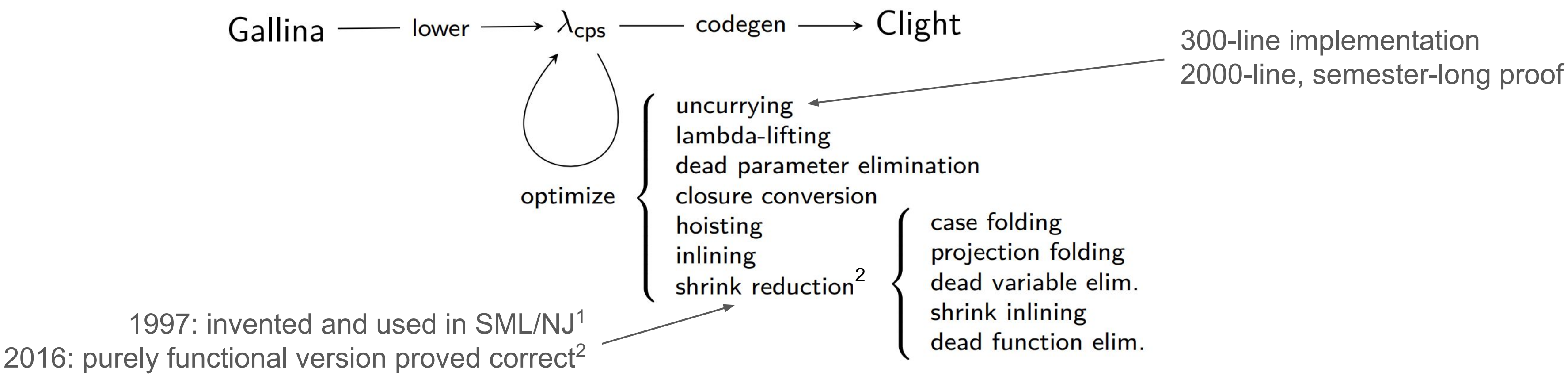
Deriving Efficient Program Transformations from Rewrite Rules

John M. Li
Andrew W. Appel

Princeton University
August 2021

Motivation

- Compilers are hard to get right[1]
- Mechanized proof is effective[1], but proofs often tedious
- Example: CertiCoq's backend



[1]Yang, Chen, Eide, and Regehr, 2011
[2]Appel and Jim, 1997
[3]Savary Bélanger and Appel, 2016

Automation to the rescue?

- Proofs about program transformations seem to have similar structure:

e and *e'* are semantically equivalent

↑ Proofs about object language semantics — Lacey and De Moor, 2001; Cobalt, Rhodium (Lerner et al., 2005); PTRANS (Mansky, 2014)

e $\longrightarrow^*$ e' where ($\longrightarrow$) is some rewriting relation

↑ Proofs about data structures and invariants — This work

transform e = e'

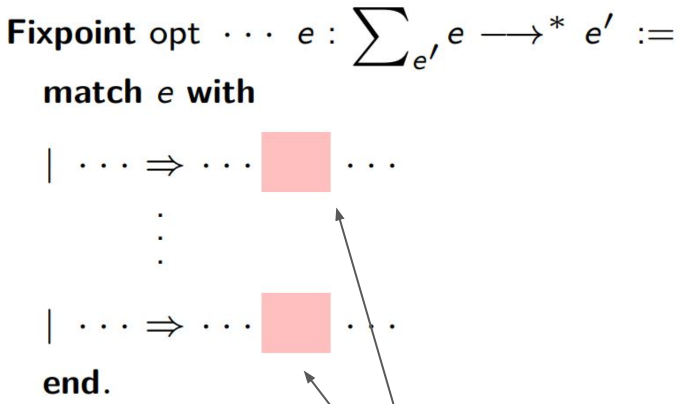
Our tool

Rewriting relation (⟶)

“Use lookup tables to store bindings in scope”
“Use a counter variable to generate fresh names”

Specified how?

Types ensure correctness



Holes to fill by hand

Specifying helper data structures: an example

b ::= true | false

e ::= x | **let** x = b **in** e | **if** x **then** e **else** e

let x = true **in** C[**if** x **then** e1 **else** e2] $\longrightarrow$ **let** x = true **in** C[e1] (case folding)

$$\frac{x \notin FV(e)}{\textbf{let } x = b \textbf{ in } e \longrightarrow e} \quad \text{(dead variable elimination)}$$

Implementing case folding

let x = true **in** C[**if** x **then** e1 **else** e2] ⟶ **let** x = true **in** C[e1] (case folding)

Pass around an extra parameter *env* mapping variables in scope to literals:

let x = b **in** ← Set *env*(x) to b

…

if x ← Lookup x in *env*; perform case folding accordingly

then a

else b

Implementing dead variable elimination

$$\frac{x \notin FV(e)}{\textbf{let}\ x = b\ \textbf{in}\ e \ \longrightarrow\ e} \quad \text{(dead variable elimination)}$$

Maintain a piece of state *uses* mapping variables to use counts.

Check if *uses*(x) = 0; delete binding if so

let x = b **in** e

Example

Set *env*(y) to true; recur

let y = true **in**
let z = false **in**
if x **then**
  if y **then** a **else** b
else
  if z **then** c **else** d

env

∅

uses

$x \mapsto 1$
$y \mapsto 1$
$z \mapsto 1$
$a \mapsto 1$
$b \mapsto 1$
$c \mapsto 1$
$d \mapsto 1$

Example

Set *env*(z) to false; recur

```
let y = true in
let z = false in
if x then
  if y then a else b
else
  if z then c else d
```

env	*uses*
y ↦ true	x ↦ 1
	y ↦ 1
	z ↦ 1
	a ↦ 1
	b ↦ 1
	c ↦ 1
	d ↦ 1

Example

```
let y = true in
let z = false in
if x then            ← x ∉ env; recur on branches
  if y then a else b
else
  if z then c else d
```

env	*uses*
$y \mapsto$ true	$x \mapsto 1$
$z \mapsto$ false	$y \mapsto 1$
	$z \mapsto 1$
	$a \mapsto 1$
	$b \mapsto 1$
	$c \mapsto 1$
	$d \mapsto 1$

Example

```
let y = true in
let z = false in
if x then
  if y then a else b
else
  if z then c else d
```

env(y) = true; case fold!

Will be deleted; decrement *uses*(b) and *uses*(y)

env	
y	↦ true
z	↦ false

uses	
x	↦ 1
y	↦ 1
z	↦ 1
a	↦ 1
b	↦ 1
c	↦ 1
d	↦ 1

Example

let y = true **in**
let z = false **in**
if x **then**
  a
else
  if z **then** c **else** d

env
y ↦ true
z ↦ false

uses
x ↦ 1
y ↦ 0
z ↦ 1
a ↦ 1
b ↦ 0
c ↦ 1
d ↦ 1

Example

```
let y = true in
let z = false in
if x then
  a
else
  if z then c else d
```

env(z) = false; case fold!

Will be deleted; decrement *uses*(c) and *uses*(z)

env

y ↦ true

z ↦ false

uses

x ↦ 1

y ↦ 0

z ↦ 1

a ↦ 1

b ↦ 0

c ↦ 1

d ↦ 1

Example

```
let y = true in
let z = false in
if x then
  a
else
  d
```

env	*uses*
y ↦ true	x ↦ 1
z ↦ false	y ↦ 0
	z ↦ 0
	a ↦ 1
	b ↦ 0
	c ↦ 0
	d ↦ 1

Example

```
let y = true in
let z = false in
if x then
  a
else
  d
```

env	*uses*
y ↦ true	x ↦ 1
z ↦ false	y ↦ 0
	z ↦ 0
	a ↦ 1
	b ↦ 0
	c ↦ 0
	d ↦ 1

Example

let y = true **in**
let z = false **in**
if x **then**
  a
else
  d

uses(z) = 0; z is dead

env
y ↦ true

uses
x ↦ 1
y ↦ 0
z ↦ 0
a ↦ 1
b ↦ 0
c ↦ 0
d ↦ 1

Example

let y = true **in**
if x **then**
  a
else
  d

env	*uses*
y ↦ true	x ↦ 1
	y ↦ 0
	z ↦ 0
	a ↦ 1
	b ↦ 0
	c ↦ 0
	d ↦ 1

Example

uses(y) = 0; y is dead

let y = true **in**
if x **then**
  a
else
  d

env

∅

uses

x ↦ 1
y ↦ 0
z ↦ 0
a ↦ 1
b ↦ 0
c ↦ 0
d ↦ 1

Example

if x **then**
  a
else
  d

env	*uses*
$\varnothing$	$x \mapsto 1$
	$y \mapsto 0$
	$z \mapsto 0$
	$a \mapsto 1$
	$b \mapsto 0$
	$c \mapsto 0$
	$d \mapsto 1$

Specifying helper data structures

- At each step, there is a subterm in focus *e* and surrounding context *C*
- Can think of implementations as state machines with configurations (C, e)
- *env* and *uses* are related to (*C*, *e*) at each step by an invariant

$$(C, e) \sim \mathit{env} \Leftrightarrow \forall\ x\ b,\ \mathit{env}(x) = b \Leftrightarrow x \text{ bound to } b \text{ in } C$$

$$(C, e) \sim \mathit{uses} \Leftrightarrow \forall\ x\ n,\ \mathit{uses}(x) = n \Leftrightarrow x \text{ used } n \text{ times in } C[e]$$

Our tool

Types ensure correctness

Rewriting relation ($\longrightarrow$)

"Use lookup tables to store bindings in scope"
"Use a counter variable to generate fresh names"

$$
\begin{array}{l}
\textbf{Fixpoint}\ \text{opt}\ \cdots\ e : \sum_{e'} e \longrightarrow^* e' := \\
\quad \textbf{match}\ e\ \textbf{with} \\
\quad \mid \cdots \Rightarrow \cdots \square \cdots \\
\quad \quad \vdots \\
\quad \mid \cdots \Rightarrow \cdots \square \cdots \\
\quad \textbf{end.}
\end{array}
$$

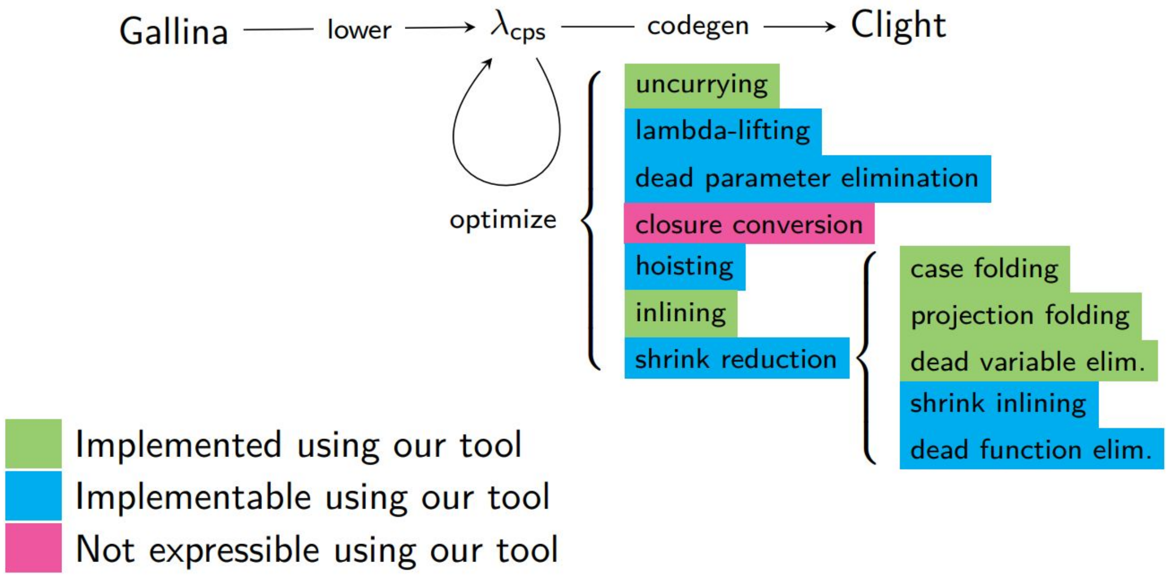
Specified how?

Answer: by <u>invariants</u>, relating each data structure to intermediate states (C, e)

Holes to fill by hand

Our tool

- Our tool also supports delayed computations and custom termination metrics
- Resulting framework is simple, but can express many of CertiCoq's passes:



Demo

Syntax

```
Definition var := positive.
Inductive exp :=
| LetIn (x : var) (b : 𝔹 ) (e : exp)
| IfThenElse (x : var) (e₁ : exp) (e₂ : exp).
```

e ::= x | **let** x = b **in** e | **if** x **then** e **else** e

```
MetaCoq Run¹
  (mk_Frame_ops
    (MPfile ["Example"; "L6"; "CertiCoq"])
    (MPfile ["Example"; "L6"; "CertiCoq"], "exp") exp
    [var; 𝔹 ]).
```

C ::= □ | **let** x = b **in** C

| **if** x **then** C **else** e

| **if** x **then** e **else** C

[1]Sozeau et al., 2020

Rewrite rules

```
Inductive rewrite_step : exp → exp → ℙ :=
(** Case folding *)
| case_fold : ∀ (C : ctx) x b e₁ e₂ e_taken,
  known_bool x b C ∧
  (if b then e₁ else e₂) = e_taken →
  C ⟦ IfThenElse x e₁ e₂ ⟧ ⟶ C ⟦ Rec e_taken ⟧
(** Dead variable elimination *)
| dead_var_elim : ∀ (C : ctx) x b e,
  ~ occurs_free x e →
  BottomUp (C ⟦ LetIn x b e ⟧ ⟶ C ⟦ e ⟧)
where "e1 --> e2" := (rewrite_step e₁ e₂).
```

let x = true **in** C[**if** x **then** e1 **else** e2]

⟶ **let** x = true **in** C[e1]

$$\frac{x \notin FV(e)}{\textbf{let}\ x = b\ \textbf{in}\ e \ \longrightarrow\ e}$$

Invariants

```
Definition env_map := M.tree 𝔹 .

Definition env_map_invariant
  {A} (C : frames_t A exp_univ_exp) (env : env_map) :=
  ∀ x b, M.get x env = Some b → known_bool x b C.
```

```
Definition uses_map := M.tree ℕ .

Definition uses_map_invariant
           {A} (C : frames_t A exp_univ_exp)
           (e : univD A)
           (uses : uses_map) :=
  ∀ x, get_count x uses = use_count x (C ⟦ e ⟧).
```

Preserving invariants across recursive calls

```
(** Obligations re: preserving invariants across recursive calls *)

Instance Preserves_env: Preserves_R (@env_map_invariant).
```

```
Instance Preserves_uses_up: Preserves_S_up (@uses_map_invariant).
```

```
Instance Preserves_uses_dn: Preserves_S_dn (@uses_map_invariant).
```

Preserving invariants across recursive calls

```
(** Obligations re: preserving invariants across recursive calls *)

Instance Preserves_env: Preserves_R (@env_map_invariant).
Proof.
  intros A B fs fs_ok f [env Henv]; destruct f;
  lazymatch goal with
  | ⊢ Param (@env_map_invariant)
        (e_map (λ fs ⇒ fs >:: LetIn2 ?x' ?b') fs) ⇒
    rename x' into x, b' into b
  | _ ⇒
    ∃ env; unerase; intros x' b' Hget';
    specialize (Henv x' b' Hget');
    destruct Henv as [D [E Hctx]];
    match goal with
    | ⊢ known_bool _ _ (_ >:: ?f) ⇒
      ∃ D, (E >:: f); now subst fs
    end
  end.
  ∃ (M.set x b env); unerase; intros x' b' Hget'; cbn in *.
  destruct (Pos.eq_dec x' x);
    [subst; rewrite M.gss in Hget';
     inversion Hget'; now ∃ fs, <[]>|].
  rewrite M.gso in Hget' by auto.
  destruct (Henv x' b' Hget') as [D [E Hctx]].
  ∃ D, (E >:: LetIn2 x b); now subst fs.
Defined.
Extraction Inline Preserves_env.
```

```
Instance Preserves_uses_up: Preserves_S_up (@uses_map_invariant).
Proof.
  intros A B fs fs_ok f x [uses Huses];
    ∃ uses; unerase; apply Huses.
Defined.
Extraction Inline Preserves_uses_up.

Instance Preserves_uses_dn: Preserves_S_dn (@uses_map_invariant).
Proof.
  intros A B fs fs_ok f x [uses Huses];
    ∃ uses; unerase; apply Huses.
Defined.
Extraction Inline Preserves_uses_dn.
```

Deriving an implementation

rewrite rules + invariants specified in goal type

```
Definition optimize :
  rewriter exp_univ_exp false (λ A C e ⇒ @measure A C e)
           rewrite_step _ (I_D_plain (D:=unit))
           _ (@env_map_invariant)
           _ (@uses_map_invariant).
Proof.
  (** Derive partial program + partial proof *)
  mk_rw.
  all: mk_easy_delay.
  (** Solve obligations related to termination *)
  all: try lazymatch goal with ⊢ MetricDecreasing → _ ⇒
    try (simpl; unfold delayD; lia);
    clear - H H1; cbn in *; subst e_taken0;
    intros _; destruct H as [_ H], b;
    apply f_equal with (f := exp_size) in H; lia end.
```

MetaCoq + Ltac

Deriving an implementation

```
2 subgoals (ID 974)

  ______________________________________
  ExtraVars "case_fold" →
  ∀ (Ans : Set) (C : erased ctx),
  e_ok C →
  ∀ (x : var) (e1 e2 : exp) (d : Delay (I_D_plain (D:=unit)) (IfThenElse x e1 e2)),
  Param (@env_map_invariant) C →
  State (@uses_map_invariant) C (delayD d) →
  (Success "case_fold" →
   ∀ (e_taken : exp) (d0 : Delay (I_D_plain (D:=unit)) e_taken) (x0 : var) (e3 e4 : exp)
     (b : 𝔹 ) (e_taken0 : exp),
   « e_map (λ C0 : ctx ⇒ known_bool x0 b C0 ∧ (if b then e3 else e4) = e_taken0) C » →
   delayD d = IfThenElse x0 e3 e4 →
   e_taken0 = delayD d0 → Param (@env_map_invariant) C → State (@uses_map_invariant) C (Rec e_taken0) → Ans) →
  (Failure "case_fold" → Ans) → Ans

subgoal 2 (ID 976) is:
 ExtraVars "dead_var_elim" →
 ∀ (Ans : Set) (C : erased ctx),
 e_ok C →
 ∀ (x : var) (b : 𝔹 ) (e : exp),
 Param (@env_map_invariant) C →
 State (@uses_map_invariant) C (LetIn x b e) →
 (Success "dead_var_elim" →
  ~ occurs_free x e → Param (@env_map_invariant) C → State (@uses_map_invariant) C e → Ans) →
 (Failure "dead_var_elim" → Ans) → Ans
```

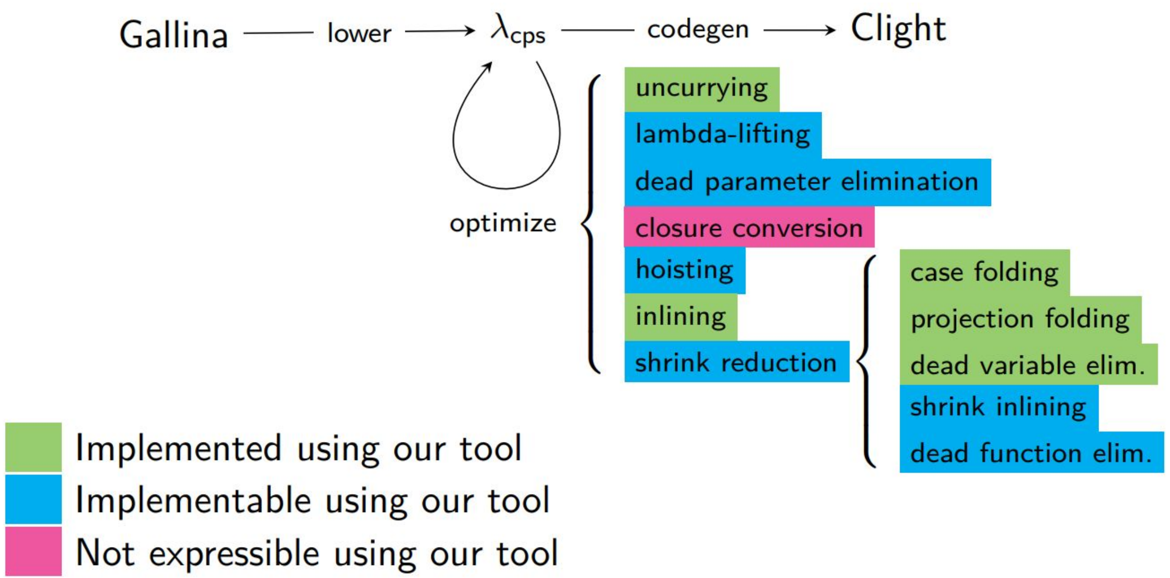
Deriving an implementation

```
(** Implement case folding *)
intros _ R C C_ok x e1 e2 d r s success failure.
destruct r as [env Henv] eqn:Hr.
(** Using env, check whether x is in scope.
    If so, perform case folding accordingly *)
destruct (M.get x env) as [b|] eqn:Hbool; [|cond_failure].
pose (d' := d : Delay (I_D_plain (D := unit))
                      (A:=exp_univ_exp)
                      (if b then e1 else e2)).
cond_success success.
specialize (success _ d' x e1 e2 b (if b then e1 else e2)).
unshelve eapply success; unerase; auto.
(** Decrement use counts of scrutinee + deleted branch *)
destruct s as [uses Huses]; destruct b;
  [∃ (upd_count decr x (decr_use_counts e2 uses))
  |∃ (upd_count decr x (decr_use_counts e1 uses))].
all: unerase; intros y; clear - Huses;
  specialize (Huses y); cbn in *;
  unfold Rec; rewrite use_count_ctx_app in *; cbn in *;
  rewrite decr_count_correct, decr_use_counts_correct; lia.
```

Deriving an implementation

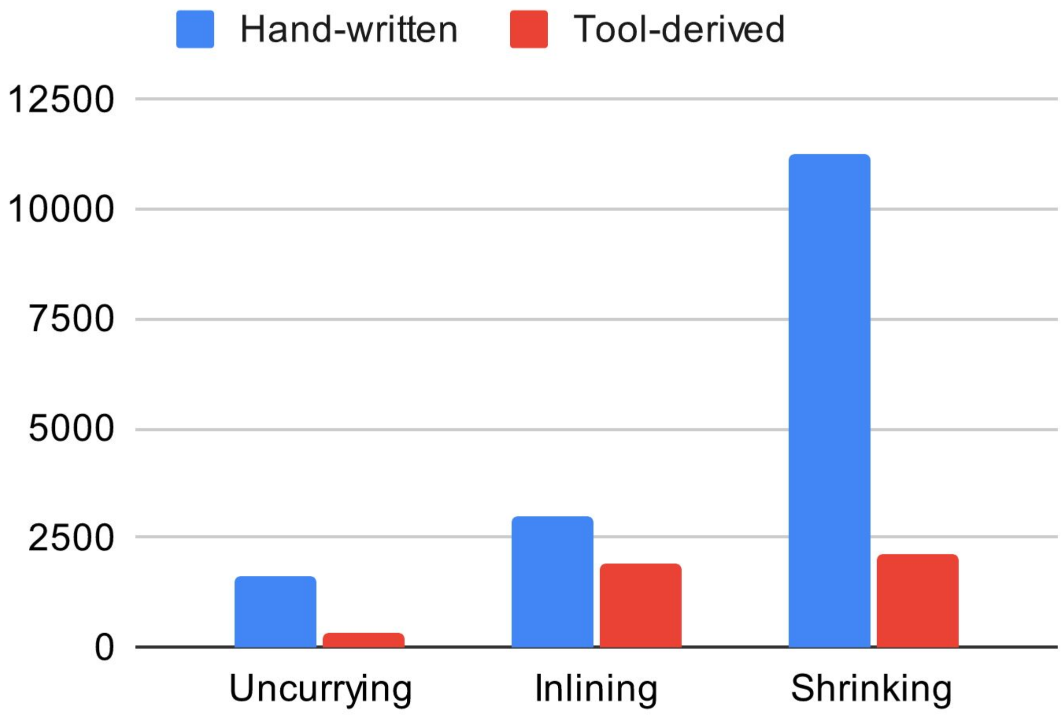
```
- (** Implement dead variable elimination *)
  clear; intros _ R C C_ok x b e r [uses Huses] success failure.
  (** Using uses, check whether x is dead.
      If so, perform dead variable elimination. *)
  destruct (M.get x uses) as [n|] eqn:Hbool; [cond_failure|].
  cond_success success.
  assert (Hget : get_count x uses = 0)
    by (unfold get_count; now rewrite Hbool).
  apply success; auto.
  + unerase. specialize (Huses x). cbn in *.
    apply use_count_zero_implies_dead.
    rewrite Huses, use_count_ctx_app in Hget.
    cbn in Hget; lia.
  + ∃ uses; unerase; intros y.
    specialize (Huses y); cbn in *.
    rewrite Huses, ?use_count_ctx_app in *; now cbn.
Defined.
```

Evaluation



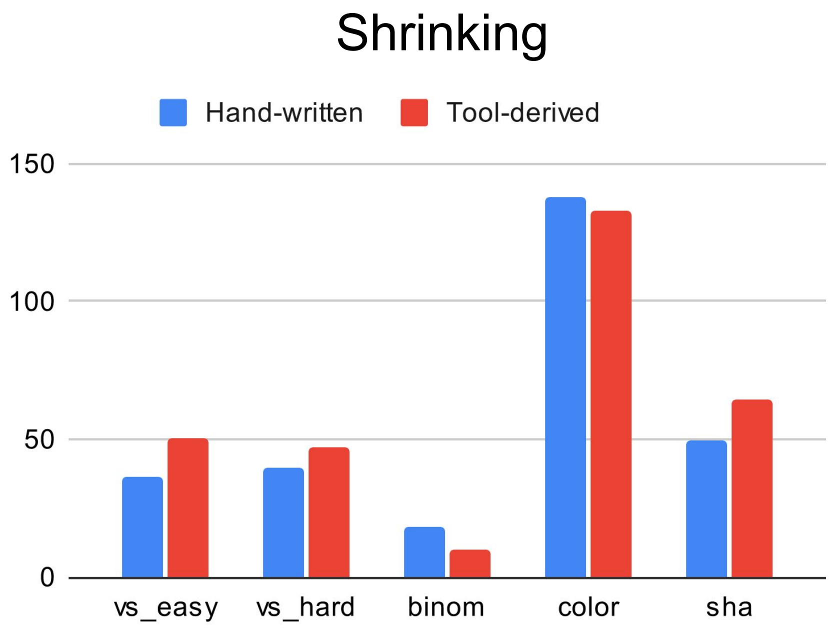
- Compared lines of code & proof to manual implementations
- Measured run-times of CertiCoq on a suite of benchmarks

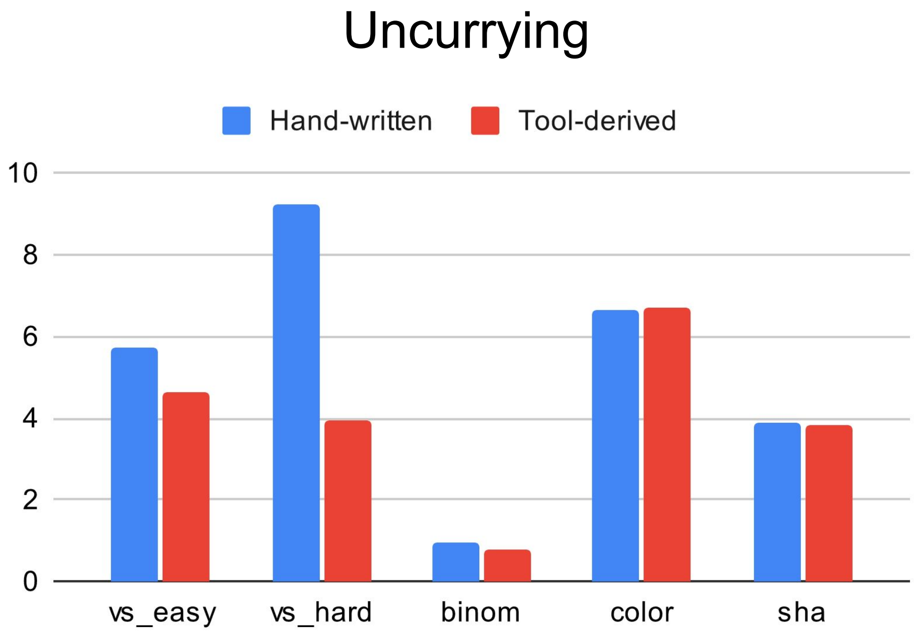
Line counts



Hand-written
Tool-derived
12500
10000
7500
5000
2500
0
Uncurrying
Inlining
Shrinking

Run times (milliseconds)





Future work

- Make the generated Coq code more human-readable
- Cross-language transformations? (e.g. CPS/ANF conversion, closure conversion)
- Implement more transformations