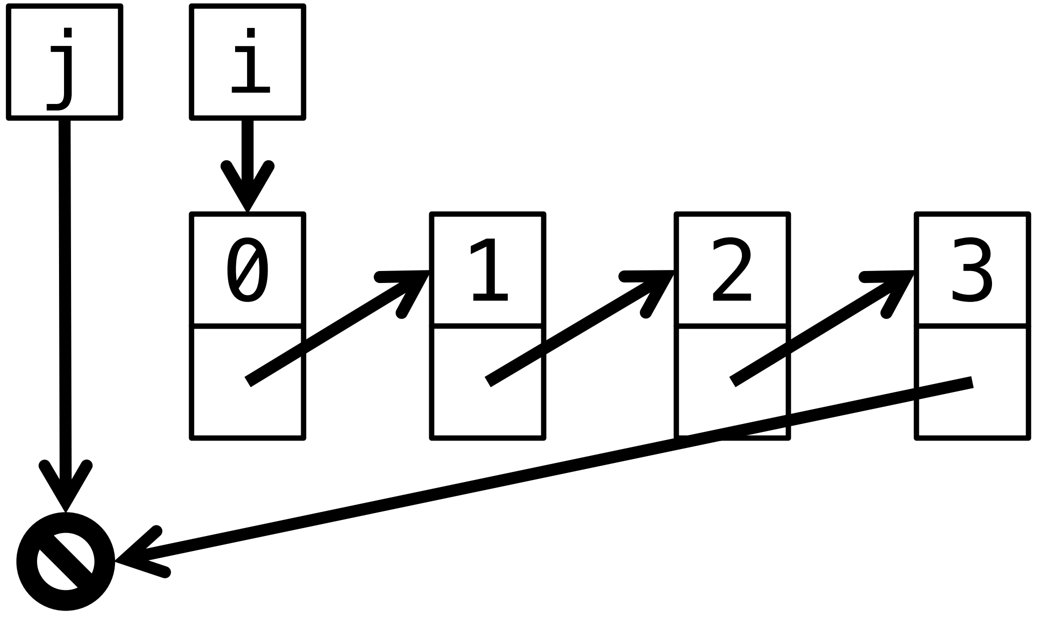
$$\{x \mapsto ? \wedge y \mapsto y_0\}$$

$$[x] := v$$

$$\{x \mapsto v \wedge ((x \neq y \wedge y \mapsto y_0) \vee (x = y \wedge y \mapsto v))\}$$

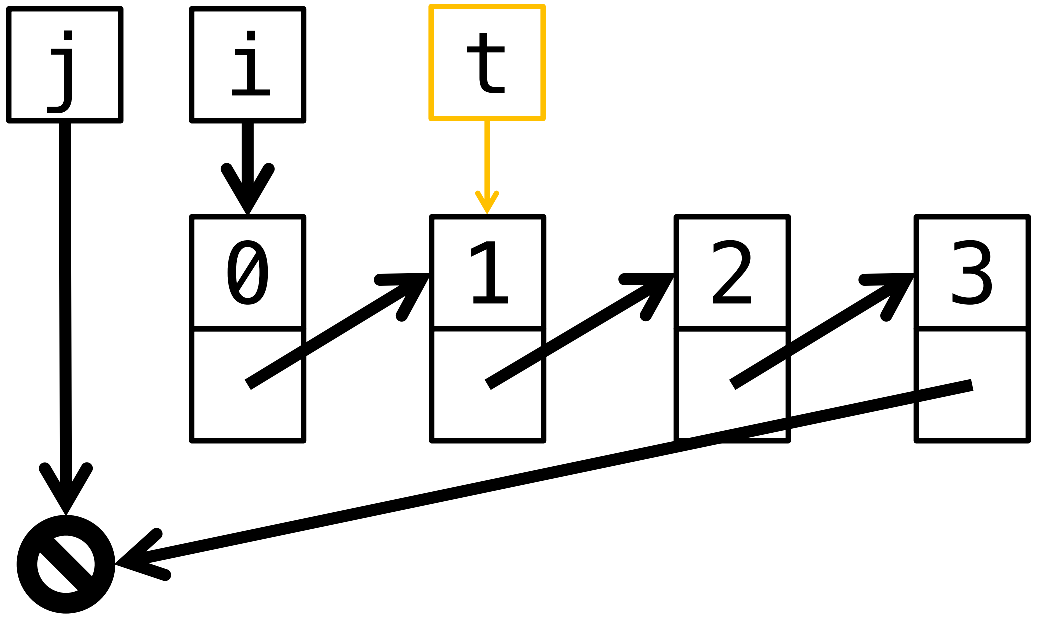
No big deal?

```
while i ≠ nil:
   tmp = [i + 1]
   [i + 1] = j
   j = i
   i = tmp
```



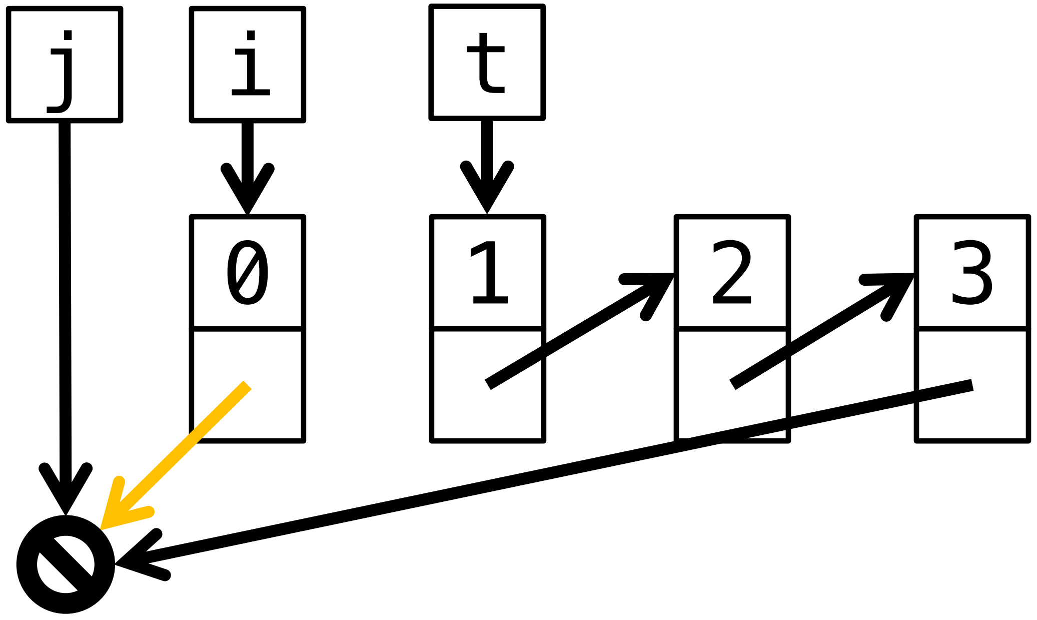
No big deal?

```
while i ≠ nil:
▷ tmp = [i + 1]
  [i + 1] = j
  j = i
  i = tmp
```



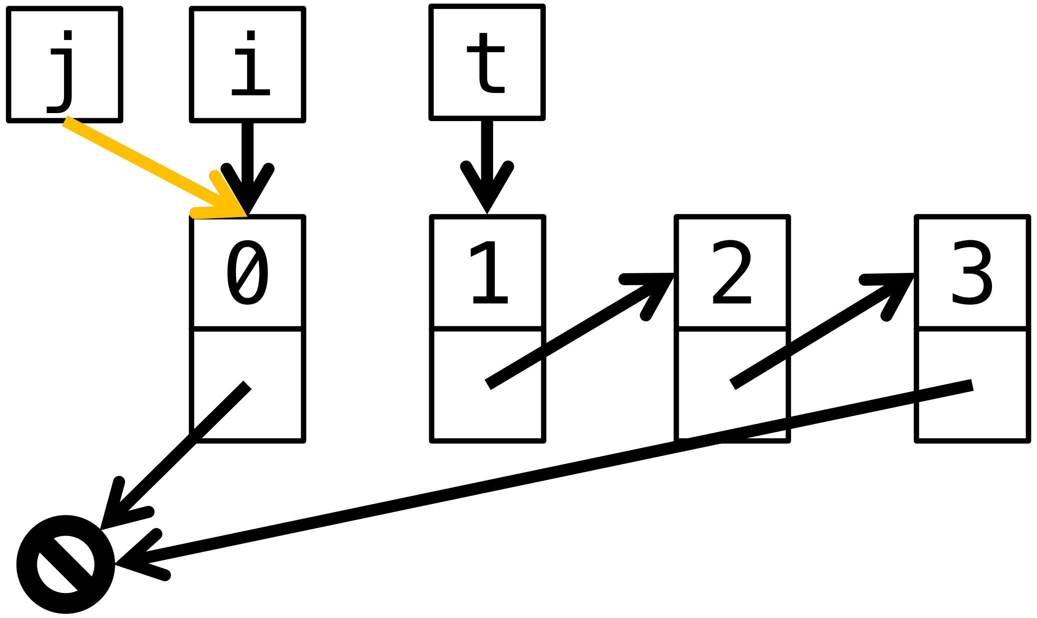
No big deal?

```
  while i ≠ nil:
    tmp = [i + 1]
▷   [i + 1] = j
    j = i
    i = tmp
```



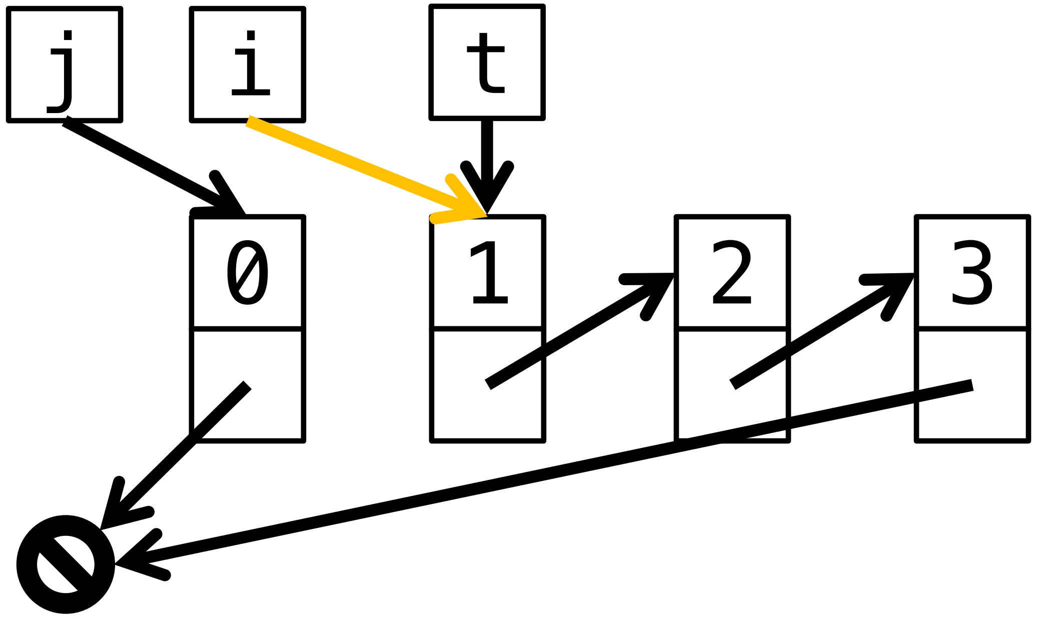
No big deal?

```
while i ≠ nil:
    tmp = [i + 1]
    [i + 1] = j
 ▷  j = i
    i = tmp
```



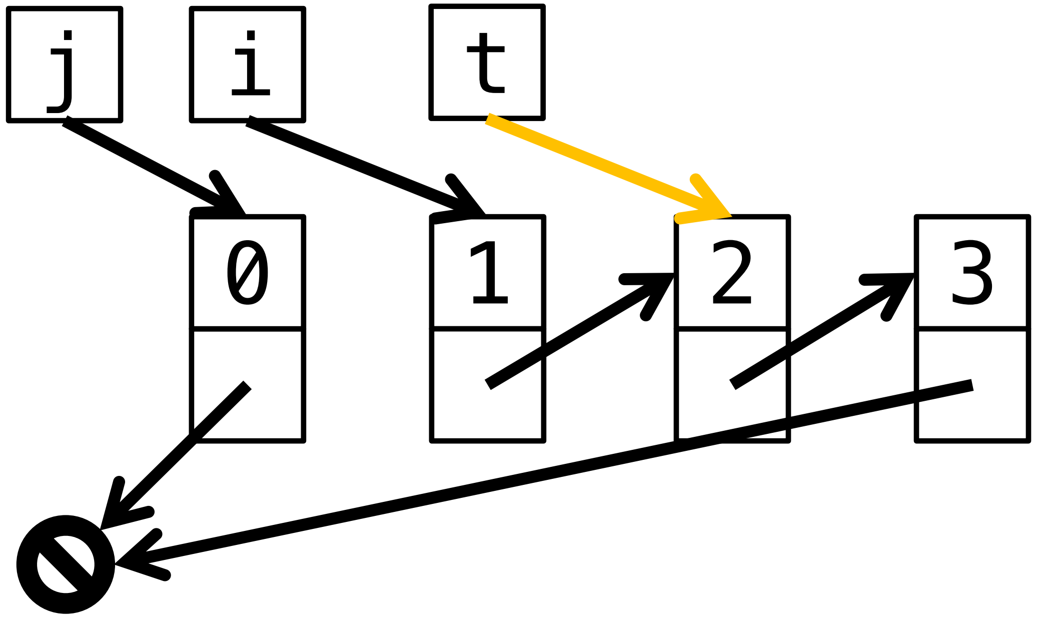
No big deal?

```
  while i ≠ nil:
    tmp = [i + 1]
    [i + 1] = j
    j = i
▷   i = tmp
```



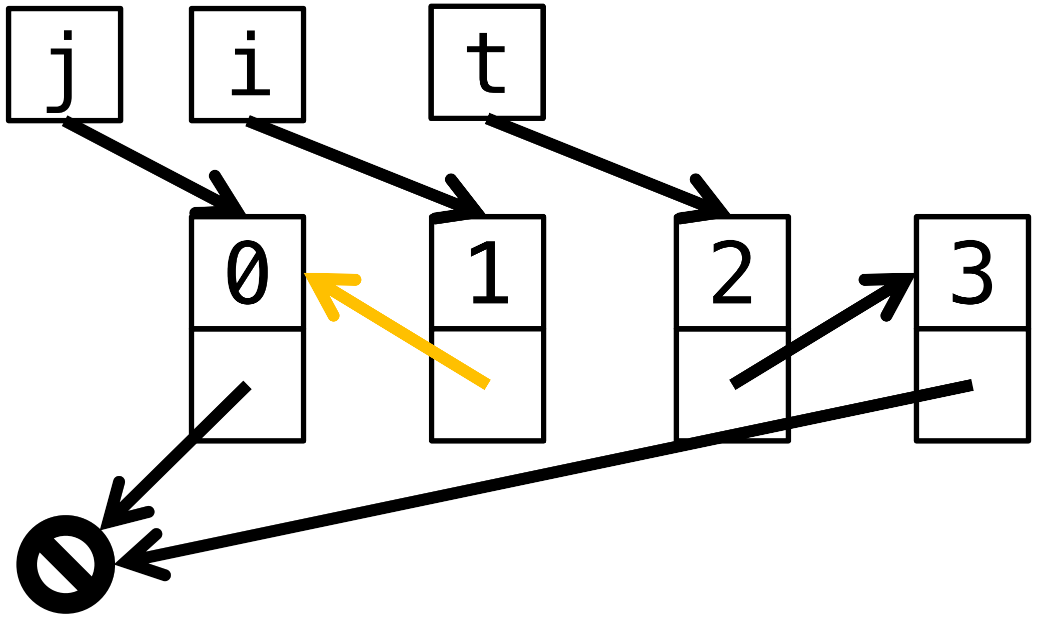
No big deal?

```
while i ≠ nil:
▷  tmp = [i + 1]
   [i + 1] = j
   j = i
   i = tmp
```



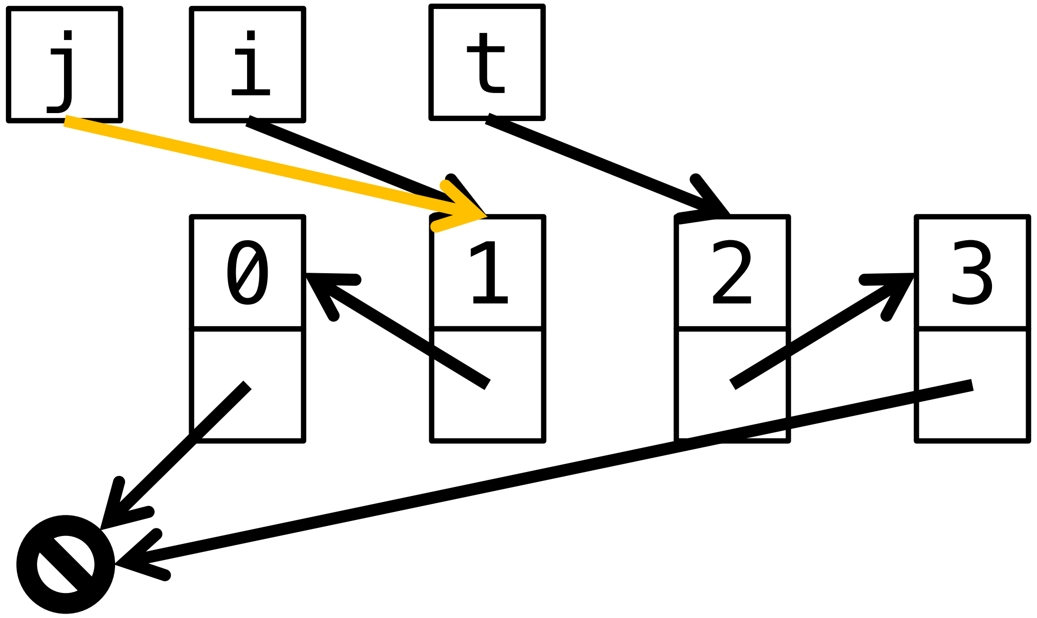
No big deal?

```
while i ≠ nil:
    tmp = [i + 1]
▷   [i + 1] = j
    j = i
    i = tmp
```



No big deal?

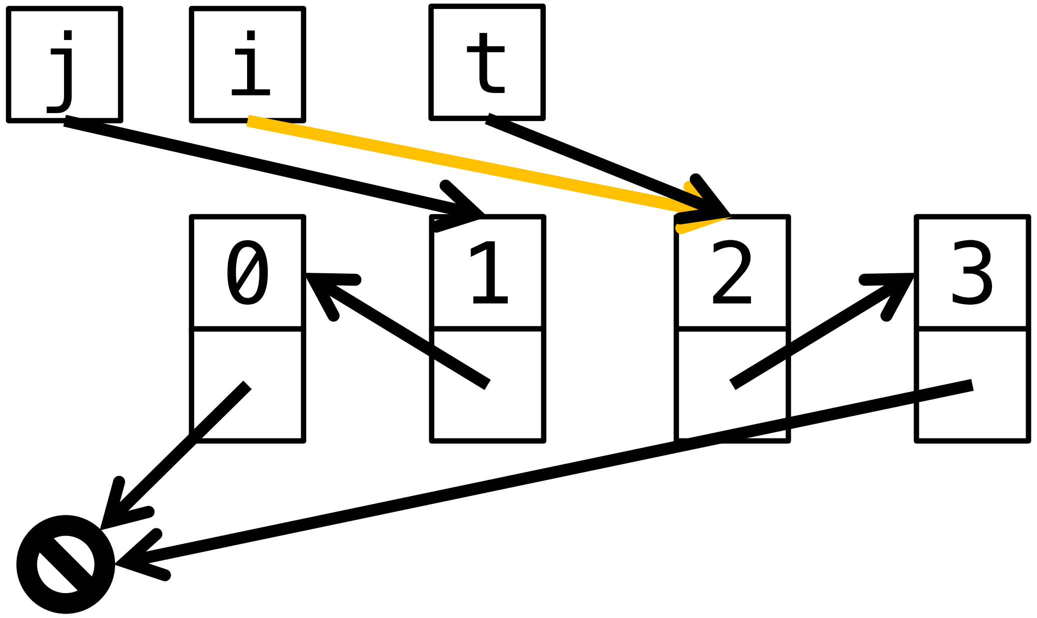
```
while i ≠ nil:
    tmp = [i + 1]
    [i + 1] = j
▷   j = i
    i = tmp
```



No big deal?

```
while i ≠ nil:
    tmp = [i + 1]
    [i + 1] = j
    j = i
▷   i = tmp
```

What is this loop's invariant?



Loop invariant (in plain logic)

(∃ α β, linkedlist(i, α)
        ∧ linkedlist(j, β)
        ∧ $\alpha_0^\dagger = \alpha^\dagger.\beta$)

∧ disjointlinkedlists(i, j)

∧ … for all other data in the program, guarantee that they are disjoint from those two lists …

Loop invariant (in separation logic)

```
(∃ α β, linkedlist(i, α)
      * linkedlist(j, β)
      ∧ α_∅† = α†.β)
```

$$(\exists\ \alpha\ \beta,\ \mathtt{linkedlist}(i, \alpha)\ * \ \mathtt{linkedlist}(j, \beta)\ \wedge\ \alpha_{\emptyset}^{\dagger} = \alpha^{\dagger}.\beta)$$

- Separation logic introduces:
  - a notion of **domain** (or **footprint**) for each assertion
  - a notion of conjunction capturing the idea that the domains are **disjoint** and **precise**

Separation logic (disjointness)

These two assertions are now different:

`x↦0 ∧ y↦0`

x points to 0,
and y points to 0

`x↦0 * y↦0`

x points to 0,
and y points to 0,
<u>and x and y do not alias</u>

Aliasing

WRONG:

```
{x↦0 ∧ y↦0} *x = 4 {x↦4 ∧ y↦0}
```

CORRECT:

```
{x↦0 * y↦0} *x = 4 {x↦4 * y↦0}
```

Verifying linked list reversal

I: $\exists\ \alpha\ \beta,\ ll(i, \alpha) * ll(j, \beta) \land \alpha_0^\dagger = \alpha^\dagger.\beta$

```
while i ≠ nil:
I ∧ i ≠ nil
    tmp = [i + 1]

    [i + 1] = j

    j = i

    i = tmp
I
```

Frame rule

- Separation logic formulae are tight:

  `{x↦0 * y↦0} [x] := 4 {x↦4}`

  `is wrong!`

- The frame rule allows relaxing them:

$$\frac{\{P\}\ S\ \{Q\}}{\{P * R\}\ S\ \{Q * R\}}$$

modularity in terms or memory!

FINAL NOTES

There’s more!

- Separation logic is one of a multitude of sub-structural logics, which let us account for facts as “resources” within logic
  - see Rust’s ownership/borrowing
- Functional languages can also benefit
  - see Hoare Type Theory
  - see work on dependent and linear types
- Lots of extensions (concurrent, probabilistic, …)