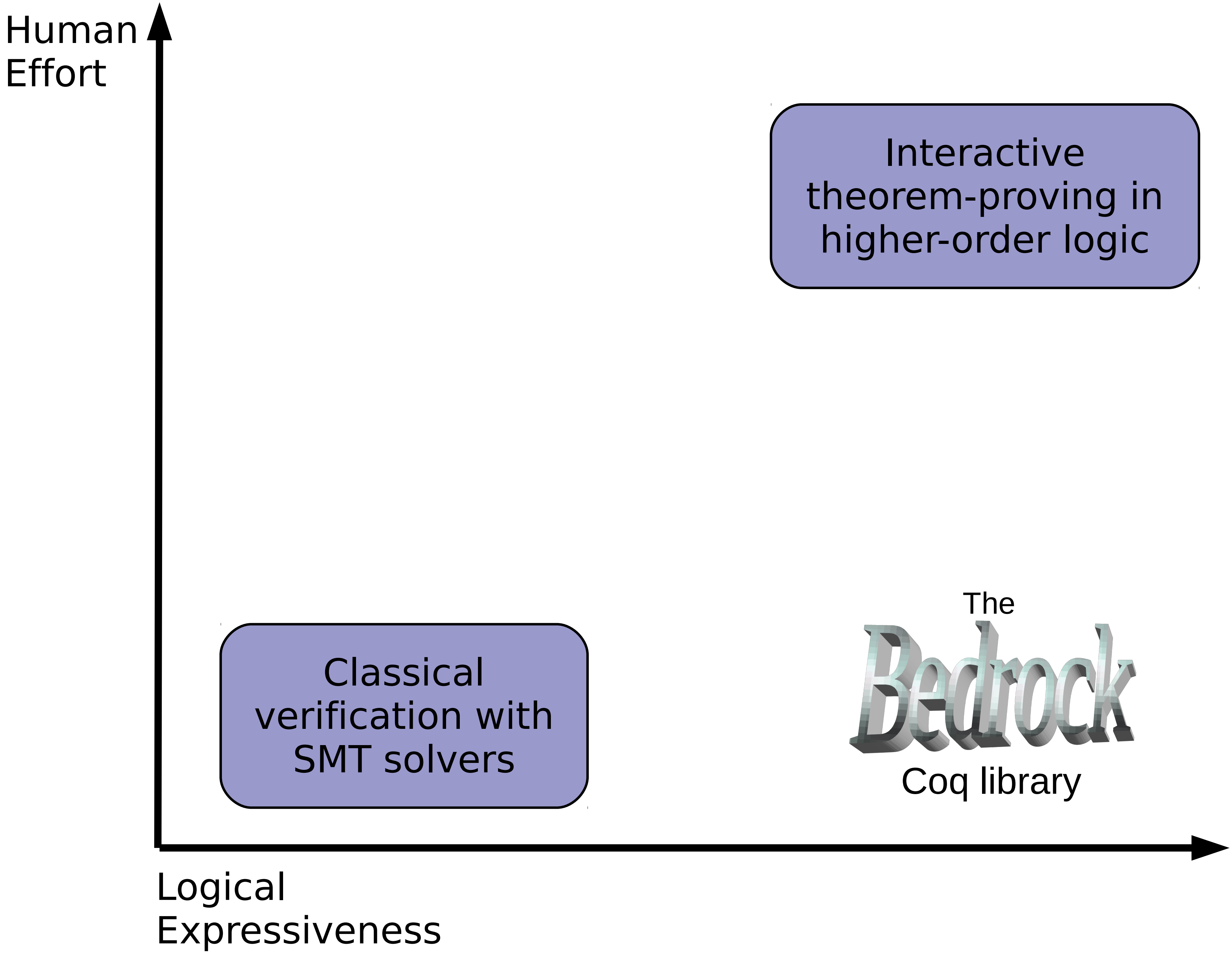
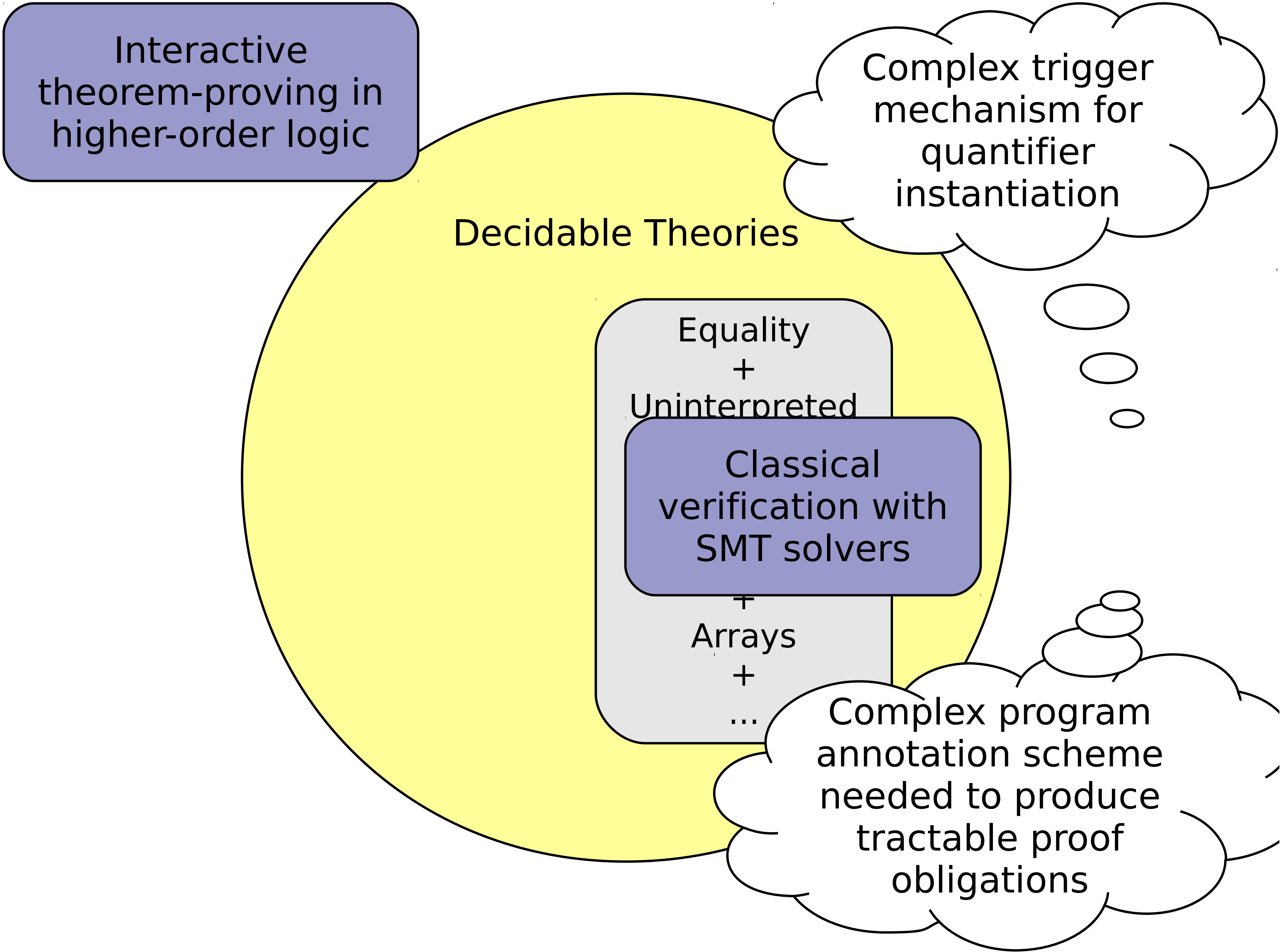
Mostly-Automated Verification of Low-Level Programs in Computational Separation Logic

Adam Chlipala
Harvard University

PLDI 2011


Human
Effort
Interactive
theorem-proving in
higher-order logic
Classical
verification with
SMT solvers
The
Bedrock
Coq library
Logical
Expressiveness


Interactive theorem-proving in higher-order logic
Complex trigger mechanism for quantifier instantiation
Decidable Theories
Equality
+
Uninterpreted
Classical verification with SMT solvers
+
Arrays
+
...
Complex program annotation scheme needed to produce tractable proof obligations

Solution: **Computational** specifications use standard programming features to avoid quantifiers in almost all specifications.

Complex trigger mechanism for quantifier instantiation

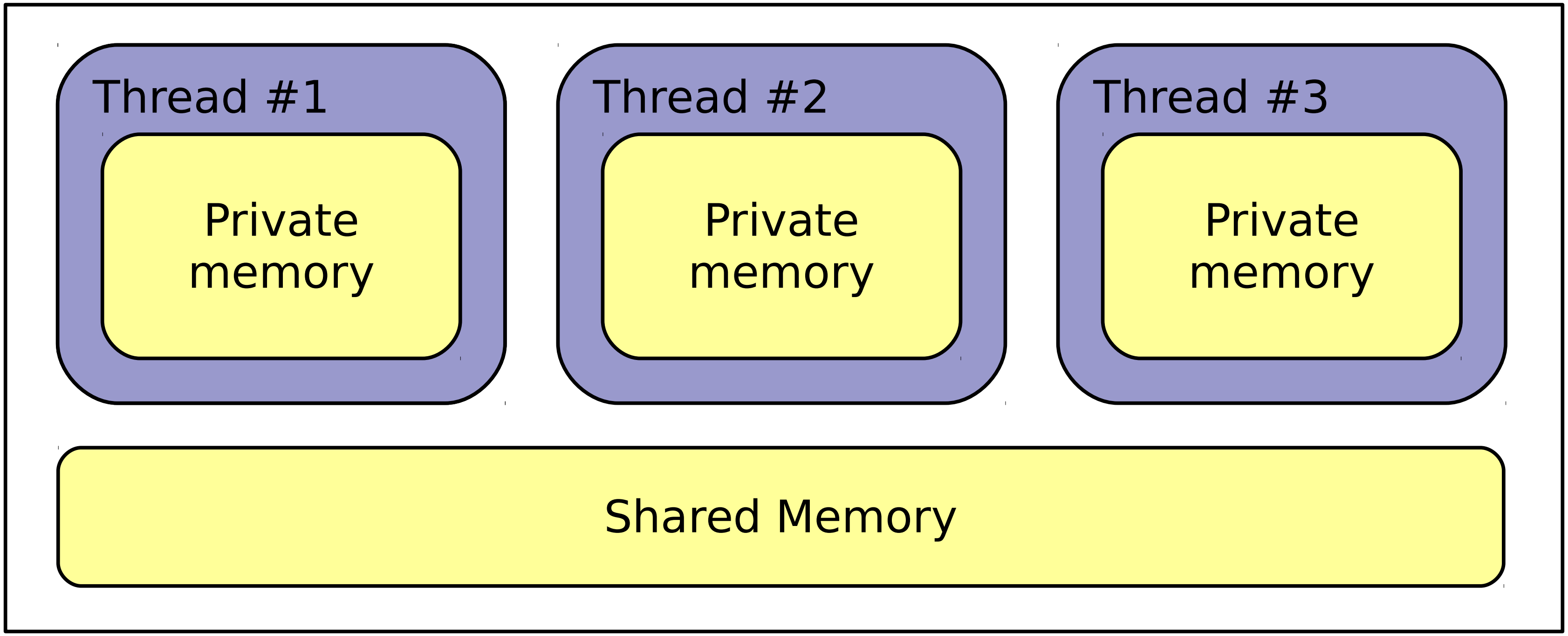
The **Bedrock** Coq library

Classical verification with SMT solvers

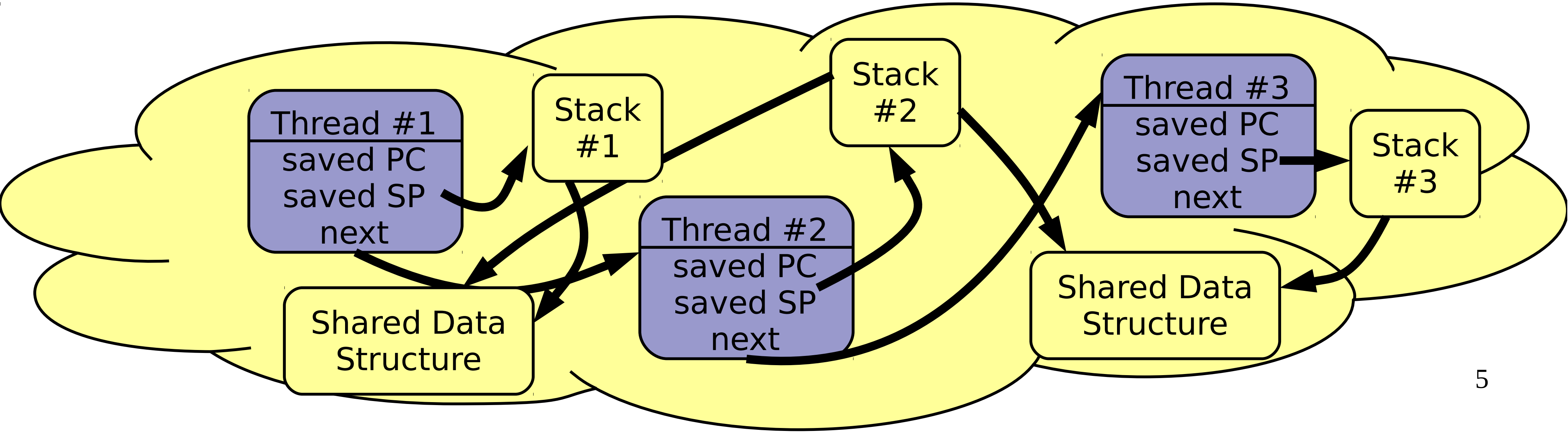
Solution: **Higher-Order Separation Logic** is expressive enough to allow direct use of the most natural specs.

Complex program annotation scheme needed to produce tractable proof obligations

A Thread Scheduler, Abstractly



A Thread Scheduler, Concretely



What does correctness mean?

"∀ sets of threads with specifications,
written in terms of local and shared heap areas,
the scheduling library satisfies all of the specs."

Quantify over *specifications*

Example: spec for `yield()` function

```
Definition yieldS := st ~> Ex fr, Ex ginv,
  Ex invs, Ex root,
      susp ginv (fun sp => sep ([< sp = st#Rsp >]
        * ![fr])%Sep) st#Rret
      /\ codesOk ginv invs
      /\ ![ !{mallocHeap 0} * st#Rsp ==> root
        * !{threads invs root}
        * ![ginv] * ![fr] ] st.
```

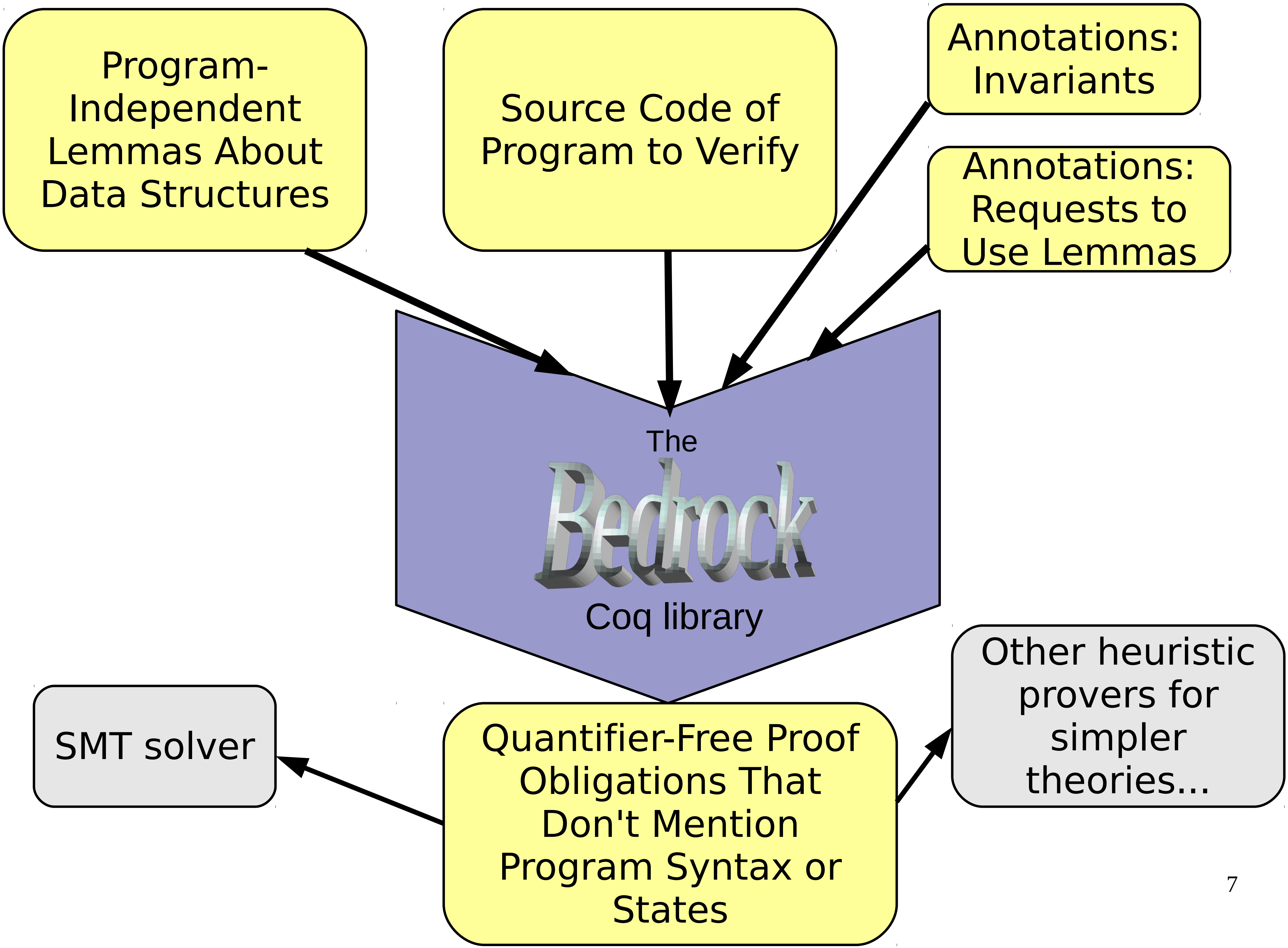
Quantify over *lists of specifications*

Higher-Order Logic + Separation Logic = Higher-Order Separation Logic

Proofs usually considered too hard to automate...

Proofs usually considered too hard to automate...

?????


Program-Independent Lemmas About Data Structures
Source Code of Program to Verify
Annotations: Invariants
Annotations: Requests to Use Lemmas
The
Bedrock
Coq library
SMT solver
Quantifier-Free Proof Obligations That Don't Mention Program Syntax or States
Other heuristic provers for simpler theories...

Computational Separation Logic

Step 1. Define data structure invariants as recursive functions.

```
(* Abstraction predicate for finite sets represented
 * with unsorted linked lists *)
Fixpoint llistOk (s : set) (ls : list nat)
     (a : nat) : sprop :=
  match ls with
    | nil => [< a = 0 /\ s = empty >]
    | x :: ls' => Ex a', [< a <> 0 /\ s x = true >]
      * a ==> x * (a+1) ==> a
      * !{llistOk (del s x) l
  end.
```

Limited form of existential quantification

Computational Separation Logic

Step 2. Prove simplification lemmas.

```
Theorem llist_empty_fwd : forall s ls a,
  a = 0
  -> llistOk s ls a ===> [< ls = nil /\ s = empty >].
  destruct ls; sepLemma.
Qed.
```

Implication in separation logic

```
Theorem llist_nonempty_fwd : forall a s ls,
  a <> 0
  -> llistOk s ls a ===> Ex x, Ex ls', Ex a',
     [< ls = x :: ls' /\ s x = true >] * a ==> x
     * (a+1) ==> a' * !{llistOk (del s x) ls' a'}.
  destruct ls; sepLemma.
Qed.
```

Computational Separation Logic

Step 3. Write annotated program.

```
Definition linkedList := bmodule {{
  bfunction "linc" [lincS] {
    [lincS]
    While (R0 != 0) {
      Use [llist_nonempty_fwd];;
      Use [llist_nonempty_bwd];;

      $[R0] <- $[R0] + 1;;
      R0 <- $[R0+1]
    };;

    Use [llist_empty_fwd];;
    Use [llist_empty_bwd];;

    Goto Rret
  }
}}.
```

Loop invariant

Request to use a lemma here

Computational Separation Logic

Step 4. Prove module correctness theorem.

```
Theorem linkedListOk : moduleOk linkedList.
  structured; sep.
Qed.
```

Bedrock tactics do almost all of the work.

Why Automating Separation Logic Proofs is Easy

$$x > 5 \ \&\& \ \cancel{x < 7}$$

Implies

$$\cancel{x < 7} \ \&\& \ x = 6$$

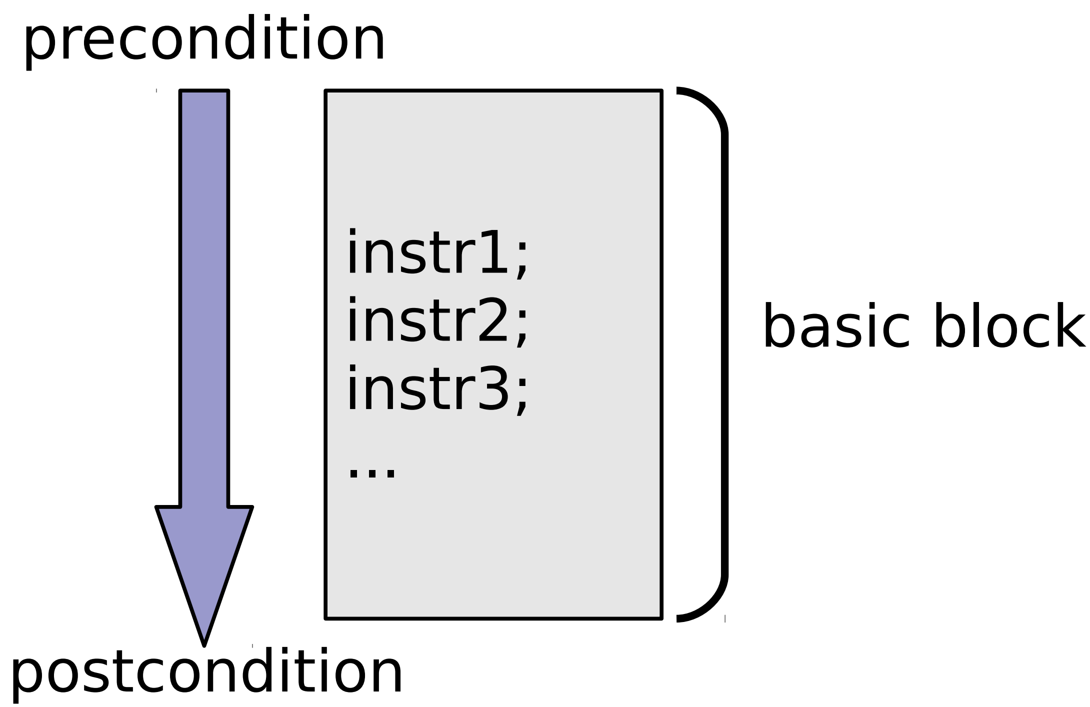
Why Automating Separation Logic Proofs is Easy

~~`llist(ls1, a1)`~~ `* ...`

Implies

`... *` ~~`llist(ls1, a1)`~~

A Simple Algorithm



1. Use annotations to expand formulas.
2. Symbolically execute block in “pre” formula.
3. Match “pre” and “post” formulas by crossing out equal parts.

Higher-Order Implications Are Easy, Too

Variable standing for thread #1's invariant over its private heap

~~`threadInv1`~~ * ~~`threadInv2`~~ * ...

Implies

~~`threadInv2`~~ * ... * ~~`threadInv1`~~

Higher-Order Implications Are Easy, Too

Assertion that every thread's invariant is satisfied

`~~allOk(invs)~~ * ...`

Implies

`... * ~~allOk(invs)~~`

Frame Rule for Free

$$\cancel{A} * \cancel{B} * C * \cancel{D} * E$$

Implies

$$\cancel{B} * \cancel{D} * \boxed{??} * \cancel{A}$$

Unification variable, which may be instantiated with any formula

Implemented Case Studies

	Library	LoC (total)	LoC (program)	LoC overhead of verification
	Malloc	267	71	2.8X
	ArrayList	771	272	1.8X
	ListSet	157	54	1.9X
	BinarySearchTree	355	127	1.8X
	AssociationList	200	85	1.4X
	Hashtable	374	90	3.2X
Higher-Order	Memoize	191	45	3.2X
Higher-Order	AppendCPS	155	56	1.8X
Higher-Order	Threads	225	67	2.4X

Last two reimplement examples from **Certified Assembly Programming** project [Shao et al.], where overhead is about **100X**.

The

Bedrock

Coq library

http://adam.chlipala.net/bedrock/