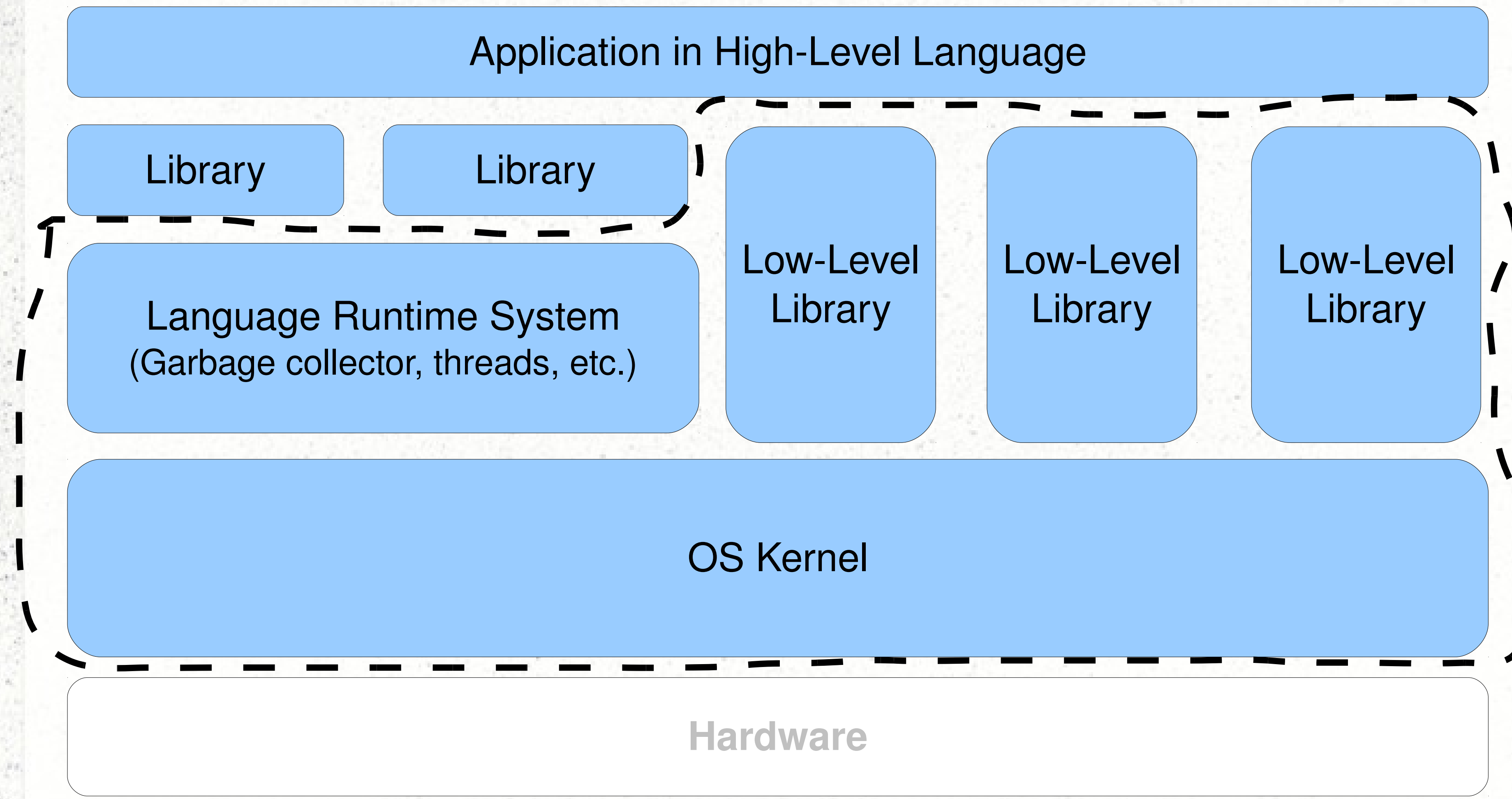
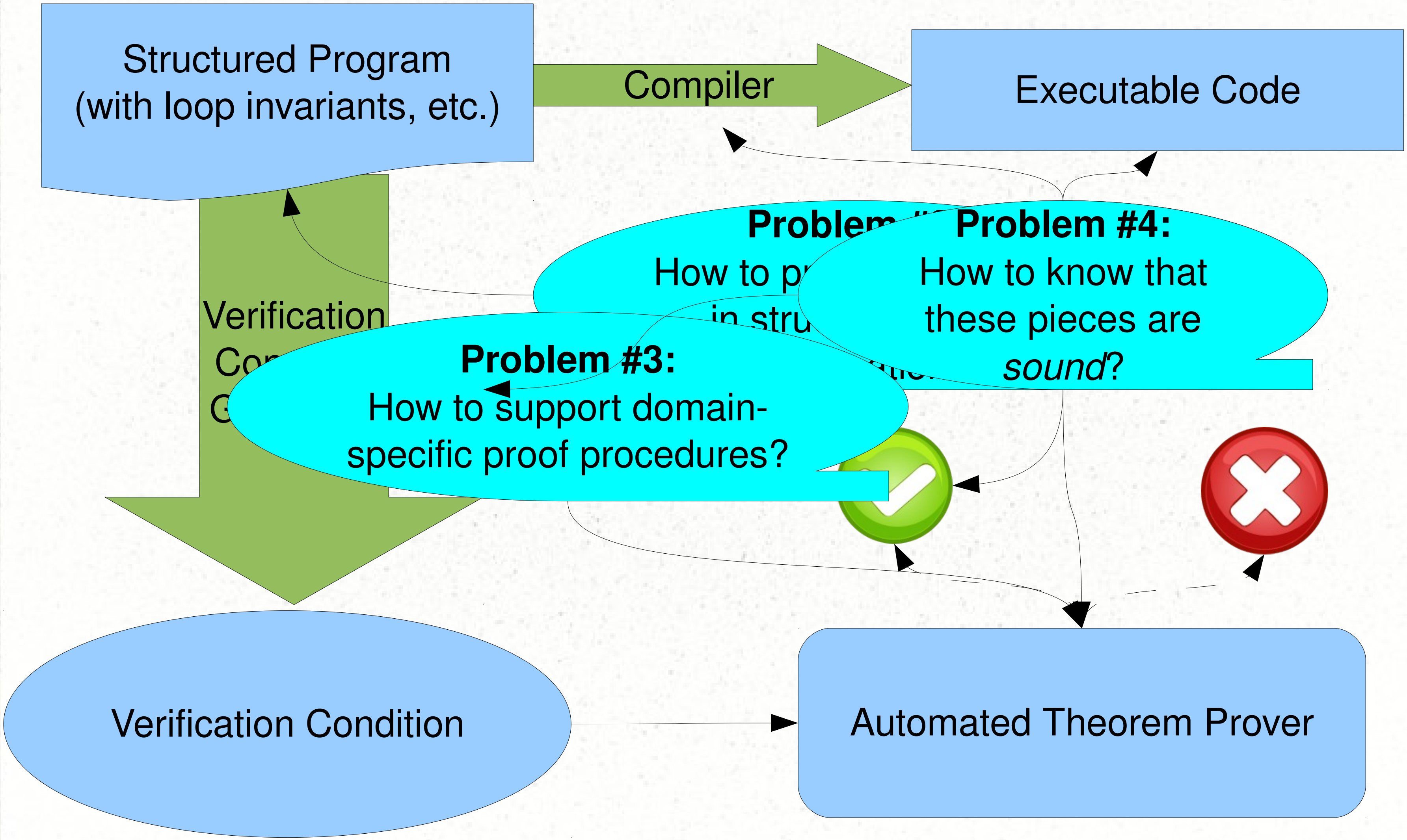
A Bottom-Up Approach to Safe Low-Level Programming

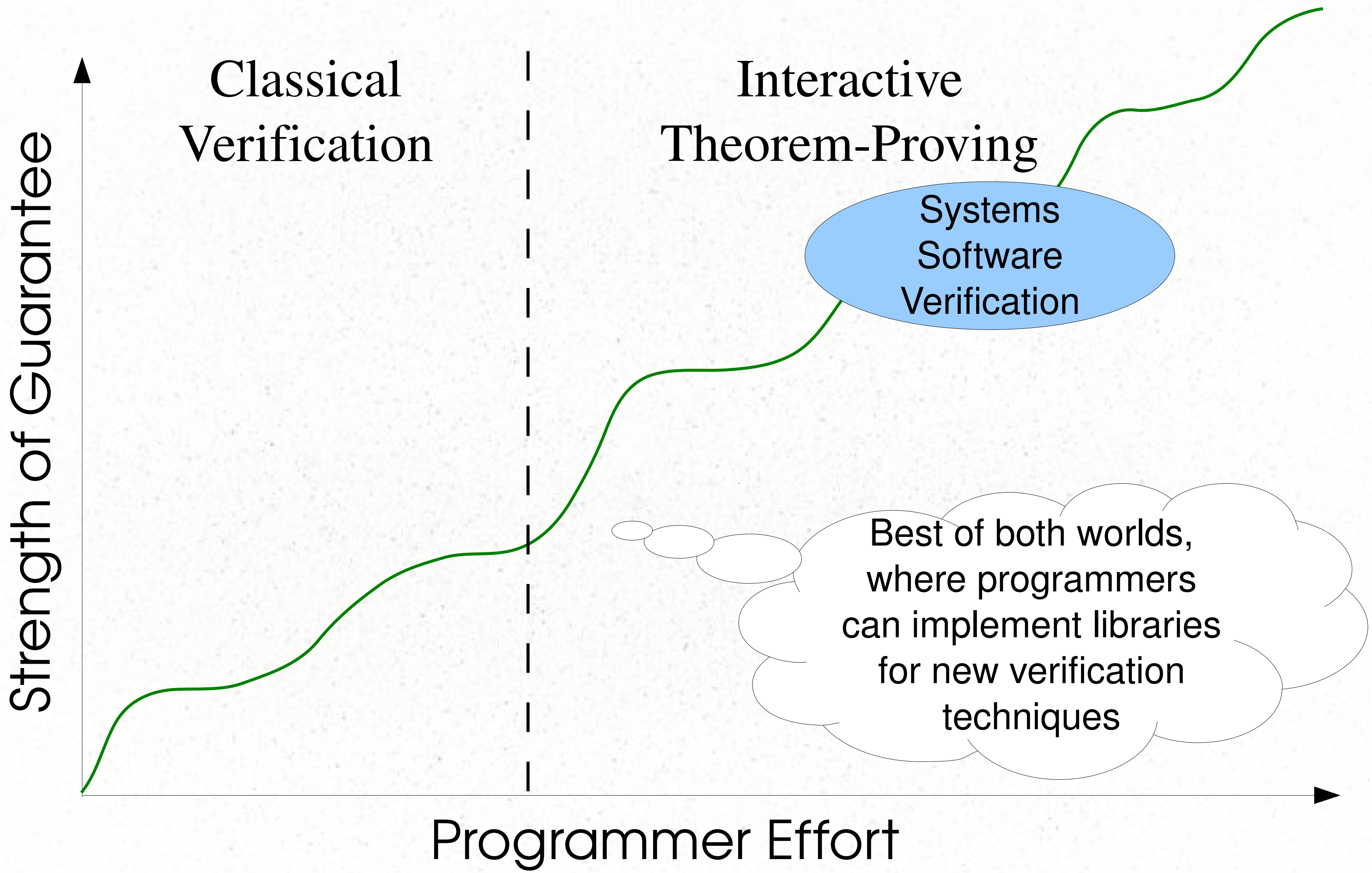
Adam Chlipala
Harvard University
MLPA 2010

Verified Software Stacks



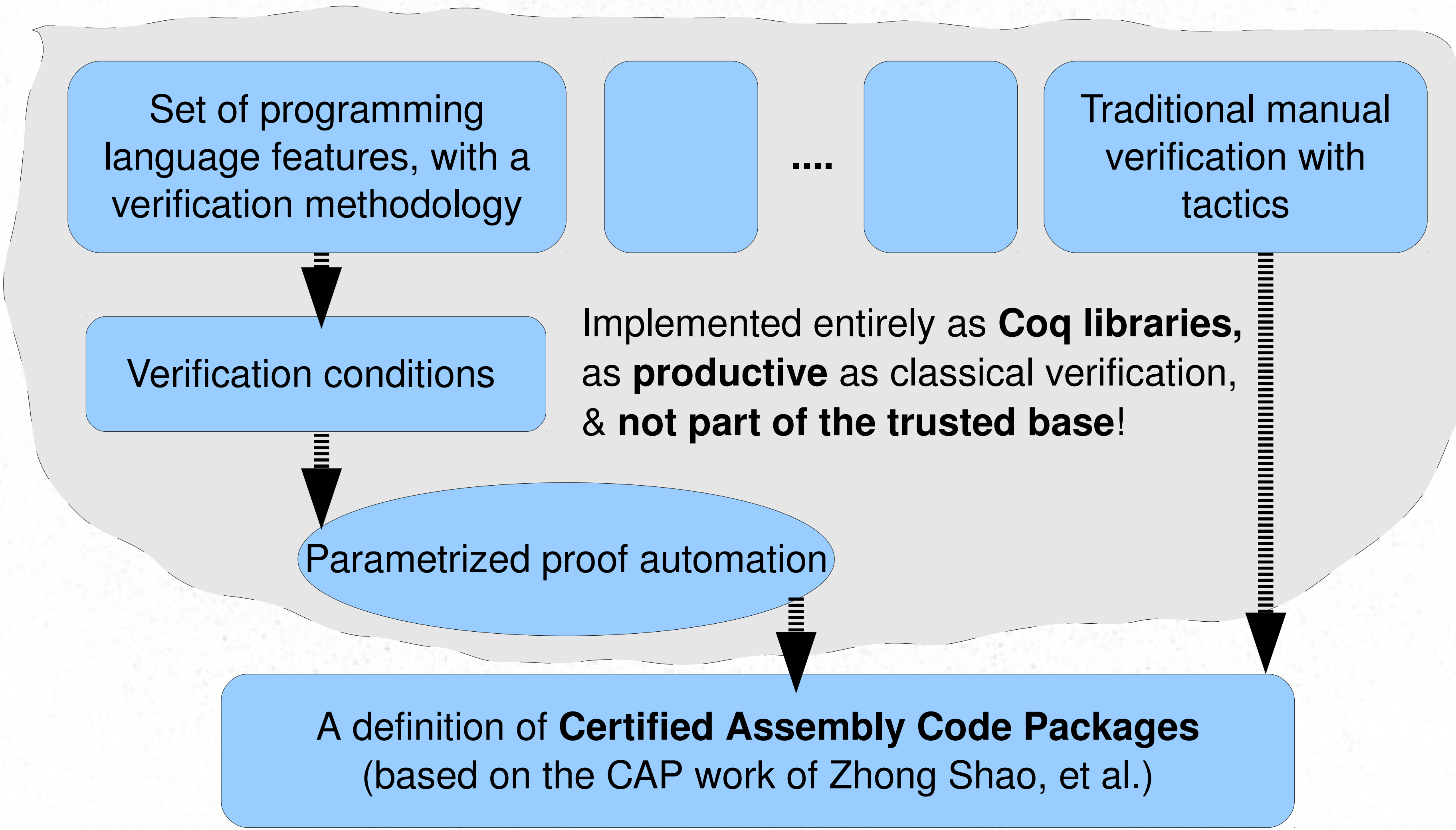
Classical Verification





Classical
Verification
Interactive
Theorem-Proving
Systems
Software
Verification
Best of both worlds,
where programmers
can implement libraries
for new verification
techniques
Strength of Guarantee
Programmer Effort

A Flexible Architecture



Code for Linked List Reverse

```
Definition linkedList := bfunction "rev" [st ~>
    Ex ls, ExX, ![ ^!{llist ls st#R0} * ![Var VO] ] st
    /\ st#Rret @@ (st' ~> [< st'#Rsp = st#Rsp >]
      /\ ![ ^!{llist (rev ls) st'#R0}
            * ![Var (VS VO)] ] st')] {
  R1 <- 0;;
  [st ~> ExX, Ex ls1, Ex ls2,
    ![ ^!{llist ls1 st#R1} * ^!{llist ls2 st#R0}
       * ![Var VO] ] st
    /\ st#Rret @@ (st' ~> [< st'#Rsp = st#Rsp >]
      /\ ![ ^!{llist (rev ls2 ++ ls1) st'#R0}
            * ![Var (VS VO)] ] st')]
  While (R0 != 0) {
    R2 <- $[R0+1];;
    $[R0+1] <- R1;;
    R1 <- R0;;
    R0 <- R2
  };;
  R0 <- R1;;
  JumpI Rret
}.
```

Precondition

Loop Invariant

Structured Control Flow

Proof of Correctness

```
Hint Extern 1 (_ ===> _) => progress unfold llist.
Hint Resolve lseg_nil_fwd lseg_cons_fwd
    llist_app_nil_fwd : Forward.
Hint Extern 1 (_ ===> lseg nil _ _) =>
    apply lseg_nil_bwd : Backward.
Hint Extern 1 (_ ===> lseg _ 0 0) =>
    apply lseg_nil_bwd : Backward.
Hint Extern 1 (_ ===> lseg ?ls ?h _) =>
    ensureUnif ls; ensureNotUnif h;
    apply lseg_cons_bwd : Backward.
```

Rules for quantifier instantiation

```
Theorem linkedListOk : package linkedList langs.
  structuredSep.
Qed.
```

Correctness proof, via domain-specific tactic

Outline

- The definition of **certified assembly packages**
- Verification frontends as libraries
  - Parsing
  - Code generation
  - Verification condition generation
- Full automation of correctness proofs
  - ...using triggers for quantifier instantiation
- Some case studies

Modular Verification

Relocatable code

```
f:
________________
________________
________________
________________
________________
________________
jmp __
L1:
________________
________________
________________
jmp __
L2:
________________
________________
________________
ret
```

Precondition for entry point

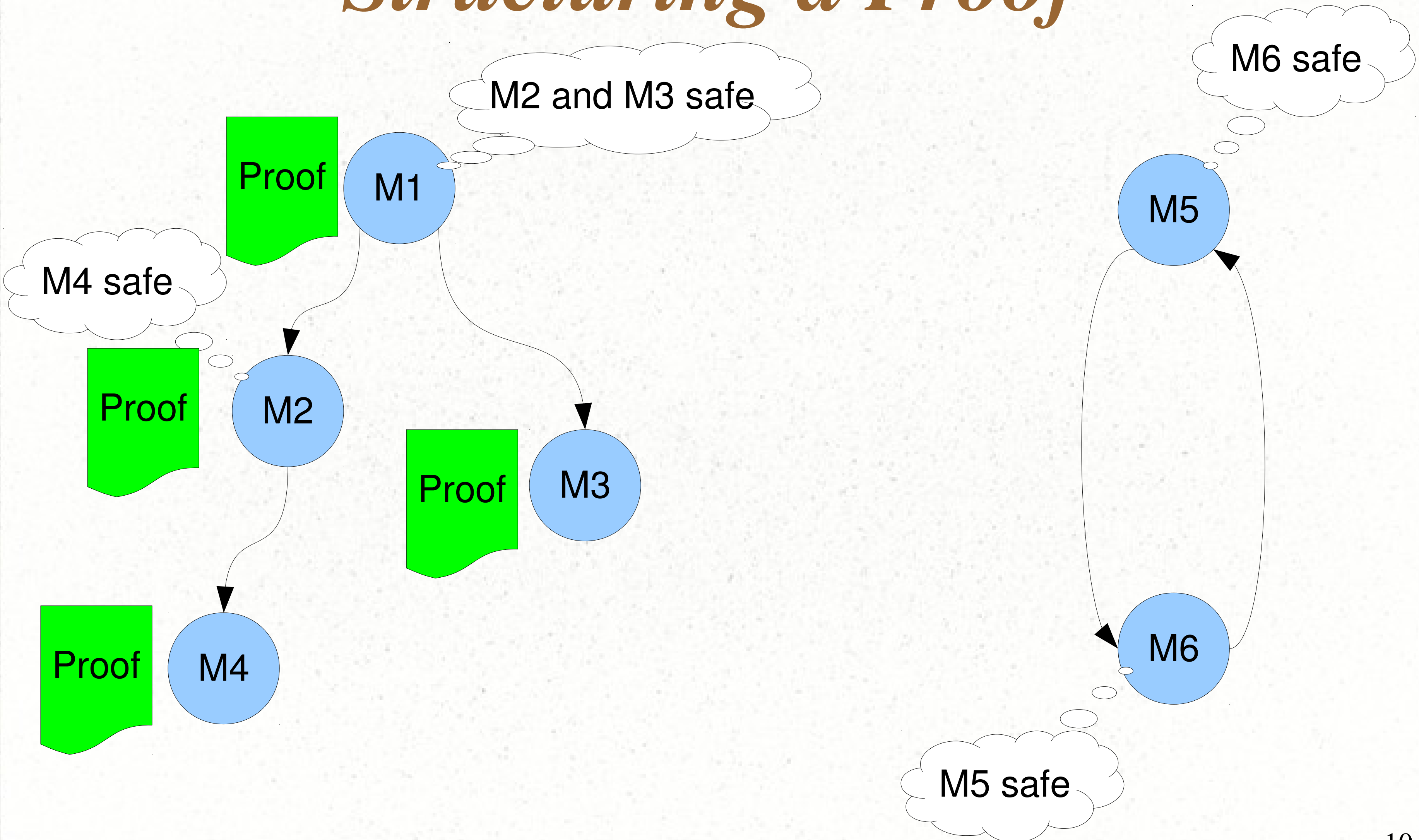
φ = ~~"L9 is safe"~~ $\wedge$ ....
= "Register R points to safe code" $\wedge$ ....

Operational Semantics
$S \rightarrow S'$

Correctness theorem (?)
$\forall S, S'.\ \varphi(S) \wedge S \rightarrow^* S' \Rightarrow \exists S''.\ S' \rightarrow S''$

```
g:
____________
call f
L9:__________
jmp __
```

Structuring a Proof



Certified Assembly Packages

Correctness proofs quantify over a
program specification Φ,
containing at least this module's preconditions.

Correctness Condition

$\forall S.\ \cancel{\varphi_1(S)}$

$\varphi_1(\Phi, S)$

Basic blocks with preconditions

$S \to S'$

L1: φ_1	**L2**: φ_2	....	**Ln**: φ_n
___	___		___
___	___		___
___	___		___
jmp *e*	jmp __		jmp __

$\varphi_2(S')$

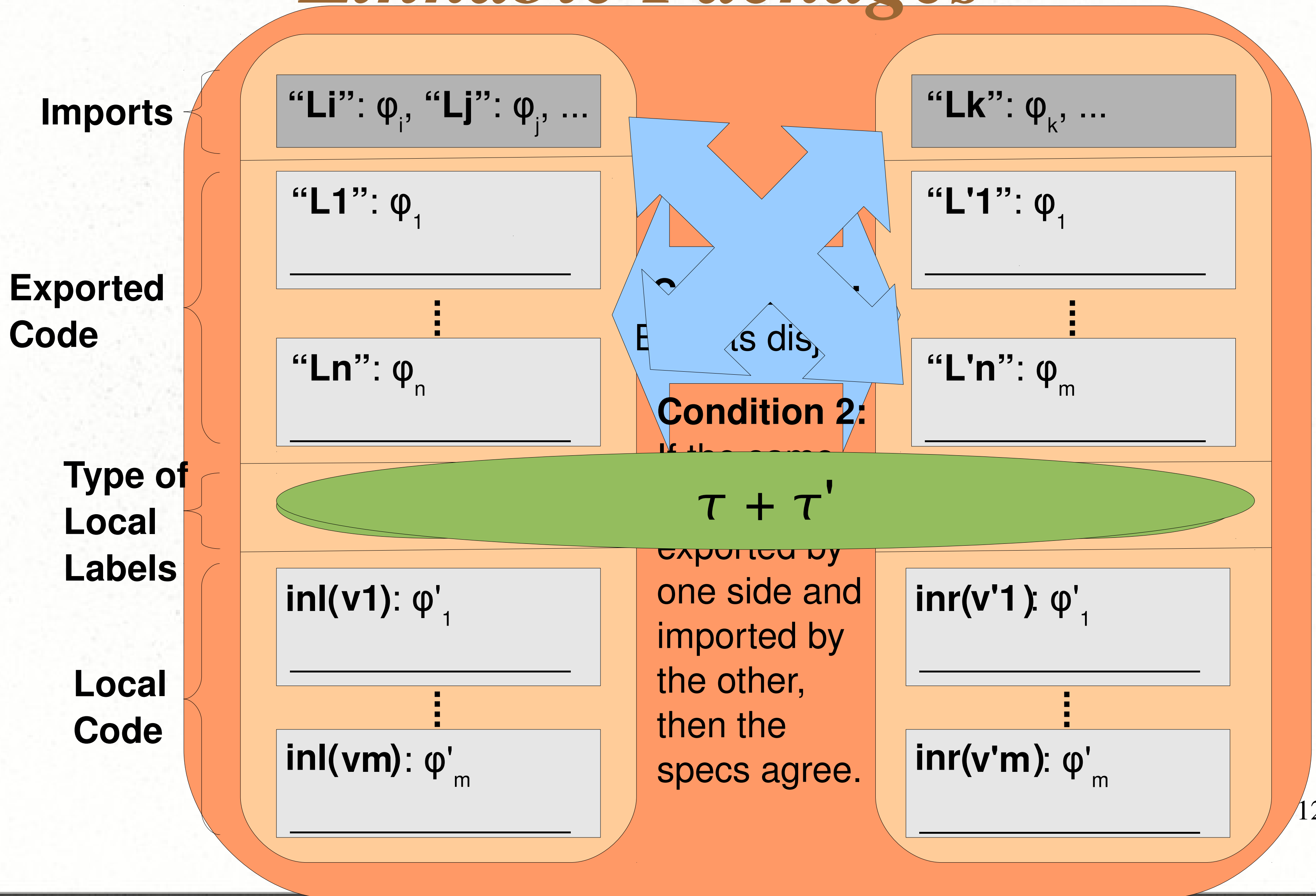
$\Phi(e(S')) = \varphi$

$\varphi(S')$

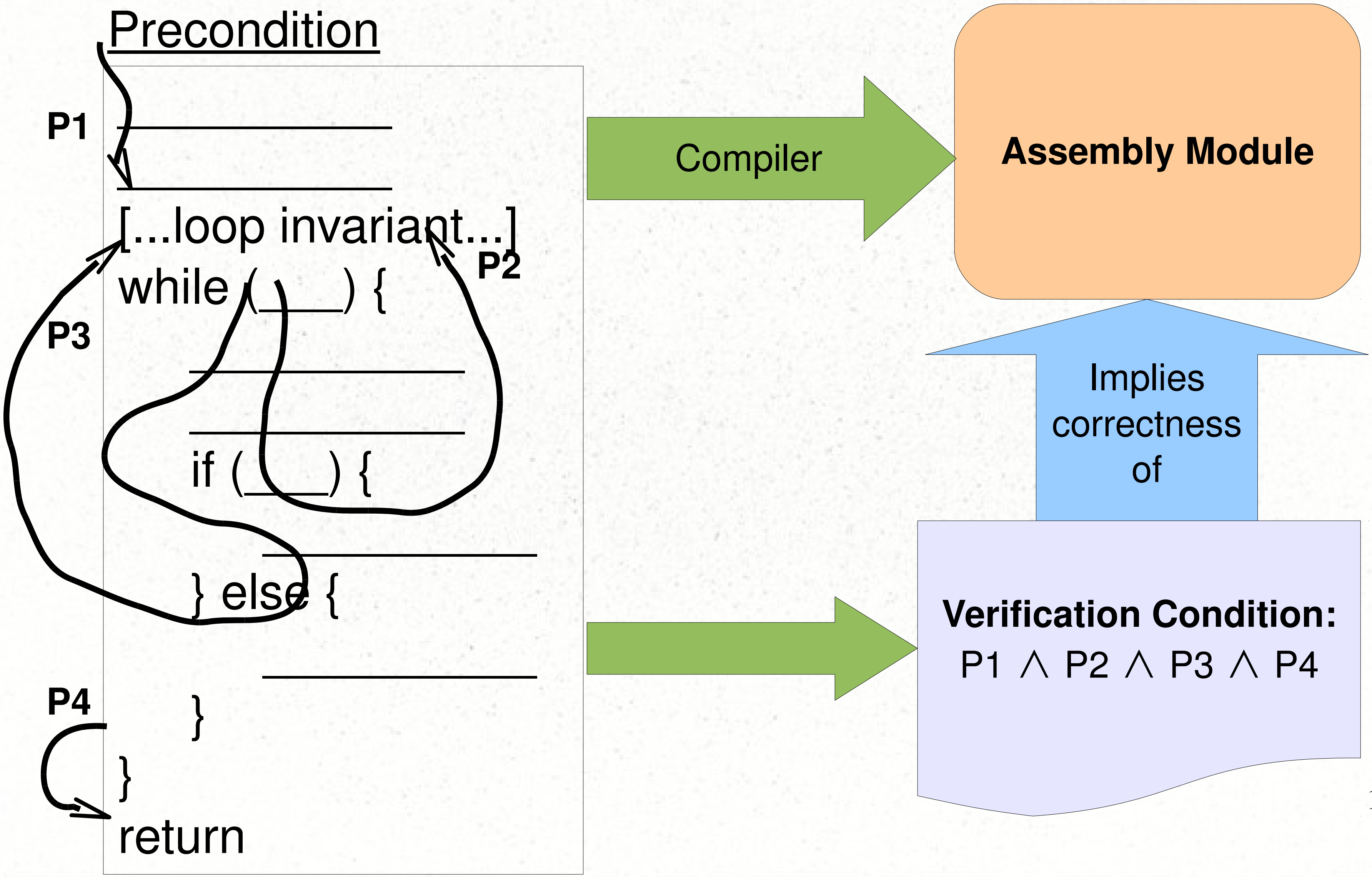
Final program correctness theorem:

1. Execution never gets stuck.
2. Whenever we enter a basic block, its precondition is satisfied.

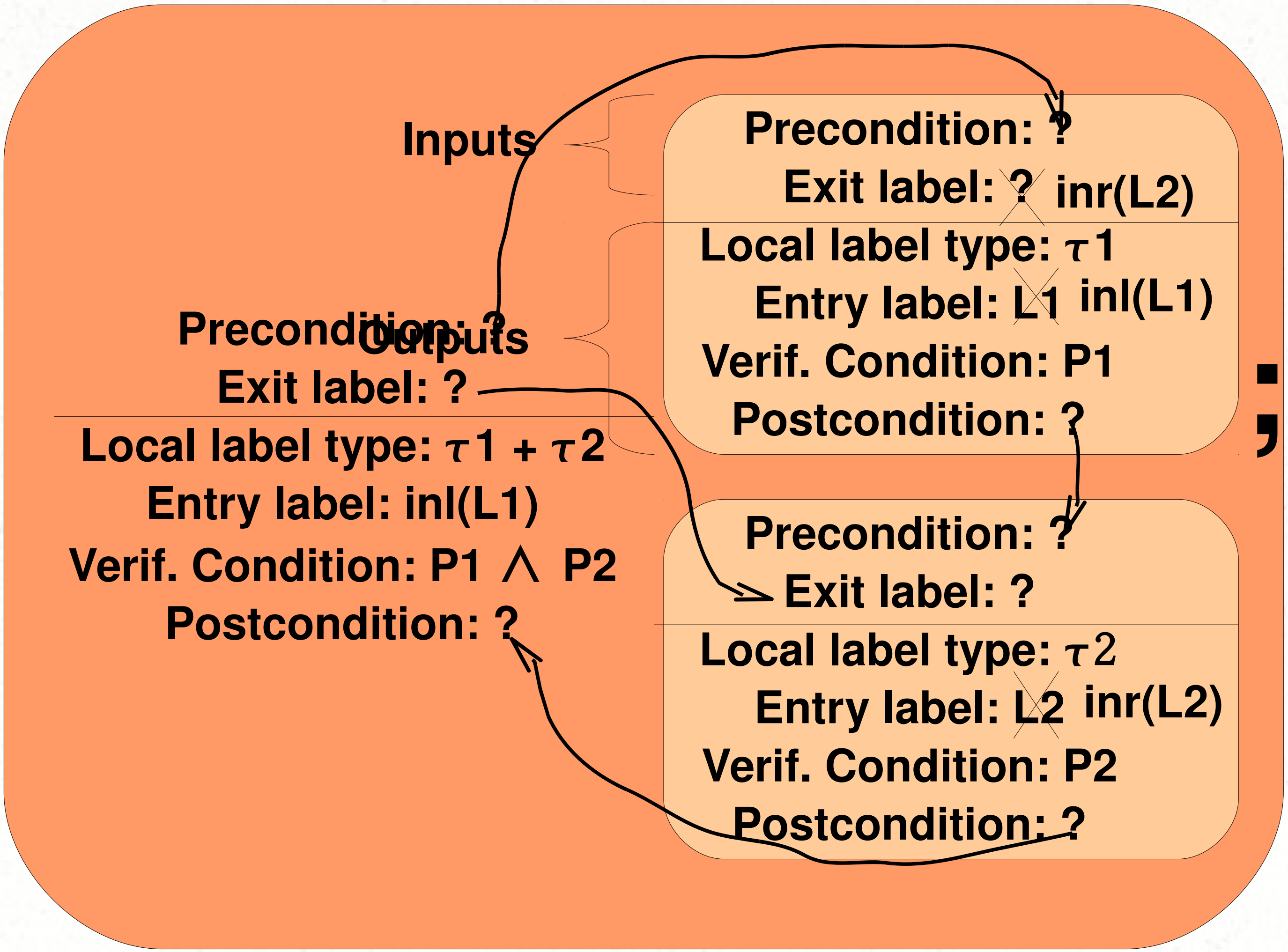
Linkable Packages



Modules from Structured Programs



Sequencing


Inputs
Outputs
Precondition: ?
Exit label: ?
Local label type: τ1 + τ2
Entry label: inl(L1)
Verif. Condition: P1 ∧ P2
Postcondition: ?
Precondition: ?
Exit label: ~~?~~ inr(L2)
Local label type: τ1
Entry label: ~~L1~~ inl(L1)
Verif. Condition: P1
Postcondition: ?
;
Precondition: ?
Exit label: ?
Local label type: τ2
Entry label: ~~L2~~ inr(L2)
Verif. Condition: P2
Postcondition: ?

Loops

Precondition: P
Exit label: L

Local label type: Test + Body(τ)
Entry label: inl(L1)
Verif. Condition: P1 $\wedge$ (P $\Rightarrow$ I) $\wedge$ (P' $\Rightarrow$ I)
Postcondition: I $\wedge$ $\neg$b

[I] while (b) {

Precondition: ~~?~~ I $\wedge$ b
Exit label: ~~?~~ Test

Local label type: τ
Entry label: ~~L1~~ Body(L1)
Verif. Condition: P1
Postcondition: ~~?~~ P'

}

L:

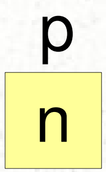
Concrete Syntax

```
[...]
While (R0 < 10) {
    If (R1 == R2) {
        R0 <- R1 * 10
    } else {
        R1 <- R1 + R2;;
        R0 <- R1
    }
}
```

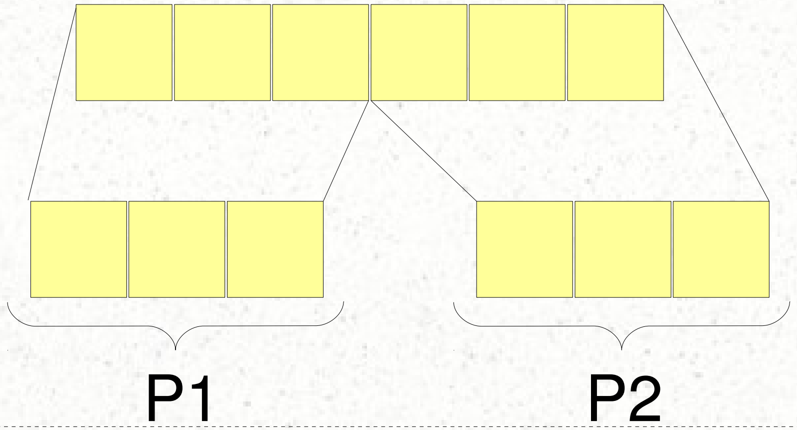
```
Coercion single : instr >-> scode.
Infix ";;" := seq
  (right associativity, at level 95) : SP_scope.
Notation "'If' c { b1 } 'else' { b2 }" :=
  (If_ (Code c) (Rval1 c) (Rval2 c) b1 b2)
  (no associativity, at level 95, c at level 0) : SP_scope.
Notation "[ p ] 'While' c { b }" :=
  (While_ p (Code c) (Rval1 c) (Rval2 c) b)
  (no associativity, at level 95, c at level 0) : SP_scope.
```

Separation Logic

p ==> n



P1 * P2



emp — The heap is empty.

[P] — The heap is empty and pure fact P is true.

allocated(p, 0) = emp
allocated(p, n) = (∃v, p ==> v) * allocated(p+1, n-1)

Abstract Predicates

Swap (inputs p and q):
Precondition: p ==> a * q ==> b
Postcondition: p ==> b * q ==> a

`malloc` (input sz, output p):
Precondition: mallocHeap
Postcondition: mallocHeap * [p <> 0] * allocated(p, sz)

Add an element to a linked list (inputs p and v, output p'):
Precondition: mallocHeap * llist(ls, p) * llist(ls', q)
Postcondition: mallocHeap * llist(v :: ls, p') * llist(ls', q)

The Frame Rule:
It is always legal to add the same formula to pre- and postconditions, using *.

Adapting to Assembly Code

Add an element to a linked list (inputs p and v, output p'):
Precondition: mallocHeap * llist(ls, p) * llist(ls', q)
Postcondition: mallocHeap * llist(v :: ls, p') * llist(ls', q)

Add an element to a linked list (inputs p, v, and R; output p'):

Return pointer

Precondition: mallocHeap * llist(ls, p)
* [R @ mallocHeap * llist(v :: ls, p')]

"It is safe to jump to R if this condition is satisfied."

Strengthened precondition: ∀P. mallocHeap * llist(ls, p) * P
* [R @ mallocHeap * llist(v :: ls, p') * P]

A Full Precondition for `malloc`

Quantifies over

Binders for machine state

Before and after versions
of machine registers

```
st ~> ExX, ![ ^!{mallocHeap st#R0}
             * ![Var VO] ] st
  /\ st#Rret @@ (st' ~>
     [< st'#R0 <> 0 /\ st'#Rsp = st#Rsp >]
     /\ ![ ^!{allocated st'#R0 (st#R1+2)}
           * ^!{mallocHeap st#R0}
           * ![Var (VS VO)] ] st')
```

Proof Obligations

Precondition

```
______________
______________
[...loop invariant...]
while (____) {
    ______________
    ______________
    if (____) {
        ______________
    } else {
        ______________
    }
}
return
```

P1
P2
P3
P4

Verification Condition:

P1 ∧ P2 ∧ P3 ∧ P4

An Extensible Prover

x : nat
l1, l2 : list nat
tl : ptr

Proof Context

In state *S*:
lseg (x :: l1) R0 tl * llist l2 tl

Pre-state

Mem[R0] < - y;
R1 <- Mem[R0+1]

Straightline Code

In state *S*[Mem[R0] := y, R1 := Mem[R0+1]]:
R0 ==> y * R0+1 ==> R1 * llist (l1 ++ l2) R1

Post-state

Normalize State Accesses

x : nat
l1, l2 : list nat
tl : ptr

Proof Context

In state *S*:
lseg (x :: l1) <u>S.R0</u> tl * llist l2 tl

Pre-state

Mem[R0] < - y;
R1 <- Mem[R0+1]

Straightline Code

In state *S*[Mem[R0] := y, R1 := Mem[R0+1]]:
<u>S.R0</u> ==> y * <u>S.R0+1</u> ==> <u>S.Mem[R0+1]</u>
* llist (l1 ++ l2) <u>S.Mem[R0+1]</u>

Post-state

Unfold Predicates in Pre-State

x : nat
l1, l2 : list nat
tl : ptr

p : ptr
p <> 0

Proof Context

In state *S*:
lseg (x :: l1) S.R0 tl * llist l2 tl
∃p. [p <> 0] * S.R0 ==> x * S.R0+1 ==> p * lseg l1 p tl

Pre-state

Mem[R0] < - y;
R1 <- Mem[R0+1]

Straightline Code

In state *S*[Mem[R0] := y, R1 := Mem[R0+1]]:
S.R0 ==> y * S.R0+1 ==> S.Mem[R0+1]
* llist (l1 ++ l2) S.Mem[R0+1]

Post-state

Simplify Memory Reads

x : nat
l1, l2 : list nat
tl : ptr

p : ptr
p <> 0

Proof Context

In state *S*:
S.R0 ==> x * S.R0+1 ==> p * lseg l1 p tl * llist l2 tl

Pre-state

Mem[R0] < - y;
R1 <- Mem[R0+1]

Straightline Code

In state *S*[Mem[R0] := y, R1 := Mem[R0+1]]:
S.R0 ==> y * S.R0+1 ==> ~~S.Mem[R0+1]~~ p
* llist (l1 ++ l2) S.Mem[R0+1]

Post-state

Execute Memory Writes

x : nat
l1, l2 : list nat
tl : ptr

p : ptr
p <> 0

Proof Context

In state *S*:
S.R0 ==> ~~x~~ * S.R0+1 ==> p * lseg l1 p tl * llist l2 tl
y

Pre-state

Mem[R0] < - y;
R1 <- Mem[R0+1]

Straightline Code

In state *S*[Mem[R0] := y, R1 := Mem[R0+1]]:
S.R0 ==> y * S.R0+1 ==> p * llist (l1 ++ l2) p

Post-state

Unfold Predicates in Post-State

x : nat
l1, l2 : list nat
tl : ptr
p : ptr
p <> 0
X : ptr

Unification variable

Proof Context

In state *S*[Mem[R0] := y, R1 := Mem[R0+1]]:
S.R0 ==> y * S.R0+1 ==> p * lseg l1 p tl * llist l2 tl

Pre-state

Mem[R0] < - y;
R1 <- Mem[R0+1]

Straightline Code

In state *S*[Mem[R0] := y, R1 := Mem[R0+1]]:
S.R0 ==> y * S.R0+1 ==> p * llist (l1 ++ l2) p

<u>∃p'. lseg l1 p p' * llist l2 p'</u>

Post-state

Cancel Equal Terms and Finish

x : nat
l1, l2 : list nat
tl : ptr

p : ptr
p <> 0
X : ptr

Proof Context

In state *S*[Mem[R0] := y, R1 := Mem[R0+1]]:
~~S.R0 ==> y~~ * ~~S.R0+1 ==> p~~ * ~~lseg l1 p tl~~ * ~~llist l2 tl~~

Pre-state

Mem[R0] < - y;
R1 <- Mem[R0+1]

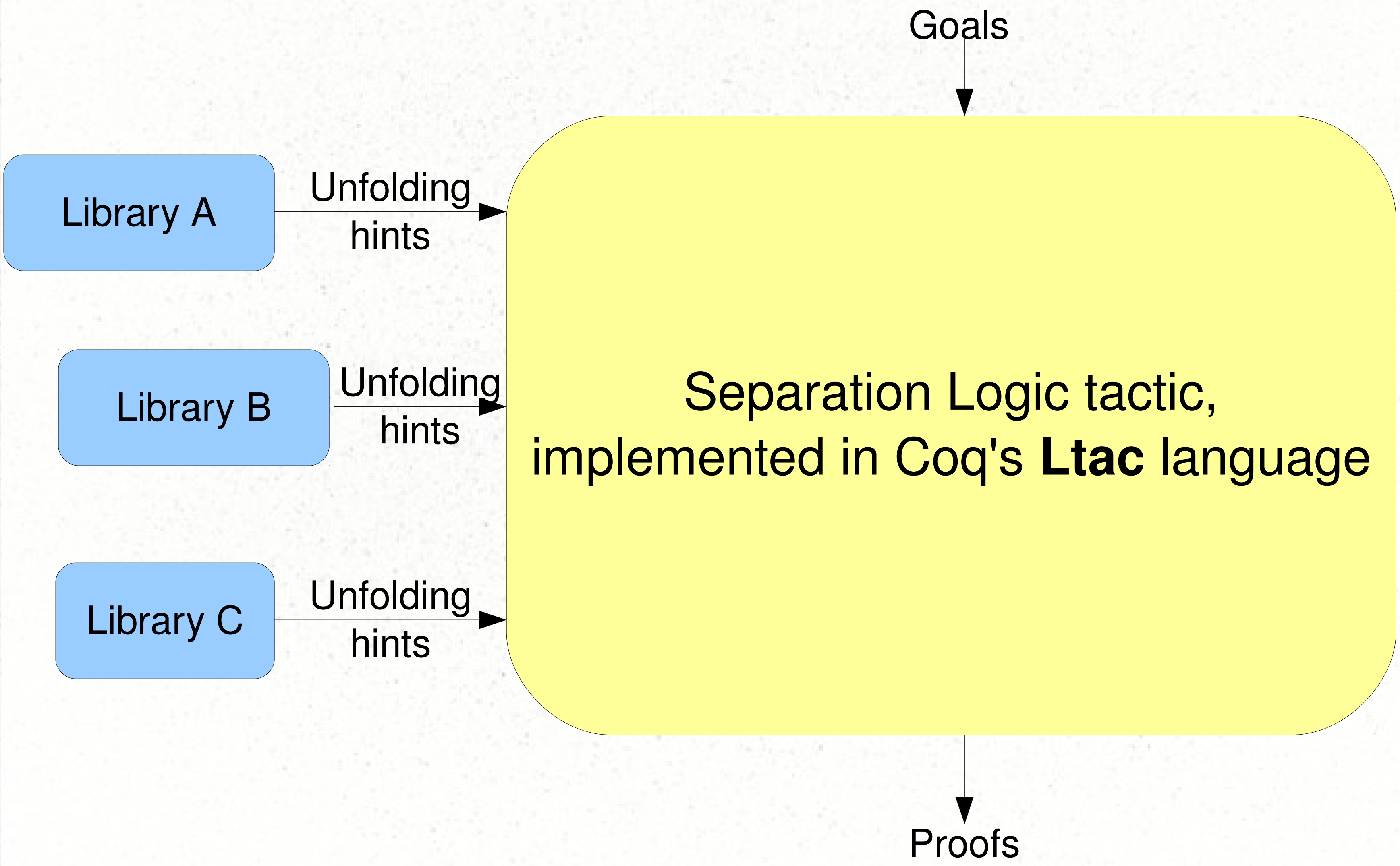
Straightline Code

In state *S*[Mem[R0] := y, R1 := Mem[R0+1]]:
~~S.R0 ==> y~~ * ~~S.R0+1 ==> p~~ * ~~lseg l1 p X~~ * ~~llist l2 X~~

Post-state

X = tl

Unfolding Hints



Proving an Unfolding Lemma

```
Theorem freeList_nonempty_fwd : forall fl flh flt,
  flh <> flt
  -> freeList fl flh flt
  ===> Ex p', Ex sz, Ex fl', [< fl = flh :: fl' >]
    * flh ==> p' * (flh+1) ==> sz
    * !{allocated (flh+2) sz} * !{freeList fl' p' flt}.
  destruct fl; sepLemma.
Qed.
```

Proof script

```
Hint Resolve freeList_nonempty_fwd : Forward.
```

Registering this lemma to use in unfolding hypotheses

A Lemma with an Inductive Proof

```
Lemma freeList_middle : forall fl2 flt p sz fl1 flh,
  flt <> 0
  -> !{freeList fl1 flh flt}
  * flt ==> p * (flt+1) ==> sz
  * !{allocated (flt+2) sz} * !{freeList fl2 p 0}
  ===> freeList (fl1 ++ flt :: fl2) flh 0.
  induction fl1; sepLemma.
Qed.
```

More Complicated Hints

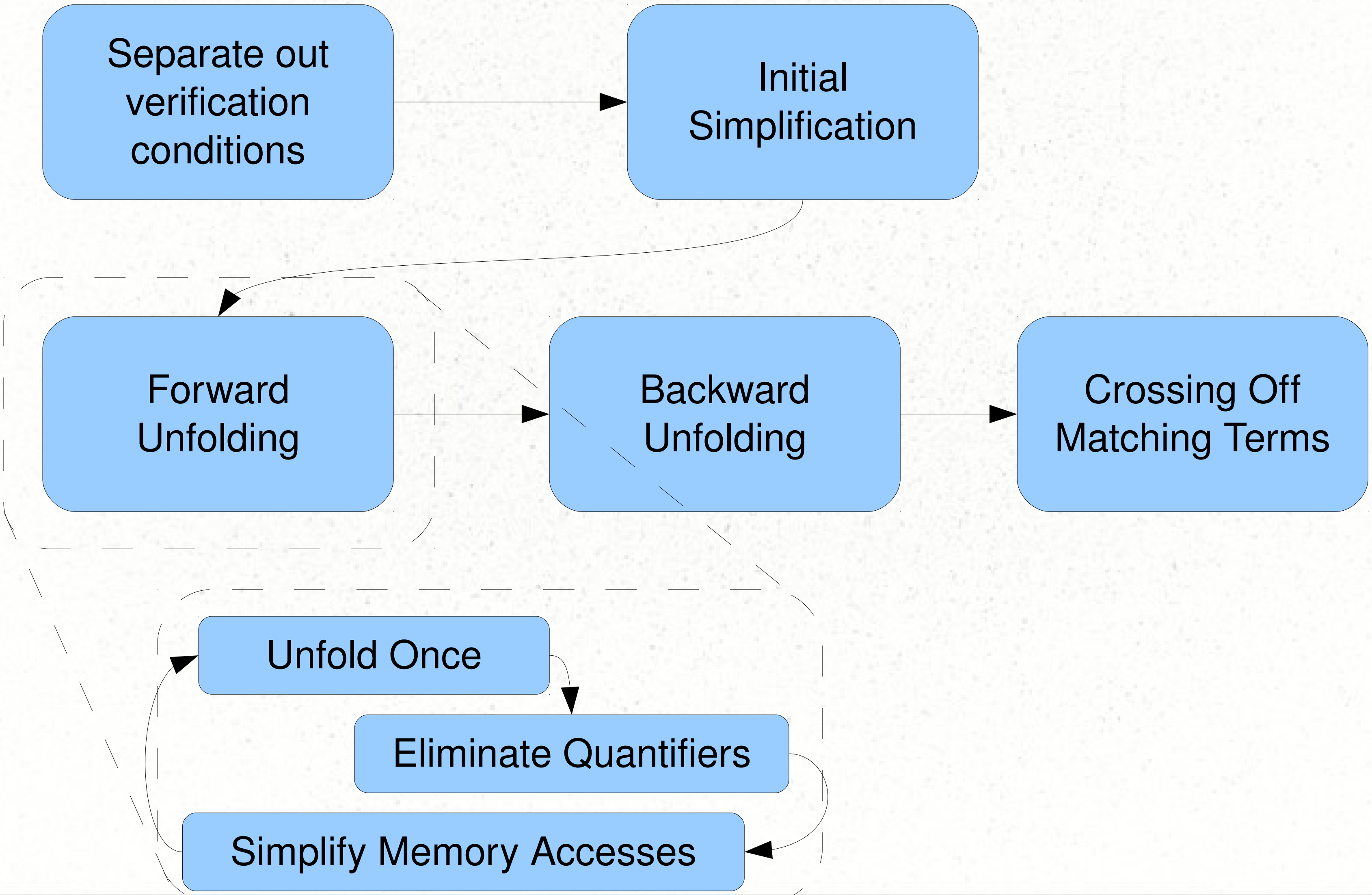
```
Hint Extern 1 (_ ===> freeList _ ?p 0) =>
  match goal with
    | [ _ : context[ptsTo ?p' 0] |- _ ] =>
      ensureEq p' p; apply freeList_create
  end : Backward.
```

Case Studies

Library	Total Lines	Lines of Proof Script
malloc/free	322	89
Linked list lemmas	128	22
Linked list free and reverse (copying)	109	34
Linked list reverse (in-place)	33	6
Linked list append (in-place, in continuation-passing style, with explicit closures)*	135	17

* Based on the most involved example from the XCAP paper by Ni and Shao, which took about 1500 lines of proof

Coding and Debugging Proofs



Conclusion

Classical Verification

Structured programs
Automated proofs

Bedrock

Interactive Theorem-Proving

Small proof checker
Flexibility
Higher-order reasoning

Hint Databases in Coq

```
Theorem plus_cong : forall n m n' m',
    n = n'
    → m = m'
    → n + m = n' + m'.
    (* ...proof... *)
Qed.

Theorem plus_comm : forall n m,
    n + m = m + n.
    (* ...proof... *)
Qed.

Hint Resolve plus_cong plus_comm : Arith.

Goal forall i j k, (i + j) + k = (j + i) + k.
   auto with Arith.
Qed.
```