Effective Interactive Proofs for Higher-Order Imperative Programs

Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, Ryan Wisnesky
Harvard University
ICFP 2009

Goal: Practical Verification of Higher-Order Imperative Programs

- We want to be able to prove **full correctness** of **mostly-functional programs** that use **imperativity for efficiency**.
  - *Challenge problems*: compilers, database servers, Internet servers, …

- *Last year's ICFP*: proof-of-concept system that you wouldn't really want to use
- *This year*: new system with effective ***proof automation***

Imperative Finite Map ADT

```
type key
type value
type map

new : unit -> map

insert : map -> key -> value -> unit

lookup : map -> key -> option value
```

With a Computation Monad

```
type key
type value
type map

new : Cmd map

insert : map -> key -> value -> Cmd unit

lookup : map -> key -> Cmd (option value)
```

With Strong Specs

```
type key
type value
type map

rep : list (key * value) -> map -> bool

new : Cmd emp (fun m : map => rep [] m)

insert : m : map -> k : key -> v : value
   -> [ls : list (key * value)]
   -> Cmd (rep ls m)
          (fun _ => rep ((k, v) :: ls) m)

lookup : m : map -> k : key
   -> [ls : list (key * value)]
   -> Cmd (rep ls m) (fun vo => match vo with
             | None => k \notin ls
             | Some v => (k, v) \in ls
          end)
```

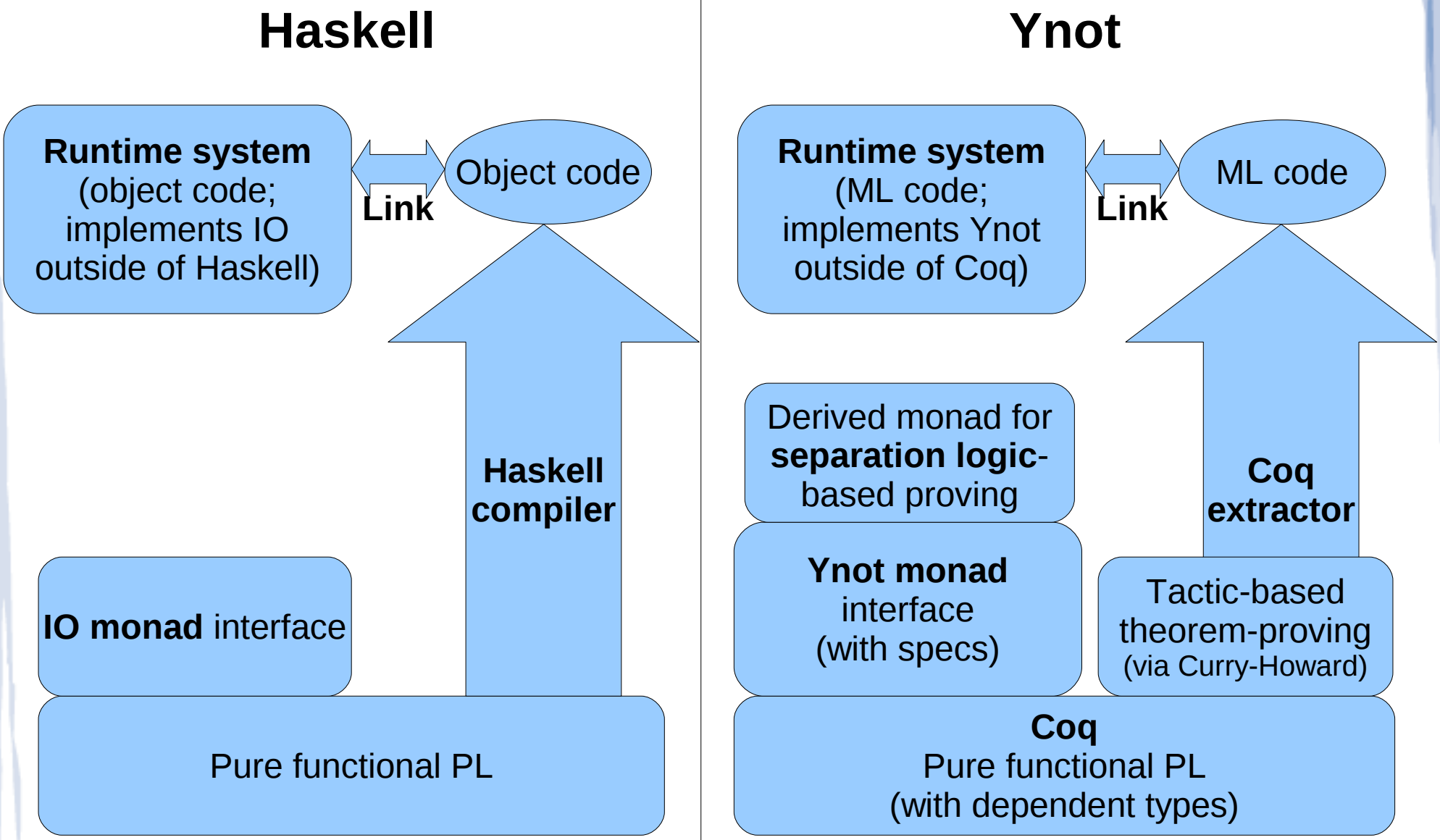
Representation invariant

Monad parameterized by *precondition and postcondition*

Dependent typing

Spec variables (erased by compiler)

Ynot (from ICFP'08)



The Burden of Proof

(proof for one method of splay trees from ICFP'08 Ynot)

```
rename l into L. rename l0 into R. rename l1 into root.
destruct H as [Ld [Llt [Rd [Rrt [rootT rest]]]]].
destruct rest as [[roottnempty bstroot] precond].
sets precond_dest precond.
destruct precond_dest as [LRh [rooth [splitsi [lrrest modelsroot]]]].
unlift_in lrrest.
destruct lrrest as [Lh [Rh [splitsLRh [lrest rrest]]]].
destruct lrest as [Lnnull [Lnode [Lddataeq modelsl]]].
destruct rrest as [Rnnull [Rnode [Rddataeq modelsr]]].
destruct modelsl as [Lptsh [L_Lh [splitsLh [Lpts modelsL_L]]]].
destruct modelsr as [Rptsh [R_Rh [splitsRh [Rpts modelsR_R]]]].
destruct rootT as [ | rootd root_Lnode root_Rnode].
  elim roottnempty...
clear roottnempty.
sets modelsroot_dest modelsroot.
simpl in modelsroot_dest. unlift_in modelsroot_dest.
destruct modelsroot_dest as [rootnnull [rootnode [rootdataeq rest]]].
destruct rest as [rootptsh [root_LRh [splitsrooth [rootpts modelsroot_LR]]]].
destruct modelsroot_LR as [root_Lh [root_Rh [splitsroot_LRh [modelsroot_L
modelsroot_R]]]].
nextvc.
splits_rewrite_in splitsrooth splitsi. splits_join_in H (1 :: 0 :: 1 :: nil).
exists rootptsh. exists (union (LRh :: root_LRh :: nil) pf0). split.
    apply* splits_commute. split.
    eexact rootpts.
    unfold nopre...
(* ok, so the read was valid and gave us back rootnode *)
nextvc. (* a: *)
  (* yay! we found v *)
  clear H. rename x into eqvrootd.
  nextvc.
    (* yes, L points to Lnode *)
    splits_rewrite_in splitsLRh splitsi.
    splits_rewrite_in splitsLh H.
    splits_join_in H0 (0 :: 1 :: 1 :: 1 :: nil).
    exists Lptsh. exists (union (L_Lh :: Rh :: rooth :: nil) pf0). split.
      apply* splits_commute. split.
      eexact Lpts.
      unfold nopre...
    (* and R points to Rnode *)
    splits_rewrite_in splitsLRh splitsi.
    splits_rewrite_in splitsRh H.
    splits_join_in H0 (1 :: 0 :: 1 :: 1 :: nil).
    exists Rptsh. exists (union (Lh :: R_Rh :: rooth :: nil) pf0). split.
      apply* splits_commute. split.
      eexact Rpts.
      unfold nopre...
    (* so we can get on with it *)
    nextvc. (* write to left *)
      splits_rewrite_in splitsLRh splitsi.
      splits_rewrite_in splitsLh H. splits_rewrite_in splitsRh H0.
      clear H H0. rename H1 into splitsipts.
      splits_join_in splitsipts (0 :: 1 :: 1 :: 1 :: 1 :: nil).
      exists Lptsh. split.
        eauto.
      exists (union (L_Lh :: Rptsh :: R_Rh :: rooth :: nil) pf0). split.
        apply* splits_commute.
      intros j Lptsh' splitsj Lpts'.
      splits_flatten_in' splitsj. clear H pf0.
    nextvc. (* write to right *)
      splits_join_in splitsj (1 :: 1 :: 0 :: 1 :: 1 :: nil).
      exists Rptsh. split.
        eauto.
      exists (union (Lptsh' :: L_Lh :: R_Rh :: rooth :: nil) pf0). split.
        apply* splits_commute.
      intros k Rptsh' splitsk Rpts'.
      splits_flatten_in' splitsk. clear H pf0.
(* and now, onto the return *)
nextvc.
  exists root. split...
(* now we get the preconditions again, and reduce them to our old ones *)
intros Llt' Rrt' rootT' Ld' Rd' precond'.
poses (splay_down_precond_uniq precond' precond). subst. clear precond'.
exists root_Lnode. exists root_Rnode. exists (data rootnode). split.
  (* show that the shape of memory is ok *)
  splits_rewrite_in splitsrooth splitsk.
  splits_rewrite_in splitsroot_LRh H.
  splits_join_in H0 (0 :: 0 :: 0 :: 1 :: 0 :: 0 :: 1 :: nil).
  splits_join_in H1 (1 :: 1 :: 0 :: 0 :: 0 :: 0 :: 0 :: nil).
  splits_join_in H2 (0 :: 0 :: 1 :: 0 :: 1 :: nil).
  splits_join_in H3 (1 :: 0 :: 1 :: 0  :: nil).
  splits_join_in H4 (1 :: 1 :: 0 :: nil).
  clear H H0 H1 H2 H3 H4. rename H5 into splitsk'.
  exists (union
            (union
               (union (L_Lh :: root_Lh :: nil) pf2 :: Lptsh' :: nil)
               pf3
             :: union
                  (union (R_Rh :: root_Rh :: nil) pf0 :: Rptsh' :: nil)
                  pf1 :: nil) pf4).
  exists rootptsh. split... split.
    exists (union (union (L_Lh :: root_Lh :: nil) pf2 :: Lptsh' :: nil) pf3).
    exists (union (union (R_Rh :: root_Rh :: nil) pf0 :: Rptsh' :: nil) pf1). split.
      exists pf4... unlift. split; split...
        exists (mkNode (data Lnode) (left_node Lnode) (left_node rootnode)). split...
          exists Lptsh'. exists (union (L_Lh :: root_Lh :: nil) pf2). split.
            apply* splits_commute. exists pf3... split...
              exists L_Lh. exists root_Lh. split.
                exists pf2...
                split...
        exists (mkNode (data Rnode) (right_node rootnode) (right_node Rnode)). split...
          exists Rptsh'. exists (union (R_Rh :: root_Rh :: nil) pf0). split.
            apply* splits_commute. exists pf1... split...
              exists root_Rh. exists R_Rh. split.
                apply* splits_commute.  exists pf0...
                split...
  split...
    exists rootnode. unlift. split...
(* show that the other properties hold *)
split.
  (* the contents of the tree are preserved *)
  simpl. do 3 rewrite app_ass. f_equal.
  (* L is smaller and R is bigger *)
```

Continued...

(~1300 lines total)

Demo

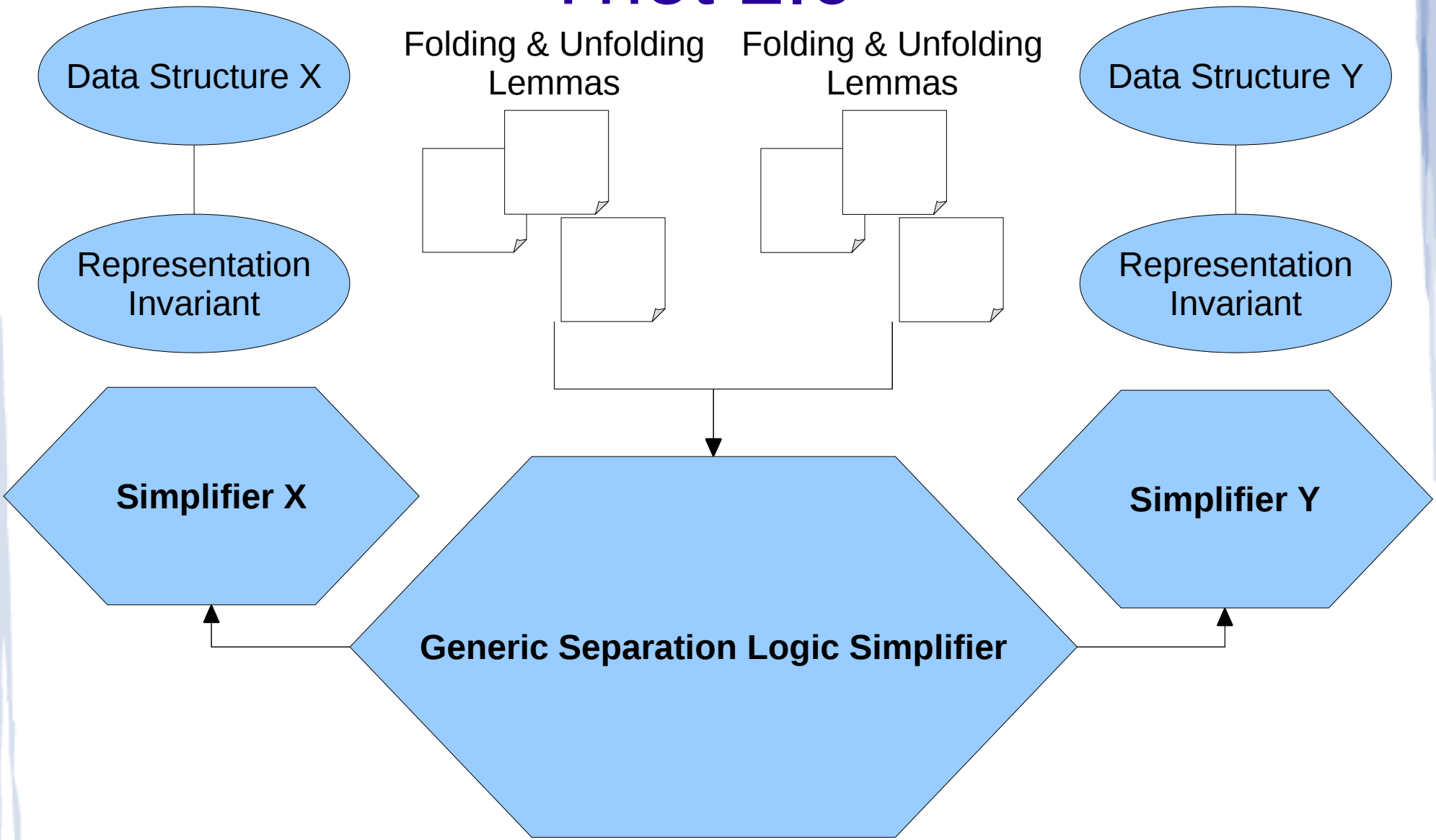
Conclusion

- Focusing on automation lets us write verified programs that seemed impossible before.
- At least an order of magnitude decrease in size of proofs and effort needed to build them.
- Paper describes implementations of imperative stack, queue, association list, hash table, binary search tree, binomial tree, and packrat parser.

Available for download from:
`http://ynot.cs.harvard.edu/`

Backup Slides

Ynot 2.0


Data Structure X
Folding & Unfolding Lemmas
Folding & Unfolding Lemmas
Data Structure Y
Representation Invariant
Representation Invariant
Simplifier X
Generic Separation Logic Simplifier
Simplifier Y