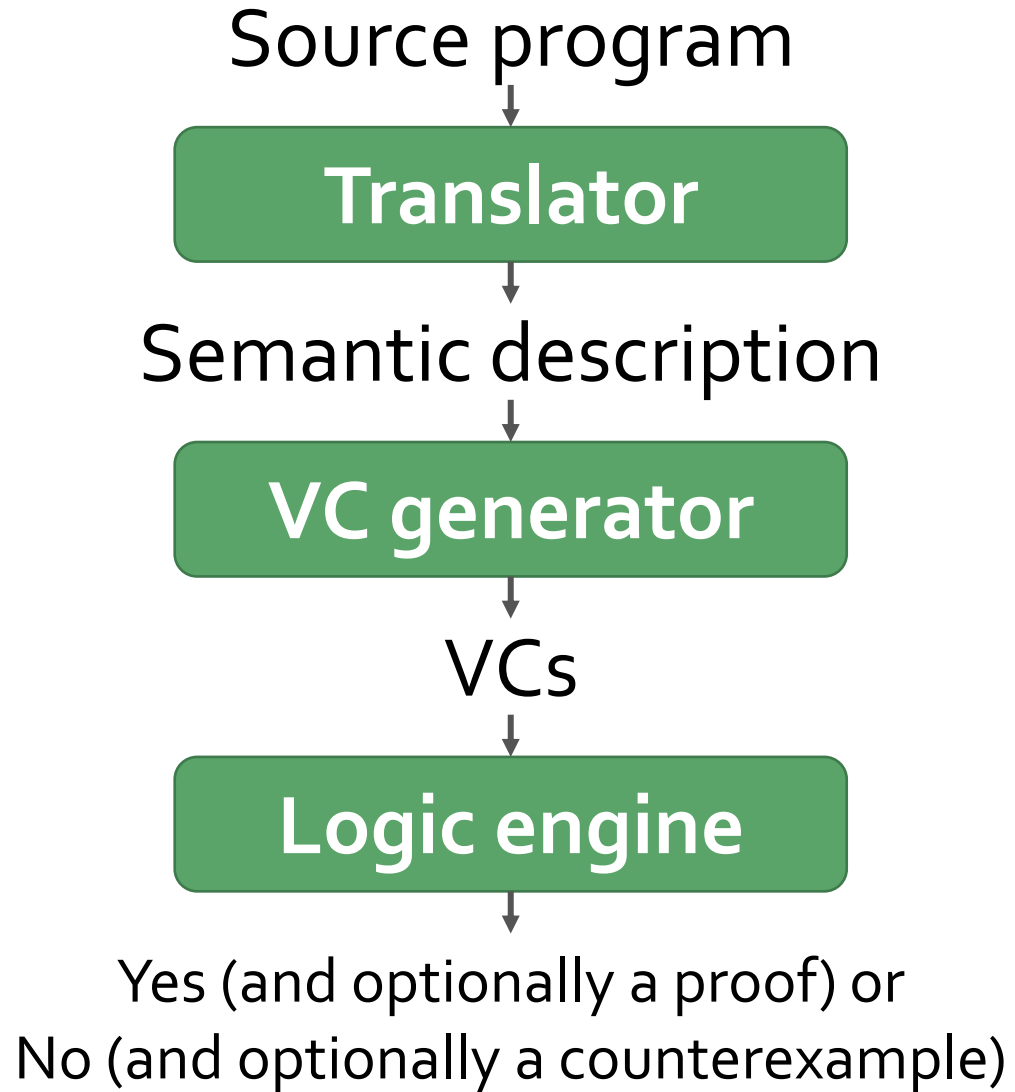


Rethinking the verification pipeline

K. Rustan M. Leino

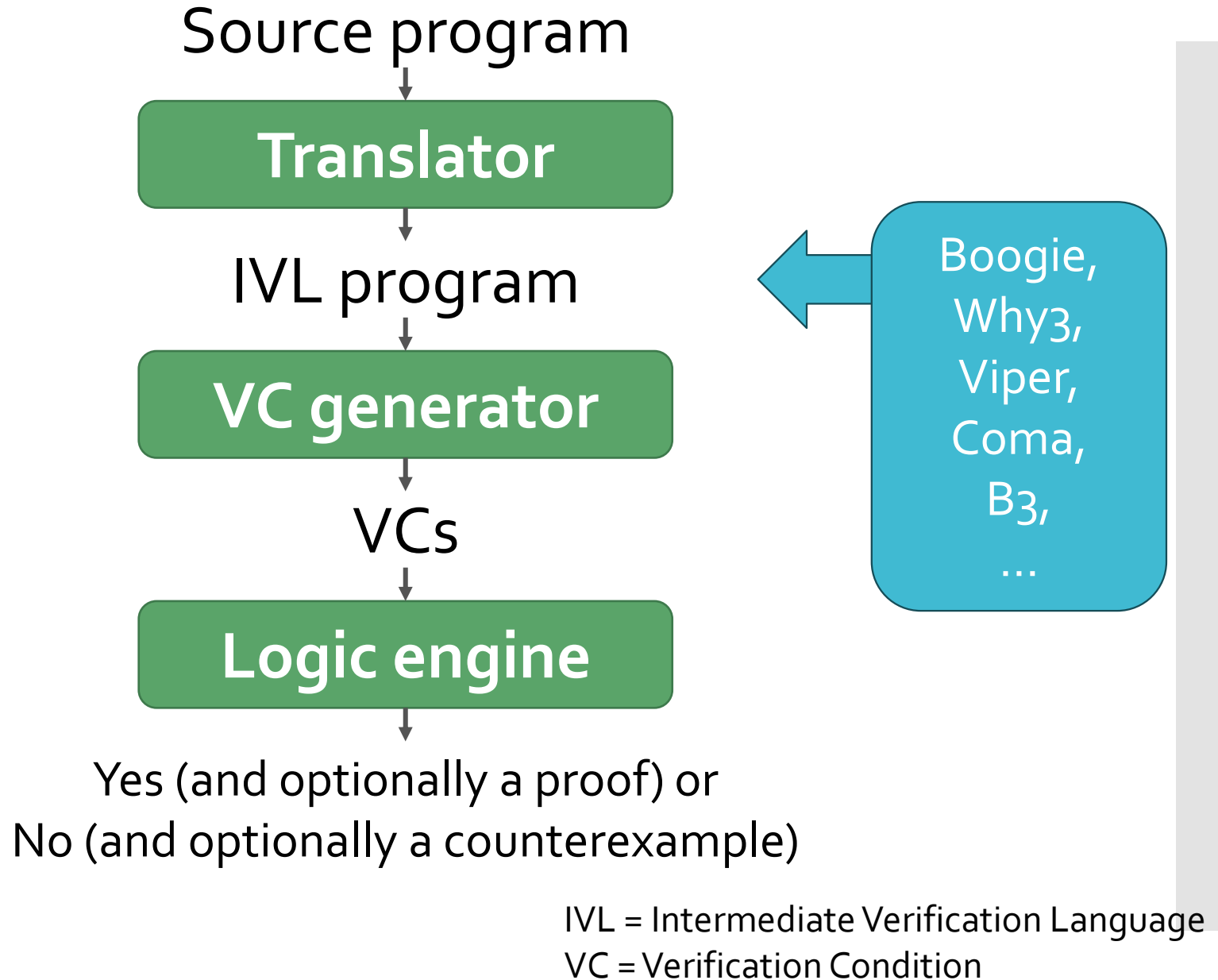
10 September 2026
IFIP WG 2.3 meeting #70
Southampton, UK

Verifier architecture

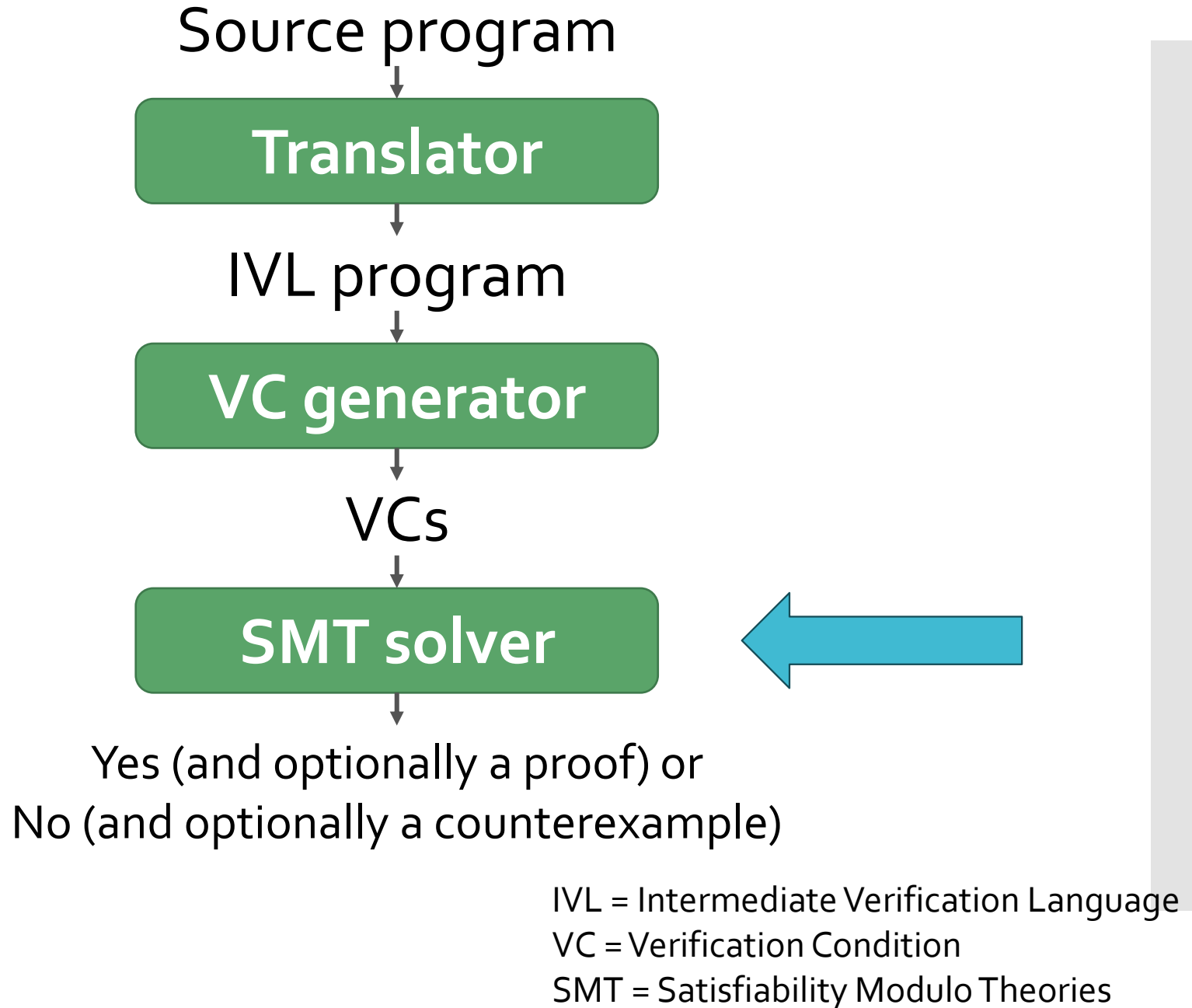


VC = Verification Condition

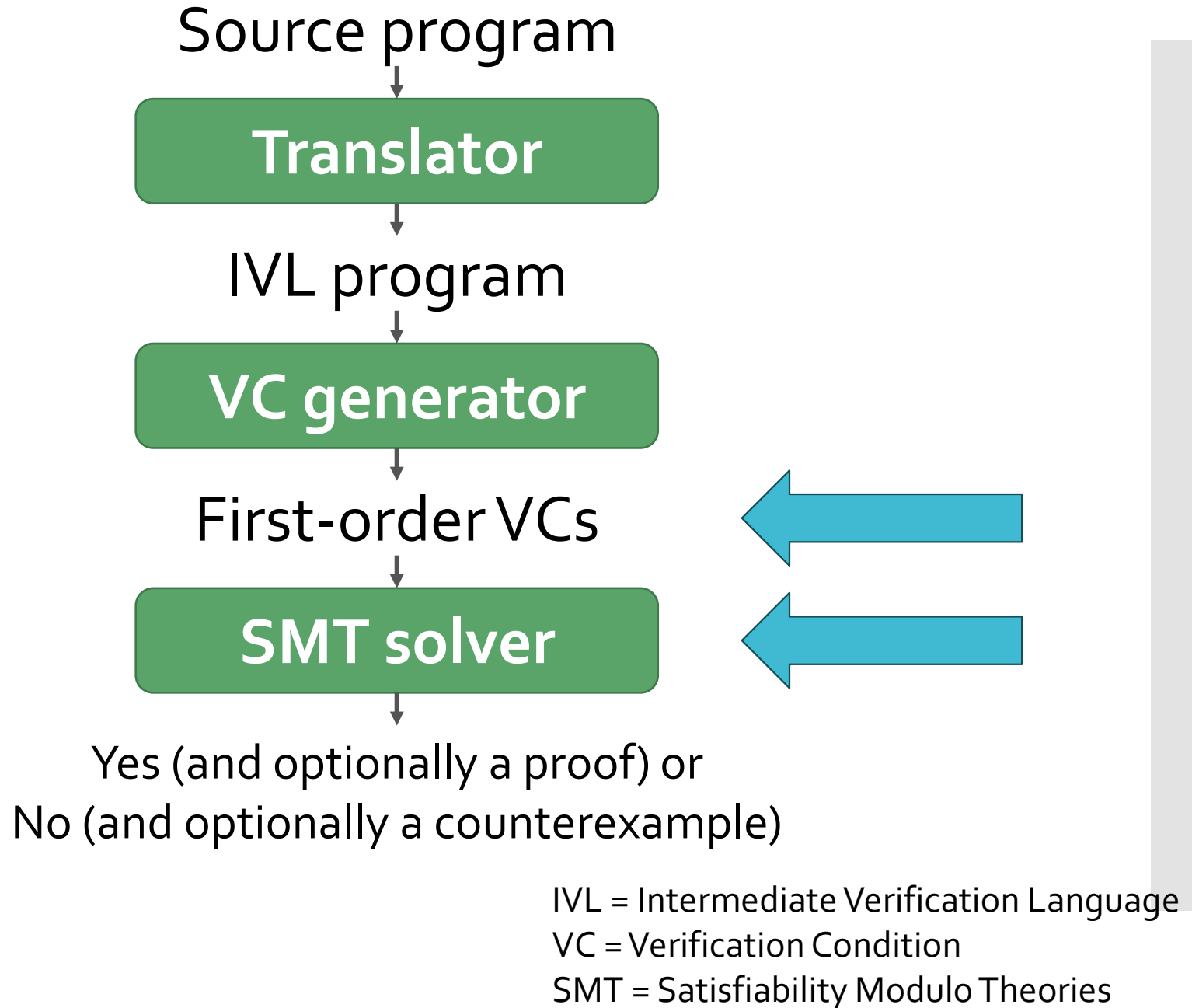
Verifier architecture



Verifier architecture



Verifier architecture



Things we treasure

- Effective programming language
 - Presents good concepts, methodology
 - Syntactic consistency
 - Familiar syntax
- Readable specifications
 - Abstraction, information hiding
- Modular verification
- IVL descriptions
 - Stateful $(x := e)$
 - Partial operations (assert)
 - Nondeterminism $(x := *)$
 - Partial statements (assume)
- Verifier interaction at source level
- Quick verifier turnaround
 - First-order
 - One-shot proofs

Or do we?

- Effective programming language
 - Presents good concepts, methodology
 - Syntactic consistency
 - Familiar syntax
- Readable specifications
 - Abstraction, information hiding
- Modular verification
- IVL descriptions
 - Stateful $(x := e)$
 - Partial operations (assert)
 - Nondeterminism $(x := *)$
 - Partial statements (assume)
- Verifier interaction at source level
- Quick verifier turnaround
 - First-order
 - One-shot proofs



Intermediate Verification Language

- Thinking tool
 - Helps you correctly define source-language semantics
- Separates front-end (semantics) and back-end (logic engine)
- 💡 Use functional language as IVL
 - Executable semantics?
 - Same/similar terms as in logic engine

State changes

```
if e {  
    x := 5  
} else {  
    y := 7  
}  
z := x + y
```

```
let z :=  
    let (x, y) :=  
        if e then (5, y) else (x, 7)  
    in  
        x + y
```

Proof obligations

```
assert wf[[e]] ∧ wf[[f]]  
r := e + f
```

```
datatype Outcome<T> =  
  | Normal(value: T)  
  | Wrong
```

```
eval(e + f) :=  
  match eval(e)  
  case Wrong ⇒ Wrong  
  case Normal(x) ⇒  
    match eval(f)  
    case Wrong ⇒ Wrong  
    case Normal(y) ⇒  
      Normal(x + y)
```

Nondeterminism

```
x := pick()
```

Nondeterminism

```
(x, st) := pick(st)
```

Memory allocation

```
p := new
```

```
(p, alloc) := (alloc, alloc + 1)
```

Memory allocation

```
p := new
```

```
if avail  $\neq \emptyset$  {  
  p : $\in$  avail  
  avail := avail - {p}  
else  
  assert false
```

Memory allocation

```
p := new
```

```
if avail  $\neq \emptyset$  {  
    p : $\in$  avail  
    avail := avail - {p}  
else  
    return Outcome.OutOfMemory
```

Memory allocation

```
p := new
```

```
if avail  $\neq \emptyset$  {  
  p : $\in$  avail  
  avail := avail - {p}  
else  
  assume false
```

Translating calc

```
calc {  
    E0;  
    == { Hint0 }  
    E1;  
    < { Hint1 }  
    E2;  
}
```

```
Hint0  
assert E0 == E1  
assume false  
[]  
Hint1  
assert E1 < E2  
assume false  
[]  
assume E0 < E2
```

Loops without explicit fixpoint

```
loop  
  invariant J  
{  
  Body  
}
```

let x denote the variables
assigned to by Body

```
assert J
```

Check the invariant on entry

```
x := *  
assume J
```

} Fast forward to an arbitrary
loop iteration

```
Body
```

Start in on the next iteration

```
assert J
```

Check invariant is maintained

```
assume false
```

Stop checking

Logic engine

- Can we design a logic engine for a stateful IVL?
 - After all, a program can be (clumsily) encoded as terms in the logic
- What gives a foundational proof assistant its power?
- SMT is almost there, but doesn't have general binders (other than $\forall$ and $\exists$)
 - Has anyone added binders to e-graphs? It seems a natural extension.