Accelerating Verified-Compiler Development with a Verified Rewriting Engine

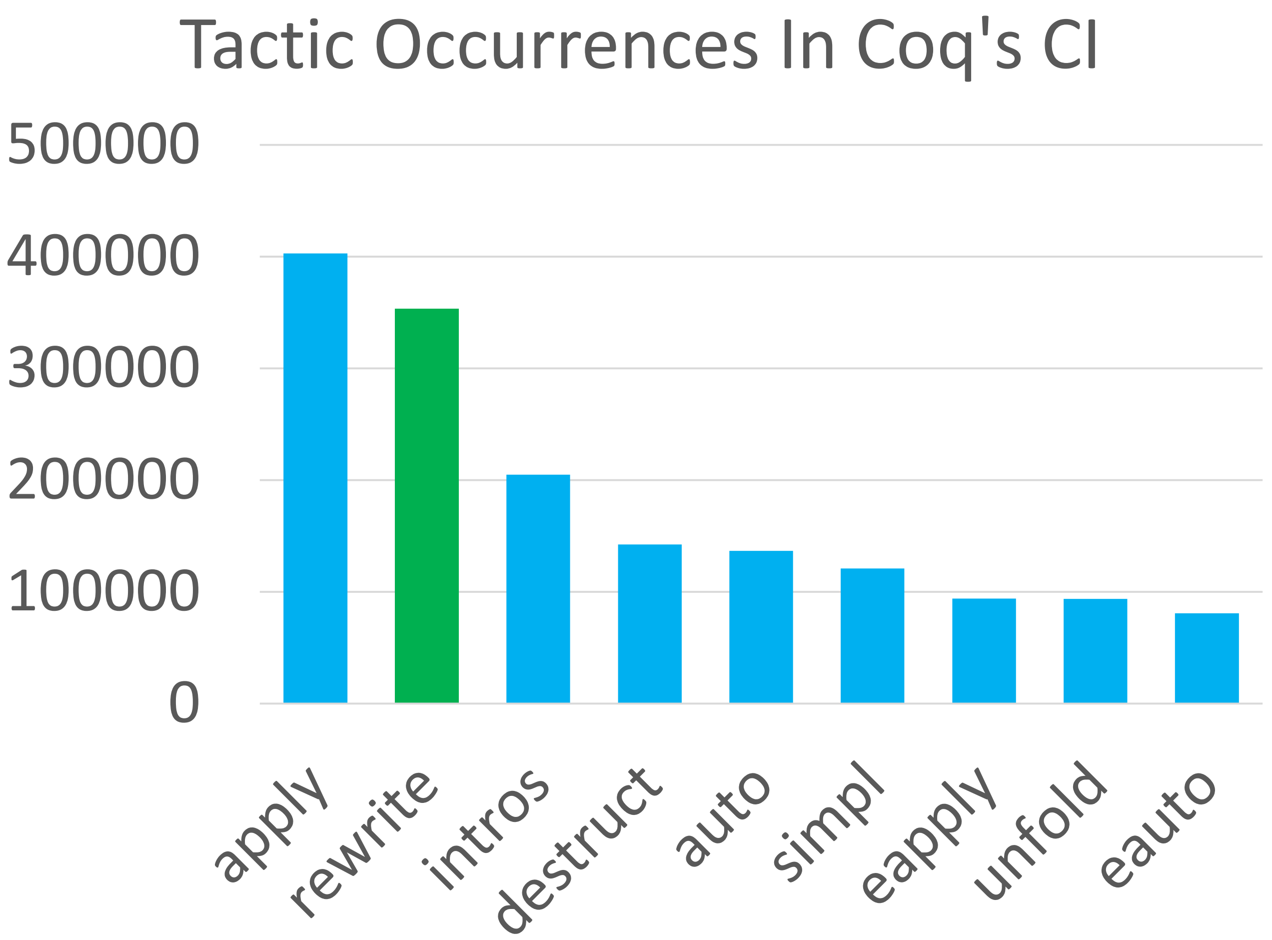
Jason Gross, Andres Erbsen, Jade Philipoom, Miraya Poddar-Agrawal, and Adam Chlipala

Rewriting: An Essential Proof Engine Component

- General equational reasoning
  - e.g., $x + x \rightsquigarrow 2x$

Can be used for:

- Arithmetic
- Code transformations
- Partial evaluation
- Implementing compilers
- Deriving optimized code
- …



Rewriting is Too Slow (for industry-scale applications)

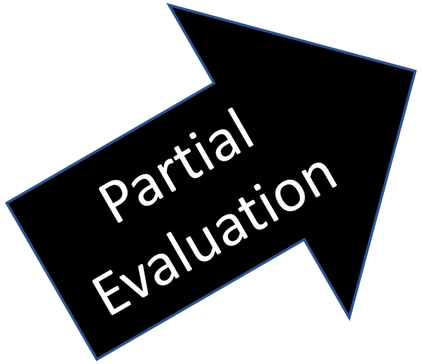
Fiat Cryptography

- Industry-scale: generates 100s – 1000s of lines of verified code in Coq
- Used in *majority* of secure connections from web browsers

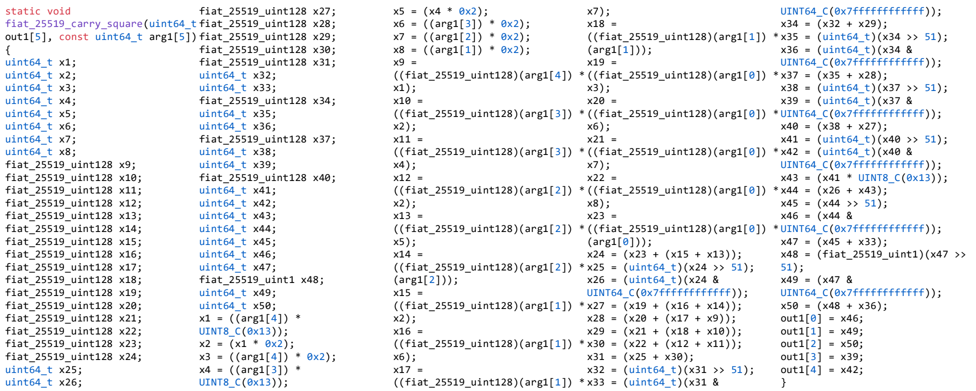
Partial Evaluation & Rewriting in Fiat Cryptography

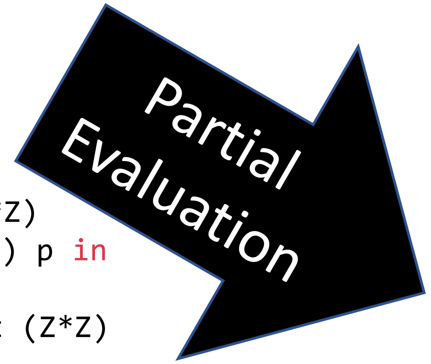
Template Code:

```
Definition mul (p q:list (Z*Z)):list (Z*Z) :=
  flat_map (fun '(w, t) =>
    map (fun '(w', t') =>
      (w * w', t * t’))
    q) p.
Fixpoint square (p:list (Z*Z)):list (Z*Z)
:= match p with
   | [] => []
   | (w, t) :: ts
     => let two_t := 2 * t in
        ((w * w, t * t)
         :: map (λ '(w', t'), (w * w', two_t * t')) ts)
        ++ square ts
   end.
Definition split (s:Z) (p:list(Z*Z)):list (Z*Z) * list (Z*Z)
:= let '(hi, lo) := partition (fun '(w, _) => w mod s =? 0) p in
   (lo, map (fun '(w, t) => (w / s, t)) hi).
Definition reduce (s:Z) (c:list (Z*Z)) (p:list (Z*Z)):list (Z*Z)
:= let '(lo, hi) := split s p in lo ++ mul c hi.
```

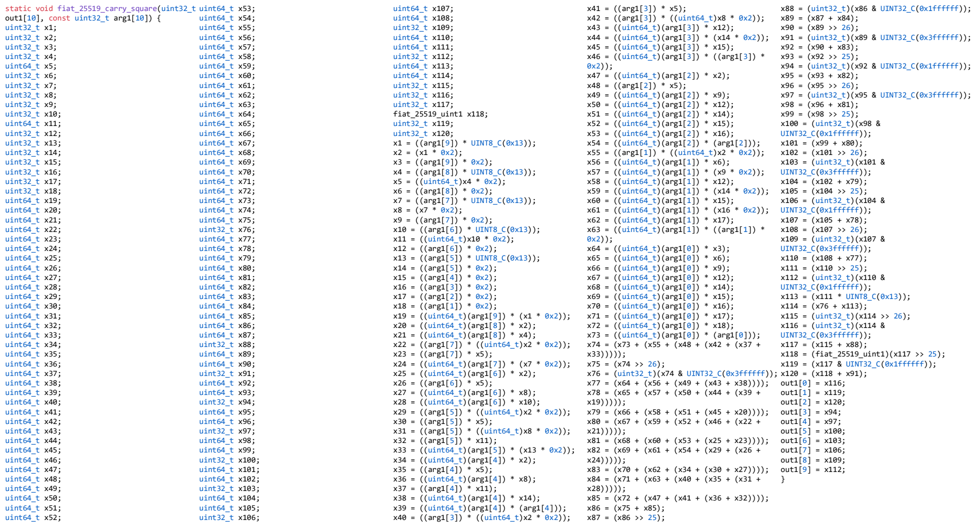


64-bit square





32-bit square



Rewriting is Too Slow
(for industry-scale applications)
How Slow is Too Slow?

Fiat Cryptography Pieces

Associational

Columns

Montgomery

Freeze

Bounds Analysis

Positional

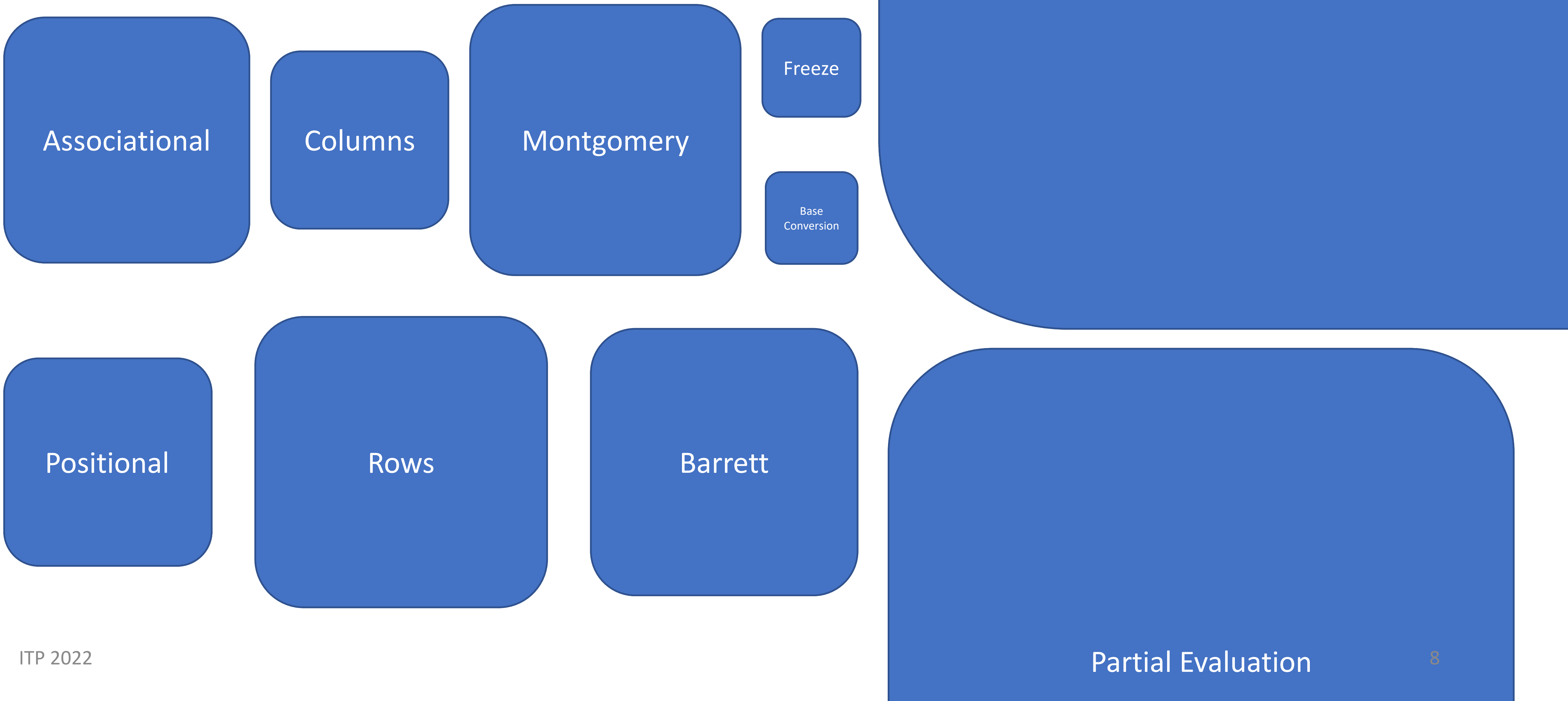
Rows

Barrett

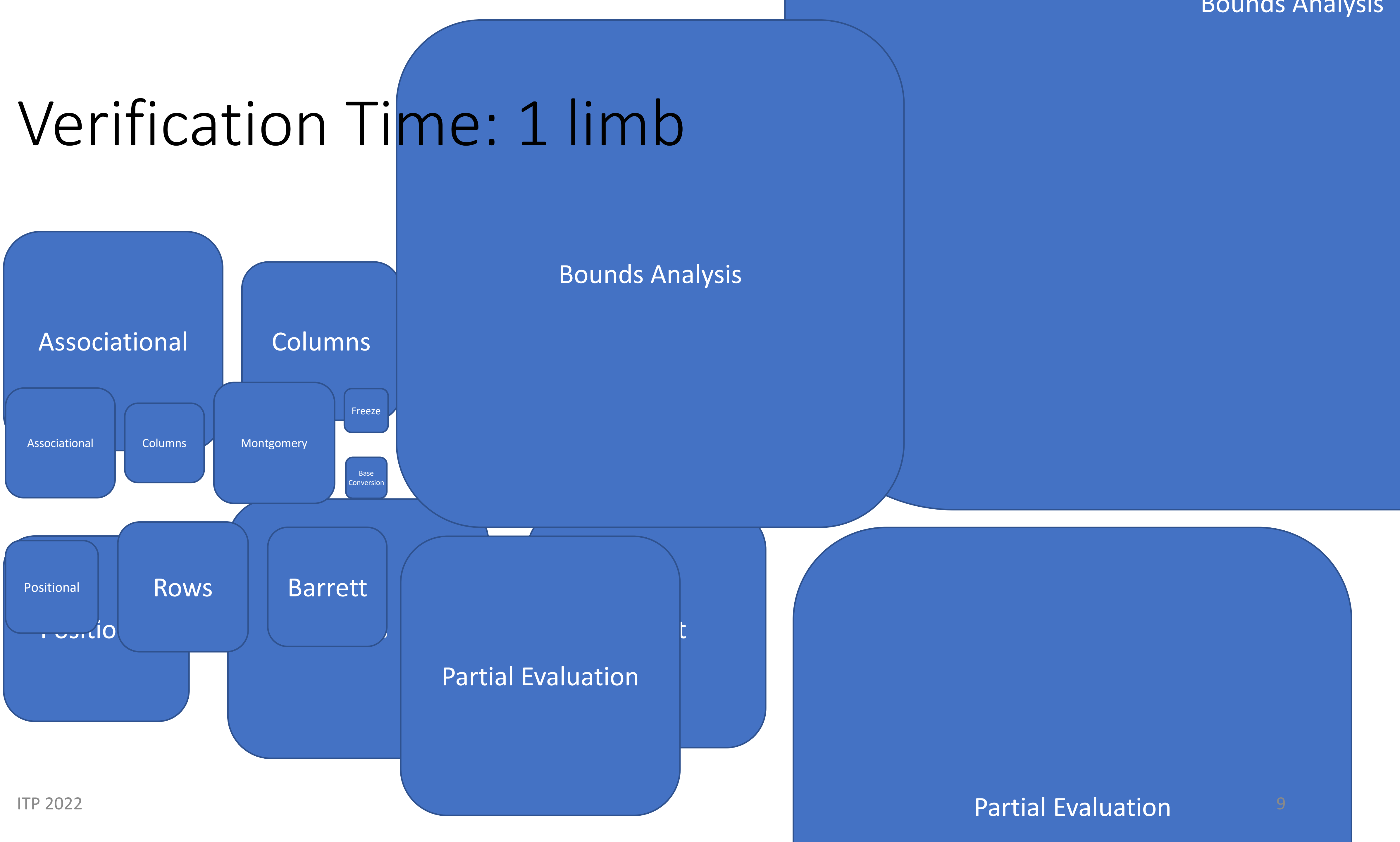
Base Conversion

Partial Evaluation

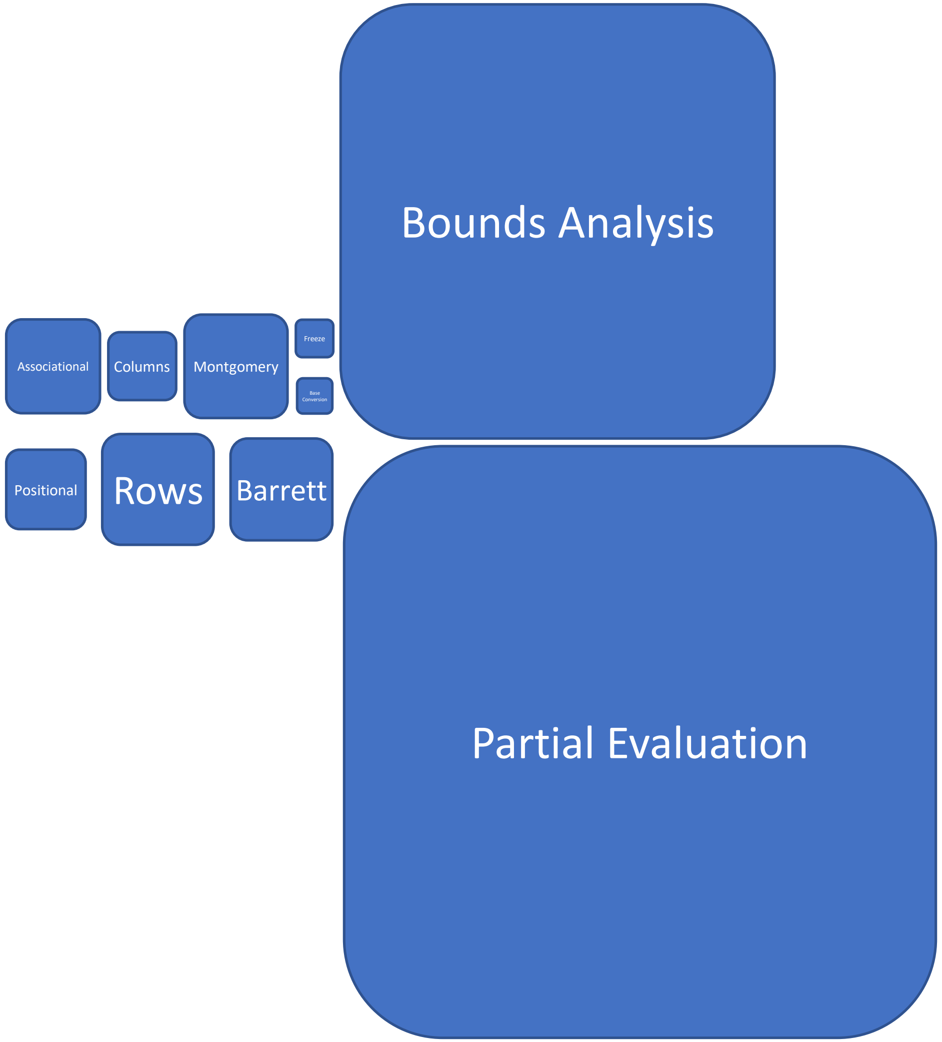
Verification Time: 1 limb



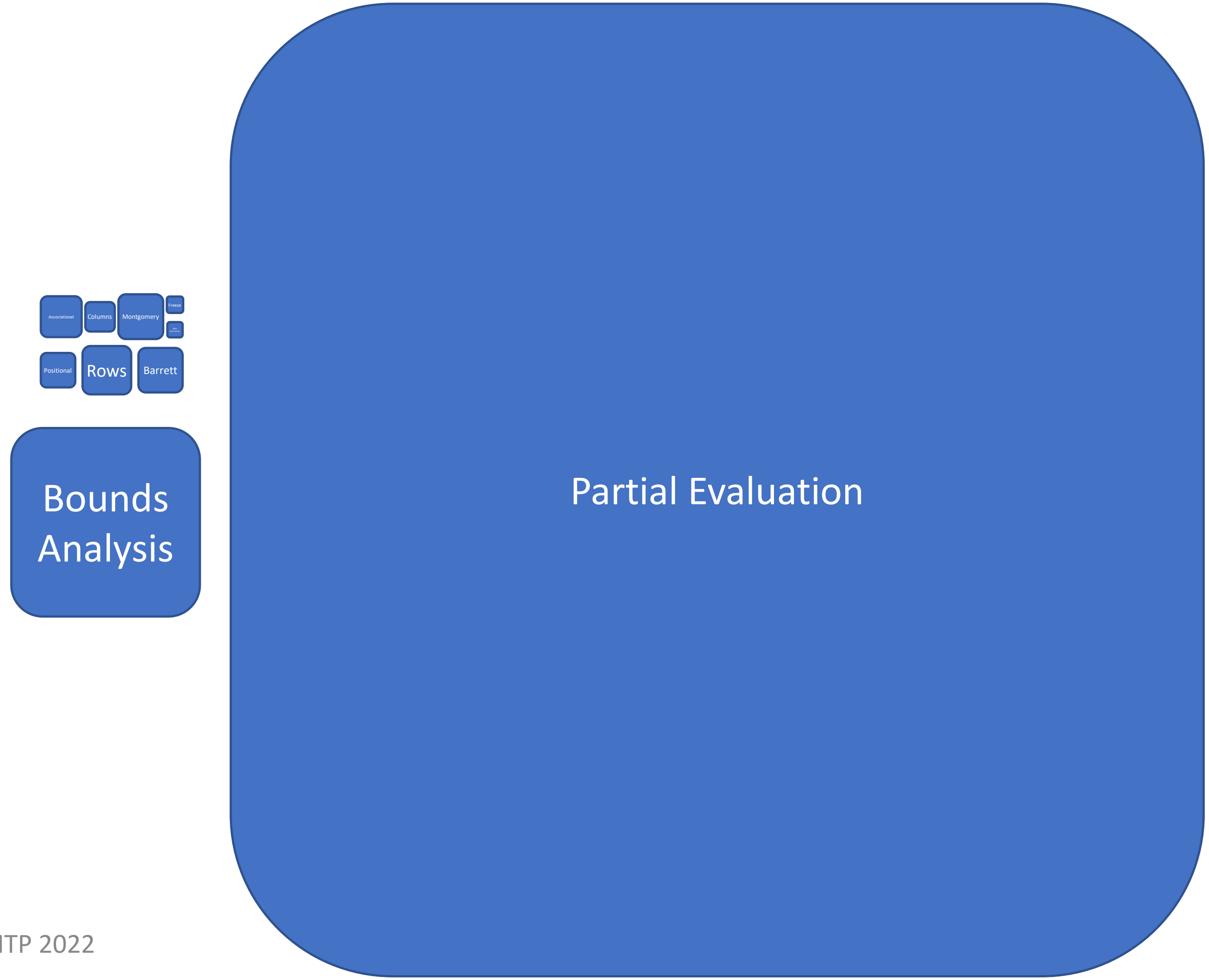
Verification Time: 1 limb



Verification Time: 2 limbs



Verification Time: 3 limbs



Two Options for Solutions

1. Seek performant modular ***proof-producing*** rewriting

2. Throw away the proof engine and
   write performant ***proven-correct*** rewriting with reflection

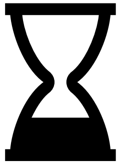
Non-Reflection Example

```
Inductive is_even:ℕ→ℙ := |zero_even : is_even 0 |two_plus_even n : is_even n → is_even (2+n).
```

```
Goal is_even 9002.
```

```
repeat constructor
```

```
Qed
```



Goal is:
`is_even 9002`

Current Proof is:

```
two_plus_even 9000
(two_plus_even 8998
(two_plus_even 8996
(two_plus_even 8994 …
```

✓

Reflection Example: Up-Front Work

```
Inductive is_even:ℕ→ℙ := |zero_even : is_even 0 |two_plus_even n : is_even n → is_even (2+n).
```

```
Inductive parity := even | odd.
```

```
Definition flip_parity p
:= match p with even => odd | odd => even end.
```

```
Fixpoint parity_of (n : nat) : parity :=
match n with
| O => even
| S n' => flip_parity (parity_of n') end.
```

```
Lemma parity_of_correct
: ∀ n, parity_of n = even → is_even n.
Proof.
  intro n; assert (H' : match parity_of n with
                       | even => is_even n
                       | odd => is_even (S n) end).
  { induction n as [|n IH]; cbn; try constructor.
    destruct (parity_of n); cbn; try constructor; try assumption. }
  intro H; rewrite H in H'; assumption.
Qed.
```

Reflection Example

```
Inductive is_even:ℕ→ℙ := |zero_even : is_even 0 |two_plus_even n : is_even n → is_even (2+n).
Inductive parity := even | odd.
Fixpoint parity_of : ℕ → parity
```

```
Lemma parity_of_correct
: ∀ n, parity_of n = even → is_even n.
```

`Goal is_even 9002.`

`apply parity_of_correct`

`vm_compute; reflexivity.`

`Qed`

Goal is:
`is_even 9002`

Goal is:
`parity_of 9002 = even`

Current Proof is:
`parity_of_correct 9002 (eq_refl even)`

✔

Man Using Laptop by Gan Khoon Lay from NounProject.com CC BY 3.0
Coq logo from https://calebstanford.com/2019/01/15/coq-vector-image/

Why isn’t there an off-the-shelf solution?

Two Issues:

- Reflective solutions are often hard to use
- Any given reflective solution can’t handle the full logic

Hard To Use

Want to write:

```
Hint Rewrite plus_n_O plus_O_n :  all.
Goal forall n, 0 + n = n + 0.
  intros; autorewrite with all.
```

In $\mathcal{R}_{tac}$, for example:

```
Notation "a @ b" :=
 (App a b) (at level 30).
Let eq_nat (a b : E) : E
:= Inj (Eq tyNat) @ a @ b.
Let plus (a b : E) : E := Inj Plus @ a @ b.
Let n (n : nat) : E := Inj (N n).

(* forall n, 0 + n = n *)
Definition RW1 : RW typ E subst :=
{| lem := {| Lemma.vars := tyNat :: nil
           ; Lemma.premises := nil
           ; Lemma.concl := (tyNat, plus (n 0) (Var 0), Var
0) |}
 ; side_solver := use_list nil
 |}.

(* forall n, n + 0 = n *)
Definition RW2 : RW typ E subst :=
{| lem := {| Lemma.vars := tyNat :: nil
           ; Lemma.premises := nil
           ; Lemma.concl := (tyNat, plus (Var 0) (n 0), Var
0) |}
 ; side_solver := use_list nil
 |}.

Let all := fun t => match t with
                    | tyNat => RW1 :: RW2 :: nil
                    | _ => nil
                    end.
Time Eval vm_compute in
    let goal := eq_nat (plus (n 0) (Var 0)) (plus (Var 0) (n
0)) in
    autorewrite all nil (tyNat :: nil) (@empty _ _ _) nil
goal.
```

~~Hard~~ Easy To Use

Want to write:

```
Hint Rewrite plus_n_O plus_O_n :  all.
Goal forall n, 0 + n = n + 0.
  intros; autorewrite with all.
```

With our tool:

```
Make all := Rewriter for (plus_n_O, plus_O_n).
Goal forall n, 0 + n = n + 0.
  intros; Rewrite_for all.
```

Alternate Usage Mode: Extracted Codegen

D:\cygwin\usr\local\bin\mosh-jgross-transparent-rooster.sh

```
$ src/ExtractionOCaml/unsaturated_solinas curve25519 64 5 2^255-19
```

```
[0] 0:emacs- 1:bash  2:htop  3:bash  4:rlwrap* 5:b> "jgross-Leopard-WS" 09:25 27-Nov-20
```

Alternate Usage Mode: Extracted Codegen

D:\cygwin\usr\local\bin\mosh-jgross-transparent-rooster.sh

```
$ src/ExtractionOCaml/unsaturated_solinas curve25519 64 5 2^255-19
```

```
[0] 0:emacs- 1:bash  2:htop  3:bash  4:rlwrap* 5:b> "jgross-Leopard-WS" 09:25 27-Nov-20
```

Making the Rewriter Usable

Challenges

- Expressing the rewrite rules
- Reification
- Gluing the reflective proof

Solution:

- Automate the boilerplate

Why isn't there an off-the-shelf solution?

Two Issues:

- Reflective solutions are often hard to use (solved: automate boilerplate)
- Any given reflective solution can't handle the full logic

Fundamental Obstacle: Gödel's Incompleteness

We can sidestep this!

Gödel’s Obstacle

Can’t have a language that can represent everything

For example:

- To encode n universes, we need to use at least $n + 1$
- To encode terminating recursion, our metalanguage needs stronger recursion
- Technically: Löb’s theorem says that any encoding of a consistent language within itself cannot have a denotation function

Our Solution

- Family of languages
- Family of denotation functions
- Instantiation is done on-the-fly, automatically, behind-the-scenes

Our Solution

Our family handles:

- Any argumentless inductive type
- Any non-dependent prenex polymorphic function
- Standard: Let binders, Lambdas, Variables, Application

Limitations:

- Named eliminators instead of (co)fixpoints
- Supported container types: option, list
- Limited support for side conditions of rewrite lemmas

Extending Our Solution (Future Work)

- Family of languages
- Family of denotation functions

Goal:

- A family that is broad enough to handle all finite fragments of Coq
- Each language in the family would itself require a larger language, still within the family, to encode it
- Entire family could be encoded, as a family, using universe polymorphism

Autogenerating Family Instantiations

```
Inductive base : Type := Bnat.(*non-function types*)
Inductive ident: type base -> Type :=(* constants *)
| iO   : ident Bnat
| iS   : ident (Bnat -> Bnat)
| iadd : ident (Bnat -> Bnat -> Bnat).
Definition base_interp (ty : type base) : Type.
Definition ident_interp {t} (idc : ident t)
  : type.interp base_interp t.
…
```

Families

```
Inductive type (base : Type) : Type
:= type_base (t : base) | arrow (s d : type base).
Inductive expr {base ident var}: type base -> Type:=
| Ident {t} (_ : ident t)
| App …
| Abs …
| Var …
| LetIn …
.
```

Why isn’t there an off-the-shelf solution?

Two Issues:

- Reflective solutions are often hard to use

  (solved: automate boilerplate)

- Any given reflective solution can’t handle the full logic

  (solved: use a family instead)

Main Components of our Rewriter

- Normalization by Evaluation (NbE) for partial evaluation
- Pattern-matching compilation for rewrite-rule selection
- Parametric Higher-Order Abstract Syntax (PHOAS)

Normalization by Evaluation

- Leverage metalanguage substitution for object-language substitution
- Fused with LetIn monad for subterm sharing preservation
- Naturally fuses with rewriting

Normalization by Evaluation: Rewrite Ordering

Ordering Rewrites = Ordering Reduction / Computation

- (in rewrite-based compiler frameworks / partial evaluation)
- asymptotic performance improvement over widely available rewrite orderings

Rewrite Ordering Options:

- topdown
- bottomup
- call-by-value
- call-by-name

Pattern-Matching Compilation

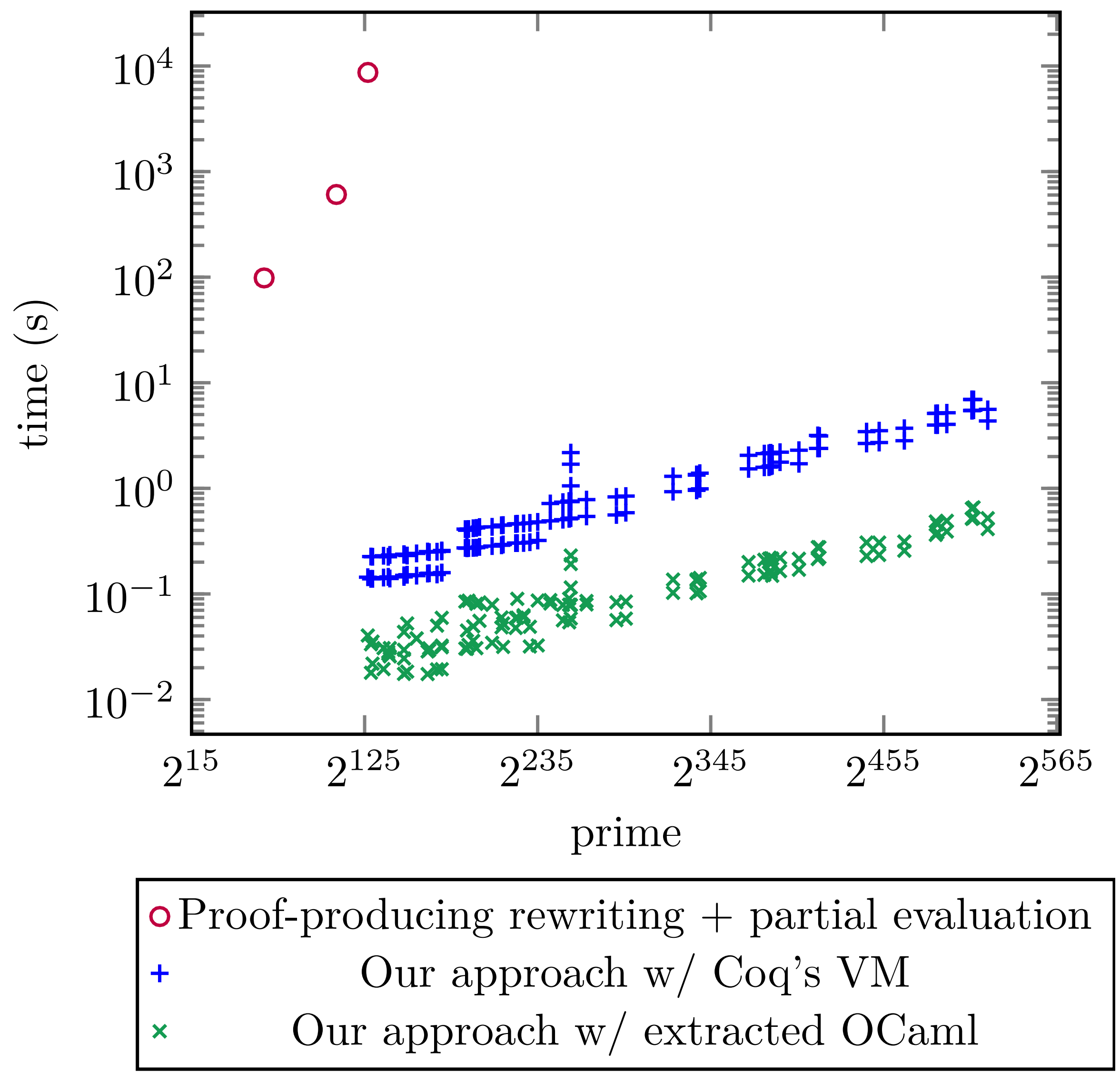
- Patterns are extracted from rewrite rules
- Standard ingredient of programming languages like ML
- Adapted to handle rewrite-rule side conditions
- Standard algorithm is essentially untyped, which is at tension with well-typed term representations
- Leverage efficient case analysis in the metalanguage for efficient case analysis in the object language
  - Relies on *inductive* type for constant codes
  - Avoid string comparisons

Parametric Higher-Order Abstract Syntax (PHOAS)

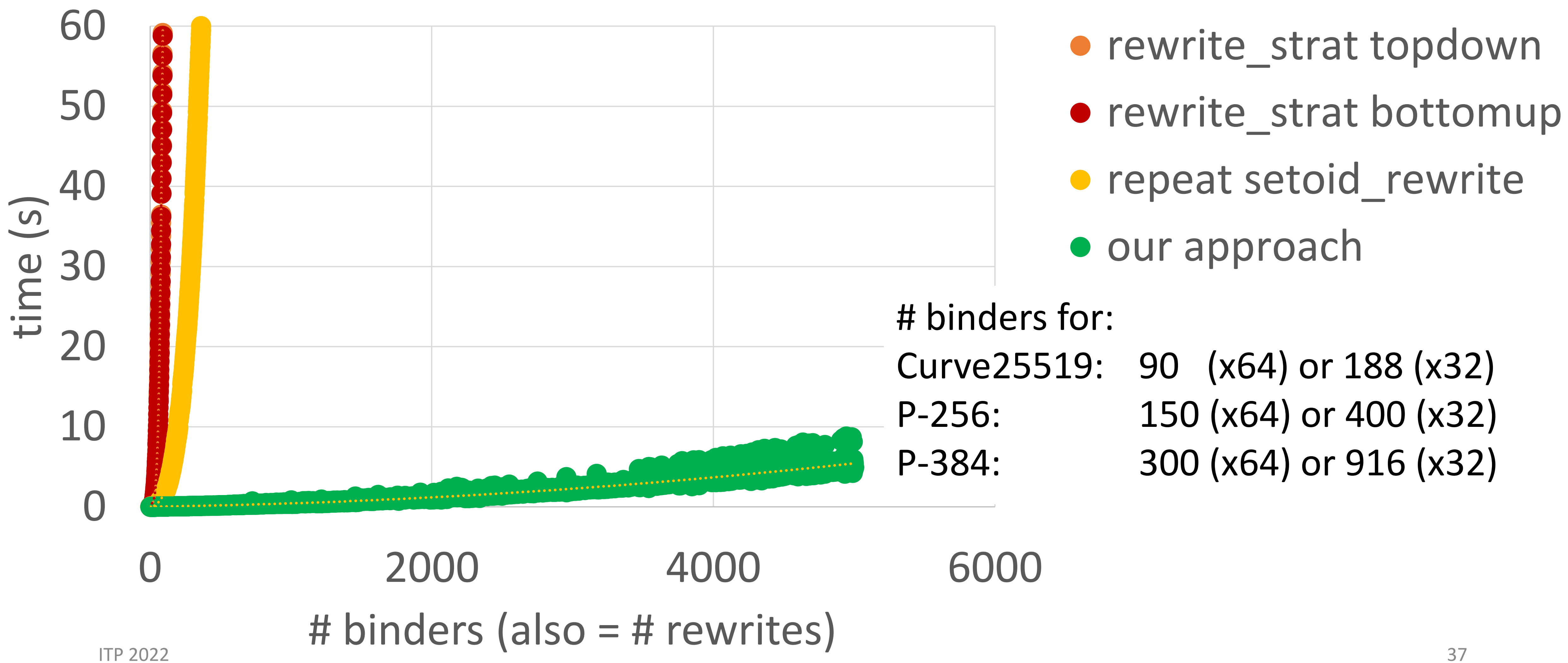
- Well-typed syntax encoding
- Avoids binder-bookkeeping
- Allows complex transformations involving binders

Performance Evaluation

Performance on Fiat Cryptography



Synthetic Performance Benchmark



binders for:
Curve25519: 90 (x64) or 188 (x32)
P-256: 150 (x64) or 400 (x32)
P-384: 300 (x64) or 916 (x32)

Future Work

Extending this to a complete proven-correct scalable performant rewriting engine for all of Coq without extending the TCB:

- Start with MetaCoq / Coq in Coq / Coq Coq Correct!
- Parameterize over [map of] named (co)inductives, universes, evars, constants, and eliminators
- Adjust well-typedness construction for denotation
- Emit necessary instantiations on the fly
- (Optional) Add constructor for efficient pattern-matching compilation

Proof Engine Interface:

- More support for side conditions

MetaCoq logo from https://metacoq.github.io/

Extra Content

Rewriting Pseudocode: No Binders

```
rw (f x) =
   let (mid, fx_mid) :=
     match rw f, rw x with
     | (f', f'f), (x', x'x) => (f' x', app_cong f'f x'x)
     | _ => (f x, eq_refl (f x))
     end in
   match rwh mid with
   | (result, mid_result) => (result, eq_trans fx_mid mid_result)
   | _ => (mid, fx_mid)
   end
```

Rewriting Pseudocode: With Binders

```
rw (fun x:T => e) =
   let rrw := (fun x:T => rw e) in
   let mid := (fun y:T => let (e', _) := beta (rrw y) in e') in
   let f_mid := functional_extensionality
                 (fun z:T => let (_, e'e) := beta (rrw z) in e'e) in
   match rwh mid with
   | (result, mid_result) => (result, eq_trans f_mid mid_result)
   | _ => (mid, f_mid)
   end
```