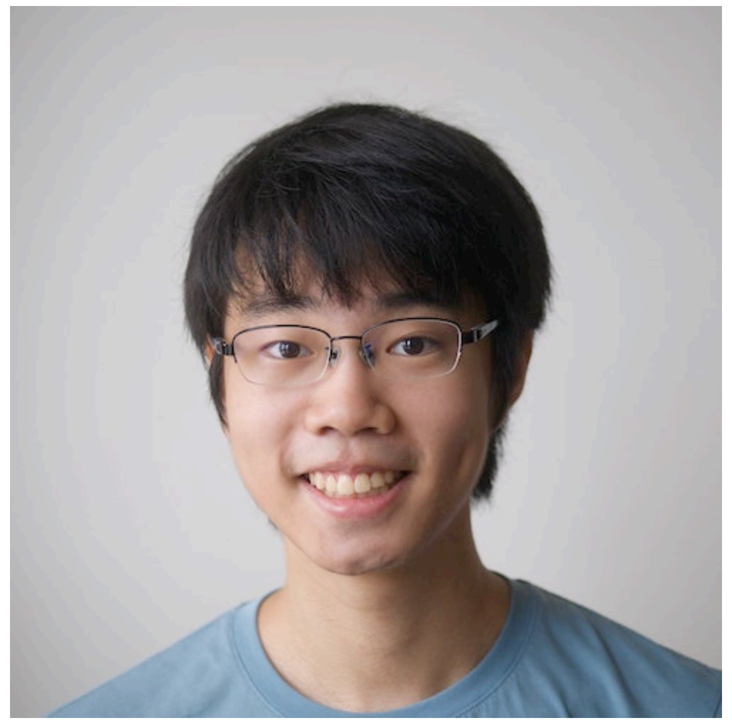
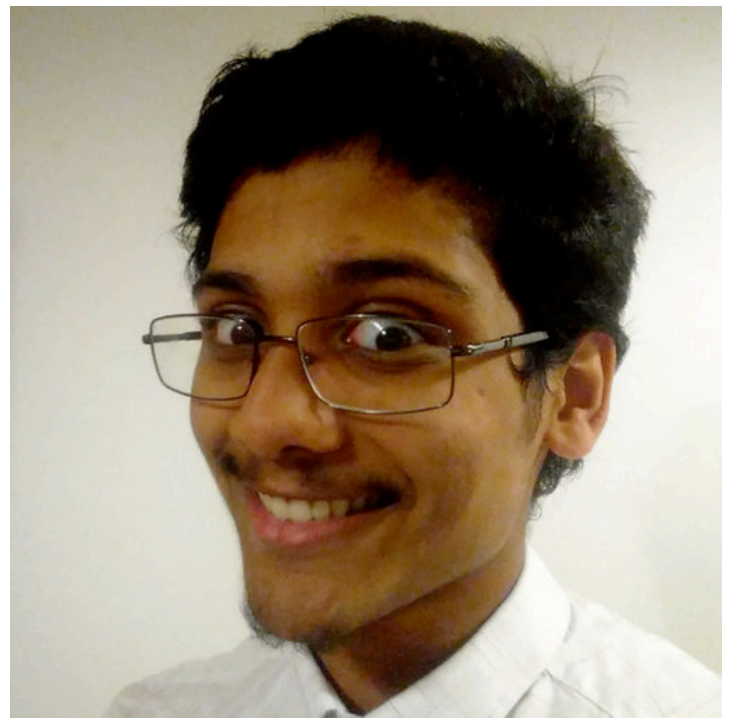
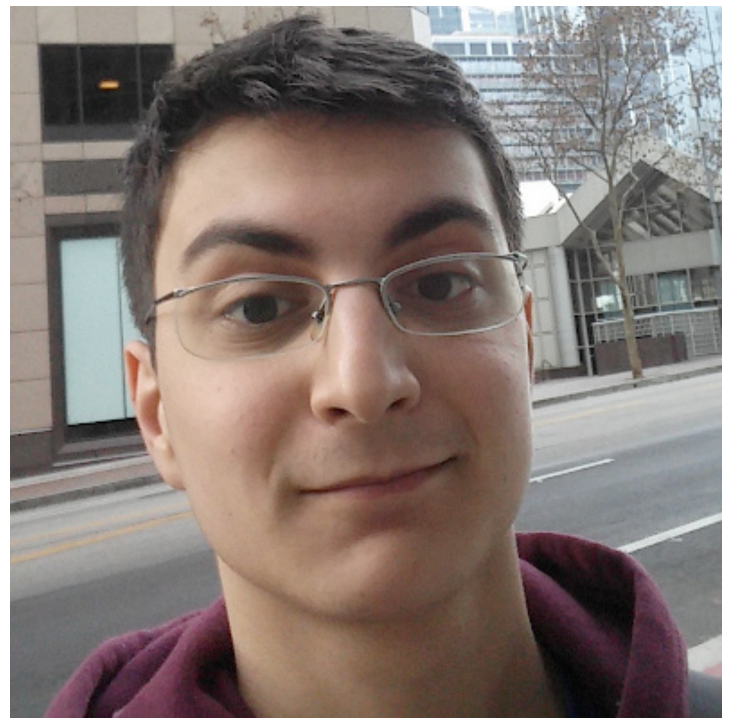
Certifying the Synthesis of Heap-Manipulating Programs

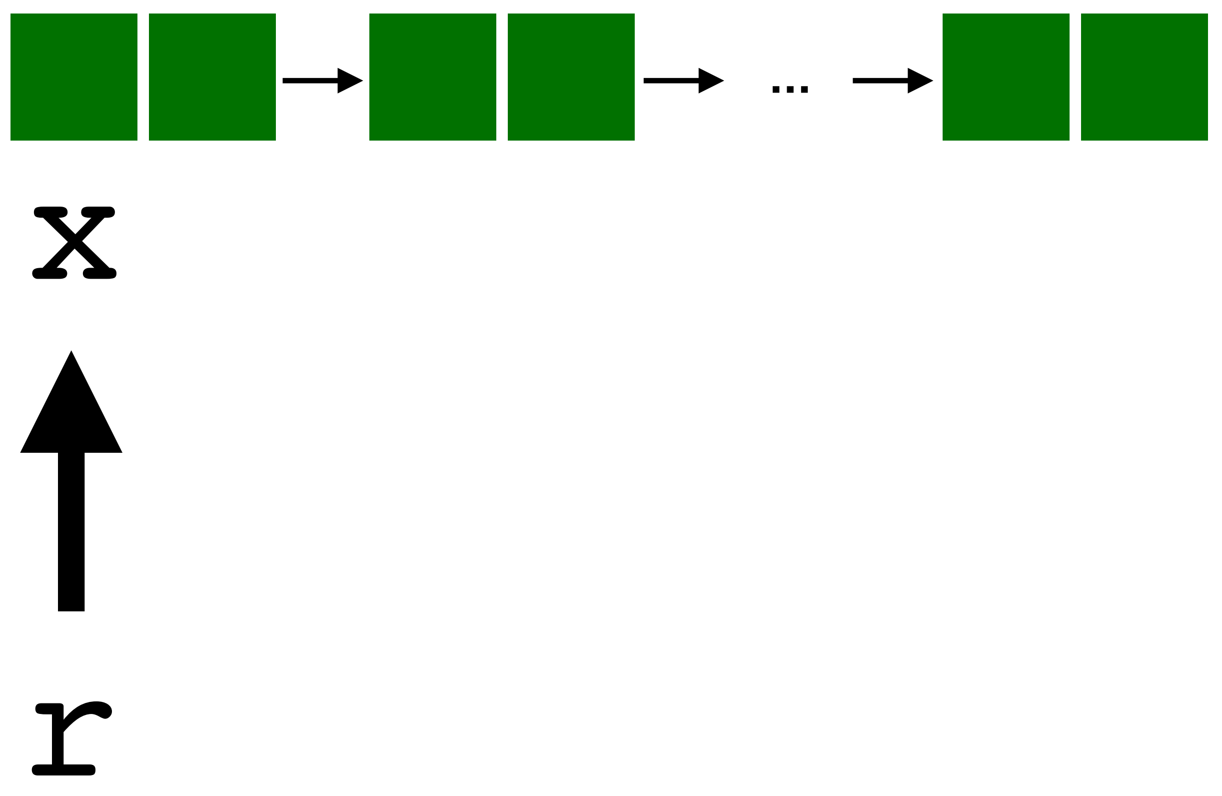
Yasunari Watanabe Kiran Gopinathan George Pîrlea Nadia Polikarpova Ilya Sergey

Yale-NUS College

NUS National University of Singapore

UC San Diego

Copying a linked list

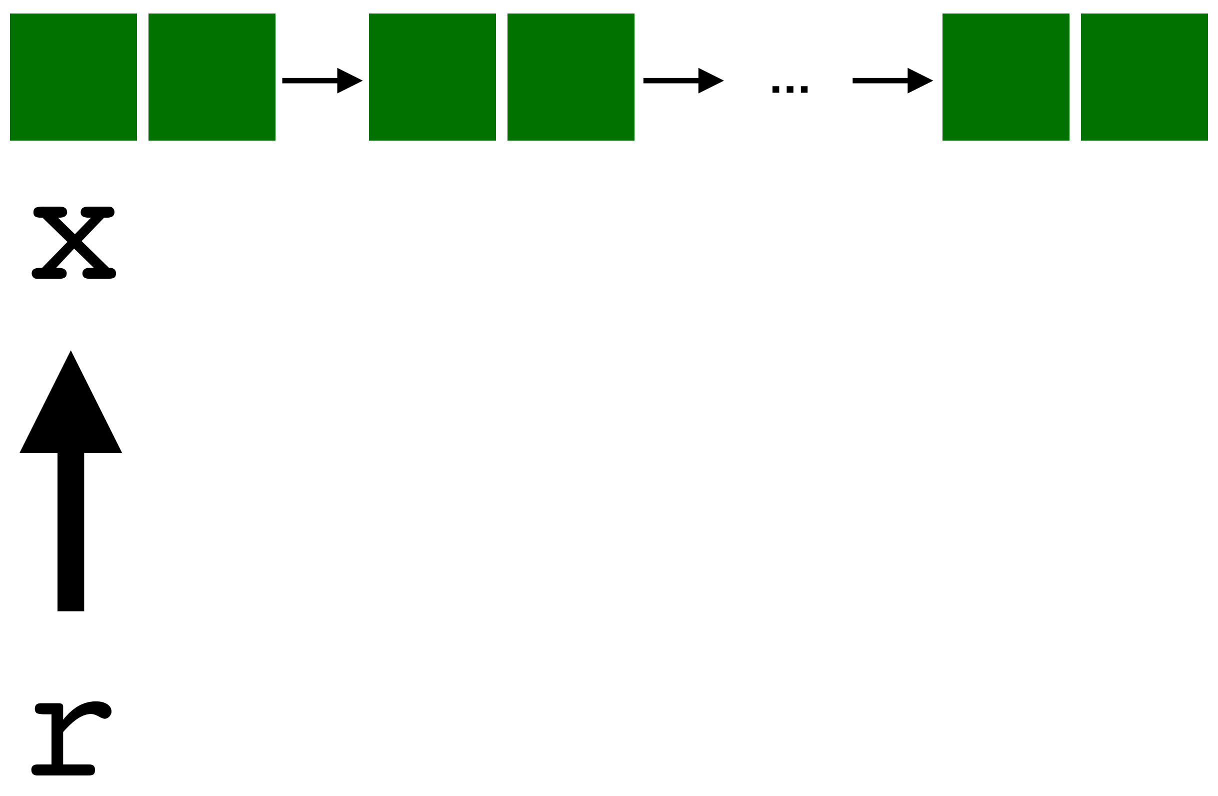


$$\{\mathtt{r} \mapsto x * \mathsf{sll}(x, s)\}$$

```
void sll_copy(loc r)
```

$$\{\mathtt{r} \mapsto y * \mathsf{sll}(x, s) * \mathsf{sll}(y, s)\}$$

Copying a linked list



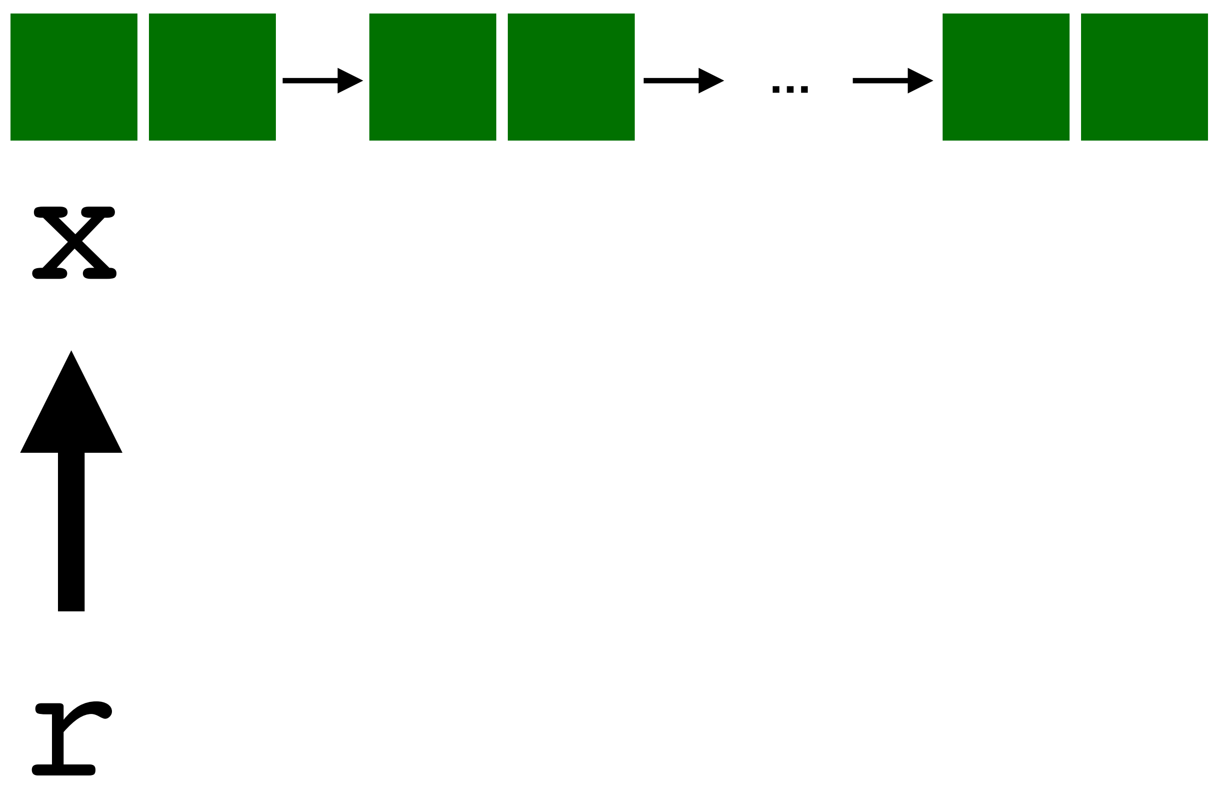
r “points-to” x

$$\{\mathtt{r} \mapsto x * \mathsf{sll}(x, s)\}$$

```
void sll_copy(loc r)
```

$$\{\mathtt{r} \mapsto y * \mathsf{sll}(x, s) * \mathsf{sll}(y, s)\}$$

Copying a linked list



r “points-to” x

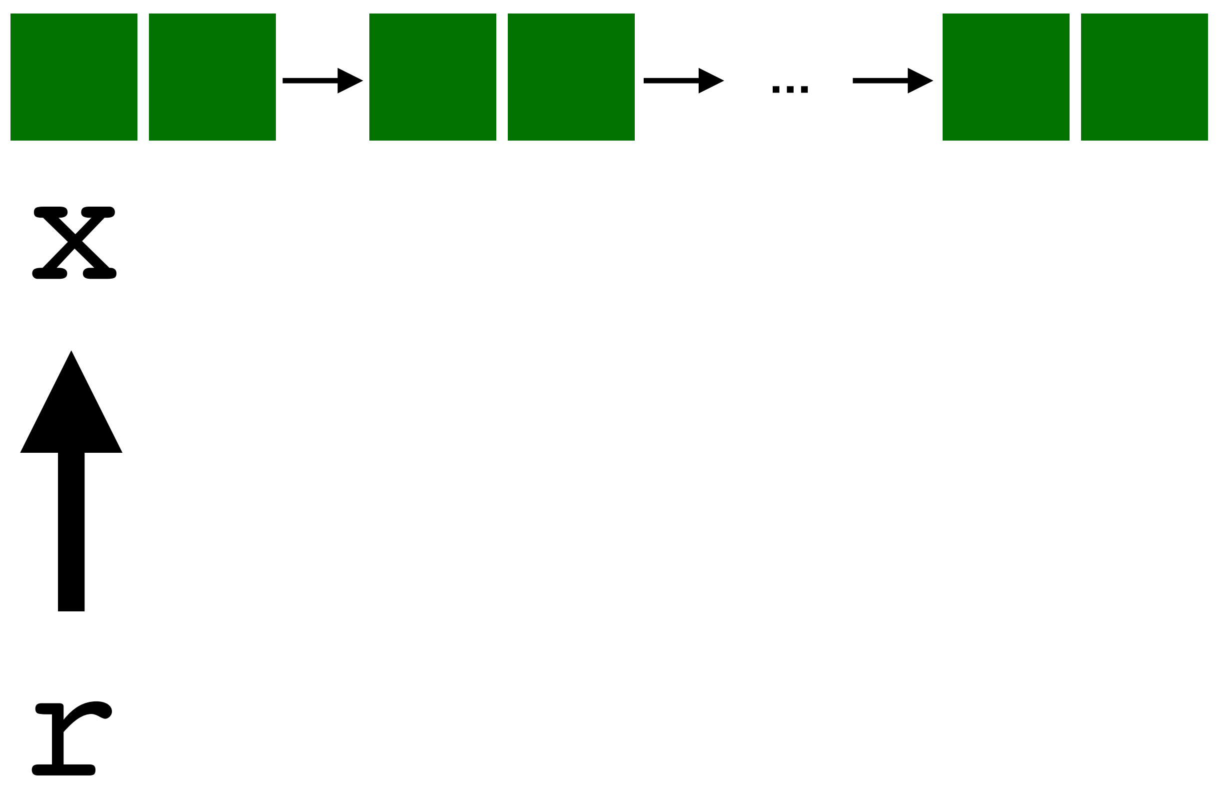
$$\{\mathtt{r} \mapsto x * \boxed{\mathsf{sll}(x, s)}\}$$

inductive predicate “singly-linked list”

```
void sll_copy(loc r)
```

$$\{\mathtt{r} \mapsto y * \mathsf{sll}(x, s) * \mathsf{sll}(y, s)\}$$

Copying a linked list



r “points-to” x

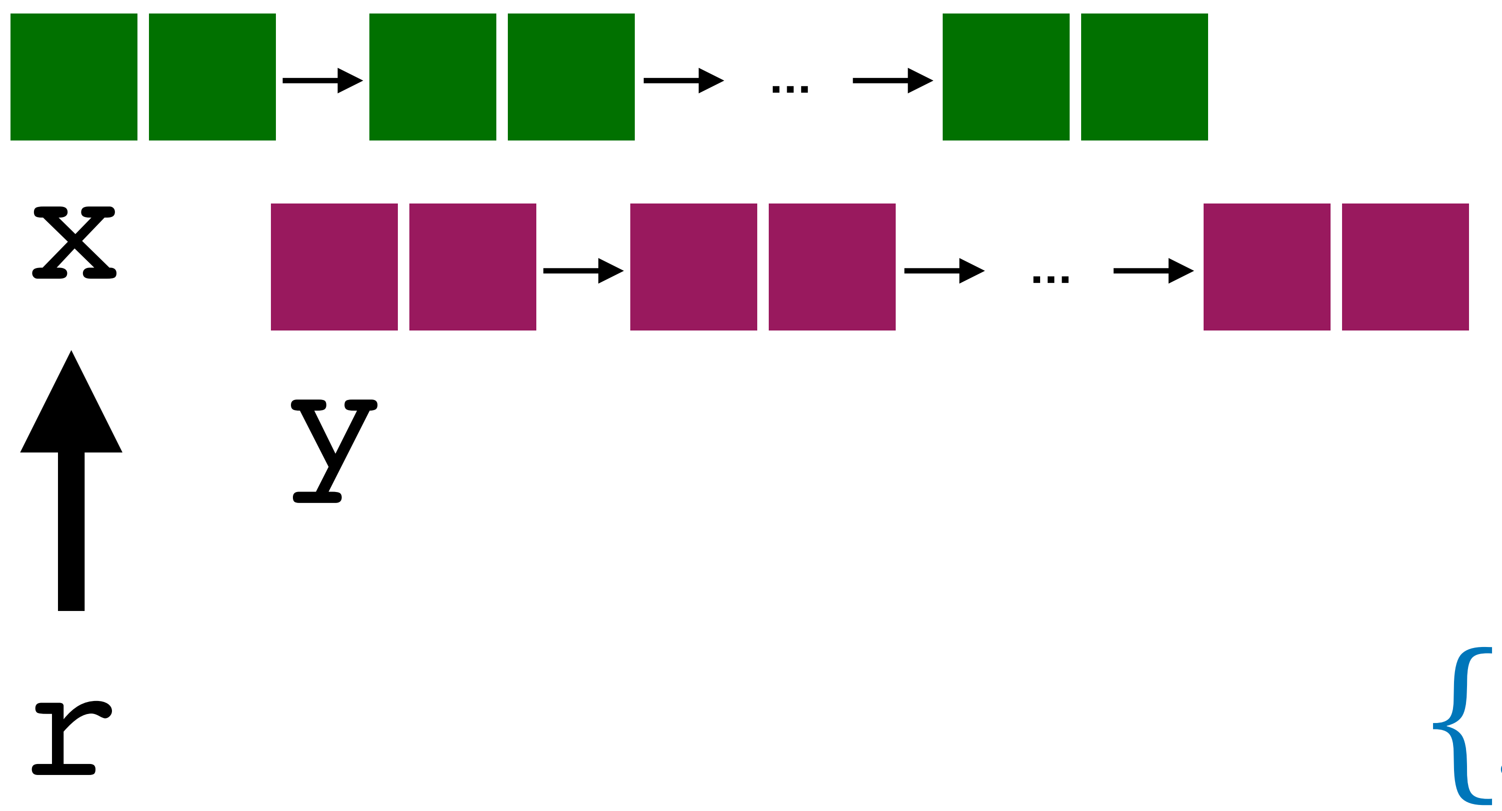
$$\{r \mapsto x * \mathsf{sll}(x, s)\}$$

inductive predicate “singly-linked list”

```
void sll_copy(loc r)
```

$$\{r \mapsto y * \boxed{\mathsf{sll}(x, s)} * \mathsf{sll}(y, s)\}$$

Copying a linked list



r “points-to” x

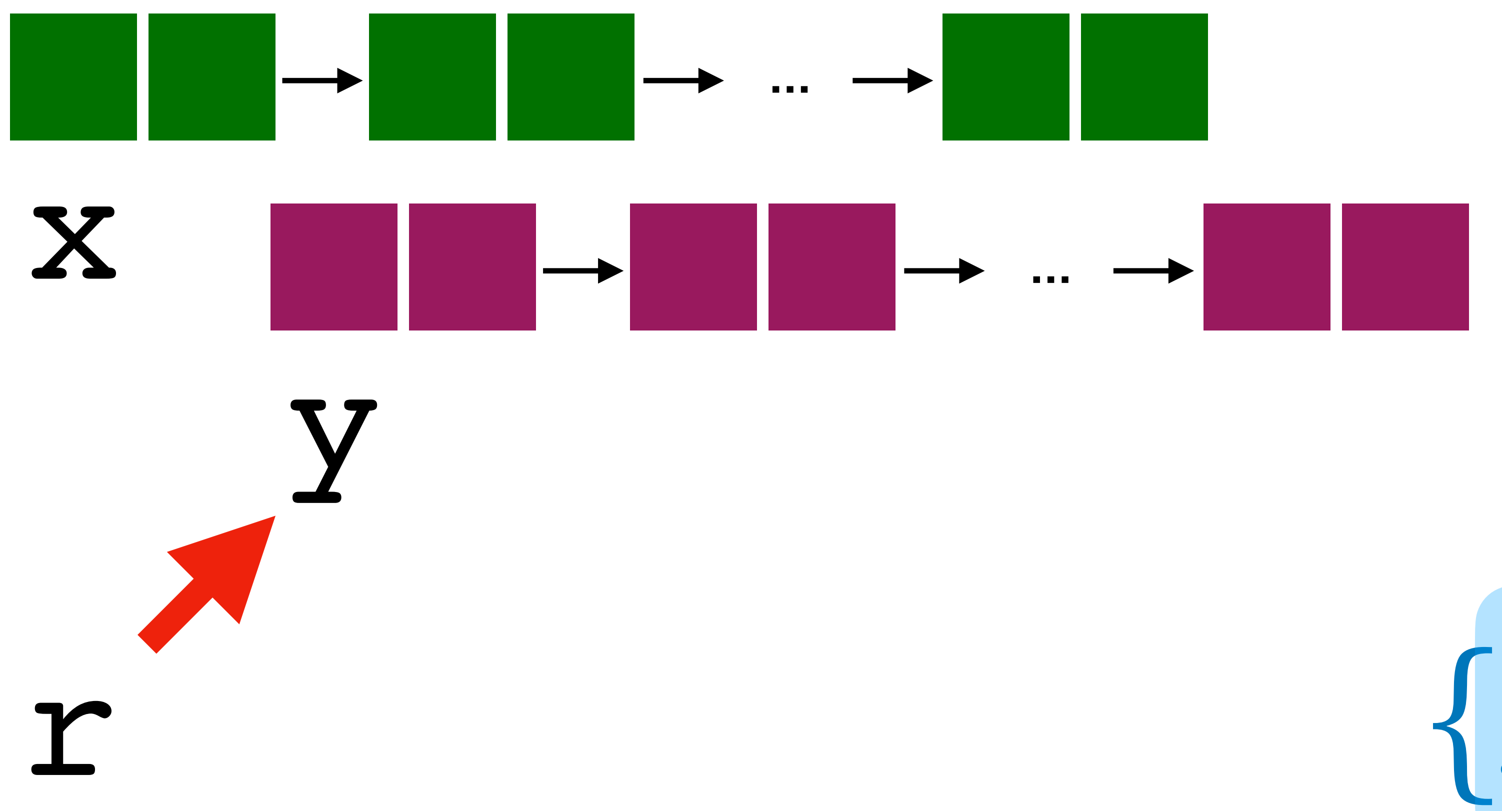
$$\{r \mapsto x * \mathsf{sll}(x, s)\}$$

inductive predicate
“singly-linked list”

```
void sll_copy(loc r)
```

$$\{r \mapsto y * \mathsf{sll}(x, s) * \boxed{\mathsf{sll}(y, s)}\}$$

Copying a linked list



r “points-to” x

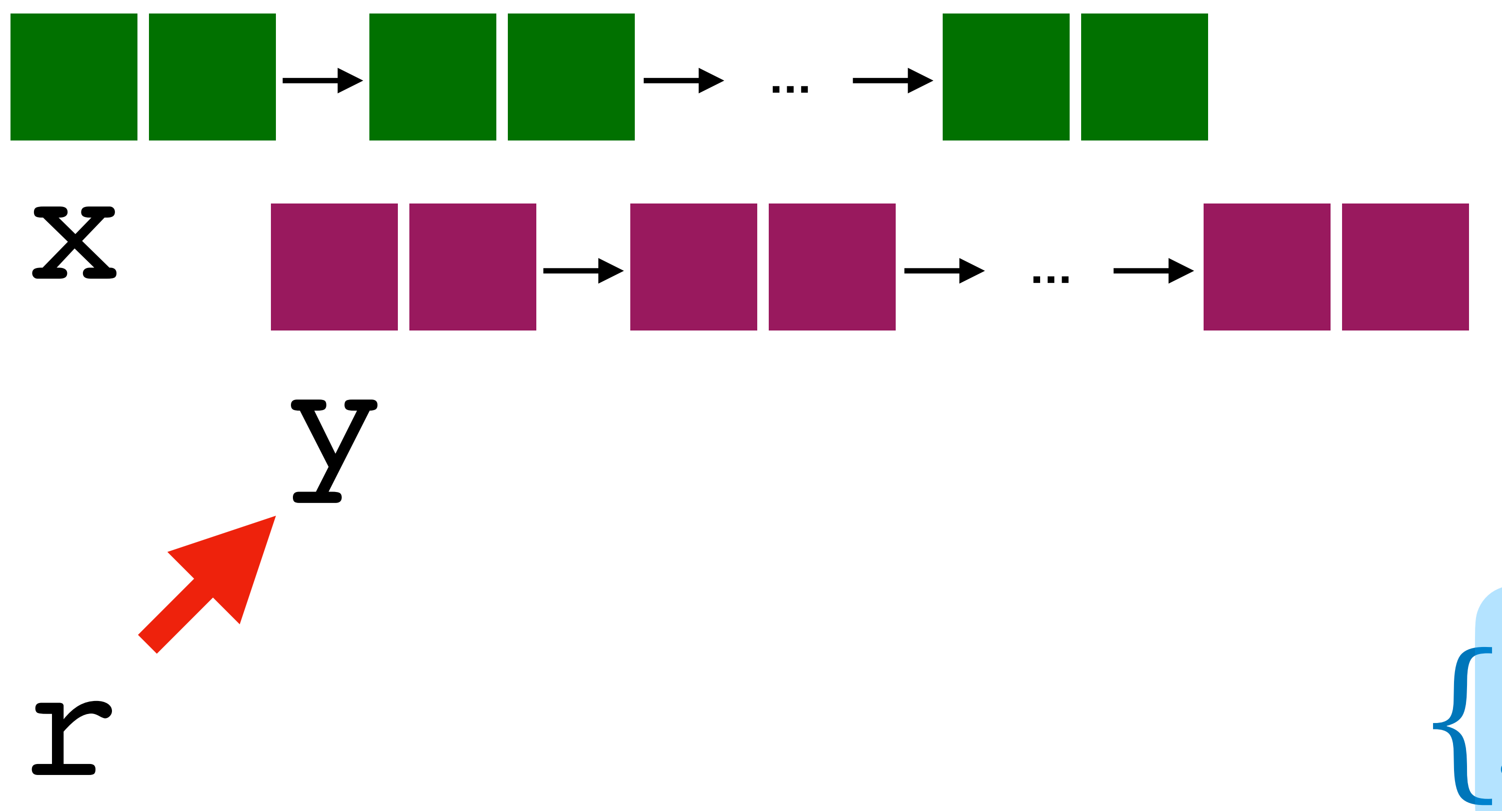
$\{\mathsf{r} \mapsto x * \mathsf{sll}(x, s)\}$

inductive predicate
“singly-linked list”

```
void sll_copy(loc r)
```

$\{\boxed{\mathsf{r} \mapsto y} * \mathsf{sll}(x, s) * \mathsf{sll}(y, s)\}$

Copying a linked list



r “points-to” x

$$\{\mathtt{r} \mapsto x * \mathsf{sll}(x, s)\}$$

inductive predicate
“singly-linked list”

```
void sll_copy(loc r)
```

$$\{\boxed{\mathtt{r} \mapsto y} * \mathsf{sll}(x, s) * \mathsf{sll}(y, s)\}$$

separating conjunction

The linked list predicate

$$\textbf{sll}(x, s) \triangleq x = 0 \land \{s = \varnothing, \textbf{emp}\}$$

$$\mid \ x \neq 0 \land \{s = \{v\} \cup s_1 \land [x,2] * \ x \mapsto v \ * (x + 1) \mapsto nxt \ * \textbf{sll}(nxt, s_1)\}$$

The linked list predicate

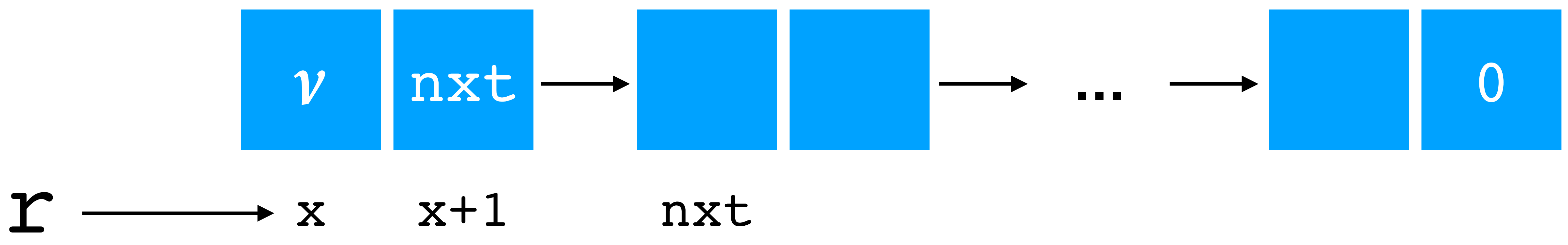
$$\textbf{sll}(x, s) \triangleq x = 0 \land \{s = \emptyset, \textbf{emp}\}$$

$$\mid \; x \neq 0 \land \{s = \{v\} \cup s_1 \land [x,2] * \; x \mapsto v \; * \; (x+1) \mapsto nxt \; * \; \textbf{sll}(nxt, s_1)\}$$

The linked list predicate

$$\textsf{sll}(x, s) \triangleq x = 0 \land \{s = \emptyset, \textsf{emp}\}$$

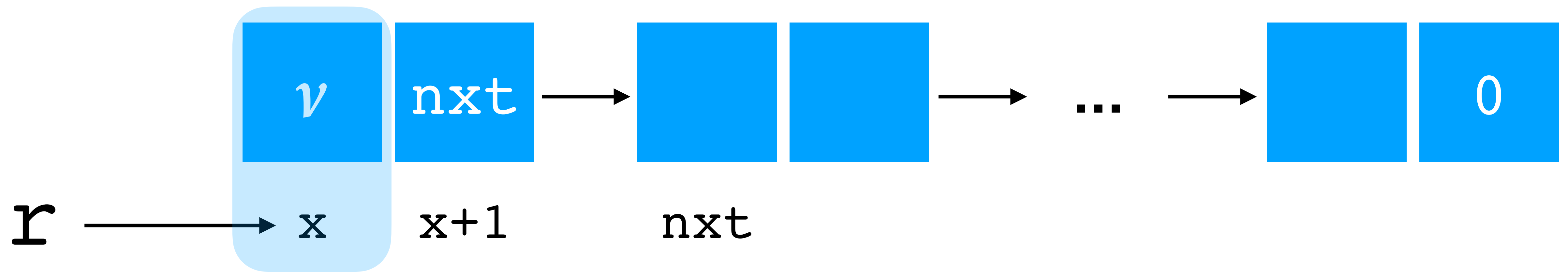
$$\mid\ x \neq 0 \land \{s = \{v\} \cup s_1 \land [x,2] * x \mapsto v * (x+1) \mapsto nxt * \textsf{sll}(nxt, s_1)\}$$



The linked list predicate

$$\textbf{sll}(x, s) \triangleq x = 0 \land \{s = \emptyset, \textbf{emp}\}$$

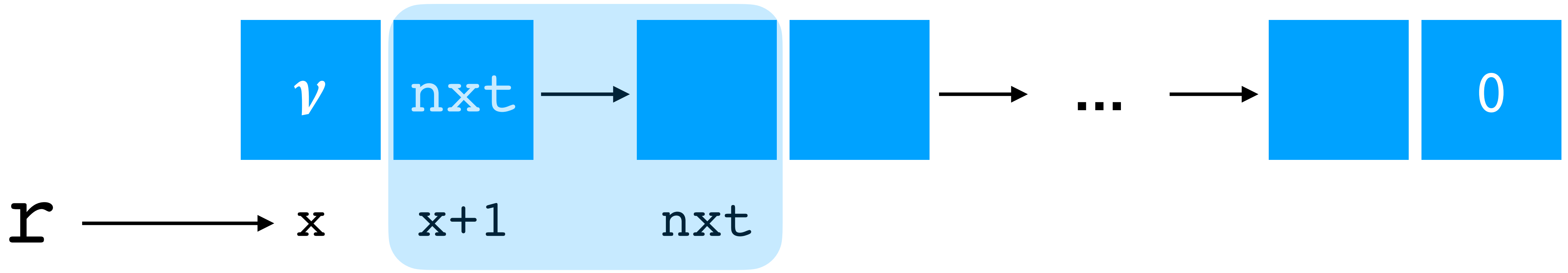
$$\mid \;\; x \neq 0 \land \{s = \{v\} \cup s_1 \land [x,2] * x \mapsto v * (x+1) \mapsto nxt * \textbf{sll}(nxt, s_1)\}$$



The linked list predicate

$$\mathbf{sll}(x, s) \triangleq x = 0 \wedge \{s = \emptyset, \mathbf{emp}\}$$

$$\mid \;\; x \neq 0 \wedge \{s = \{v\} \cup s_1 \wedge [x,2] * x \mapsto v * (x+1) \mapsto nxt * \mathbf{sll}(nxt, s_1)\}$$



Write your own code?

Spec

$\{\mathsf{r} \mapsto x * \mathsf{sll}(x, s)\}$

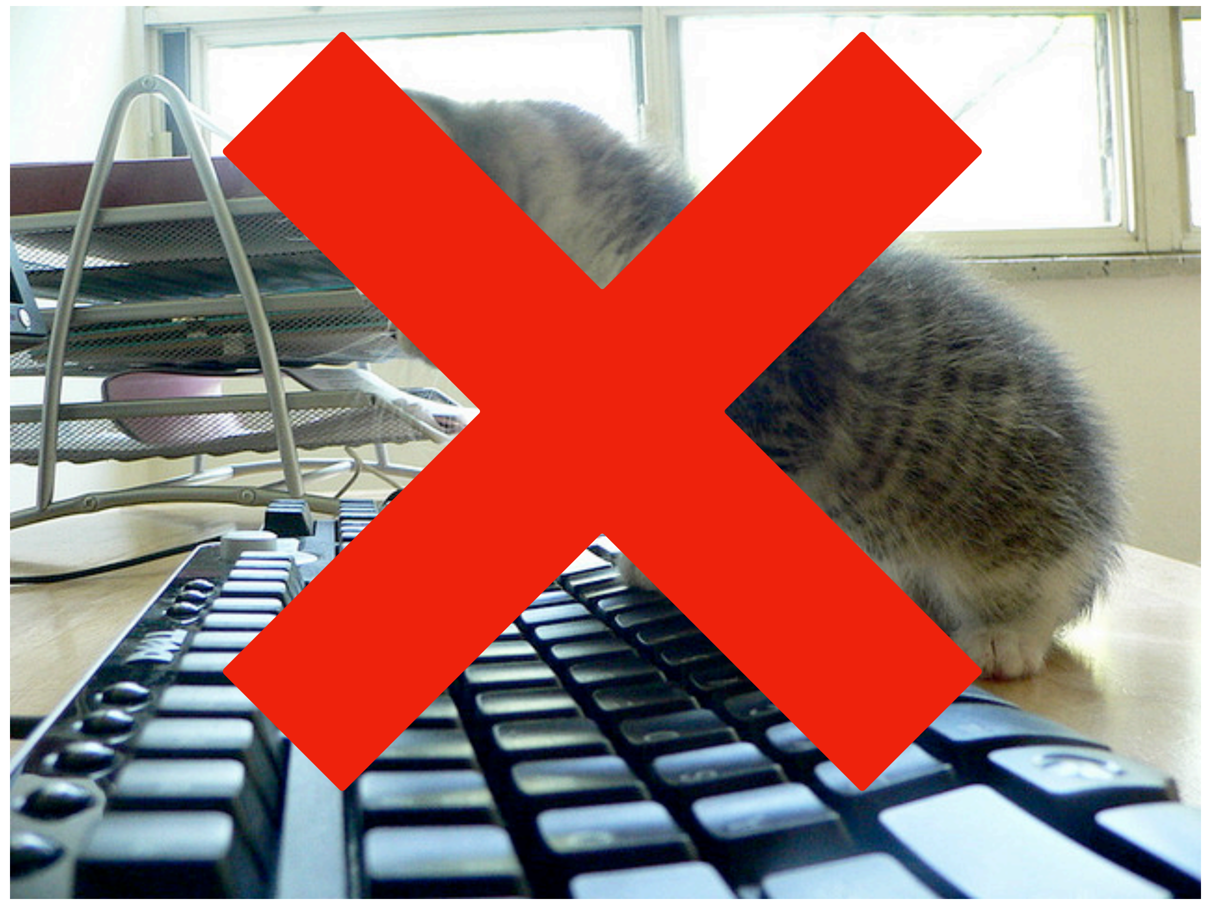
```
void sll_copy(loc r)
```

$\{\mathsf{r} \mapsto y * \mathsf{sll}(x, s) * \mathsf{sll}(y, s)\}$

Program

Write your own code?

Spec



Program

$\{\mathsf{r} \mapsto x * \mathsf{sll}(x, s)\}$

```
void sll_copy(loc r)
```

$\{\mathsf{r} \mapsto y * \mathsf{sll}(x, s) * \mathsf{sll}(y, s)\}$

Synthesize with SuSLik[1]!

Spec → → Program

$\{\mathsf{r} \mapsto x * \mathsf{sll}(x, s)\}$

```
void sll_copy(loc r)
```

$\{\mathsf{r} \mapsto y * \mathsf{sll}(x, s) * \mathsf{sll}(y, s)\}$

[1] Polikarpova and Sergey ’19

Synthesize with SuSLik[1]!

Spec → Program

$\{\mathtt{r} \mapsto x * \mathsf{sll}(x, s)\}$

```
void sll_copy(loc r)
```

$\{\mathtt{r} \mapsto y * \mathsf{sll}(x, s) * \mathsf{sll}(y, s)\}$

Automatically produce an implementation

[1] Polikarpova and Sergey '19

Synthesize with SuSLik[1]!

Spec → **Program**

$\{r \mapsto x * \mathsf{sll}(x, s)\}$

```
void sll_copy(loc r)
```

$\{r \mapsto y * \mathsf{sll}(x, s) * \mathsf{sll}(y, s)\}$

Automatically produce an implementation

```
void sll_copy (loc r) {
  let x2 = *r;
  if (x2 == 0) {}
  else {
    let v = *x2;
    let nxt = *(x2 + 1);
    *r = nxt;
    sll_copy(r);
    let y12 = *r;
    let y2 = malloc(2);
    *r = y2;
    *(y2 + 1) = y12;
    *y2 = v;
  }
}
```

[1] Polikarpova and Sergey ’19

Can I trust this result?

```
void sll_copy (loc r) {
  let x2 = *r;
  if (x2 == 0) {}
  else {
    let v = *x2;
    let nxt = *(x2 + 1);
    *r = nxt;
    sll_copy(r);
    let y12 = *r;
    let y2 = malloc(2);
    *r = y2;
    *(y2 + 1) = y12;
    *y2 = v;
  }
}
```

Can I trust this result?

```
void sll_copy (loc r) {
  let x2 = *r;
  if (x2 == 0) {}
  else {
    let v = *x2;
    let nxt = *(x2 + 1);
    *r = nxt;
    sll_copy(r);
    let y12 = *r;
    let y2 = malloc(2);
    *r = y2;
    *(y2 + 1) = y12;
    *y2 = v;
  }
}
```

What if there's a bug in the synthesizer?

WANTED

A formal guarantee of correctness

This work

This work

Formally guarantee correctness of synthesized programs

This work

Formally guarantee correctness of synthesized programs

with **proof certificates**

This work

Formally guarantee correctness of synthesized programs

with **proof certificates**

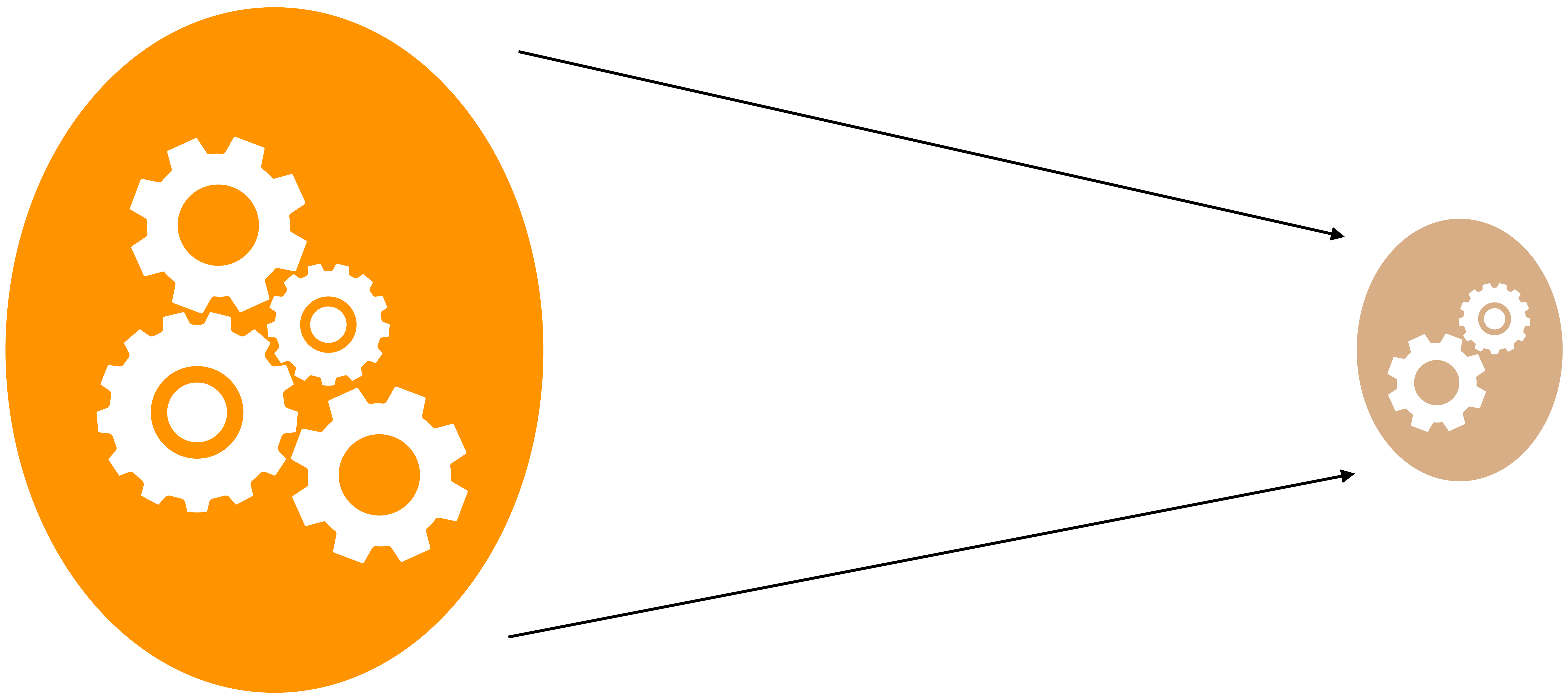
generated via **deductive insight** from synthesis

Shifting the burden of trust

Shifting the burden of trust

SuSLik: Large TCB

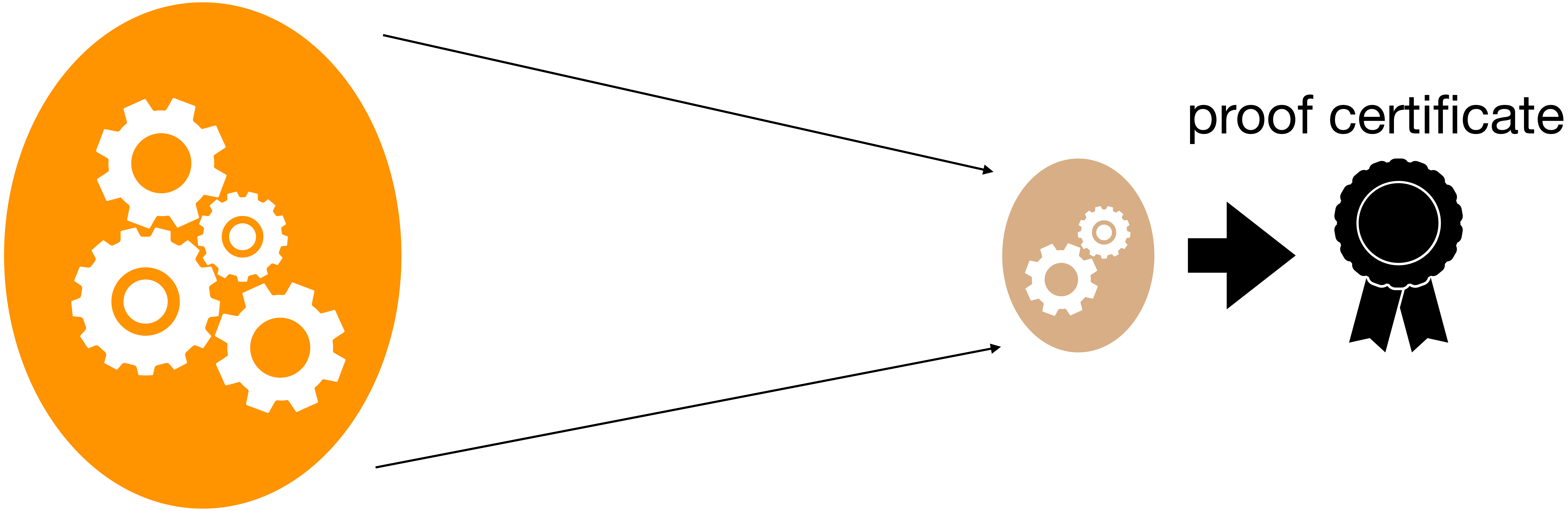
Shifting the burden of trust



SuSLik: Large TCB

Coq: Small TCB

Shifting the burden of trust



SuSLik: Large TCB

Coq: Small TCB

SuSLik codebase: too large to verify

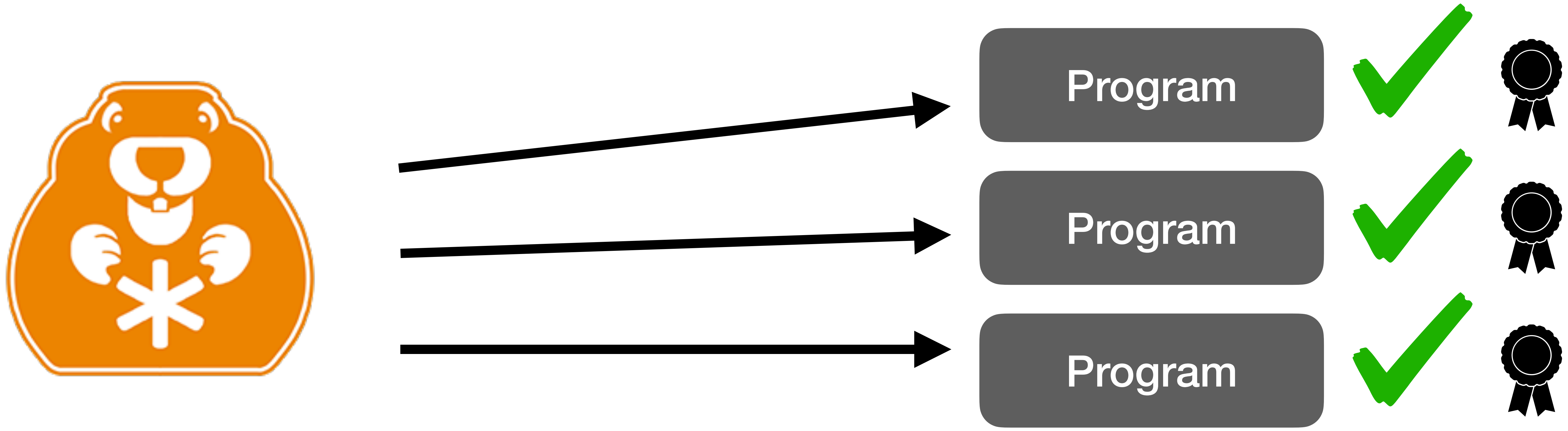
```scala
protected def synthesize(goal: Goal)
                        (stats: SynStats): Option[Solution] = {
  init(goal)
  processWorkList(stats, goal.env.config)
}

@tailrec final def processWorkList(implicit
                                   stats: SynStats,
                                   config: SynConfig): Option[Solution] = {
  // Check for timeouts
  if (!config.interactive && stats.timedOut) {
    throw SynTimeOutException(s"\n\nThe derivation took too long: more than ${config.timeOut} seconds.\n")
  }

  val sz = worklist.length
  log.print(s"Worklist ($sz): ${worklist.map(n => s"${n.pp()}[${n.cost}]").mkString(" ")}", Console.YELLOW
  log.print(s"Succeeded leaves (${successLeaves.length}): ${successLeaves.map(n => s"${n.pp()}").mkStri
  log.print(s"Memo (${memo.size}) Suspended (${memo.suspendedSize})", Console.YELLOW, 2)
  stats.updateMaxWLSize(sz)

  if (worklist.isEmpty) None // No more goals to try: synthesis failed
  else {
    val (node, addNewNodes) = popNode // Select next node to expand
    val goal = node.goal
    implicit val ctx: log.Context = log.Context(goal)
    stats.addExpandedGoal(node)
    log.print(s"Expand: ${node.pp()}[${node.cost}]", Console.YELLOW) //      <goal: ${node.goal.label.pp}>
    log.print(s"${goal.pp}", Console.BLUE)
    trace.add(node)

    // Lookup the node in the memo
    val res = memo.lookup(goal) match {
      case Some(Failed) => { // Same goal has failed before: record as failed
        log.print("Recalled FAIL", Console.RED)
        trace.add(node, Failed, Some("cache"))
        node.fail
        None
      }
      case Some(Succeeded(sol, id)) =>
      { // Same goal has succeeded before: return the same solution
        log.print(s"Recalled solution ${sol._1.pp}", Console.RED)
```

```scala
object OperationalRules extends SepLogicUtils with RuleUtils {

  val exceptionQualifier: String = "rule-operational"

  import Statements._

  /*
  Write rule: create a new write from where it's possible

  Γ ; {φ ; x.f -> l' * P} ; {ψ ; x.f -> l' * Q} ---> S   GV(l) = GV(l') = Ø
  ---------------------------------------------------------------------------- [write]
  Γ ; {φ ; x.f -> l * P} ; {ψ ; x.f -> l' * Q} ---> *x.f := l' ; S

  */
  bject WriteRule extends SynthesisRule with GeneratesCode with InvertibleRule {

      ide def toString: Ident = "Write"

        y(goal: Goal): Seq[RuleResult] = {
          = goal.pre
           = goal.post

          ets have no ghosts
          hosts: Heaplet => Boolean = {
          e PointsTo(x@Var(_), _, e) => !goal.isGhost(x) && e.vars.forall(v => !goal.isGhost(v))
      case _ => false
    }

    // When do two heaplets match
    def isMatch(hl: Heaplet, hr: Heaplet) = sameLhs(hl)(hr) && !sameRhs(hl)(hr) && noGhosts(hr)

    findMatchingHeaplets(_ => true, isMatch, goal.pre.sigma, goal.post.sigma) match {
      case None => Nil
      case Some((hl@PointsTo(x@Var(_), offset, e1), hr@PointsTo(_, _, e2))) =>
        val newPre = Assertion(pre.phi, goal.pre.sigma - hl)
        val newPost = Assertion(post.phi, goal.post.sigma - hr)
        val subGoal = goal.spawnChild(newPre, newPost)
        val kont: StmtProducer = PrependProducer(Store(x, offset, e2)) >> ExtractHelper(goal)

        List(RuleResult(List(subGoal), kont, this, goal))
      case Some((hl, hr)) =>
        ruleAssert(assertion = false, s"Write rule matched unexpected heaplets ${hl.pp} and ${hr.pp}")
        Nil
```

Deductive insight

Deductive insight → post-hoc certification

Deductive insight → post-hoc certification



Program synthesis with SuSLik

Synthetic separation logic (SSL)

SSL rule

Synthetic separation logic (SSL)

Initial goal $\{P\} \rightsquigarrow \{Q\}$

SSL rule

Synthetic separation logic (SSL)

Initial goal $\{P\} \leadsto \{Q\}$

SSL rule

Transformed goal $\{P'\} \leadsto \{Q'\}$

Synthetic separation logic (SSL)

Initial goal $\{P\} \leadsto \{Q\}$

SSL rule

`program statement`

Transformed goal $\{P'\} \leadsto \{Q'\}$

Searching for a program

Initial specification

$\{\mathtt{r} \mapsto x * \mathsf{sll}(x, s)\}$

```
void sll_copy(loc r)
```

$\{\mathtt{r} \mapsto y * \mathsf{sll}(x, s) * \mathsf{sll}(y, s)\}$

Searching for a program

Initial specification

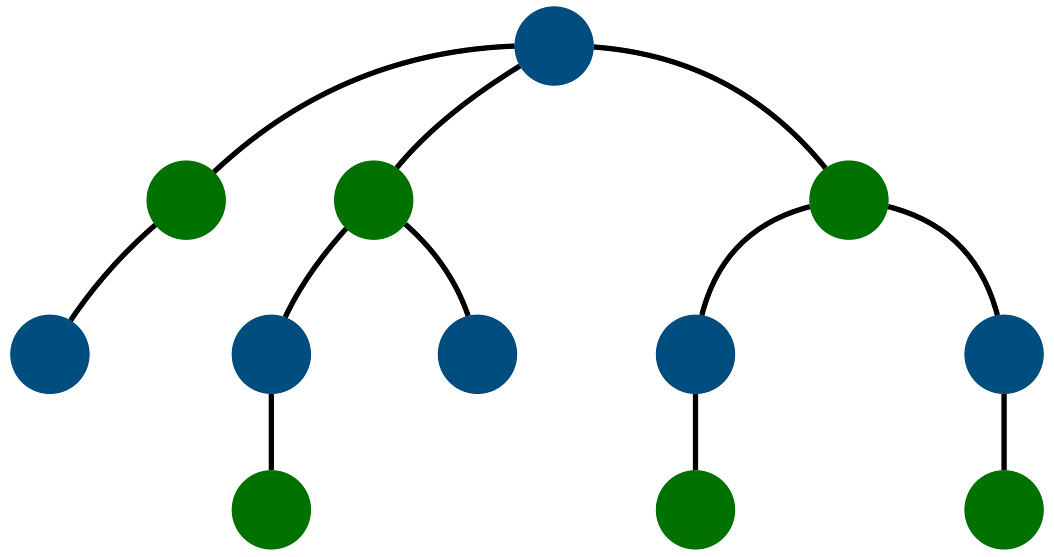
$\{r \mapsto x * \mathsf{sll}(x, s)\}$

```
void sll_copy(loc r)
```

$\{r \mapsto y * \mathsf{sll}(x, s) * \mathsf{sll}(y, s)\}$

➡

Enumerative proof search



Searching for a program

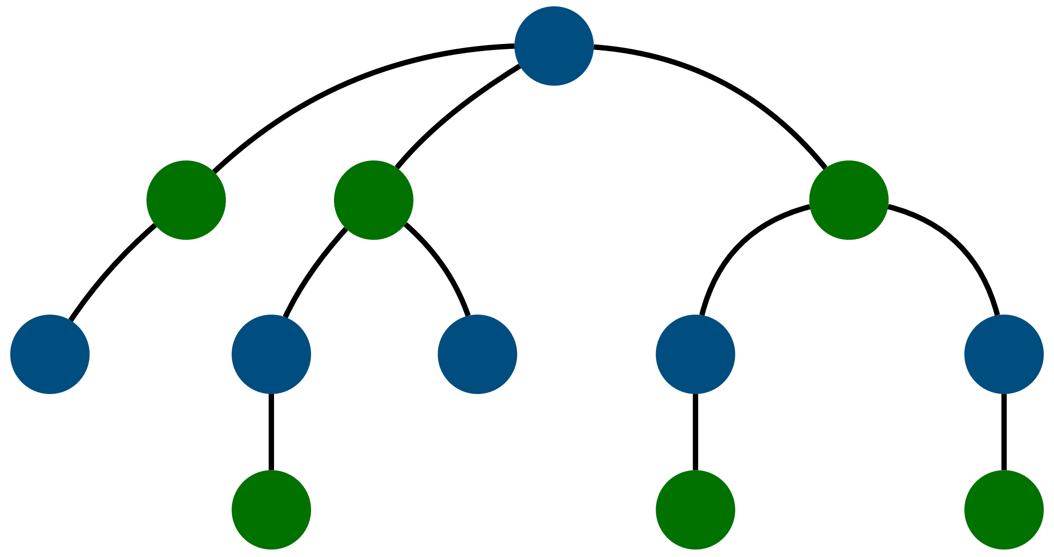
Initial specification

$$\{\mathtt{r} \mapsto x * \mathsf{sll}(x, s)\}$$

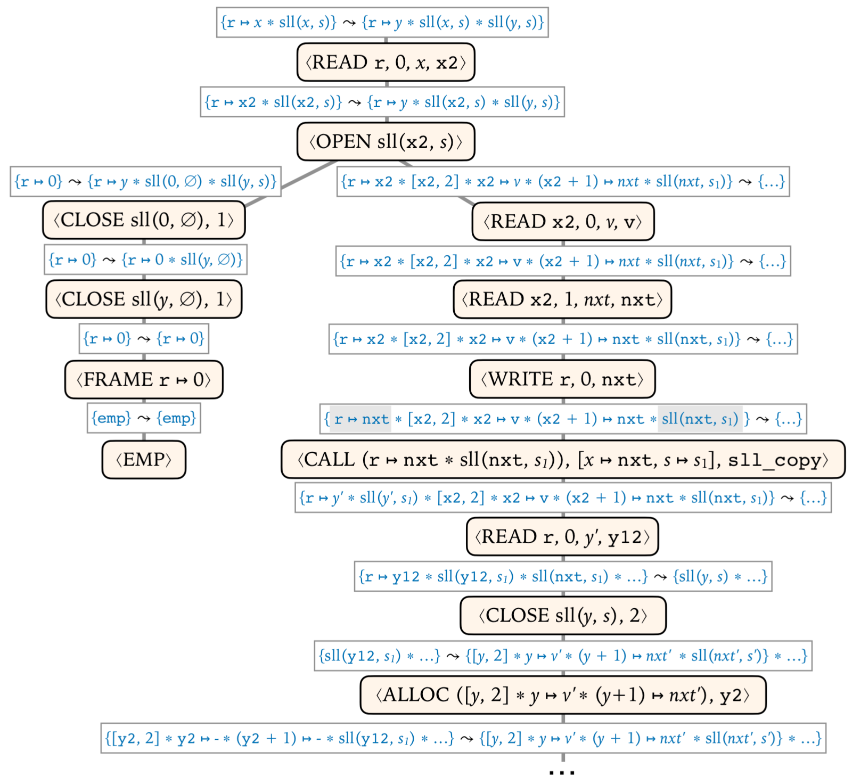
```
void sll_copy(loc r)
```

$$\{\mathtt{r} \mapsto y * \mathsf{sll}(x, s) * \mathsf{sll}(y, s)\}$$

Enumerative proof search



Proof tree



Searching for a program

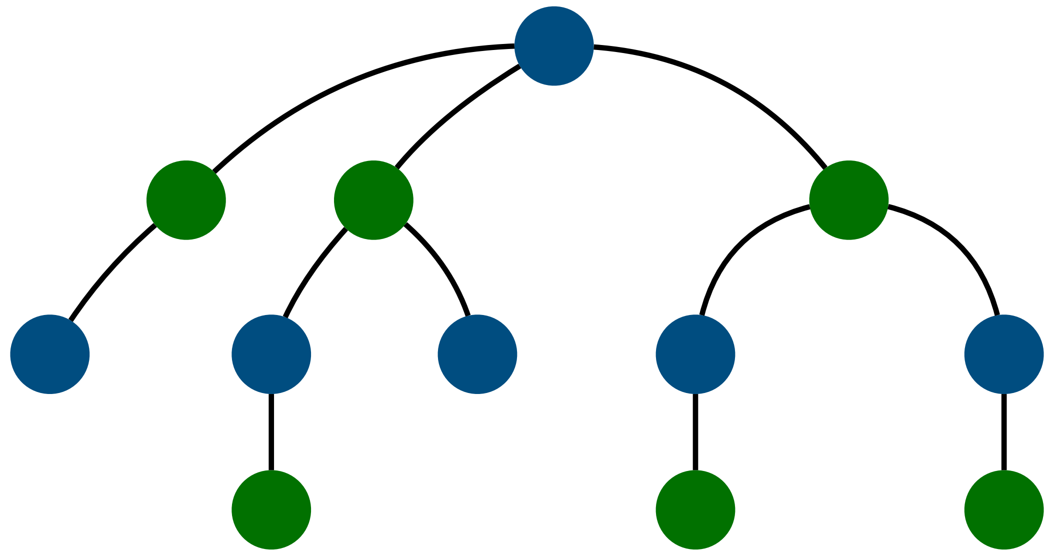
Initial specification

$$\{\mathtt{r} \mapsto x * \mathsf{sll}(x, s)\}$$

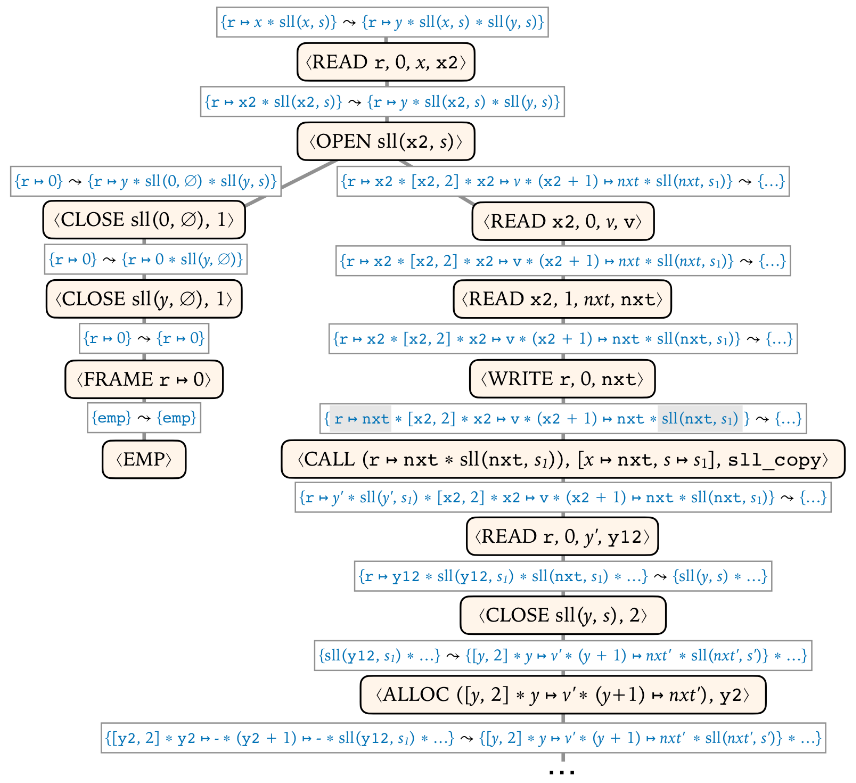
```
void sll_copy(loc r)
```

$$\{\mathtt{r} \mapsto y * \mathsf{sll}(x, s) * \mathsf{sll}(y, s)\}$$

Enumerative proof search



Proof tree



Program (byproduct)

```
void sll_copy (loc r) {
  let x2 = *r;
  if (x2 == 0) {}
  else {
    let v = *x2;
    let nxt = *(x2 + 1);
    *r = nxt;
    sll_copy(r);
    let y12 = *r;
    let y2 = malloc(2);
    *(y2 + 1) = y12;
    *y2 = v;
  }
}
```

Searching for a program

Initial specification

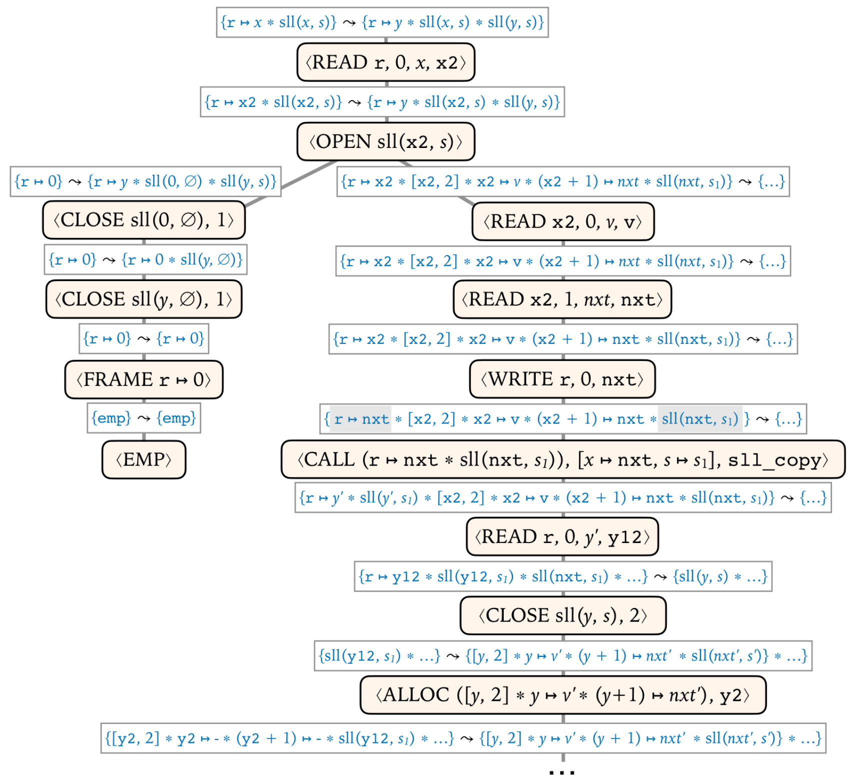
$\{r \mapsto x * \mathsf{sll}(x, s)\}$

```
void sll_copy(loc
```

$\{r \mapsto y * \mathsf{sll}(x, s) * \mathsf{sll}(y, s)\}$

All we need to build a certificate?

Proof tree



Program (byproduct)

```
void sll_copy (loc r) {
  let x2 = *r;
  if (x2 == 0) {}
  else {
    let v = *x2;
    let nxt = *(x2 + 1);
    *r = nxt;
    sll_copy(r);
    let y12 = *r;
    let y2 = malloc(2);
    *(y2 + 1) = y12;
    *y2 = v;
  }
}
```

And yet, a fundamental gap

synthesis ⟷ **verification**

And yet, a fundamental gap

synthesis ⟷ **verification**

Verification: symbolic execution

known program structure

```
let a = *x;

*x = b;
```

Verification: symbolic execution

Symbolically execute
to transform
the *precondition* only

```
let a = *x;


*x = b;
```

Verification: symbolic execution

Symbolically execute
to transform
the *precondition* only

$\{P_0\} \rightsquigarrow \{Q\}$

```
let a = *x;
```

```
apply: bnd_readR=>//=.
```

$\{P_1\} \rightsquigarrow \{Q\}$

```
*x = b;
```

```
apply: bnd_writeR=>//=.
```

$\{P_2\} \rightsquigarrow \{Q\}$

$\vdots$

```
apply: val_ret.
```

$\{P_m\} \rightsquigarrow \{Q\}$

Verification: symbolic execution

Symbolically execute
to transform
the *precondition* only

```
let a = *x;

*x = b;
```

$\{P_0\} \rightsquigarrow \{Q\}$

`apply: bnd_readR=>//=.`

$\{P_1\} \rightsquigarrow \{Q\}$

`apply: bnd_writeR=>//=.`

$\{P_2\} \rightsquigarrow \{Q\}$

$\vdots$

`apply: val_ret.`

$\{P_m\} \rightsquigarrow \{Q\}$

Verification: symbolic execution

$$\{P_0\} \rightsquigarrow \{Q\}$$

```
let a = *x;
```

```
apply: bnd_readR=>//=.
```

$$\{P_1\} \rightsquigarrow \{Q\}$$

```
*x = b;
```

```
apply: bnd_writeR=>//=.
```

$$\{P_2\} \rightsquigarrow \{Q\}$$

$$\vdots$$

```
apply: val_ret.
```

$$\{P_m\} \rightsquigarrow \{Q\}$$

Verification: symbolic execution

$\{P_0\} \rightsquigarrow \{Q\}$

```
let a = *x;
```

```
apply: bnd_readR=>//=.
```

$\{P_1\} \rightsquigarrow \{Q\}$

```
*x = b;
```

```
apply: bnd_writeR=>//=.
```

$\{P_2\} \rightsquigarrow \{Q\}$

$\vdots$

```
apply: val_ret.
```

$\{P_m\} \rightsquigarrow \{Q\}$

Verification: symbolic execution

Postcondition unification delayed to the end!

$$P_m \vdash Q$$

$\{P_0\} \rightsquigarrow \{Q\}$

```
let a = *x;
```

```
apply: bnd_readR=>//=.
```

$\{P_1\} \rightsquigarrow \{Q\}$

```
*x = b;
```

```
apply: bnd_writeR=>//=.
```

$\{P_2\} \rightsquigarrow \{Q\}$

$\vdots$

```
apply: val_ret.
```

$\{P_m\} \rightsquigarrow \{Q\}$

Synthesis: transforming the whole goal

$\{P_0\} \rightsquigarrow \{Q_0\}$

⟨READ ...⟩

$\{P_1\} \rightsquigarrow \{Q_1\}$

⟨WRITE ...⟩

$\{P_2\} \rightsquigarrow \{Q_2\}$

$\vdots$

⟨EMP⟩

$\{\mathsf{emp}\} \rightsquigarrow \{\mathsf{emp}\}$

Can't rely on program structure!

Synthesis: transforming the whole goal

$\{P_0\} \leadsto \{Q_0\}$

$\langle$READ ...$\rangle$

$\{P_1\} \leadsto \{Q_1\}$

$\langle$WRITE ...$\rangle$

$\{P_2\} \leadsto \{Q_2\}$

$\vdots$

$\langle$EMP$\rangle$

$\{\mathsf{emp}\} \leadsto \{\mathsf{emp}\}$

Can't rely on program structure!

Synthesis: transforming the whole goal

$\{P_0\} \rightsquigarrow \{Q_0\}$

⟨READ ...⟩

```
let a = *x;
```

$\{P_1\} \rightsquigarrow \{Q_1\}$

⟨WRITE ...⟩

```
*x = b;
```

$\{P_2\} \rightsquigarrow \{Q_2\}$

$\vdots$

⟨EMP⟩

$\{\mathsf{emp}\} \rightsquigarrow \{\mathsf{emp}\}$

Synthesis: transforming the whole goal

$\{P_0\} \leadsto \{Q_0\}$

$\langle \textsc{Read} \ldots \rangle$

```
let a = *x;
```

$\{P_1\} \leadsto \{Q_1\}$

$\langle \textsc{Write} \ldots \rangle$

```
*x = b;
```

$\{P_2\} \leadsto \{Q_2\}$

$\vdots$

$\langle \textsc{Emp} \rangle$

$\{\mathsf{emp}\} \leadsto \{\mathsf{emp}\}$

Both
pre- and postcondition
can be transformed

A question

A question

Rules that transform the postcondition

A question

Rules that transform the postcondition

need to *delay* their insight

A question

Rules that transform the postcondition

need to *delay* their insight

→ How to bridge the gap?

Another question

Another question

Multiple program verifiers available

Another question

Multiple program verifiers available

1. **Hoare Type Theory (HTT)** — Nanevski et al. '10

Another question

Multiple program verifiers available

1. **Hoare Type Theory (HTT)** — Nanevski et al. ’10
2. **Verified Software Toolchain (VST)** — Appel ’11

Another question

Multiple program verifiers available

1. **Hoare Type Theory (HTT)** — Nanevski et al. ’10
2. **Verified Software Toolchain (VST)** — Appel ’11
3. **IRIS** — Jung et al. ’18; Krebbers et al. ’17

Another question

Multiple program verifiers available

1. **Hoare Type Theory (HTT)** — Nanevski et al. ’10

2. **Verified Software Toolchain (VST)** — Appel ’11

3. **IRIS** — Jung et al. ’18; Krebbers et al. ’17

→ How to support verifiers uniformly?

Two motivating challenges…

How to bridge the gap?

How to support verifiers uniformly?

Two motivating challenges…

How to bridge the gap?

“interpret” synthesis trace into verification proof

How to support verifiers uniformly?

Two motivating challenges…

How to bridge the gap?

"interpret" synthesis trace into verification proof

How to support verifiers uniformly?

an abstract framework that each verifier can instantiate

Our contributions

Our contributions

- An **abstract proof evaluator** framework

Our contributions

- An **abstract proof evaluator** framework
- Instantiations for **3 target verifiers** (HTT, VST, IRIS)

Our contributions

- An **abstract proof evaluator** framework
- Instantiations for **3 target verifiers** (HTT, VST, IRIS)
- **Evaluation** on characteristic benchmarks

Our contributions

- An **abstract proof evaluator** framework
- Instantiations for **3 target verifiers** (HTT, VST, IRIS)
- **Evaluation** on characteristic benchmarks

Custom proof step interpreters

HTT Interpreter

IRIS Interpreter

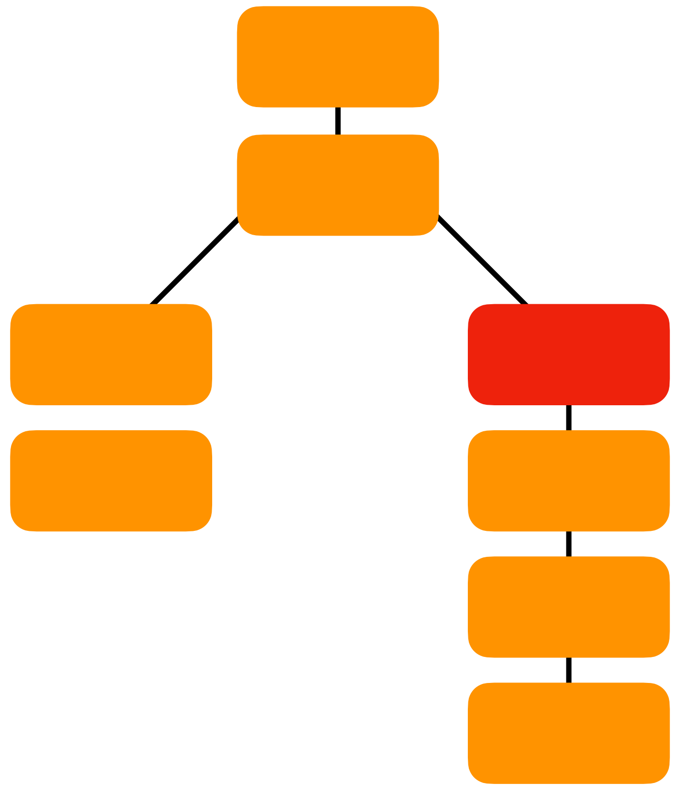
VST Interpreter

Custom proof step interpreters

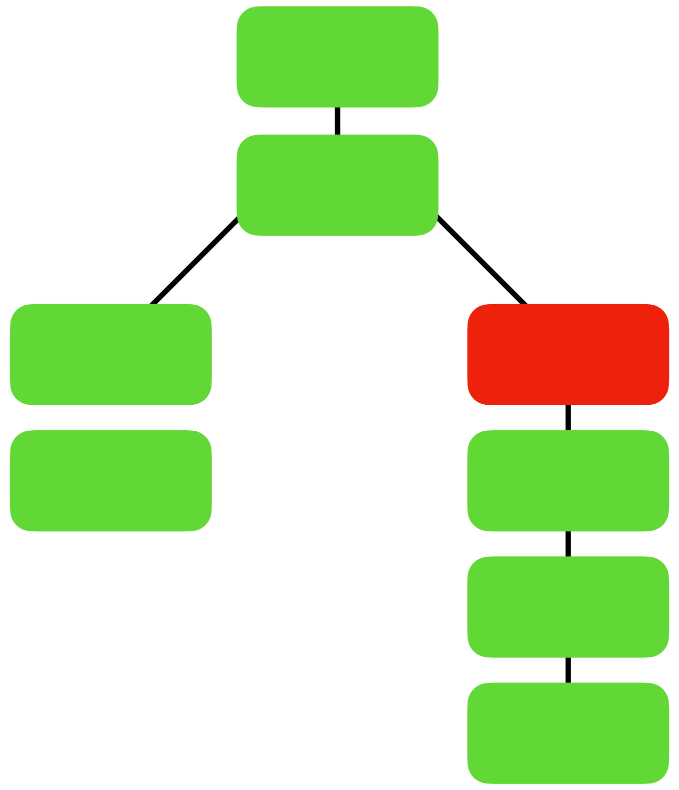
$$I_{htt} : \text{Step}_{ssl} \longrightarrow \text{Step}_{htt}$$

HTT Interpreter

Abstract Evaluator



synthesis proof tree



HTT certificate

Custom proof step interpreters

$$I_{htt} : \mathsf{Step}_{ssl} \longrightarrow \mathsf{Step}_{htt}$$

HTT Interpreter

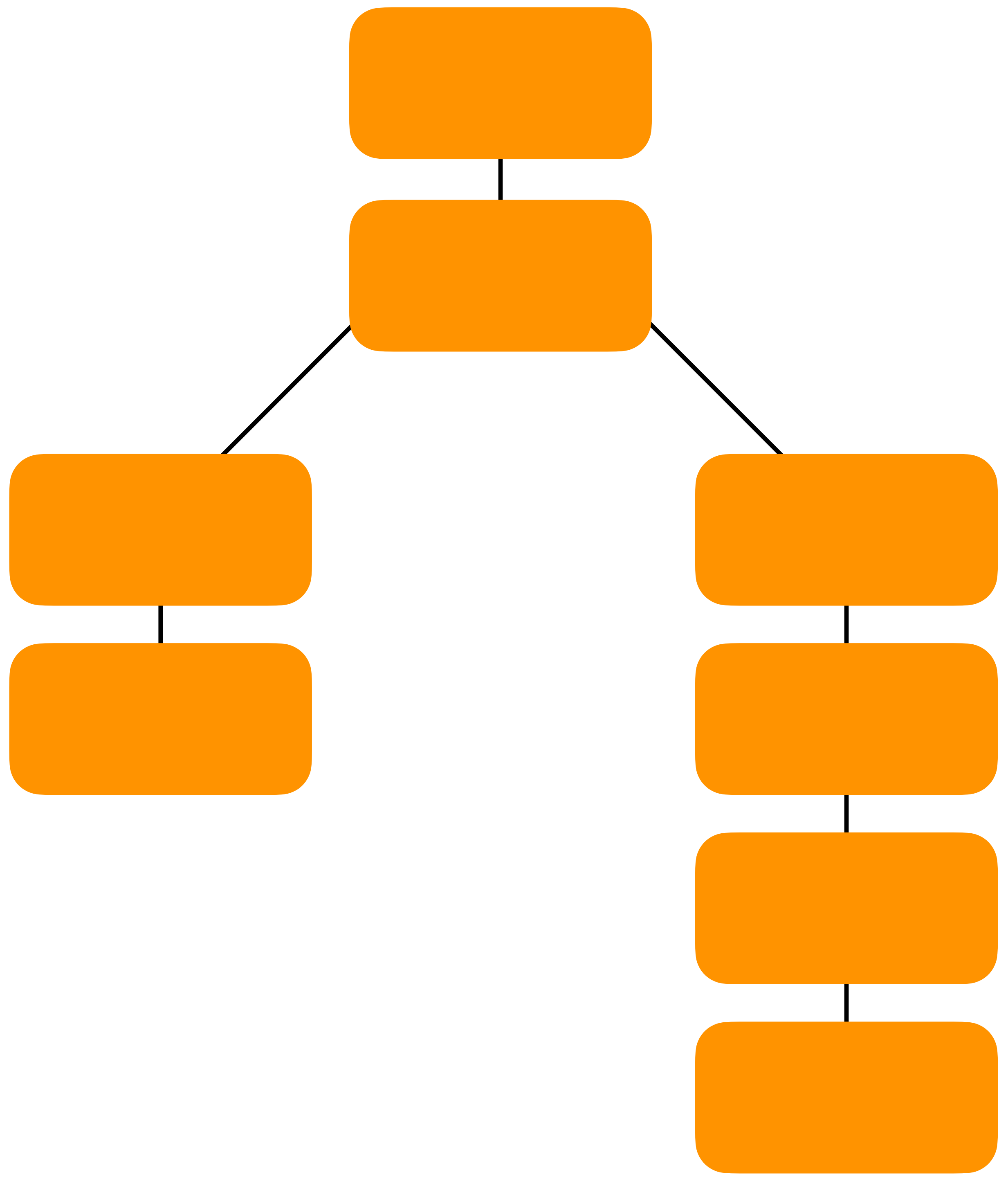
Abstract Evaluator

synthesis proof tree

HTT certificate

The evaluator in action

SuSLik proof tree

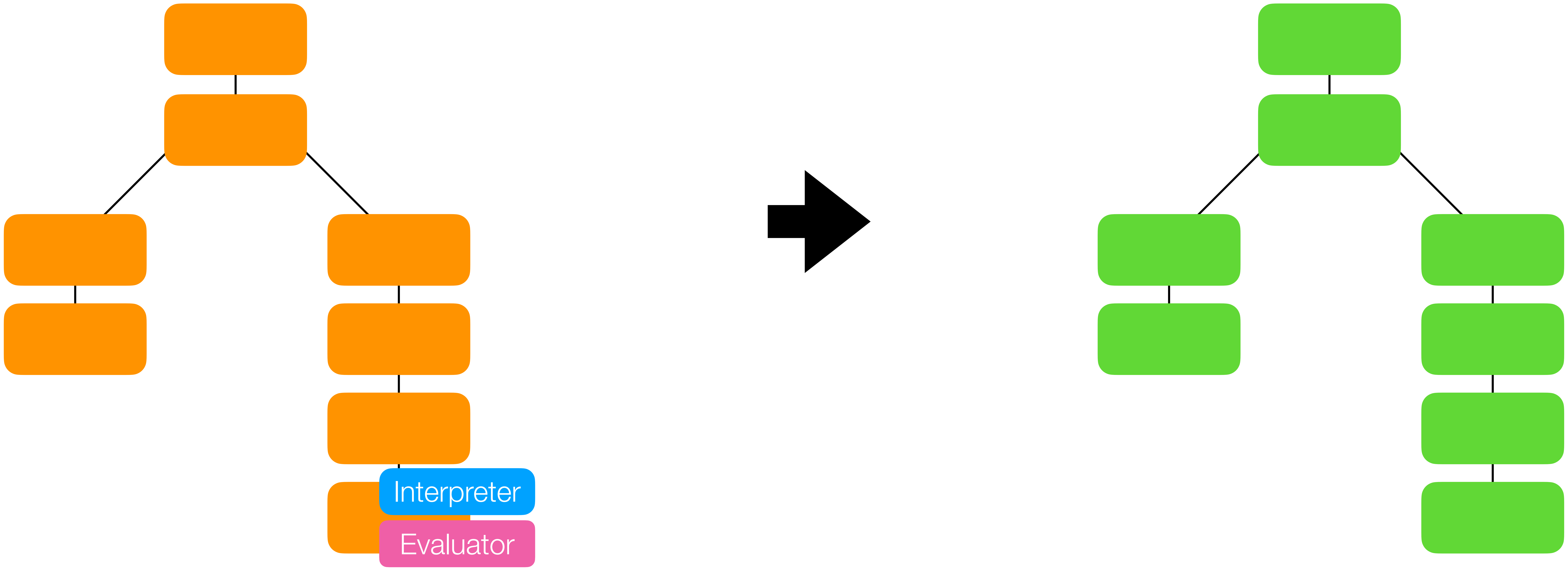


Coq proof certificate

The evaluator in action

SuSLik proof tree

Coq proof certificate



Two strategies to bridge the gap

Two strategies to bridge the gap

1. **Deferred proof steps**

Two strategies to bridge the gap

1. Deferred proof steps

delay proof step appearance

Two strategies to bridge the gap

1. Deferred proof steps

delay proof step appearance

like a

continuation

Two strategies to bridge the gap

1. Deferred proof steps

delay proof step appearance

like a

continuation

2. Proof contexts

Two strategies to bridge the gap

1. Deferred proof steps

delay proof step appearance

like a

continuation

2. Proof contexts

track bookkeeping information

Two strategies to bridge the gap

1. Deferred proof steps

delay proof step appearance

like a

continuation

2. Proof contexts

track bookkeeping information

like an

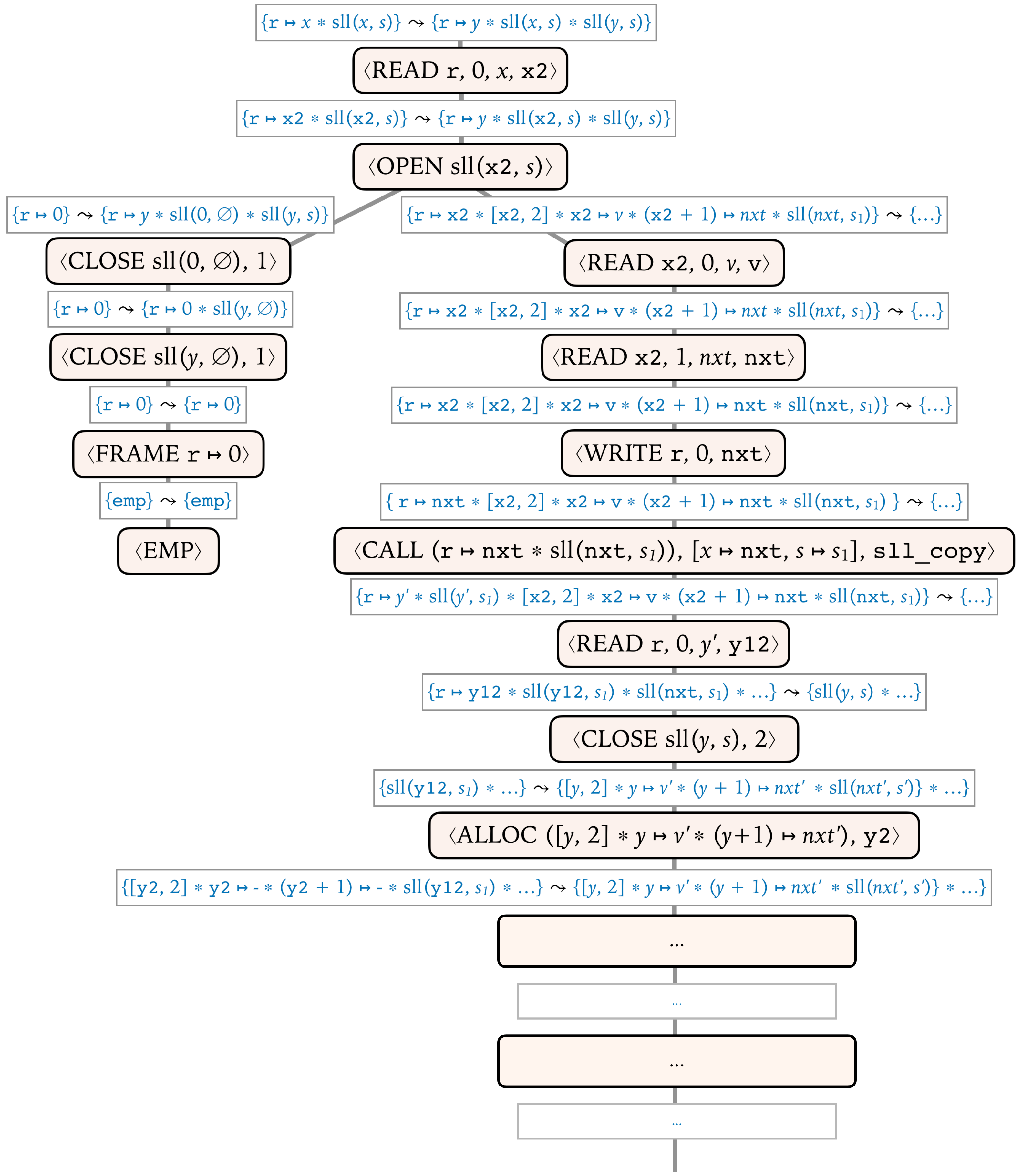
accumulator

Example with Hoare Type Theory (HTT)

Let's certify `sll_copy`

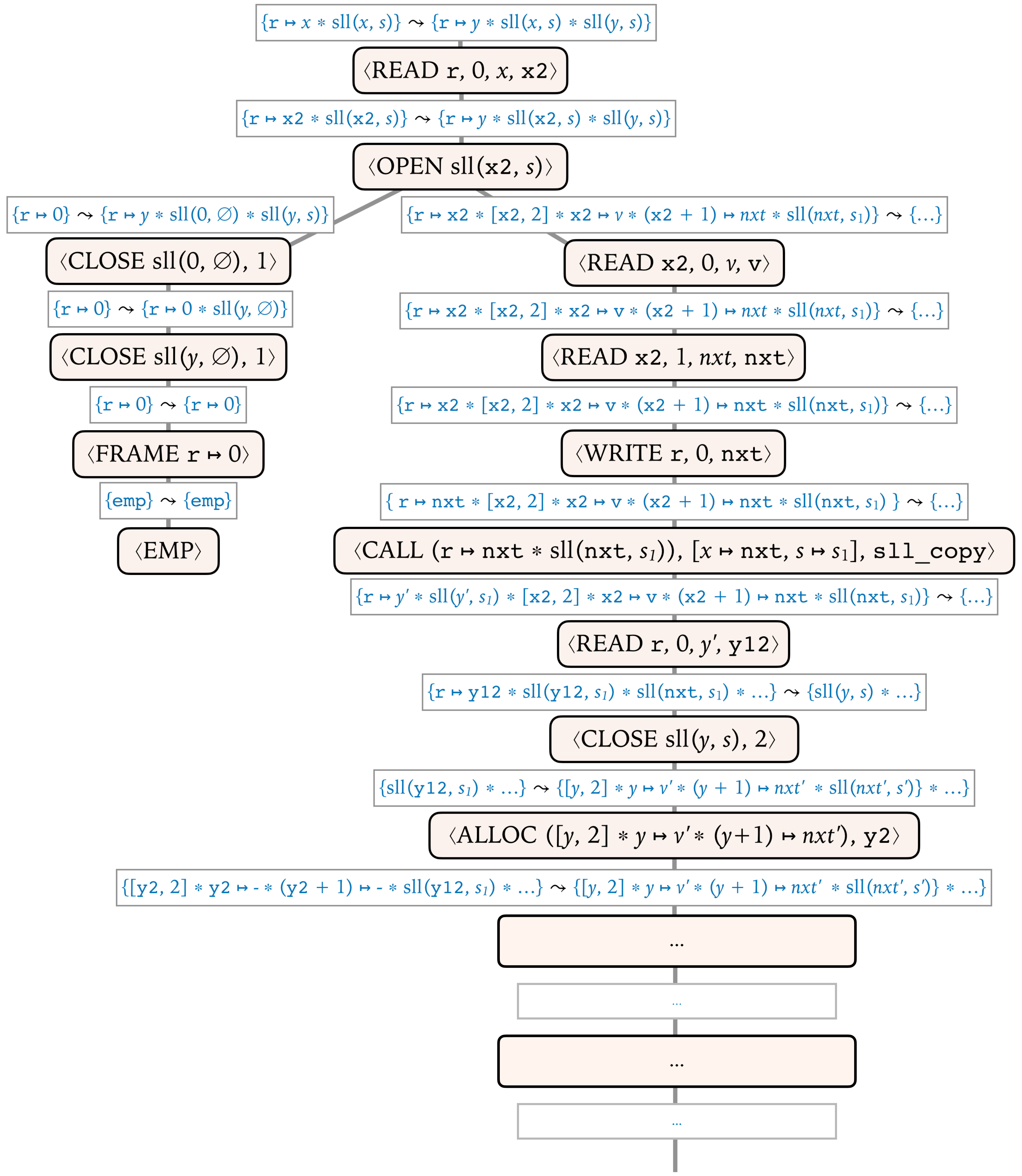
Let's certify `sll_copy`

SuSLik proof tree



Let's certify `sll_copy`

SuSLik proof tree

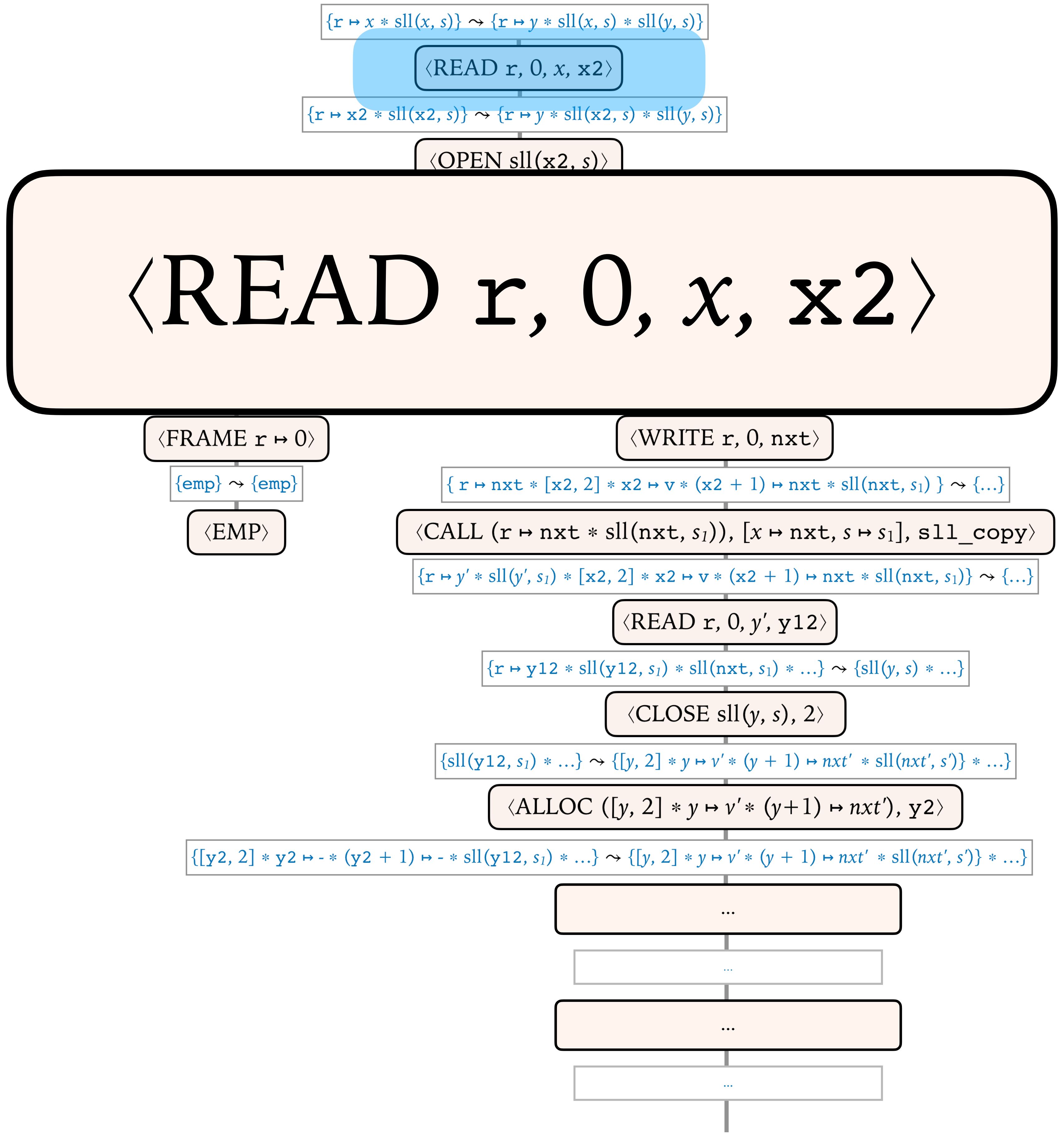


Coq proof certificate (HTT)

```
Next Obligation.
(* Initialize HTT proof context *)
 apply: ghR; move=>h_self[x2 s][h'][->]Hsll _.
(* Read *) apply: bnd_readR=>/=.
(* Open (unfold) SLL instance in the precondition *)
 case: Hsll; case: ifP; move=>IfCond//_;
 [move=>[?]->|move=>[v][s1][nxt][h1][?][->]H1].
 (* Case: empty list (x2 = 0) *)
 - move:IfCond=>/eqP->. (* substitute x2 ↦ 0 *)
  (* Emp *) apply: val_ret; ∃ null, Unit, Unit;
  (* Close (unfold) SLL instance in postcondition *)
   repeat split=>//=; do?[hhauto|constructor 1].
 (* Case: non-empty list *)
 - (* Read *) apply: bnd_readR=>//=.
  (* Read *) apply: bnd_readR=>//=.
  (* Write *) apply: bnd_writeR=>//=.
  (* Call *)
   rewrite (joinC _ h1) joinA; apply: bnd_seq.
   apply: (gh_ex (nxt, s1)); apply: val_do=>//=_.
   ∃ h1; split=>//=.
   move=>h_call [y12][h11][h21][->][H2 H3]_.
  (* Read *) apply: bnd_readR=>//=.
  (* Alloc *) apply: bnd_allocbR=>y2//=.
  (* Write *) apply: bnd_writeR=>//=.
  (* Write *) apply: bnd_writeR=>//=.
  (* Write *) apply: bnd_writeR=>//=.
  (* Emp *)
   apply: val_ret; rewrite defPtUn0; case/andP=>?.
   ∃ y2, (x2 ↦ v • (x2 +1) ↦ nxt • h11),
       (y2 ↦ v • (y2 +1) ↦ y12 • h21).
   repeat split=>//=; first by hhauto.
  + (* Close SLL instance 1 in postcondition *)
    by constructor 2=>//=; ∃ v, s1, nxt, h11.
  + (* Close SLL instance 2 in postcondition *)
    constructor 2=>//=; first by apply negbTE.
    by ∃ v, s1, y12, h21.
Qed.
```

Let's certify `sll_copy`

SuSLik proof tree



Coq proof certificate (HTT)

```
Next Obligation.
(* Initialize HTT proof context *)
  apply: ghR; move=>h_self[x2 s][h'][->]Hsll _.
(* Read *) apply: bnd_readR=>/=.
(* Open (unfold) SLL instance in the precondition *)
  case: Hsll; case: ifP; move=>IfCond//_;
  [move=>[?]->|move=>[v][s1][nxt][h1][?][->]H1].
```

apply: bnd_readR=>//=.

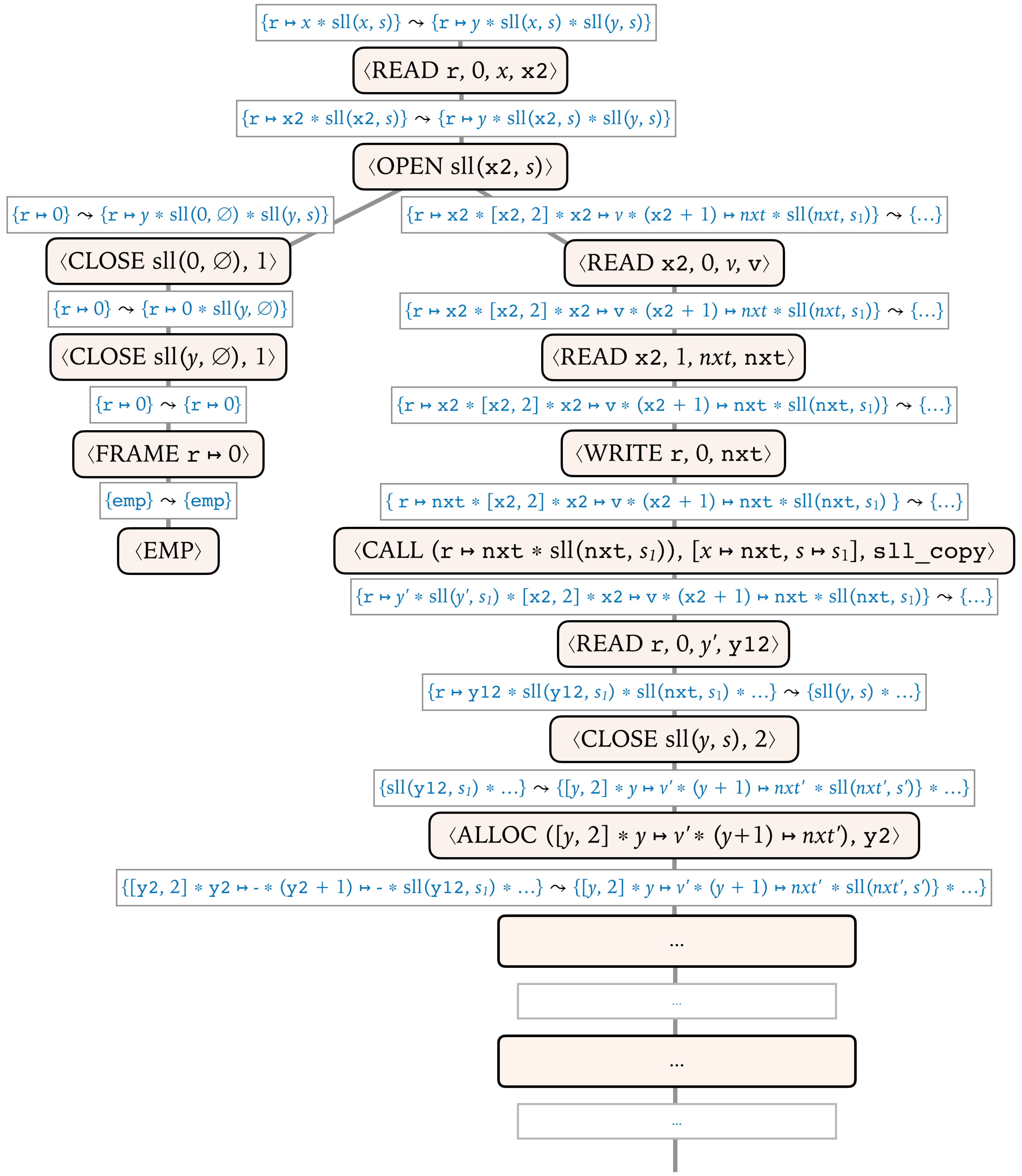
```
- (* Read *) apply: bnd_readR=>//=.
  (* Read *) apply: bnd_readR=>//=.
  (* Write *) apply: bnd_writeR=>//=.
  (* Call *)
   rewrite (joinC _ h1) joinA; apply: bnd_seq.
   apply: (gh_ex (nxt, s1)); apply: val_do=>//=_.
   ∃ h1; split=>//=.
   move=>h_call [y12][h11][h21][->][H2 H3]_.
  (* Read *) apply: bnd_readR=>//=.
  (* Alloc *) apply: bnd_allocbR=>y2//=.
  (* Write *) apply: bnd_writeR=>//=.
  (* Write *) apply: bnd_writeR=>//=.
  (* Write *) apply: bnd_writeR=>//=.
  (* Emp *)
   apply: val_ret; rewrite defPtUn0; case/andP=>?.
   ∃ y2, (x2 ↦ v • (x2 +1) ↦ nxt • h11),
         (y2 ↦ v • (y2 +1) ↦ y12 • h21).
   repeat split=>//=; first by hhauto.
  + (* Close SLL instance 1 in postcondition *)
    by constructor 2=>//=; ∃ v, s1, nxt, h11.
  + (* Close SLL instance 2 in postcondition *)
    constructor 2=>//=; first by apply negbTE.
    by ∃ v, s1, y12, h21.
Qed.
```

Handling the READ rule for HTT

$I_{htt}\langle \text{READ}, x, \iota, e, \mathrm{y}\rangle \triangleq$ [`apply: bnd_readR=>//=.`]

Problem: out of order appearance

SuSLik proof tree

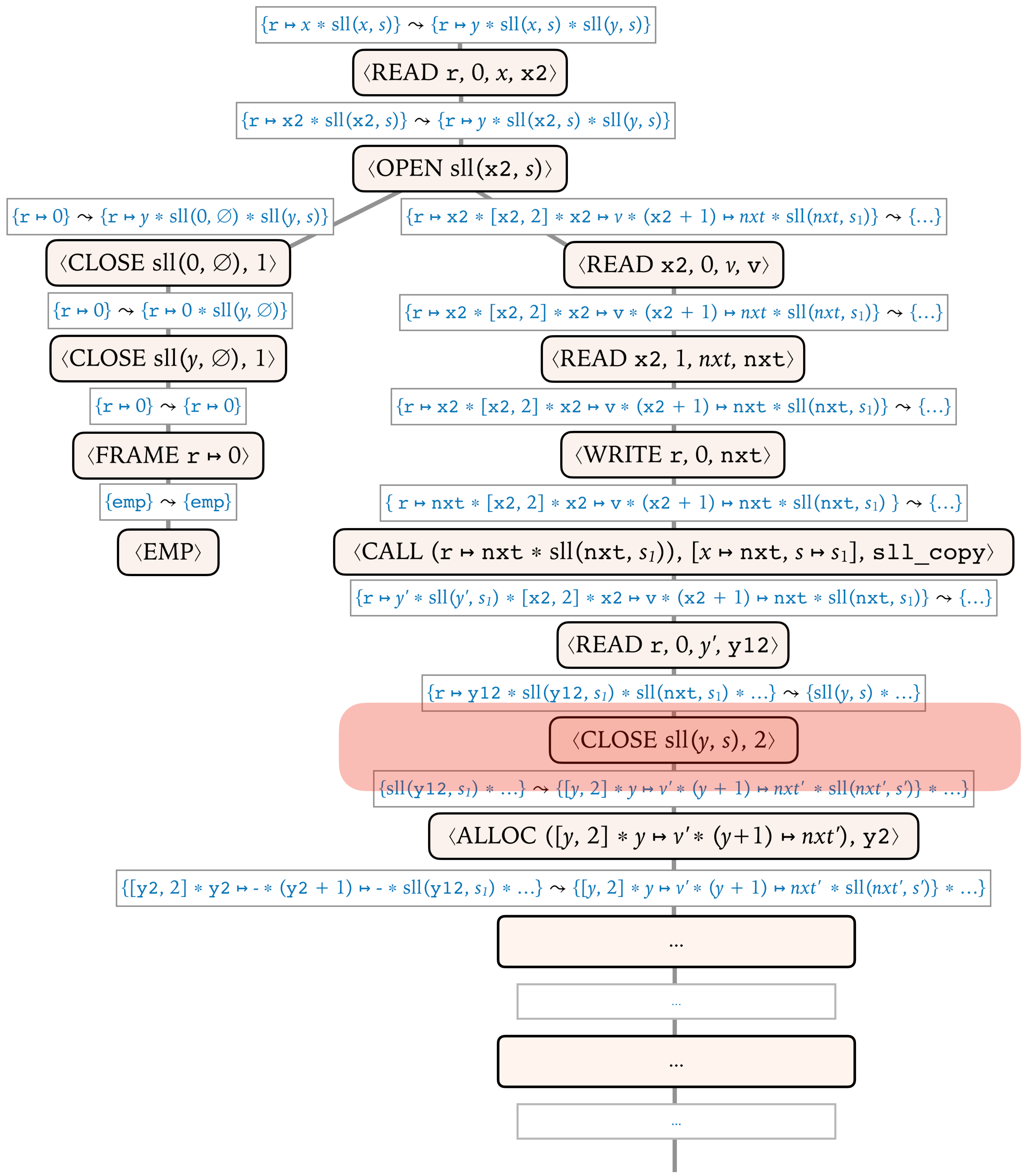


Coq proof certificate (HTT)

```
Next Obligation.
(* Initialize HTT proof context *)
 apply: ghR; move=>h_self[x2 s][h'][->]Hsll _.
(* Read *) apply: bnd_readR=>/=.
(* Open (unfold) SLL instance in the precondition *)
 case: Hsll; case: ifP; move=>IfCond//_;
 [move=>[?]->|move=>[v][s1][nxt][h1][?][->]H1].
 (* Case: empty list (x2 = 0) *)
 - move:IfCond=>/eqP->. (* substitute x2 ↦ 0 *)
  (* Emp *) apply: val_ret; ∃ null, Unit, Unit;
  (* Close (unfold) SLL instance in postcondition *)
   repeat split=>//=; do?[hhauto|constructor 1].
 (* Case: non-empty list *)
 - (* Read *) apply: bnd_readR=>//=.
  (* Read *) apply: bnd_readR=>//=.
  (* Write *) apply: bnd_writeR=>//=.
  (* Call *)
   rewrite (joinC _ h1) joinA; apply: bnd_seq.
   apply: (gh_ex (nxt, s1)); apply: val_do=>//=_.
   ∃ h1; split=>//=.
   move=>h_call [y12][h11][h21][->][H2 H3]_.
  (* Read *) apply: bnd_readR=>//=.
  (* Alloc *) apply: bnd_allocbR=>y2//=.
  (* Write *) apply: bnd_writeR=>//=.
  (* Write *) apply: bnd_writeR=>//=.
  (* Write *) apply: bnd_writeR=>//=.
  (* Emp *)
   apply: val_ret; rewrite defPtUn0; case/andP=>?.
   ∃ y2, (x2 ↦ v • (x2+1) ↦ nxt • h11),
       (y2 ↦ v • (y2+1) ↦ y12 • h21).
   repeat split=>//=; first by hhauto.
  + (* Close SLL instance 1 in postcondition *)
    by constructor 2=>//=; ∃ v, s1, nxt, h11.
  + (* Close SLL instance 2 in postcondition *)
    constructor 2=>//=; first by apply negbTE.
    by ∃ v, s1, y12, h21.
Qed.
```

Problem: out of order appearance

SuSLik proof tree



Coq proof certificate (HTT)

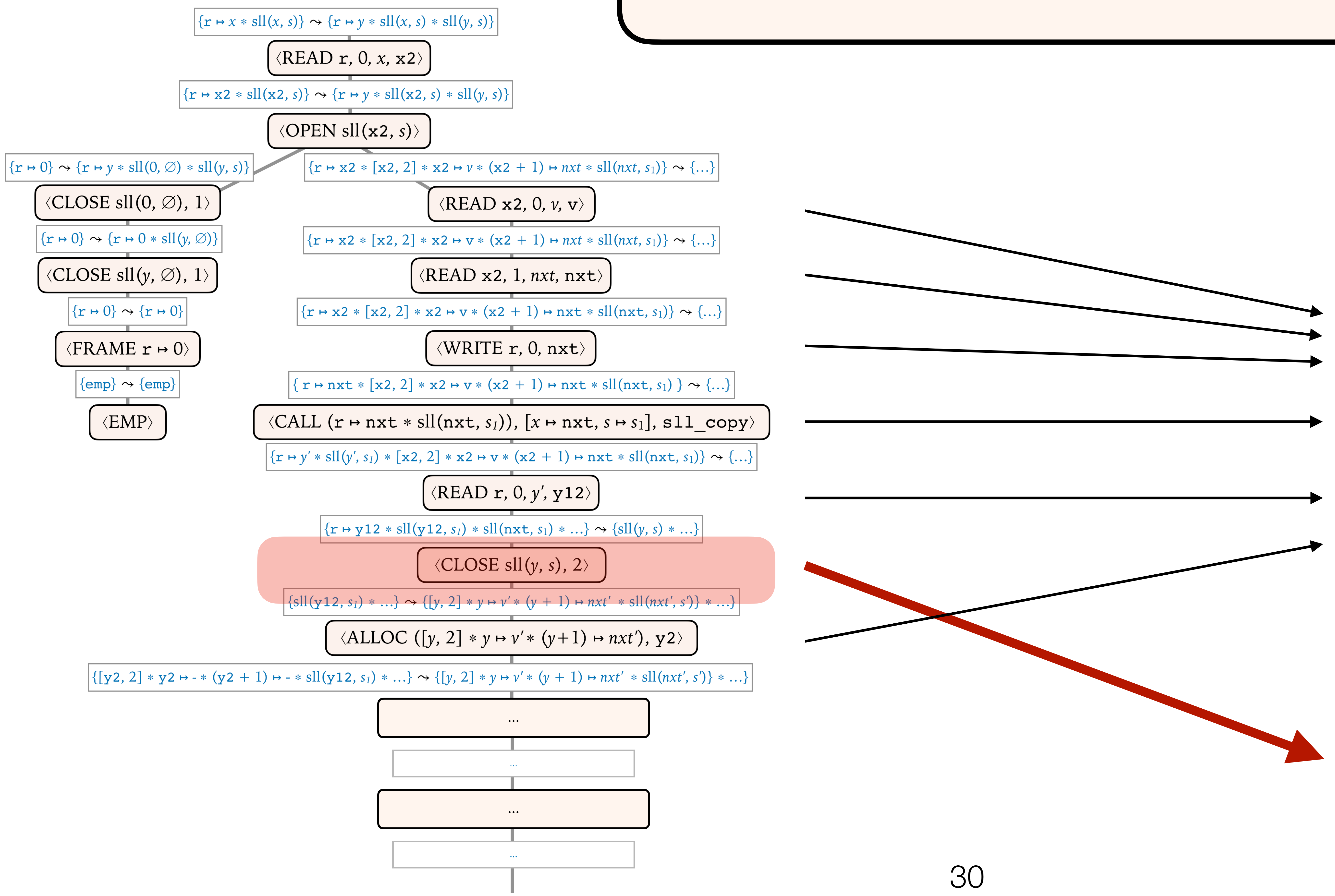
```
Next Obligation.
(* Initialize HTT proof context *)
 apply: ghR; move=>h_self[x2 s][h'][->]Hsll _.
(* Read *) apply: bnd_readR=>/=.
(* Open (unfold) SLL instance in the precondition *)
 case: Hsll; case: ifP; move=>IfCond//_;
 [move=>[?]->|move=>[v][s1][nxt][h1][?][->]H1].
  (* Case: empty list (x2 = 0) *)
 - move:IfCond=>/eqP->. (* substitute x2 ↦ 0 *)
   (* Emp *) apply: val_ret; ∃ null, Unit, Unit;
   (* Close (unfold) SLL instance in postcondition *)
    repeat split=>//=; do?[hhauto|constructor 1].
  (* Case: non-empty list *)
 - (* Read *) apply: bnd_readR=>//=.
   (* Read *) apply: bnd_readR=>//=.
   (* Write *) apply: bnd_writeR=>//=.
   (* Call *)
    rewrite (joinC _ h1) joinA; apply: bnd_seq.
    apply: (gh_ex (nxt, s1)); apply: val_do=>//=_.
    ∃ h1; split=>//=.
    move=>h_call [y12][h11][h21][->][H2 H3]_.
   (* Read *) apply: bnd_readR=>//=.
   (* Alloc *) apply: bnd_allocbR=>y2//=.
   (* Write *) apply: bnd_writeR=>//=.
   (* Write *) apply: bnd_writeR=>//=.
   (* Write *) apply: bnd_writeR=>//=.
   (* Emp *)
    apply: val_ret; rewrite defPtUn0; case/andP=>?.
    ∃ y2, (x2 ↦ v • (x2+1) ↦ nxt • h11),
           (y2 ↦ v • (y2+1) ↦ y12 • h21).
    repeat split=>//=; first by hhauto.
   + (* Close SLL instance 1 in postcondition *)
     by constructor 2=>//=; ∃ v, s1, nxt, h11.
   + (* Close SLL instance 2 in postcondition *)
     constructor 2=>//=; first by apply negbTE.
     by ∃ v, s1, y12, h21.
Qed.
```

Problem: out of order appearance

Need to defer!

SuSLik proof tre[e] — [Pr]oof certificate (HTT)

$\langle$CLOSE sll(y, s), 2$\rangle$




```
...igation.
(* ...alize HTT proof context *)
apply: ghR; move=>h_self[x2 s][h'][->]Hsll _.
(* Read *) apply: bnd_readR=>/=.
(* Open (unfold) SLL instance in the precondition *)
case: Hsll; case: ifP; move=>IfCond//_;
[move=>[?]->|move=>[v][s1][nxt][h1][?][->]H1].
(* Case: empty list (x2 = 0) *)
- move:IfCond=>/eqP->. (* substitute x2 ↦ 0 *)
  (* Emp *) apply: val_ret; ∃ null, Unit, Unit;
  (* Close (unfold) SLL instance in postcondition *)
   repeat split=>//=; do?[hhauto|constructor 1].
(* Case: non-empty list *)
- (* Read *) apply: bnd_readR=>//=.
  (* Read *) apply: bnd_readR=>//=.
  (* Write *) apply: bnd_writeR=>//=.
  (* Call *)
   rewrite (joinC _ h1) joinA; apply: bnd_seq.
   apply: (gh_ex (nxt, s1)); apply: val_do=>//=_.
   ∃ h1; split=>//=.
   move=>h_call [y12][h11][h21][->][H2 H3]_.
  (* Read *) apply: bnd_readR=>//=.
  (* Alloc *) apply: bnd_allocbR=>y2//=.
  (* Write *) apply: bnd_writeR=>//=.
  (* Write *) apply: bnd_writeR=>//=.
  (* Write *) apply: bnd_writeR=>//=.
  (* Emp *)
   apply: val_ret; rewrite defPtUn0; case/andP=>?.
   ∃ y2, (x2 ↦ v • (x2 +1) ↦ nxt • h11),
         (y2 ↦ v • (y2 +1) ↦ y12 • h21).
   repeat split=>//=; first by hhauto.
  + (* Close SLL instance 1 in postcondition *)
    by constructor 2=>//=; ∃ v, s1, nxt, h11.
  + (* Close SLL instance 2 in postcondition *)
    constructor 2=>//=; first by apply negbTE.
    by ∃ v, s1, y12, h21.
Qed.
```

CLOSE unfolds a postcondition predicate

$$\{\mathtt{r} \mapsto x * \mathsf{sll}(x, s)\}$$

```
void sll_copy(loc r)
```

$$\{\mathtt{r} \mapsto y * \mathsf{sll}(x, s) * \mathsf{sll}(y, s)\}$$

CLOSE unfolds a postcondition predicate

$$\{\mathtt{r} \mapsto x * \mathsf{sll}(x, s)\}$$

```
void sll_copy(loc r)
```

$$\{\mathtt{r} \mapsto y * \mathsf{sll}(x, s) * \boxed{\mathsf{sll}(y, s)\}}$$

CLOSE unfolds a postcondition predicate

$$\{\mathtt{r} \mapsto x * \mathsf{sll}(x, s)\}$$

```
void sll_copy(loc r)
```

$$\{\mathtt{r} \mapsto y * \mathsf{sll}(x, s) * \underbrace{\boxed{\mathsf{sll}(y, s)}}_{[y, 2] * y \mapsto v' * (y + 1) \mapsto nxt' * \mathsf{sll}(nxt', s')}\}$$

constructor 2 of the `sll` predicate

Recall…

Recall…

Rules that transform the postcondition

Recall…

Rules that transform the postcondition

need to *delay* their insight

Deferred proof steps to the rescue!

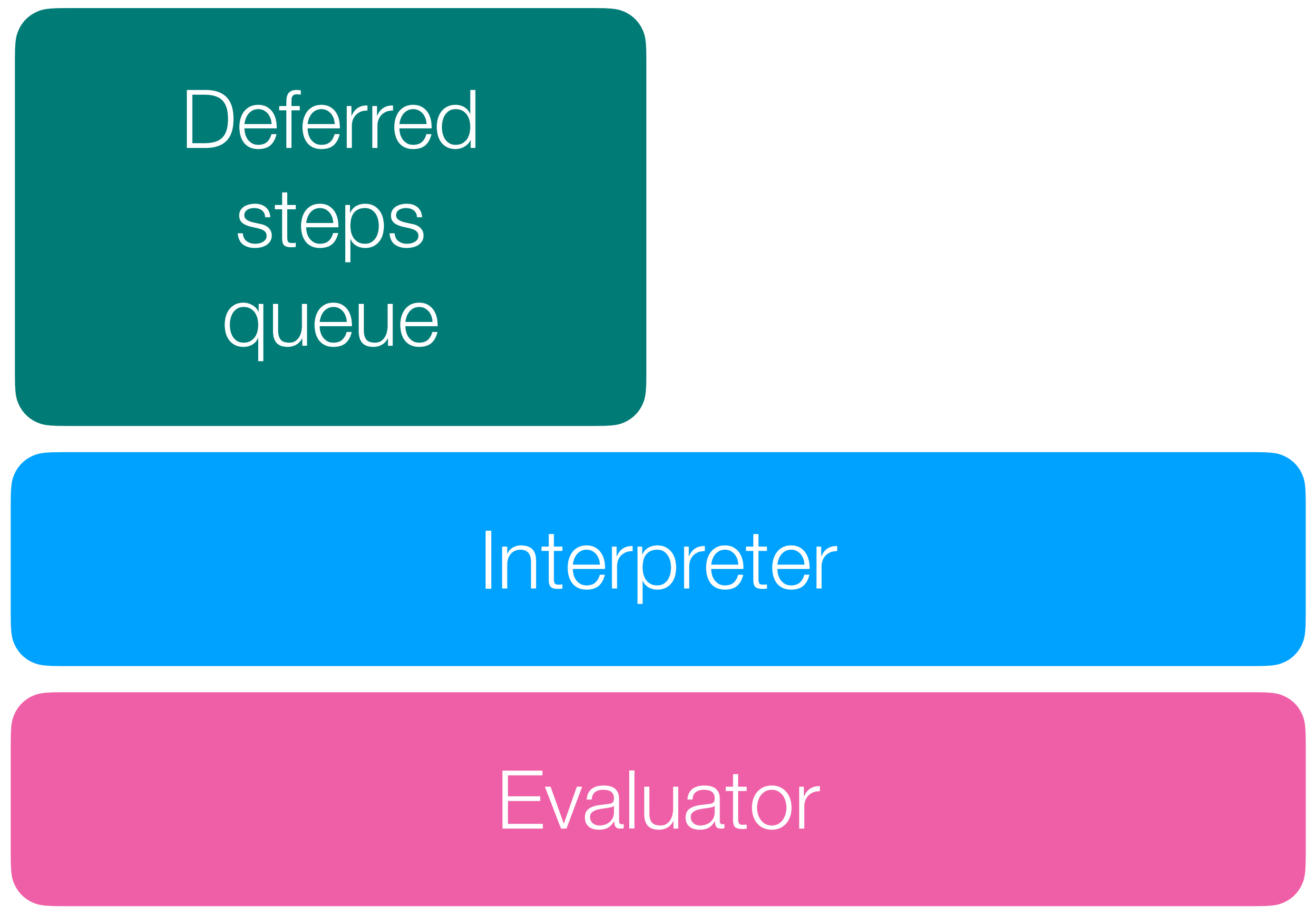
$$I_{htt} : \mathrm{Step}_{ssl} \longrightarrow \mathrm{Step}_{htt}$$

Interpreter

Evaluator

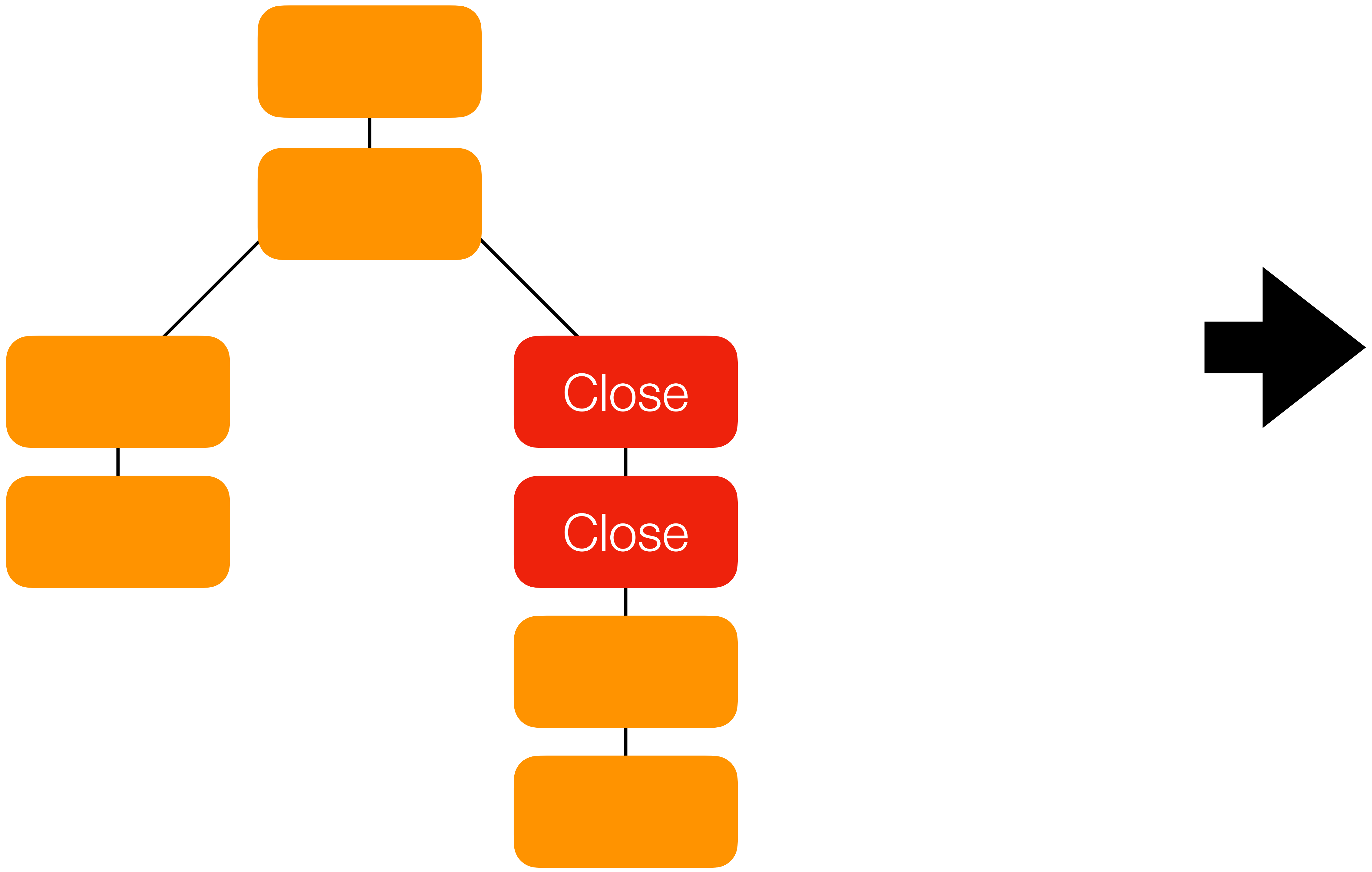
Deferred proof steps to the rescue!

$$I_{htt} : \mathsf{Step}_{ssl} \longrightarrow \mathsf{Step}_{htt} \times \mathsf{DeferredStep}_{htt}$$



Deferred proof steps in action

SuSLik proof tree

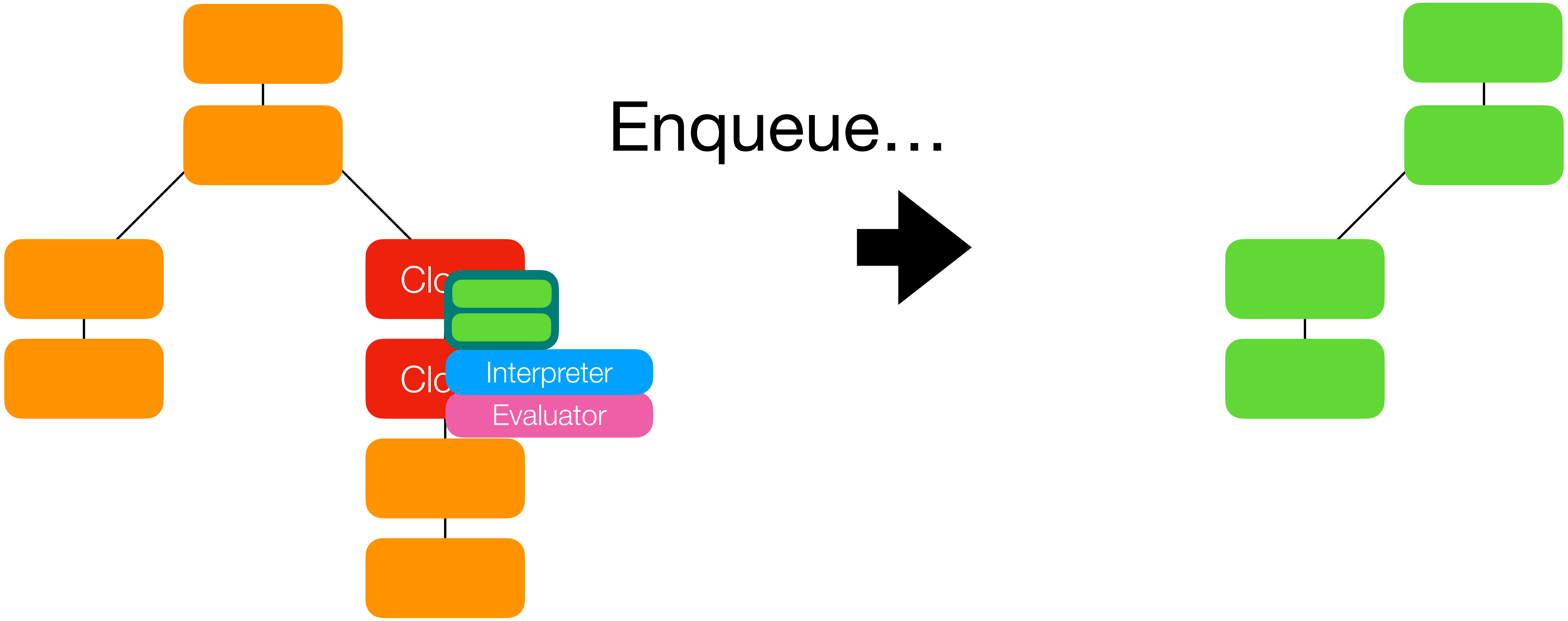


Coq proof certificate

Deferred proof steps in action

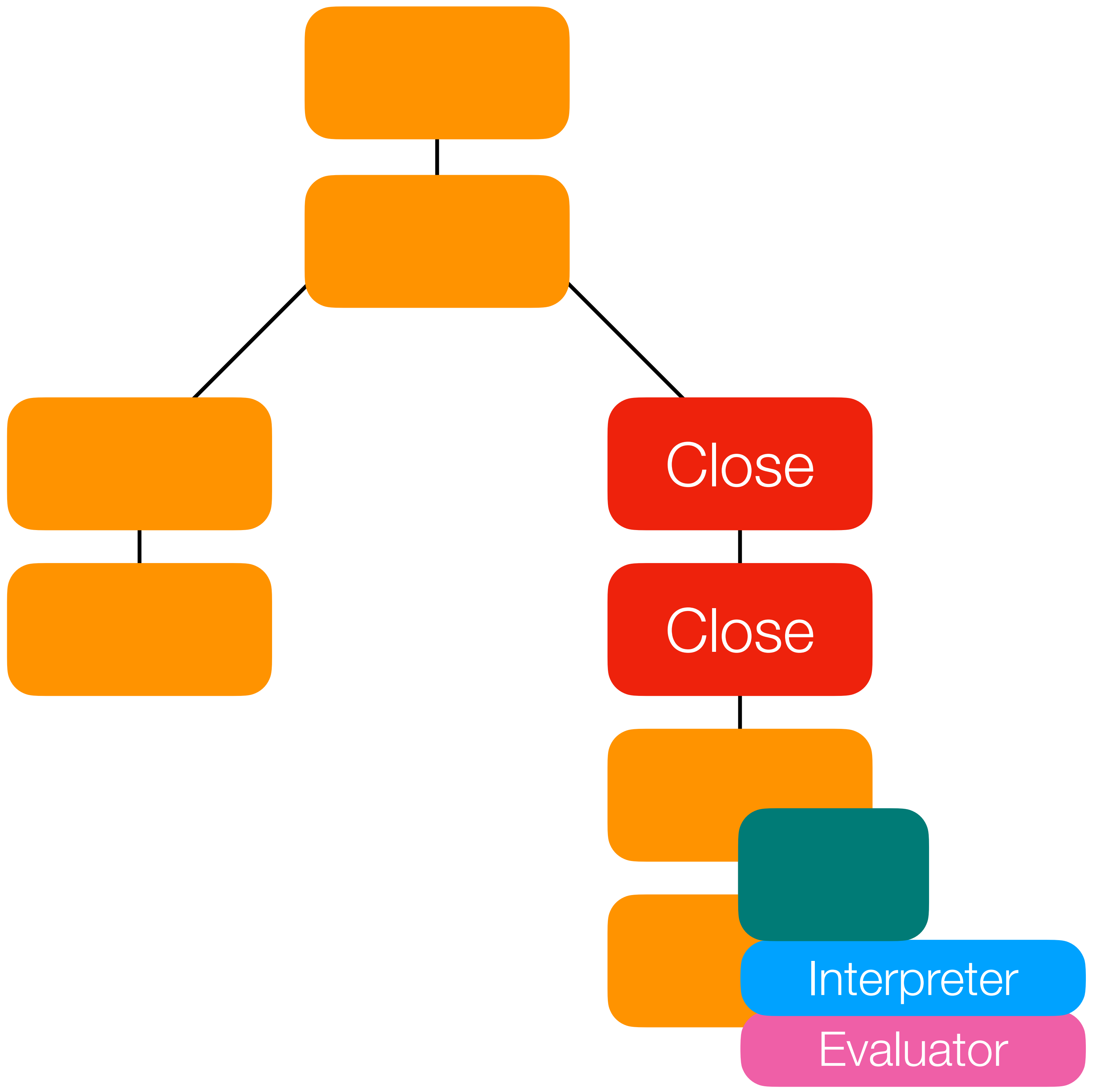
SuSLik proof tree

Coq proof certificate



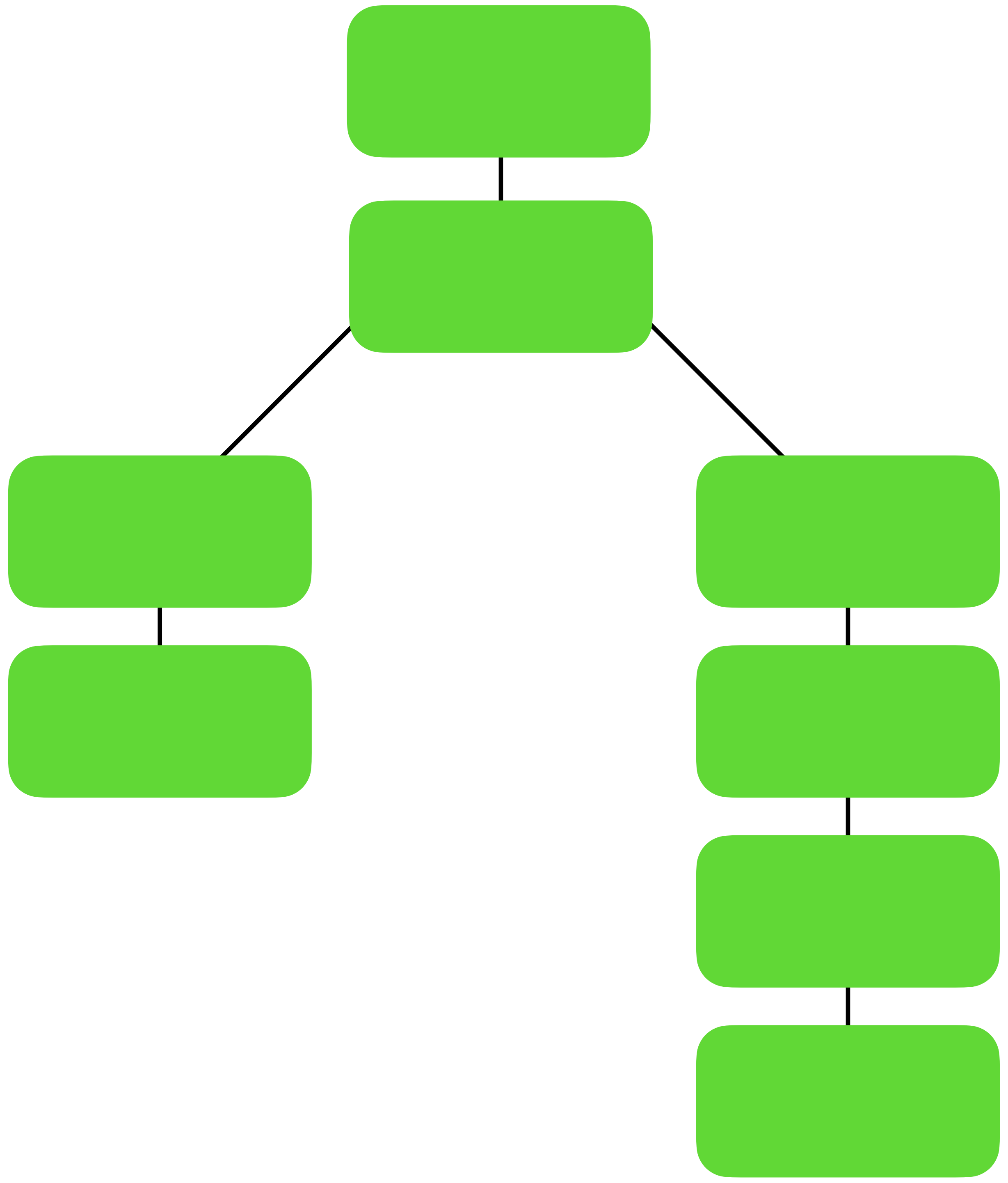
Deferred proof steps in action

SuSLik proof tree

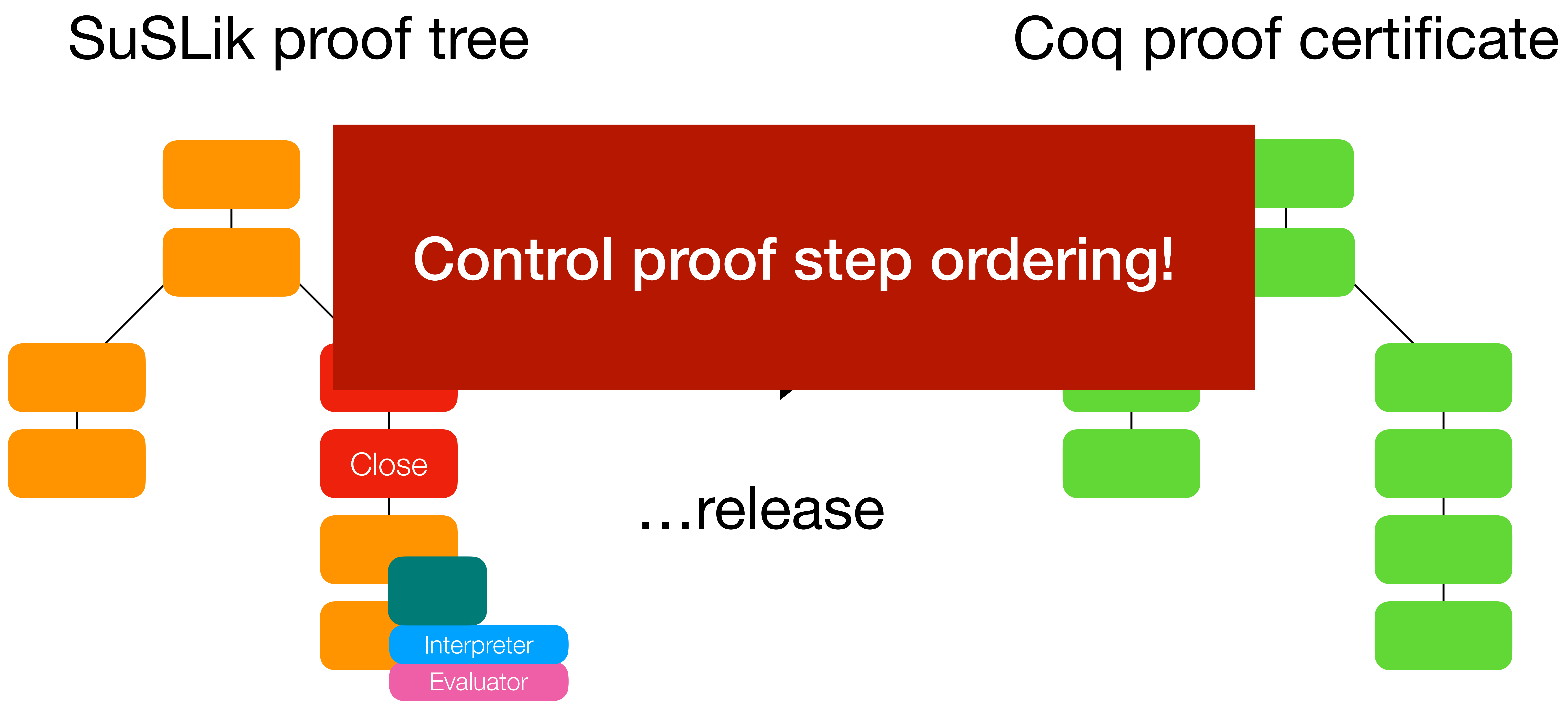


...release

Coq proof certificate



Deferred proof steps in action



However…

$\{\mathtt{r} \mapsto \mathtt{y12} * \mathrm{sll}(\mathtt{y12}, s_1) * \mathrm{sll}(\mathtt{nxt}, s_1) * \ldots\} \rightsquigarrow \{\mathrm{sll}(y, s) * \ldots\}$

$\langle \text{CLOSE } \mathrm{sll}(y, s), 2 \rangle$

$\{\mathrm{sll}(\mathtt{y12}, s_1) * \ldots\} \rightsquigarrow \{[y, 2] * y \mapsto v' * (y + 1) \mapsto nxt' * \mathrm{sll}(nxt', s') * \ldots\}$

However…

What we get

```
by constructor 2=>//=; ∃ v', s', nxt', h11.
```

However...

What we get

```
by constructor 2=>//=; ∃ v', s', nxt', h11.
```

is not what we need!

```
by constructor 2=>//=; ∃ v, s1, nxt, h11.
```

Initially, no existential values given

`by constructor 2=>//=;` $\exists\, v',\ s',\ nxt',$ `h11.`

A few nodes later, s' is unified with s_1

```
by constructor 2=>//=; ∃ v', s', nxt', h11.
```

A few nodes later, s’ is unified with s_1

`by constructor 2=>//=;` $\exists\, v', s', nxt'$, `h11.`

$\langle \ldots\ldots \rangle$

$\{\mathsf{sll}(\mathtt{y12}, s_1) * \ldots\} \rightsquigarrow \{\mathsf{sll}(nxt', s') * \mathsf{sll}(nxt'', s') * \ldots\}$

$\langle \textsc{Unify}\ s_1,\ s' \rangle$

$\{\mathsf{sll}(\mathtt{y12}, s_1) * \ldots\} \rightsquigarrow \{\mathsf{sll}(nxt', s_1) * \mathsf{sll}(nxt'', s_1) * \ldots\}$

$\langle \ldots\ldots \rangle$

A few nodes later, s' is unified with s_1

`by constructor 2=>//=;` $\exists\, v'$, `s1,` nxt', `h11.`

$s' \rightarrow s_1$

$\langle \ldots\ldots \rangle$

$\{\mathsf{sll}(\mathtt{y12}, s_1) * \ldots\} \rightsquigarrow \{\mathsf{sll}(nxt', s') * \mathsf{sll}(nxt'', s') * \ldots\}$

$\langle \textsc{Unify}\ s_1,\ s' \rangle$

$\{\mathsf{sll}(\mathtt{y12}, s_1) * \ldots\} \rightsquigarrow \{\mathsf{sll}(nxt', s_1) * \mathsf{sll}(nxt'', s_1) * \ldots\}$

$\langle \ldots\ldots \rangle$

Eventually, all existentials are provided

```
by constructor 2=>//=; ∃ v', s1, nxt', h11.
```

$s' \rightarrow s_1$

Eventually, all existentials are provided

```
by constructor 2=>//=; ∃ v, s1, nxt, h11.
```

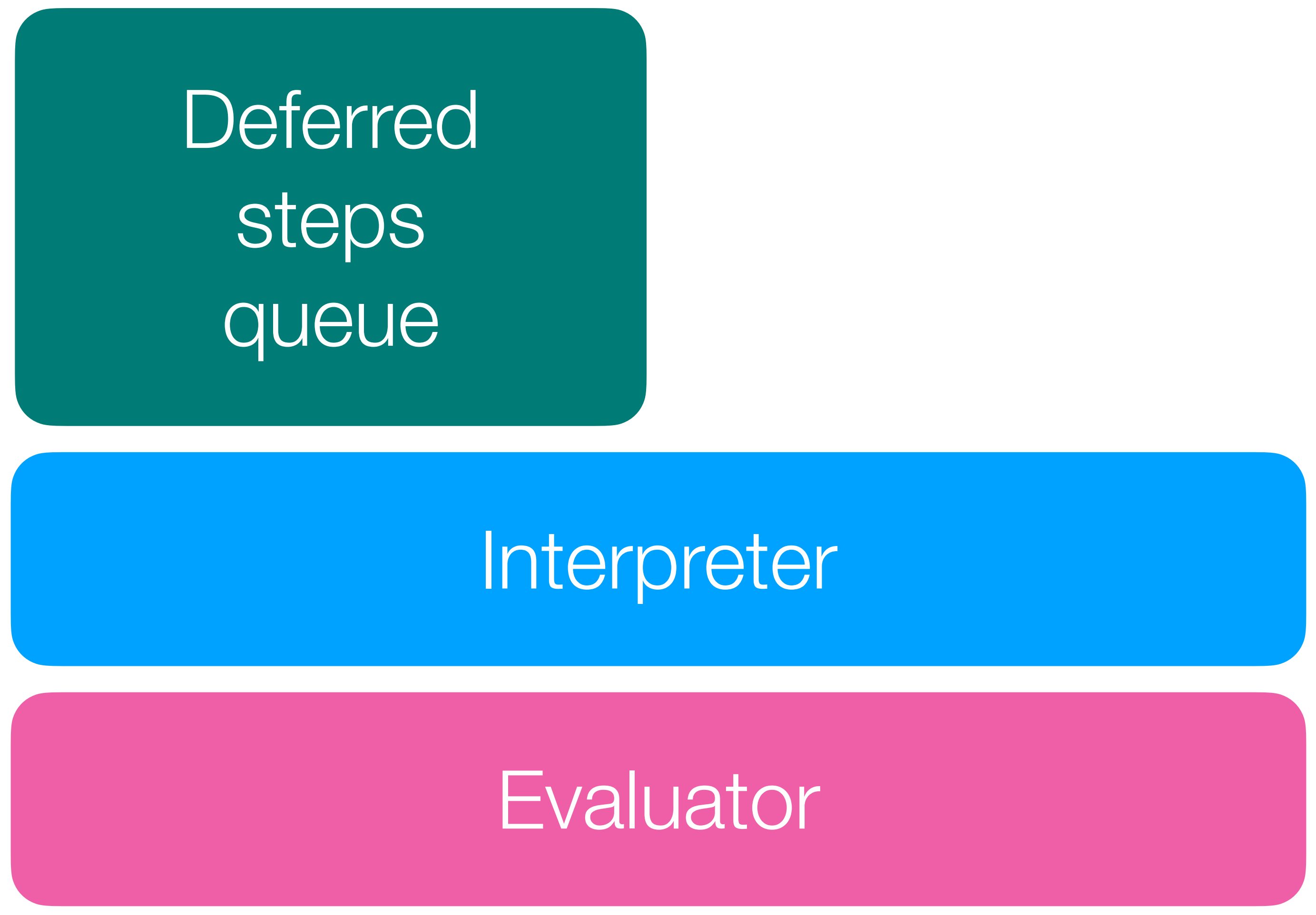
$s' \Rightarrow s_1$

$v' \Rightarrow v$

$nxt' \Rightarrow nxt$

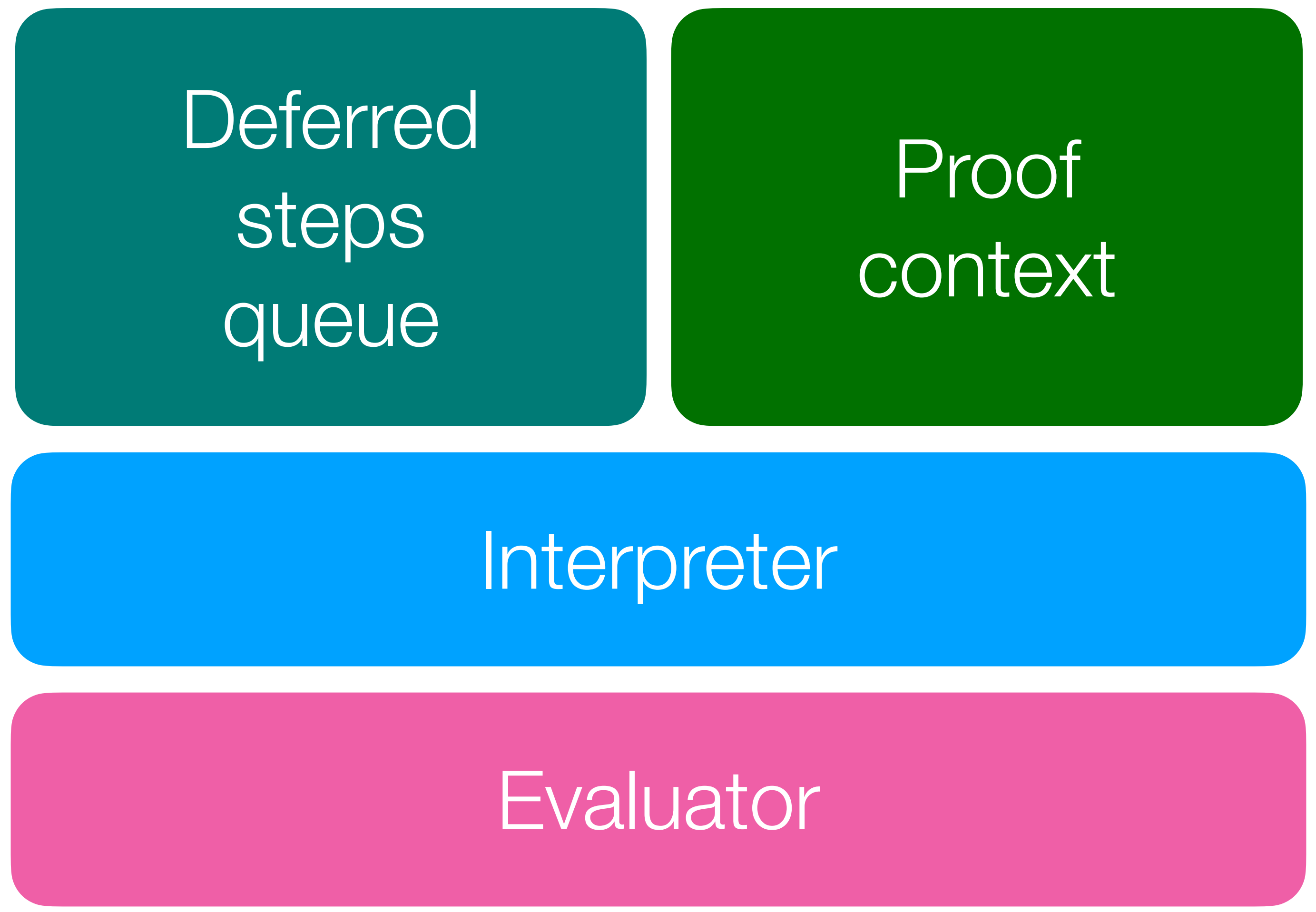
A proof context to track substitutions

$$I_{htt} : \mathsf{Step}_{ssl} \longrightarrow \mathsf{Step}_{htt} \times \mathsf{DeferredStep}_{htt}$$

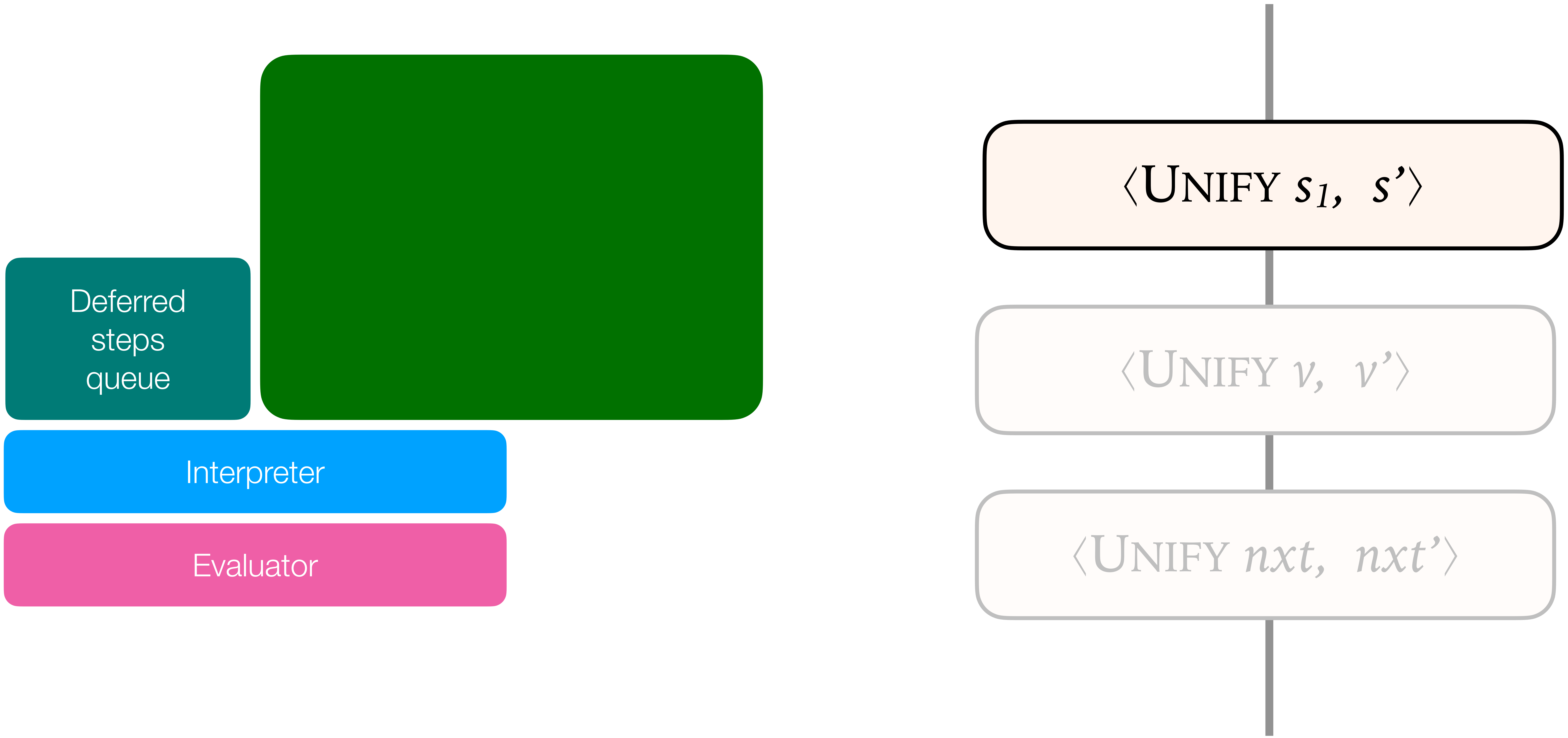


A proof context to track substitutions

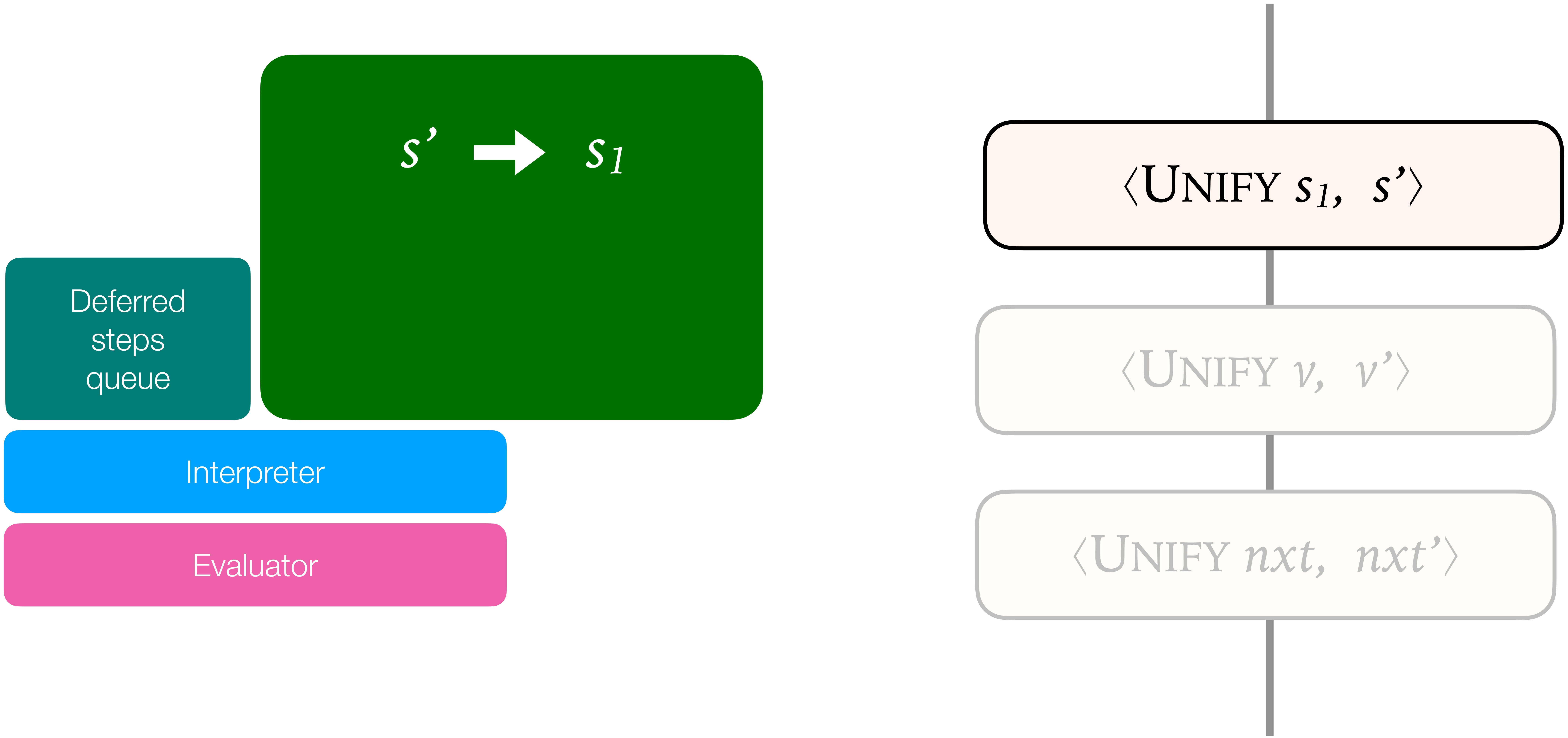
$$I_{htt} : \mathsf{Step}_{ssl} \longrightarrow \mathsf{Context}_{ssl} \longrightarrow \mathsf{Step}_{htt} \times \mathsf{Context}_{htt} \times \mathsf{DeferredStep}_{htt}$$



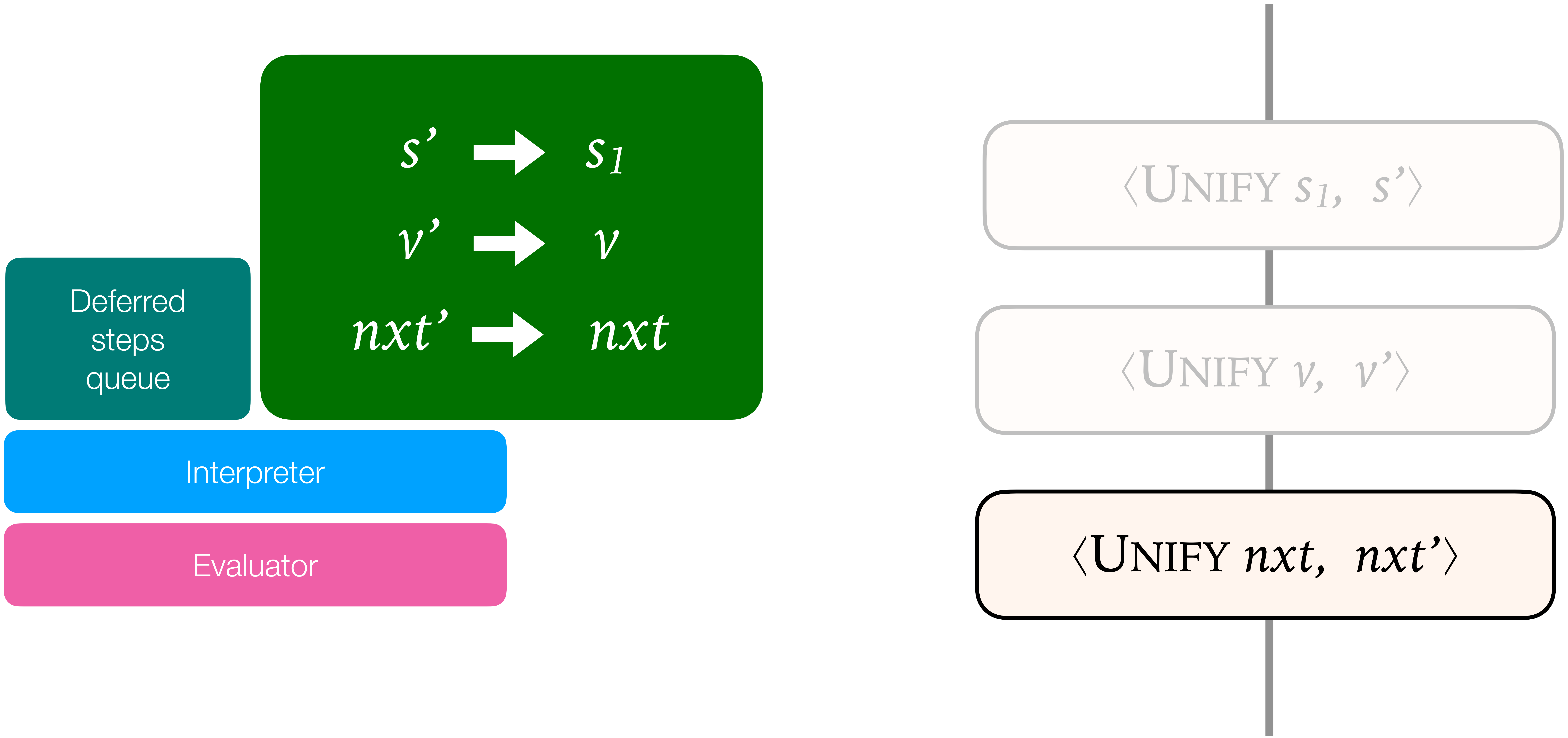
Updating the proof context



Updating the proof context



Updating the proof context



Using proof context information

$s' \Rightarrow s_1$
$v' \Rightarrow v$
$nxt' \Rightarrow nxt$

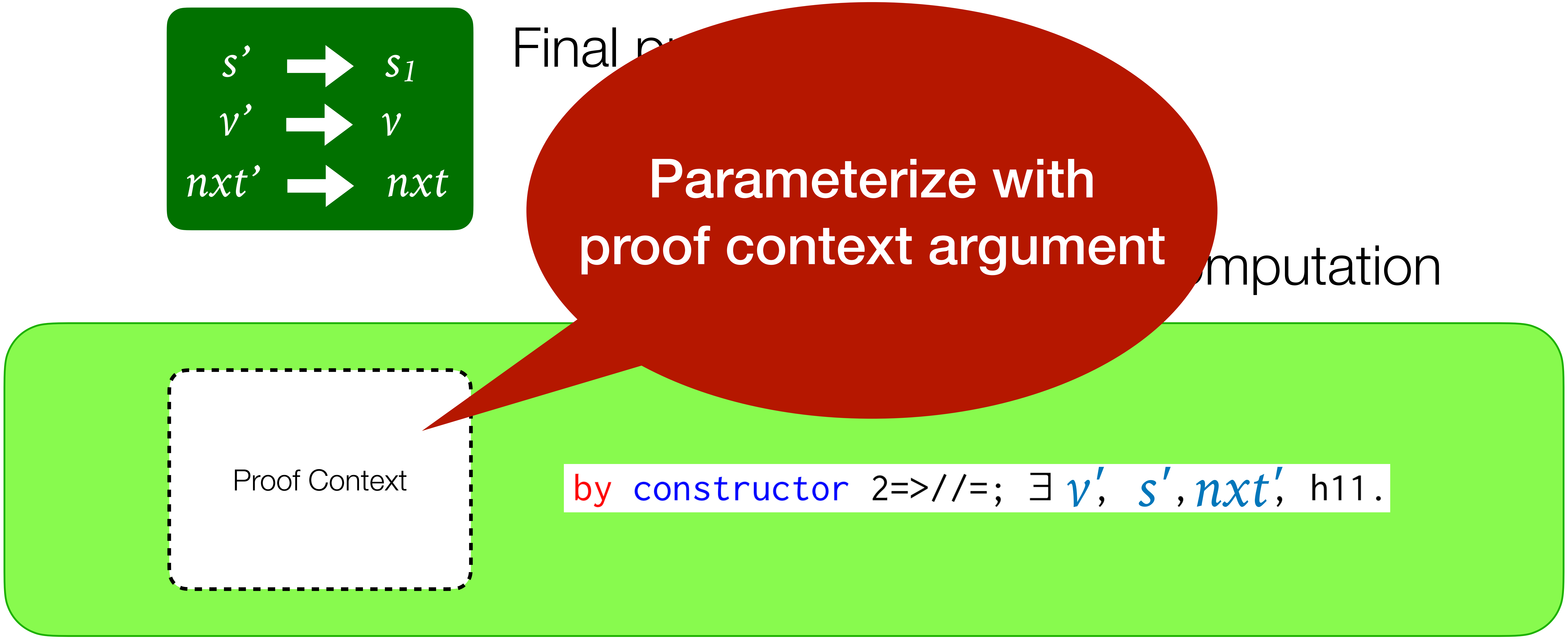
Final proof context

Deferred step computation

Proof Context

```
by constructor 2=>//=; ∃ v', s', nxt', h11.
```

Using proof context information



Using proof context information

$s' \Rightarrow s_1$

$v' \Rightarrow v$

$nxt' \Rightarrow nxt$

```
by constructor 2=>//=; ∃ v', s', nxt', h11.
```

The correct HTT step!

```
by constructor 2=>//=; ∃ v, s1, nxt, h11.
```

Two strategies to bridge the gap

1. Deferred proof steps

delay proof step appearance

like a

continuation

2. Proof contexts

track bookkeeping information

like an

accumulator

Two strategies to bridge the gap

1. Deferred proof steps

delay proof step appearance

like a

continuation

2. Proof contexts

track bookkeeping information

like an

accumulator

Two strategies to bridge the gap

1. Deferred proof steps

delay proof step appearance

like a

continuation

2. Proof contexts

track bookkeeping information

like an

accumulator

Research questions

Research questions

1. Efficiency of certification,
   wrt. proof size and checking time

Research questions

1. Efficiency of certification,
   wrt. proof size and checking time

2. Synthesizer/verifier design choices
   that complicate automated certification

Research questions

1. **Efficiency of certification, wrt. proof size and checking time**
2. Synthesizer/verifier design choices that complicate automated certification

Concise proof sizes

Group	Description	Synthesis Time	HTT			Iris			VST		
			Spec	Proofs	Time	Spec	Proofs	Time	Spec	Proofs	Time
Integers	max	<0.1	55	18	54.3	25	17	3.4	21	20	6.4
	min	<0.1	55	18	50.0	25	17	3.5	21	20	79.2
	swap2	<0.1	49	15	3.7	23	15	3.7	20	14	132.7
	swap4	<0.1	53	23	8.6	29	21	6.5	20	22	649.6
Singly-Linked Lists	length	0.6	68	100	2.8	34	53	6.9	-	-	-
	maximum	0.5	68	99	2.8	-	-	-	21	57	244.8
	minimum	0.5	68	99	2.7	-	-	-	21	57	242
	append	0.2	61	89	6.2	34	50	7.4	23	52	312.9
	copy	0.4	70	103	64.9	38	67	8.6	33	63	370.1
	two-element	0.3	57	50	2.7	28	47	7.3	34	36	171.5
	dispose	<0.1	55	46	1.7	30	31	4.3	31	28	7.8
	singleton	<0.1	55	37	2.3	24	32	4.7	34	26	127.4
DLLs	append	2.3	74	154	7.4	51	98	18.4	24	97	594.6
	singleton	<0.1	55	37	3.1	25	33	6.2	34	27	128.3
Trees	copy	1.3	73	135	6.5	45	83	15.6	32	77	516.5
	flatten	0.2	92	138	5.9	56	75	13.5	58	76	685.7
	dispose	<0.1	58	62	2.4	34	37	5.7	31	32	10.8
	size	0.5	64	92	5.4	37	58	9.8	-	-	-

Concise proof sizes

Group	Description	Synthesis Time	HTT			Iris			VST		
			Spec	Proofs	Time	Spec	Proofs	Time	Spec	Proofs	Time
Integers	max	<0.1	55	18	54.3	25	17	3.4	21	20	6.4
	min	<0.1	55	18	50.0	25	17	3.5	21	20	79.2
	swap2	<0.1	49	15	3.7	23	15	3.7	20	14	132.7
	swap4	<0.1	53	23	8.6	29	21	6.5	20	22	649.6
Singly-Linked Lists	length	0.6	68	100	2.8	34	53	6.9	-	-	-
	maximum	0.5	68	99	2.8	-	-	-	21	57	244.8
	minimum	0.5	68	99	2.7	-	-	-	21	57	242
	append	0.2	61	89	6.2	34	50	7.4	23	52	312.9
	copy	0.4	70	103	64.9	38	67	8.6	33	63	370.1
	two-element	0.3	57	50	2.7	28	47	7.3	34	36	171.5
	dispose	<0.1	55	46	1.7	30	31	4.3	31	28	7.8
	singleton	<0.1	55	37	2.3	24	32	4.7	34	26	127.4
DLLs	append	2.3	74	154	7.4	51	98	18.4	24	97	594.6
	singleton	<0.1	55	37	3.1	25	33	6.2	34	27	128.3
Trees	copy	1.3	73	135	6.5	45	83	15.6	32	77	516.5
	flatten	0.2	92	138	5.9	56	75	13.5	58	76	685.7
	dispose	<0.1	58	62	2.4	34	37	5.7	31	32	10.8
	size	0.5	64	92	5.4	37	58	9.8	-	-	-

2-20s checking times for HTT/Iris, longer for VST

Group	Description	Synthesis Time	HTT			Iris			VST		
			Spec	Proofs	Time	Spec	Proofs	Time	Spec	Proofs	Time
Integers	max	<0.1	55	18	54.3	25	17	3.4	21	20	6.4
	min	<0.1	55	18	50.0	25	17	3.5	21	20	79.2
	swap2	<0.1	49	15	3.7	23	15	3.7	20	14	132.7
	swap4	<0.1	53	23	8.6	29	21	6.5	20	22	649.6
Singly-Linked Lists	length	0.6	68	100	2.8	34	53	6.9	-	-	-
	maximum	0.5	68	99	2.8	-	-	-	21	57	244.8
	minimum	0.5	68	99	2.7	-	-	-	21	57	242
	append	0.2	61	89	6.2	34	50	7.4	23	52	312.9
	copy	0.4	70	103	64.9	38	67	8.6	33	63	370.1
	two-element	0.3	57	50	2.7	28	47	7.3	34	36	171.5
	dispose	<0.1	55	46	1.7	30	31	4.3	31	28	7.8
	singleton	<0.1	55	37	2.3	24	32	4.7	34	26	127.4
DLLs	append	2.3	74	154	7.4	51	98	18.4	24	97	594.6
	singleton	<0.1	55	37	3.1	25	33	6.2	34	27	128.3
Trees	copy	1.3	73	135	6.5	45	83	15.6	32	77	516.5
	flatten	0.2	92	138	5.9	56	75	13.5	58	76	685.7
	dispose	<0.1	58	62	2.4	34	37	5.7	31	32	10.8
	size	0.5	64	92	5.4	37	58	9.8	-	-	-

2-20s checking times for HTT/Iris, longer for VST

Group	Description	Synthesis Time	HTT Spec	HTT Proofs	HTT Time	Iris Spec	Iris Proofs	Iris Time	VST Spec	VST Proofs	VST Time
Integers	max	<0.1	55	18	54.3	25	17	3.4	21	20	6.4
	min	<0.1	55	18	50.0	25	17	3.5	21	20	79.2
	swap2	<0.1	49	15	3.7	23	15	3.7	20	14	132.7
	swap4	<0.1	53	23	8.6	29	21	6.5	20	22	649.6
Singly-Linked Lists	length	0.6	68	100	2.8	34	53	6.9	-	-	-
	maximum	0.5	68	99	2.8	-	-	-	21	57	244.8
	minimum	0.5	68	99	2.7	-	-	-	21	57	242
	append	0.2	61	89	6.2	34	50	7.4	23	52	312.9
	copy	0.4	70	103	64.9	38	67	8.6	33	63	370.1
	two-element	0.3	57	50	2.7	28	47	7.3	34	36	171.5
	dispose	<0.1	55	46	1.7	30	31	4.3	31	28	7.8
	singleton	<0.1	55	37	2.3	24	32	4.7	34	26	127.4
DLLs	append	2.3	74	154	7.4	51	98	18.4	24	97	594.6
	singleton	<0.1	55	37	3.1	25	33	6.2	34	27	128.3
Trees	copy	1.3	73	135	6.5	45	83	15.6	32	77	516.5
	flatten	0.2	92	138	5.9	56	75	13.5	58	76	685.7
	dispose	<0.1	58	62	2.4	34	37	5.7	31	32	10.8
	size	0.5	64	92	5.4	37	58	9.8	-	-	-

2-20s checking times for HTT/Iris, longer for VST

Group	Description	Synthesis Time	HTT			Iris			VST		
			Spec	Proofs	Time	Spec	Proofs	Time	Spec	Proofs	Time
Integers	max	<0.1	55	18	54.3	25	17	3.4	21	20	6.4
	min	<0.1	55	18	50.0	25	17	3.5	21	20	79.2
	swap2	<0.1	49	15	3.7	23	15	3.7	20	14	132.7
	swap4	<0.1	53	23	8.6	29	21	6.5	20	22	649.6
Singly-Linked Lists	length	0.6	68	100	2.8	34	53	6.9	-	-	-
	maximum	0.5	68	99	2.8	-	-	-	21	57	244.8
	minimum	0.5	68	99	2.7	-	-	-	21	57	242
	append	0.2	61	89	6.2	34	50	7.4	23	52	312.9
	copy	0.4	70	103	64.9	38	67	8.6	33	63	370.1
	two-element	0.3	57	50	2.7	28	47	7.3	34	36	171.5
	dispose	<0.1	55	46	1.7	30	31	4.3	31	28	7.8
	singleton	<0.1	55	37	2.3	24	32	4.7	34	26	127.4
DLLs	append	2.3	74	154	7.4	51	98	18.4	24	97	594.6
	singleton	<0.1	55	37	3.1	25	33	6.2	34	27	128.3
Trees	copy	1.3	73	135	6.5	45	83	15.6	32	77	516.5
	flatten	0.2	92	138	5.9	56	75	13.5	58	76	685.7
	dispose	<0.1	58	62	2.4	34	37	5.7	31	32	10.8
	size	0.5	64	92	5.4	37	58	9.8	-	-	-

Research questions

1. Efficiency of certification,
   wrt. proof size and checking time

2. **Synthesizer/verifier design choices that complicate automated certification**

Two challenges

Synthesizer/verifier design choices
that complicate automated certification

- Implementation experience for the 3 target verifiers
- Recreating synthesis steps not recoverable from proof tree

Two challenges

Synthesizer/verifier design choices
that complicate automated certification

- **Implementation experience for the 3 target verifiers**
- Recreating synthesis steps not recoverable from proof tree

Our experience with HTT

HTT | VST | IRIS

Our experience with HTT

HTT | VST | IRIS

- The simplest framework

Our experience with HTT

HTT | VST | IRIS

- The simplest framework
- Shallow embedding

Our experience with HTT

HTT | VST | IRIS

- The simplest framework
- Shallow embedding
- No need to distinguish program and proof terms

Our experience with VST

HTT | **VST** | IRIS

Our experience with VST

HTT | **VST** | IRIS

- VST certifies real C programs

Our experience with VST

HTT | **VST** | IRIS

- VST certifies real C programs
- Lots of custom notation used to simplify proofs

Our experience with VST

HTT | **VST** | IRIS

- VST certifies real C programs
- Lots of custom notation used to simplify proofs
- Need understanding of implementation details to write tactics for proof automation

Our experience with IRIS

HTT | VST | **IRIS**

Our experience with IRIS

HTT | VST | IRIS

- Most difficult

Our experience with IRIS

HTT | VST | **IRIS**

- Most difficult
- With human-oriented approach, proofs need to manage lots of goal/hypothesis information

Our experience with IRIS

HTT | VST | **IRIS**

Our experience with IRIS

HTT | VST | IRIS

- Alternative, "SUSLIK-style" approach relies on heap unification to avoid the trouble

Our experience with IRIS

HTT | VST | **IRIS**

- Alternative, "SUSLIK-style" approach relies on heap unification to avoid the trouble
- But IRIS's heap unification tactics are fragile

Two challenges

Synthesizer/verifier design choices
that complicate automated certification

- Implementation experience for the 3 target verifiers
- Recreating synthesis steps not recoverable from proof tree

Two challenges

Synthesizer/verifier design choices
that complicate automated certification

- Implementation experience for the 3 target verifiers
- **Recreating synthesis steps not recoverable from proof tree**

SuSLik solves pure assertions with SMT

SuSLik solves pure assertions with SMT

pure assertions

$$\vdash \Phi \Rightarrow \Psi$$

SuSLik solves pure assertions with SMT

pure assertions

$$\vdash \Phi \Rightarrow \Psi$$

Synthesis

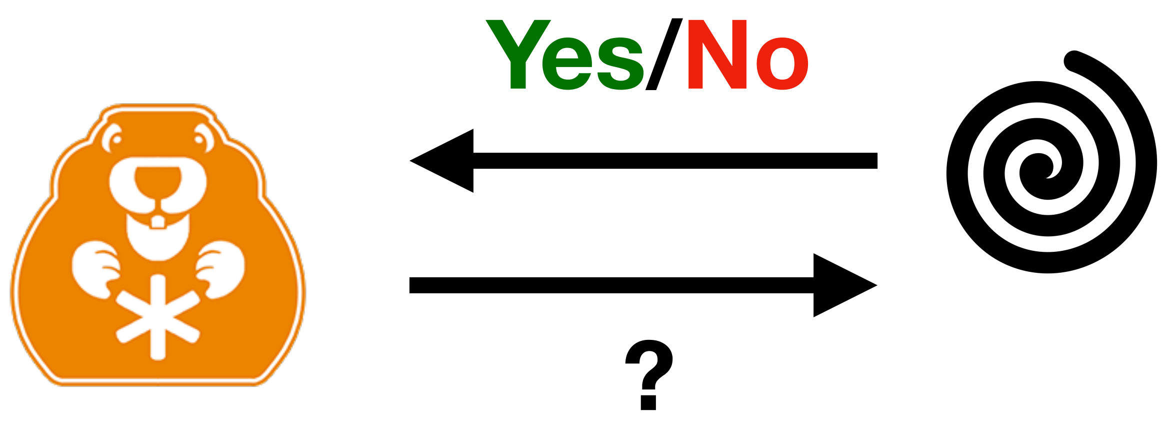
SMT solver

SuSLik solves pure assertions with SMT

pure assertions

$$\vdash \Phi \Rightarrow \Psi$$

Synthesis



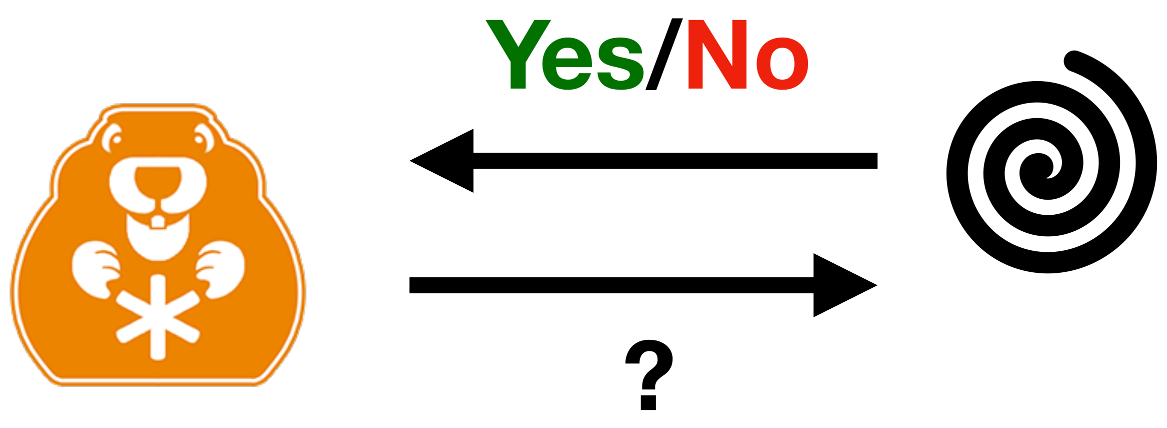
SMT solver

SuSLik solves pure assertions with SMT

pure assertions

$$\vdash \Phi \Rightarrow \Psi$$

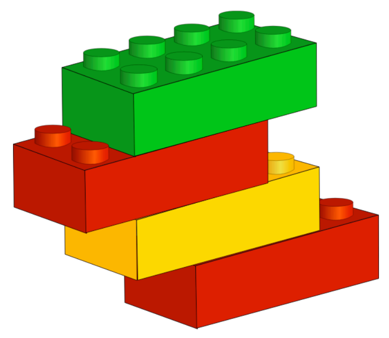
Synthesis



SMT solver

Verification

HTT



constructive proof

SuSLik solves pure assertions with SMT

pure assertions

$$\vdash \Phi \Rightarrow \Psi$$

Synthesis

Yes/No

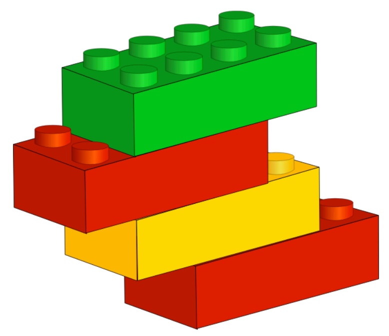
?

SMT solver

Verification

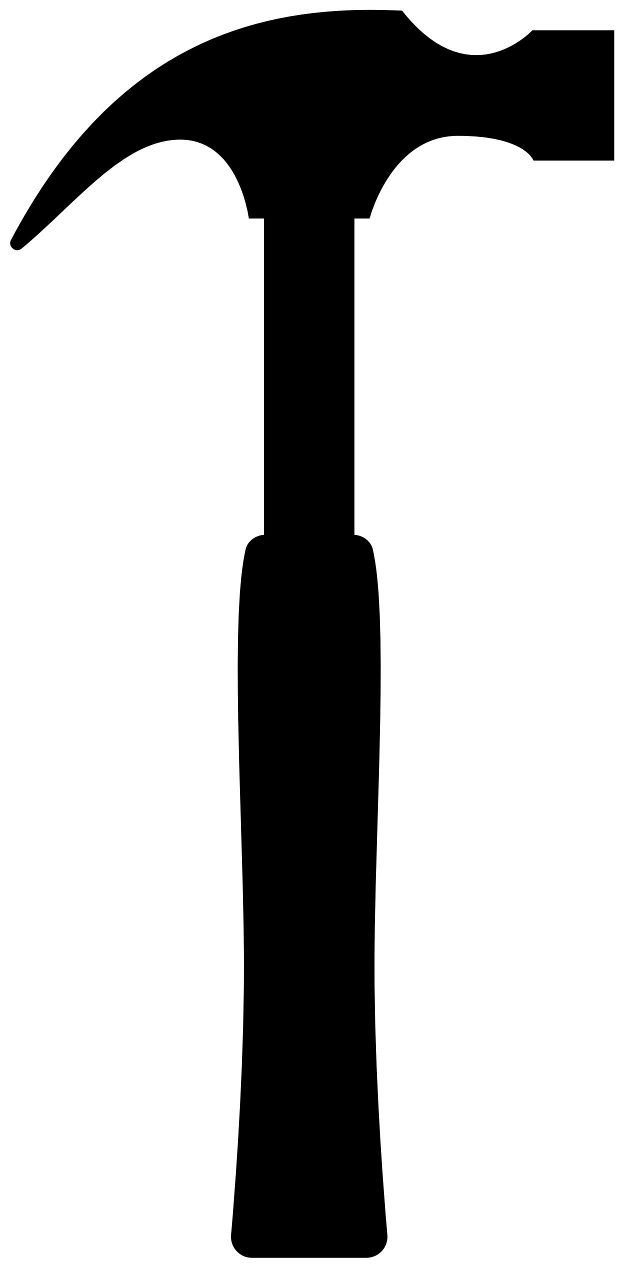
HTT

```
apply …
rewrite …
apply …
```



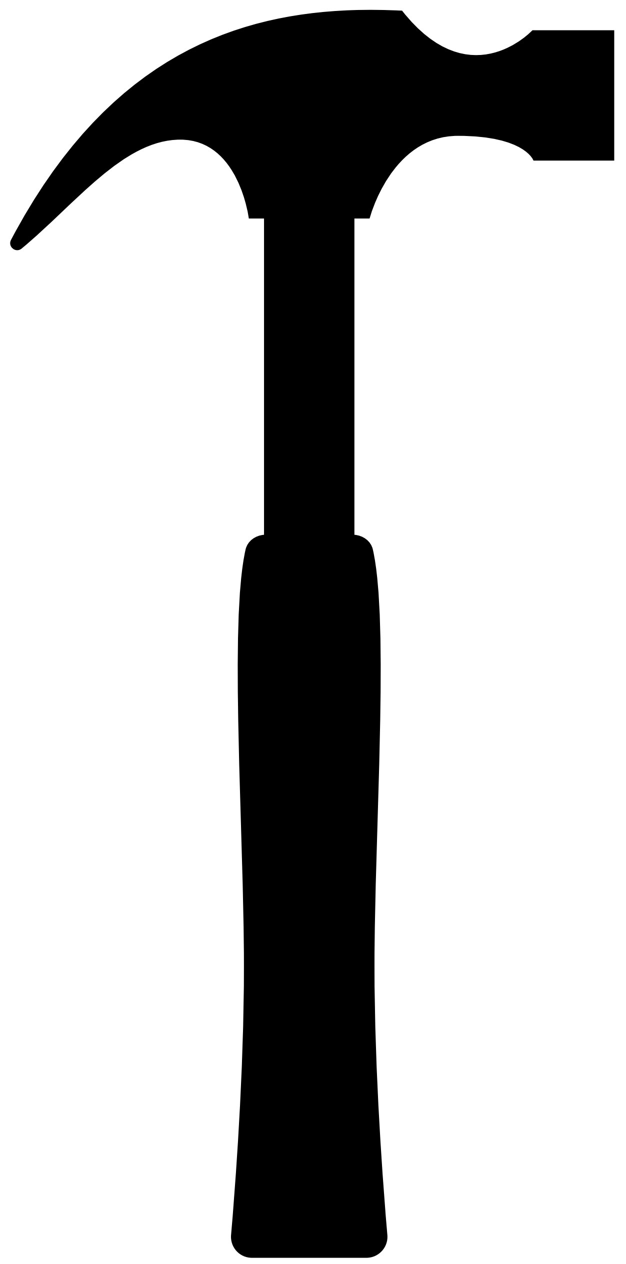
constructive proof

Solution: certified solvers (hammers)



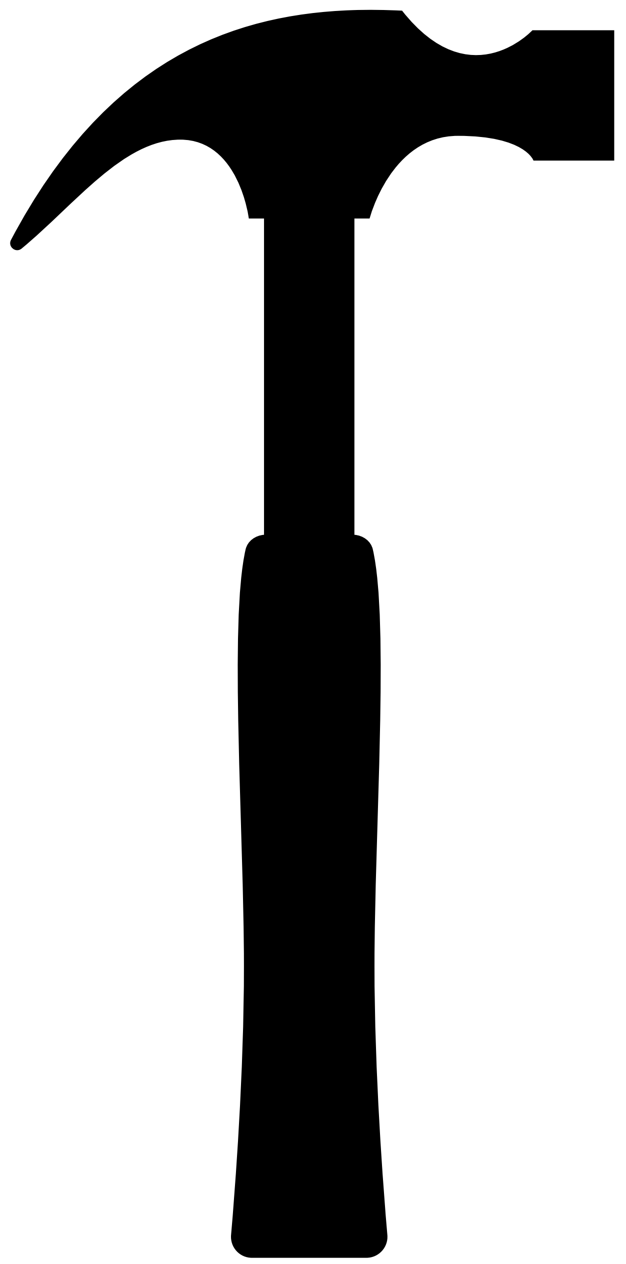
Solution: certified solvers (hammers)

- Single-line commands



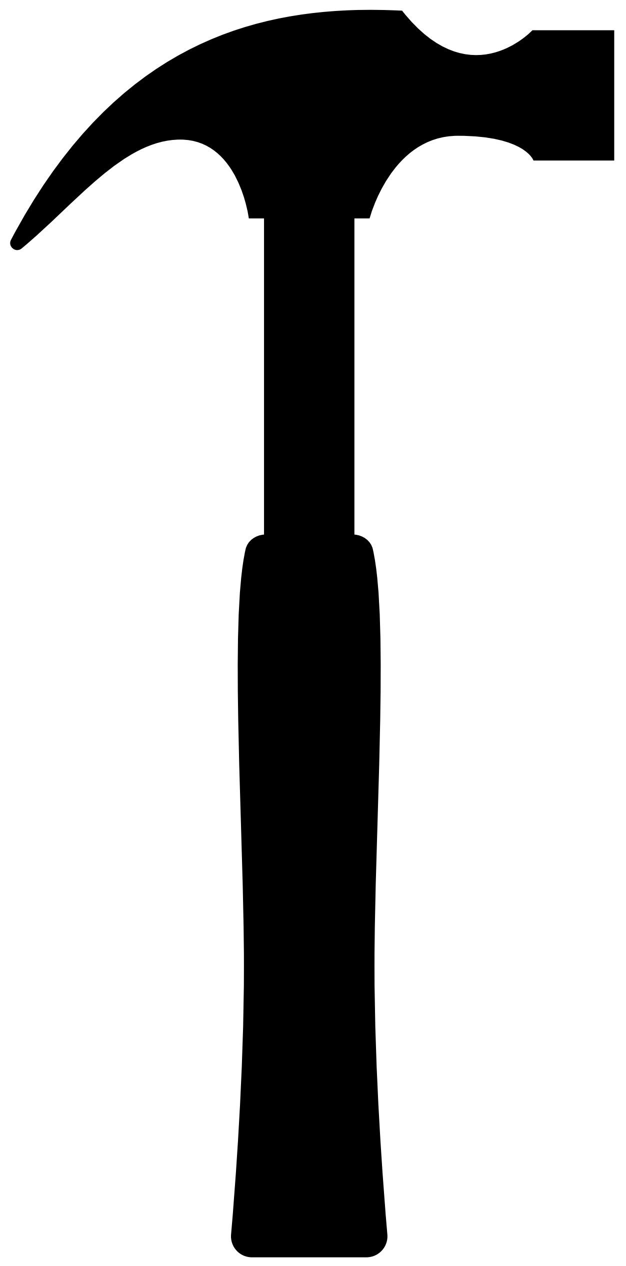
Solution: certified solvers (hammers)

- Single-line commands
- Powerful proof automation



Solution: certified solvers (hammers)

- Single-line commands
- Powerful proof automation
- Advanced ATP-guided proof search on available lemmas



Hammer time!

```
Lemma pure_example k2 vx2 lo1x :
  vx2 <= lo1x -> 0 <= vx2 -> vx2 <= 7 ->
  0 <= k2 -> ¬(vx2 <= k2) -> k2 <= 7 ->
  k2 <= (if vx2 <= lo1x then vx2 else lo1x).
```

Hammer time!

Capture and extract entailments into lemmas

```
Lemma pure_example k2 vx2 lo1x :
  vx2 <= lo1x -> 0 <= vx2 -> vx2 <= 7 ->
  0 <= k2 -> ¬(vx2 <= k2) -> k2 <= 7 ->
  k2 <= (if vx2 <= lo1x then vx2 else lo1x).
```

Hammer time!

```
Lemma pure_example k2 vx2 lo1x :
  vx2 <= lo1x -> 0 <= vx2 -> vx2 <= 7 ->
  0 <= k2 -> ¬(vx2 <= k2) -> k2 <= 7 ->
  k2 <= (if vx2 <= lo1x then vx2 else lo1x).
```

Hammer time!

Prove extracted lemma with CoqHammer[2]

```
Lemma pure_example k2 vx2 lo1x :
  vx2 <= lo1x -> 0 <= vx2 -> vx2 <= 7 ->
  0 <= k2 -> ¬(vx2 <= k2) -> k2 <= 7 ->
  k2 <= (if vx2 <= lo1x then vx2 else lo1x).

Proof. intros. hammer. Qed.
```

[2] Czajka and Kaliszyk ’18

Lemma becomes usable for automation

Main proof

???

```
Lemma pure_example k2 vx2 lo1x :
  vx2 <= lo1x -> 0 <= vx2 -> vx2 <= 7 ->
  0 <= k2 -> ¬(vx2 <= k2) -> k2 <= 7 ->
  k2 <= (if vx2 <= lo1x then vx2 else lo1x).

Proof. intros. hammer. Qed.
```

Lemma becomes usable for automation

Main proof

```
Lemma pure_example k2 vx2 lo1x :
  vx2 <= lo1x -> 0 <= vx2 -> vx2 <= 7 ->
  0 <= k2 -> ¬(vx2 <= k2) -> k2 <= 7 ->
  k2 <= (if vx2 <= lo1x then vx2 else lo1x).

Proof. intros. hammer. Qed.
```

Some advanced benchmarks need manual help with auxiliary lemmas

Group	Program	Synt. time	Coq time	Lemmas	Manual
Doubly-Linked Lists	copy	8.7	22.1	7	4
	two-element	0.6	11.6	3	3
	from-sll	1.2	18.2	5	2
Binary Search Trees	find-smallest	2.9	12.2	7	1
	insert	33.8	72.9	15	6
	rotate-left	6.3	16.9	4	2
	rmv-root-left	3.4	26.5	6	2
	rmv-root-right	34.7	25.1	6	2
	rotate-right	5.6	15.9	4	2
Sorted Lists	insertion-sort	1.8	10.7	7	0
	insert-sort-free	0.6	9.5	5	0
	insert	9.7	26.8	18	8
	prepend	0.3	6.6	2	0

Some advanced benchmarks need manual help with auxiliary lemmas

Group	Program	Synt. time	Coq time	Lemmas	Manual
Doubly-Linked Lists	copy	8.7	22.1	7	4
	two-element	0.6	11.6	3	3
	from-sll	1.2	18.2	5	2
Binary Search Trees	find-smallest	2.9	12.2	7	1
	insert	33.8	72.9	15	6
	rotate-left	6.3	16.9	4	2
	rmv-root-left	3.4	26.5	6	2
	rmv-root-right	34.7	25.1	6	2
	rotate-right	5.6	15.9	4	2
Sorted Lists	insertion-sort	1.8	10.7	7	0
	insert-sort-free	0.6	9.5	5	0
	insert	9.7	26.8	18	8
	prepend	0.3	6.6	2	0

Some advanced benchmarks need manual help with auxiliary lemmas

Group	Program	Synt. time	Coq time	Lemmas	Manual
[illegible]	copy	8.7	22.1	7	4
	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
Sorted Lists	insert-sort-free	0.6	9.5	5	0
	insert	9.7	26.8	18	8
	prepend	0.3	6.6	2	0

Future work:

a middle ground between

automation and **interactivity**?

We addressed a fundamental gap

synthesis ⟷ **verification**

The takeaway

The takeaway

- An **abstract proof evaluator** framework

The takeaway

- An **abstract proof evaluator** framework
- Instantiations for **3 target verifiers** (HTT, VST, IRIS)

The takeaway

- An **abstract proof evaluator** framework
- Instantiations for **3 target verifiers** (HTT, VST, IRIS)
- **Evaluation** on characteristic benchmarks (~15 shared by all three verifiers)

The takeaway

- An **abstract proof evaluator** framework
- Instantiations for **3 target verifiers** (HTT, VST, IRIS)
- **Evaluation** on characteristic benchmarks (~15 shared by all three verifiers)

Fully certified program synthesis!