Performance Engineering of Proof-Based Software Systems at Scale

Jason Gross
Ph.D. Defense
MIT CSAIL

Good morning! I would like to start with my belated thanksgiving toasts. First thank you Adam, for all of your support and guidance and encouragement through the years! I obviously don't know what it's like to have any other PhD advisor, but I can't imagine having a PhD advisor who would have been better for my mental health than you.
I want to thank my coworkers, with special thanks to you Andres for many engaging conversations and rich and productive collaborations! I want to thank you, mom for taking every opportunity to enrich my life and setting me on this path. Thank you Rachel, Dad, and the rest of my family, you've always been kind and supportive. Thank you to the several teachers in the audience from my 24 years of education, for inspiring me and nourishing me. Thank you, Allison, for your friendship through the years. Thank you Rajee for your faith in me and for everything you've done to help me excel. Finally, thank you to everyone who has made the time to be here, for your support and interest.

I will be defending my thesis titled …

Takeaways

- Opportunity: Automate Verification to Enable Innovation
- Big Problem: Asymptotic Performance
- My Contribution: Reflective Partial Evaluation
- Important Next Steps

The things I want you to takeaway from this talk…
The big problem in automating verification in interactive theorem provers is …
I will talk about my contribution to solving this problem through my work in reflective partial evaluation.
And conclude with what I see as important next steps for the field of verification.

Fiat Crypto

- Joint work with Andres Erbsen, Jade Philipoom, Adam Chlipala, et al
- Used in *majority* of secure connections from web browsers

ive.mozilla.org/foundation/identity-guidelines/firefox
HTTPS image modified from image by Sean MacEntee, CC BY 2.0, via Wikimedia Commons
Chrome logo from https://www.logo.wine/logo/Google_Chrome ©2018 Google LLC All rights reserved. Chrome is a trademark of Google LLC.
Go Logo By The Go Authors - https://blog.golang.org/go-brand, Public Domain, https://commons.wikimedia.org/w/index.php?curid=82371649
File:Logo of WireGuard.svg. (2020, April 21). Wikimedia Commons, the free media repository. Retrieved 23:20, November 28, 2020 from Wikimedia Commons
Libra logo from By Libra Association - https://libra.org/, Public Domain, https://commons.wikimedia.org/w/index.php?curid=79808006

The context of this talk will be the project fiat crypto, where crypto here is cryptography and not dogecoin. Fiat Crypto is joint work with Andres, Jade, and Adam.
The code we generate is now used in the majority of secure connections from web browsers. Our code implements the elliptic curve cryptography underneath HTTPS which is used wherever you see the secure lock icon. Our code is used in Chrome and Firefox… It was picked up just 3 years after we started work on it, which is highly atypical for academic formal verification projects. I believe a large part of this is the way fiat crypto enables developers to more quickly and confidently innovate using crypto.

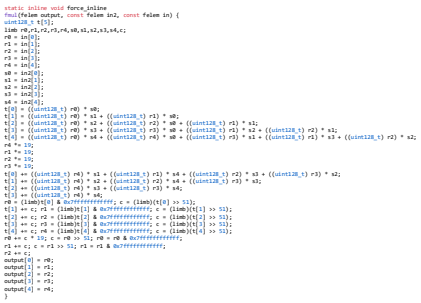
Misc Notes: Fiat Crypto first commit Sep 10, 2015
Fiat Crypto in Chrome Canary by Jan 2018

Innovation with Cryptography

"Better! Faster! Cheaper!"

- Hedging against more powerful attackers
- More mathematical security
- Reduce costs (server & user)

Mathematical Specification: $(a \cdot b) \bmod p$



```
static inline void force_inline
fmul(felem output, const felem in2, const felem in) {
uint128_t t[5];
limb r0,r1,r2,r3,r4,s0,s1,s2,s3,s4,c;
r0 = in[0];
r1 = in[1];
r2 = in[2];
r3 = in[3];
r4 = in[4];
s0 = in2[0];
s1 = in2[1];
s2 = in2[2];
s3 = in2[3];
s4 = in2[4];
t[0] = ((uint128_t) r0) * s0;
t[1] = ((uint128_t) r0) * s1 + ((uint128_t) r1) * s0;
t[2] = ((uint128_t) r0) * s2 + ((uint128_t) r2) * s0 + ((uint128_t) r1) * s1;
t[3] = ((uint128_t) r0) * s3 + ((uint128_t) r3) * s0 + ((uint128_t) r1) * s2 + ((uint128_t) r2) * s1;
t[4] = ((uint128_t) r0) * s4 + ((uint128_t) r4) * s0 + ((uint128_t) r3) * s1 + ((uint128_t) r1) * s3 + ((uint128_t) r2) * s2;
r4 *= 19;
r1 *= 19;
r2 *= 19;
r3 *= 19;
t[0] += ((uint128_t) r4) * s1 + ((uint128_t) r1) * s4 + ((uint128_t) r2) * s3 + ((uint128_t) r3) * s2;
t[1] += ((uint128_t) r4) * s2 + ((uint128_t) r2) * s4 + ((uint128_t) r3) * s3;
t[2] += ((uint128_t) r4) * s3 + ((uint128_t) r3) * s4;
t[3] += ((uint128_t) r4) * s4;
r0 = (limb)t[0] & 0x7ffffffffffff; c = (limb)(t[0] >> 51);
t[1] += c; r1 = (limb)t[1] & 0x7ffffffffffff; c = (limb)(t[1] >> 51);
t[2] += c; r2 = (limb)t[2] & 0x7ffffffffffff; c = (limb)(t[2] >> 51);
t[3] += c; r3 = (limb)t[3] & 0x7ffffffffffff; c = (limb)(t[3] >> 51);
t[4] += c; r4 = (limb)t[4] & 0x7ffffffffffff; c = (limb)(t[4] >> 51);
r0 += c * 19; c = r0 >> 51; r0 = r0 & 0x7ffffffffffff;
r1 += c; c = r1 >> 51; r1 = r1 & 0x7ffffffffffff;
r2 += c;
output[0] = r0;
output[1] = r1;
output[2] = r2;
output[3] = r3;
output[4] = r4;
}
```

"Don't touch it; it works!"

- Lots of room for error
- Enormous cost of error
- Hard to find errors

In crypto, there is often a small mathematical specification such as… . In ECC there are many long and complicated implementations of this spec for reasons of security and performance.

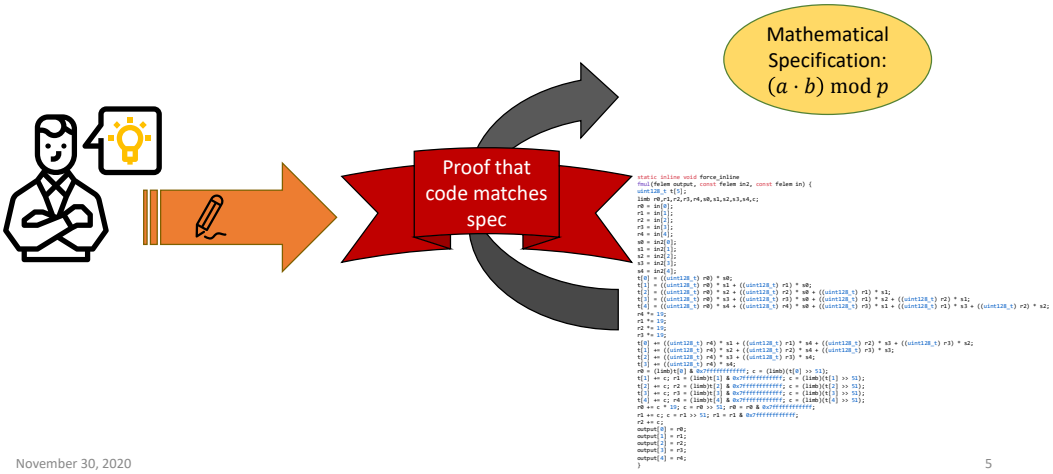
There is good reason to innovate here. As computers get more powerful and attackers have access to more computational power we need to improve the mathematical security of our crypto. Additionally, we can reduce costs, both server-side and user-side, by optimizing the implementations of our cryptography.

However, these optimizations complicate the code, and because the implementation is so far from the specification, there is lot of room for error.

Such errors have an enormous cost because attackers have financial incentives to exploit any bugs.

Since it's so hard to find the errors---even billions of random tests are generally not sufficient---there's an attitude of "don't touch it; it works!" even when this gets in the way of innovation.

The Promise of Verification



November 30, 2020

5

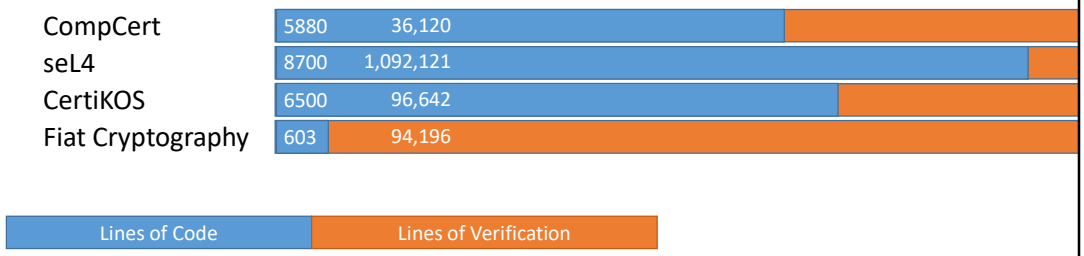
Thinking modified from image by ArmOkay from the Noun Project CC BY 3.0

Verification can fix this!

People, represented by Lightbulbman here, can write proofs that the code matches the spec, and these proofs will potentially let us deploy new crypto with confidence by virtually eliminating the possibility of bugs in the code.

The Overhead of Verification

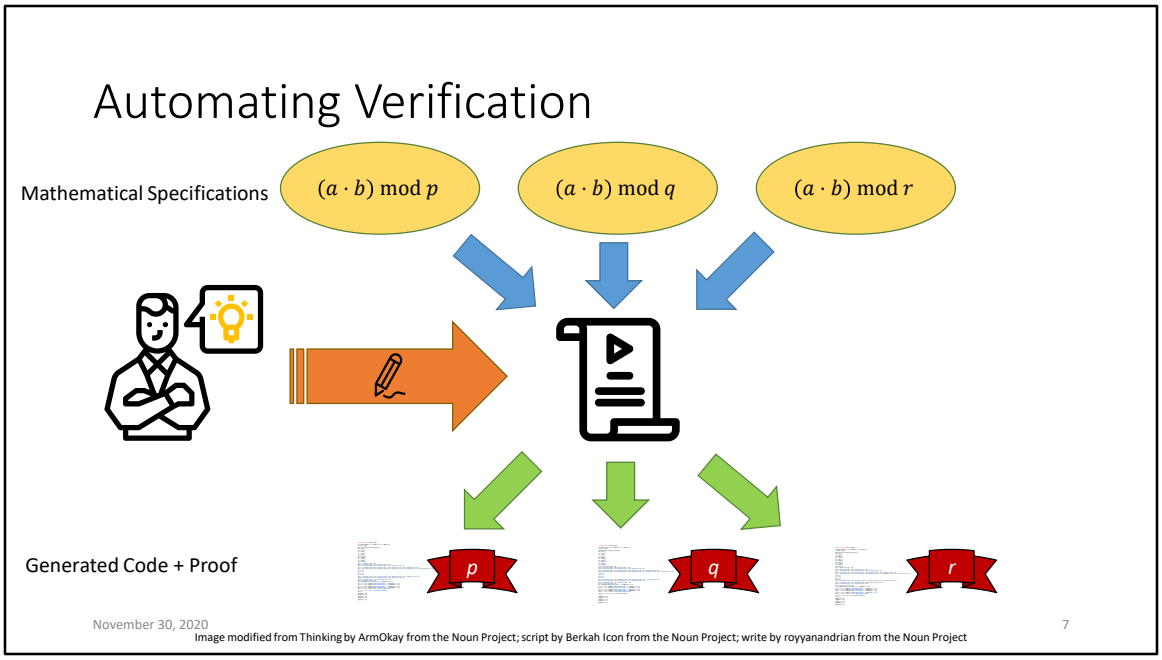
- 10x—100x overhead



November 30, 2020 6

The problem is that there is a 10x to 100x overhead in lines of code of verification over lines of code being verified. Here you can see the numbers for some successful verification projects. This means that when you want to verify a modest program, you need to put in an enormous amount of effort.

Definitions Specification Proofs Verification
seL4 8700 10600 1081521 1092121
CertiKOS 6500 6642 90000 96642
Fiat-Crypto 603 2585 91611 94196
CompCert 5880 4200 31920 36120



Automating verification will fix this problem. One way of automating verification is for lightbulbman here to write a script that generates both the code to be verified and a proof that this code matches the spec.

In this way, when we have many related specifications, as we do in ECC, we can write just one script parametrized over the differences in the specification and thereby pay the upfront cost once and get the verification of a significantly larger amount of code for free. This allows us to significantly reduce the marginal cost of verifying new code.

Our script is run and checked by…

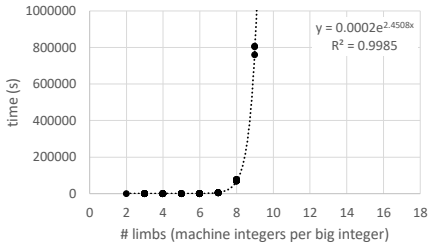
- Dependently typed, interactive, tactic-driven proof assistants
- Dependently typed proof assistants are expressive
- Interactivity allows easy insertion of human ingenuity
- Tactics allow automation

Coq logo from https://calebstanford.com/2019/01/15/coq-vector-image/

… DTITDPAs, like Coq.
…Tactics allow the aforementioned automation of verification.

The Big Problem in Automating Verification

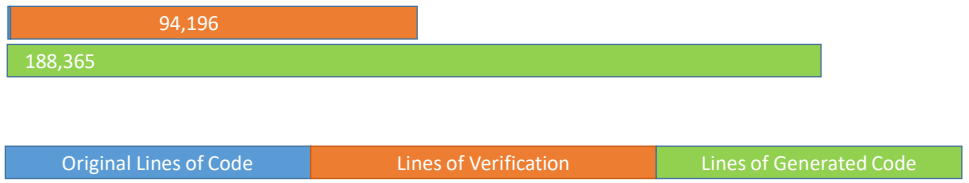
- **Asymptotic performance**
- We can automate verification of toy examples in the proof engine
- BUT this automation takes way too long on real examples
- My work has been fixing this performance problem



Lightbulbman writing the script removes the overhead of writing things over and over. But running the script in the proof assistant still takes too long.
So the big problem in automating verification is asymptotic performance.
(after bullets)
As you see in this plot, in an earlier version of fiat-crypto, the automated verification that worked fine on an example of size 2, taking 17 seconds, on an example of size 17, was projected to take over 4,000 millennia!
Size is measured in the number of machine integers …
My work has …
By the way, none of the performance issues in this talk are about brute force search, they're all about manipulating expressions.

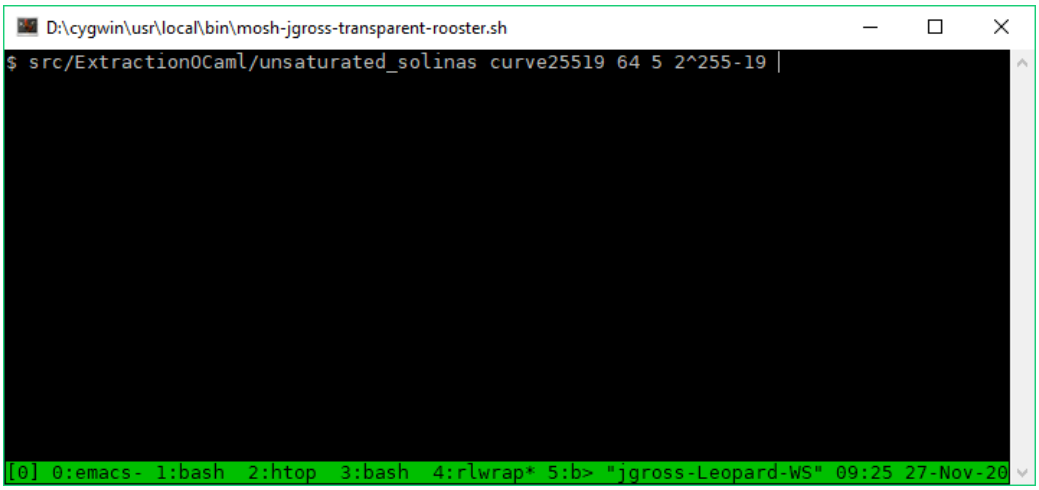
The Potential of Automating Verification

Fiat Cryptography:



After integrating my contribution into fiat crypto we are able to generate 100s of thousands of lines of verified code without needing to do any marginal work. This means that engineers who want to innovate with different crypto parameters can deploy verified code with ease. We're turning engineers into lightbulbmen.

It’s really easy to use!

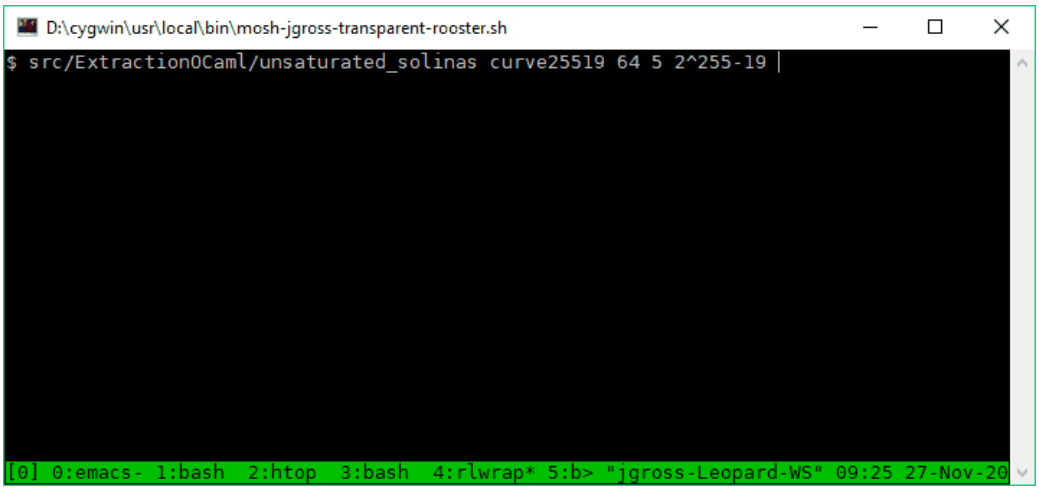


The code we wrote makes a pretty cool tool that can be run on the command line. (DO NOT call the binary “generated”)

Our output artifact can automatically generate verified code on the command line, in seconds (not hours or days or weeks), for given just the prime, the bitwidth, and the name of the high-level algorithm

I think the ease with which our tool can be used contributed significantly to its adoption in industry.

It’s really easy to use!



The code we wrote makes a pretty cool tool that can be run on the command line. (DO NOT call the binary “generated”)

Our output artifact can automatically generate verified code on the command line, in seconds (not hours or days or weeks), for given just the prime, the bitwidth, and the name of the high-level algorithm

I think the ease with which our tool can be used contributed significantly to its adoption in industry.

Requirements

1. Code we generate must be fast and constant time

   Justification: server load, security

2. Easy to add and prove new algorithm, prime, architecture, …

   Justification: scalability of human effort, edit-compile-debug loops

3. Verification should not run forever

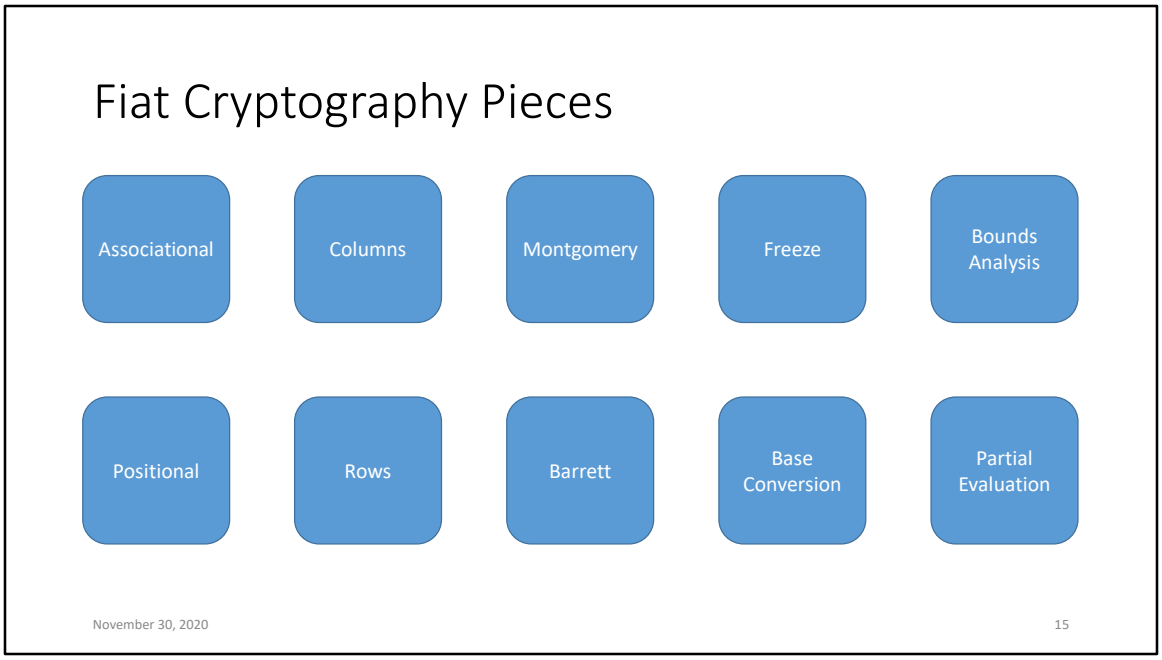
   Justification: usability

The rest of the talk is about the work that went into making Fiat Crypto. If you came here just to support me, you are welcome to go to sleep, I'll let you know when you can wake up.
So, the requirements for fiat crypto were

2nd: this is what enables innovation
3rd: if verification doesn't finish in reasonable time, it's not actually of any use

Alternate 3rd: Verification should complete in reasonable time
Alternate 3rd justification: Needs to be checkable in time for industry deadlines, in time to be usable
Alternate 3rd: Coq should not run forever
Alternate 3rd justification: Obvious

Where was the asymptotic performance issue?



In fiat-crypto, we carved the low-level code up into these neatly-separated conceptually distinct units that you see on the screen, which are small enough to not hit asymptotic issues during interactive verification and during running automated verification. While abstraction is widely used to manage conceptual burden, here, additionally, we use it reduce burden on the proof assistant. While this is not a way in which abstraction being talked about in the literature, this part of the approach not the focus of this talk.
My colleagues worked on most of the individual pieces and I helped them see how this impacts the performance of running the proof script.
I worked primarily on the two compilation stages on the far right. These are the pieces where generate the code from the other pieces.

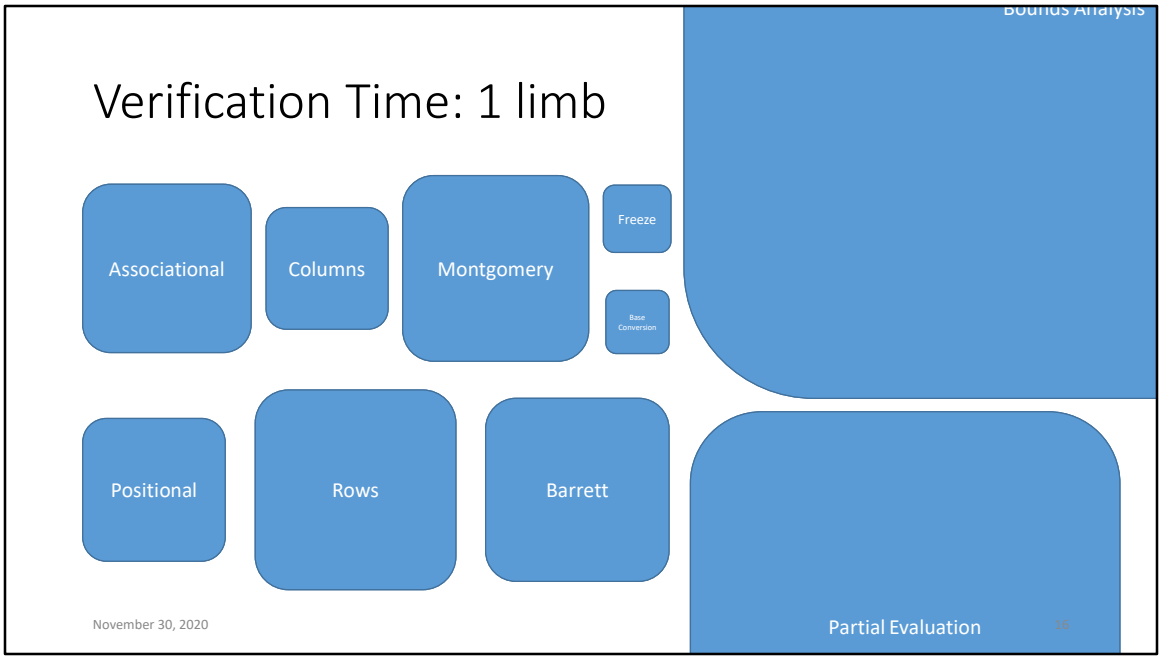
```
$ find src/Arithmetic -name "*.v" | xargs coqwc | sort -h
    spec    proof comments
      27       68       0 src/Arithmetic/CoreExtra.v
      29       75       1 src/Arithmetic/Primitives.v
      32       40       0 src/Arithmetic/SaturatedAssociational.v
      37       77       3 src/Arithmetic/FancyMontgomeryReduction.v
      39       39      36 src/Arithmetic/BarrettReduction/Wikipedia.v
```

```
  49      53      38 src/Arithmetic/BarrettReduction/HAC.v
  52      71       3 src/Arithmetic/Partition.v
  53      75       1 src/Arithmetic/ModularArithmeticPre.v
  58     133       2 src/Arithmetic/UniformWeight.v
  62      99      38 src/Arithmetic/BarrettReduction/Generalized.v
  69       0      91 src/Arithmetic/MontgomeryReduction/Definition.v
  70     163      37 src/Arithmetic/PrimeFieldTheorems.v
 102     152       4 src/Arithmetic/MontgomeryReduction/Proofs.v
 104     172      65 src/Arithmetic/BarrettReduction/RidiculousFish.v
 134     177      12 src/Arithmetic/ModularArithmeticTheorems.v
 150     109       9 src/Arithmetic/SaturatedColumns.v
 181     135      14 src/Arithmetic/Freeze.v
 181      31       3 src/Arithmetic/ModOps.v
 197     335      14 src/Arithmetic/BarrettReduction.v
 215     145      58 src/Arithmetic/CoreAssociational.v
 225     101      21 src/Arithmetic/BaseConversion.v
 266     198      14 src/Arithmetic/CorePositional.v
 357     503      50 src/Arithmetic/BYInv.v
 366     243      25 src/Arithmetic/SaturatedRows.v
 482     721      12 src/Arithmetic/WordByWordMontgomery.v
 512     407      72 src/Arithmetic/Core.v
 552     409      34 src/Arithmetic/Saturated.v
4601    4731     657 total

$ git ls-files "*.v" | grep -v Util | grep -v Demo | xargs coqwc | sort -h
  spec  proof comments
   5     152       3 src/Rewriter/Language/IdentifiersGenerateProofs.v
   9       0       0 src/Rewriter/Rewriter/Examples/PerfTesting/Settings.v
   9      10       0 src/Rewriter/Rewriter/Examples/PerfTesting/ListRectInstances.v
  16     109       0 src/Rewriter/Language/IdentifiersBasicLibrary.v
  18     705      39 src/Rewriter/Language/IdentifiersGenerate.v
  23       0       3 src/Rewriter/Language/PreCommon.v
  23     539      15 src/Rewriter/Rewriter/ProofsCommonTactics.v
  24    1524      12 src/Rewriter/Language/IdentifiersBasicGenerate.v
  44      33       0 src/Rewriter/Language/PreLemmas.v
  58      15       0 src/Rewriter/Rewriter/Examples/PrefixSums.v
  66       3       3 src/Rewriter/Rewriter/Examples.v
  73       0       6 src/Rewriter/Language/Pre.v
 138     191      41 src/Rewriter/Rewriter/Examples/PerfTesting/LiftLetsMap.v
 142      12      40 src/Rewriter/Rewriter/Examples/PerfTesting/UnderLetsPlus0.v
 171     208      14 src/Rewriter/Rewriter/AllTactics.v
 187     320      22 src/Rewriter/Language/IdentifiersLibraryProofs.v
```

```
     203       5      47
src/Rewriter/Rewriter/Examples/PerfTesting/SieveOfEratosthenes.v
     238       0       4 src/Rewriter/Language/UnderLets.v
     273      71       8 src/Rewriter/Rewriter/Examples/PerfTesting/Harness.v
     299       3     117 src/Rewriter/Rewriter/Examples/PerfTesting/Plus0Tree.v
     543     442      15 src/Rewriter/Rewriter/Wf.v
     582     275      26 src/Rewriter/Language/IdentifiersLibrary.v
     662     875       8 src/Rewriter/Rewriter/InterpProofs.v
     780     299       0 src/Rewriter/Language/Inversion.v
     900      17      54 src/Rewriter/Rewriter/Examples/PerfTesting/Sample.v
     962     946      22 src/Rewriter/Language/Wf.v
    1015     141      24 src/Rewriter/Rewriter/Reify.v
    1220     676       6 src/Rewriter/Language/UnderLetsProofs.v
    1310      95      68 src/Rewriter/Rewriter/Rewriter.v
    1630     138      96 src/Rewriter/Language/Language.v
    1813    1908      31 src/Rewriter/Rewriter/ProofsCommon.v
   13436    9712     724 total
$ git ls-files 'src/AbstractInterpretation/*.v' | xargs coqwc
    spec    proof comments
     519       0      23 src/AbstractInterpretation/AbstractInterpretation.v
     697     763       1 src/AbstractInterpretation/Proofs.v
     452     679       8 src/AbstractInterpretation/Wf.v
      24       0       0 src/AbstractInterpretation/WfExtra.v
     736      65      25 src/AbstractInterpretation/ZRange.v
     276     354       2 src/AbstractInterpretation/ZRangeProofs.v
    2704    1861      59 total
```

```
$ echo "defn$(printf '\t')$(coqwc src/Arithmetic/Core.v | head -1)"; for i in $(git ls-files 'src/Arithmetic/*.v'); do echo "$(cat $i | tr '\n' '~' | sed s'/`/,/g' | sed s'/\.[ ~]/`/g' | grep -o 'Definition [^`]*`' | tr '~' '\n' | wc -l)$(printf '\t')$(coqwc $i | tail -1)"; done | sort –h

$ echo "defn$(printf '\t')$(coqwc src/Arithmetic/Core.v | head -1)"; for i in $(git ls-files 'src/Arithmetic/*.v'); do echo "$(cat $i | tr '\n' '~' | sed s'/`/,/g' | sed s'/\.[ ~]/`/g' | grep -o 'Definition [^`]*`' | tr '~' '\n' | wc -l)$(printf '\t')$(coqwc $i | tail -1)"; done | sort -h
defn        spec    proof comments
0          102     152       4 src/Arithmetic/MontgomeryReduction/Proofs.v
0           39      39      36 src/Arithmetic/BarrettReduction/Wikipedia.v
0           62      99      38 src/Arithmetic/BarrettReduction/Generalized.v
```

2	52	71	3 src/Arithmetic/Partition.v
3	181	31	3 src/Arithmetic/ModOps.v
3	49	53	38 src/Arithmetic/BarrettReduction/HAC.v
3	58	133	2 src/Arithmetic/UniformWeight.v
4	134	177	12 src/Arithmetic/ModularArithmeticTheorems.v
7	29	75	1 src/Arithmetic/Primitives.v
7	37	77	3 src/Arithmetic/FancyMontgomeryReduction.v
8	70	163	37 src/Arithmetic/PrimeFieldTheorems.v
18	181	135	14 src/Arithmetic/Freeze.v
35	53	75	1 src/Arithmetic/ModularArithmeticPre.v
39	225	101	21 src/Arithmetic/BaseConversion.v
39	69	0	91 src/Arithmetic/MontgomeryReduction/Definition.v
46	104	172	65 src/Arithmetic/BarrettReduction/RidiculousFish.v
50	197	335	14 src/Arithmetic/BarrettReduction.v
63	482	721	12 src/Arithmetic/WordByWordMontgomery.v
66	357	503	50 src/Arithmetic/BYInv.v
123	552	409	34 src/Arithmetic/Saturated.v
153	512	407	72 src/Arithmetic/Core.v



Here you see the areas of the pieces scaled to match the running times of the proof scripts.
Size of partial eval on ex size 1: 98.616s, 495 rewrites
Size of partial eval on ex size 2: 607.765s, 1269 rewrites

1/2th size
Sizes:
$ python3 -c 'import math; sizes=[("Bounds",326.19), ("Associational",9.79+5.49), ("Rows", 21.61), ("Base Conversion", 2.16), ("BarrettReduction",18.06), ("WordByWordMontgomery", 18.44), ("Positional",10.91), ("Columns",7.94),("Freeze",2.44)]; print("\n".join(f"{r:<20} {math.sqrt(v)/2.0}" for r, v in sizes))'
Bounds 9.030365441110343
Associational 1.9544820285692064
Rows 2.324327859833892
Base Conversion 0.7348469228349535
BarrettReduction 2.124852936087578
WordByWordMontgomery 2.147091055358389
Positional 1.6515144564913744

Columns 1.408900280360537
Freeze 0.7810249675906654

```
git show 4844fa07f958215bbb30bdca58d0dd0c9d927575 | grep 'After \|Total Time\|Arithmetic/\|----' | less -S
(export COQPATH=/home/jgross/Documents/repos/fiat-crypto/coqprime/src:/home/jgross/Documents/repos/fiat-crypto/rupicola/src:/home/jgross/Documents/repos/fiat-crypto/rupicola/bedrock2/bedrock2/src:/home/jgross/Documents/repos/fiat-crypto/rupicola/bedrock2/deps/coqutil/src:/home/jgross/Documents/repos/fiat-crypto/rewriter/src; for i in src/Arithmetic/*.v; do command time -f "${i}o (real: %e, user: %U, sys: %S, mem: %M ko)" "coqc"   -q -w +implicit-core-hint-db,+implicits-in-term,+non-reversible-notation,+deprecated-intros-until-0,+deprecated-focus,+unused-intro-pattern,+variable-collision,-deprecated-hint-constr,-fragile-hint-constr,+omega-is-deprecated,+deprecated-instantiate-syntax,+non-recursive -w -notation-overridden,-undeclared-scope  -R src Crypto $i; done) 2>&1 | tee log

$ grep -o 'Arithmetic/.*user[^,]*' log | sed s'/^\([^ ]*\).*user: \([^ ,]*\).*/\2\t\1/g' | sort -h
0.50    Arithmetic/ModularArithmeticPre.vo
1.24    Arithmetic/CoreExtra.vo
1.33    Arithmetic/Partition.vo
1.56    Arithmetic/PrimeFieldTheorems.vo
1.69    Arithmetic/ModOps.vo
1.80    Arithmetic/ModularArithmeticTheorems.vo
2.16    Arithmetic/BaseConversion.vo
2.44    Arithmetic/Freeze.vo
2.53    Arithmetic/Primitives.vo
4.06    Arithmetic/UniformWeight.vo
5.49    Arithmetic/SaturatedAssociational.vo
5.83    Arithmetic/BYInv.vo
7.94    Arithmetic/SaturatedColumns.vo
9.46    Arithmetic/FancyMontgomeryReduction.vo
9.79    Arithmetic/CoreAssociational.vo
10.91   Arithmetic/CorePositional.vo
17.72   Arithmetic/Core.vo
18.06   Arithmetic/BarrettReduction.vo
18.44   Arithmetic/WordByWordMontgomery.vo
21.61   Arithmetic/SaturatedRows.vo
33.76   Arithmetic/Saturated.vo
```

```
 1m14.22s |  967412 ko | AbstractInterpretation/Proofs.vo                        |
1m12.40s |  967344 ko || +0m01.81s ||       68 ko |   +2.51% |
    2m17.93s |  960828 ko | AbstractInterpretation/Wf.vo                          |
2m18.00s |  960820 ko || -0m00.06s ||        8 ko |   -0.05% |
    1m14.14s | 1131868 ko | AbstractInterpretation/ZRangeProofs.vo                  |
1m14.12s | 1124580 ko || +0m00.01s ||     7288 ko |   +0.02% |
    0m02.17s |  752944 ko | AbstractInterpretation/ZRange.vo                       |
0m02.05s |  750316 ko || +0m00.12s ||     2628 ko |   +5.85% |
    0m01.96s |  563392 ko | AbstractInterpretation/AbstractInterpretation.vo
|  0m02.11s |  562712 ko || -0m00.14s ||      680 ko |   -7.10% |
    0m01.12s |  556432 ko | AbstractInterpretation/WfExtra.vo                      |
0m01.18s |  555212 ko || -0m00.05s ||     1220 ko |   -5.08% |

 1m14.22s |  967412 ko | AbstractInterpretation/Proofs.vo                        |
1m12.40s |  967344 ko || +0m01.81s ||       68 ko |   +2.51% |
    2m17.93s |  960828 ko | AbstractInterpretation/Wf.vo                          |
2m18.00s |  960820 ko || -0m00.06s ||        8 ko |   -0.05% |
    1m14.14s | 1131868 ko | AbstractInterpretation/ZRangeProofs.vo                  |
1m14.12s | 1124580 ko || +0m00.01s ||     7288 ko |   +0.02% |
    0m15.65s |  880424 ko | Util/ZRange/LandLorBounds.vo                           |
0m15.02s |  880216 ko || +0m00.63s ||      208 ko |   +4.19% |
    0m09.34s |  715912 ko | Util/ZRange/CornersMonotoneBounds.vo
|  0m09.37s |  715836 ko || -0m00.02s ||       76 ko |   -0.32% |
    0m06.10s |  707744 ko | Util/ZRange/BasicLemmas.vo                             |
0m06.23s |  708104 ko || -0m00.13s ||     -360 ko |   -2.08% |
    0m02.17s |  752944 ko | AbstractInterpretation/ZRange.vo                       |
0m02.05s |  750316 ko || +0m00.12s ||     2628 ko |   +5.85% |
    0m01.96s |  563392 ko | AbstractInterpretation/AbstractInterpretation.vo
|  0m02.11s |  562712 ko || -0m00.14s ||      680 ko |   -7.10% |
    0m01.80s |  697128 ko | Util/ZRange/SplitRangeBounds.vo                        |
0m01.80s |  697036 ko || +0m00.00s ||       92 ko |   +0.00% |
    0m01.76s |  731112 ko | Util/ZRange/SplitBounds.vo                             |
0m01.90s |  731080 ko || -0m00.13s ||       32 ko |   -7.36% |
    0m01.12s |  556432 ko | AbstractInterpretation/WfExtra.vo                      |
0m01.18s |  555212 ko || -0m00.05s ||     1220 ko |   -5.08% |
    0m00.78s |  493584 ko | Util/ZRange/OperationsBounds.vo                        |
0m00.81s |  493820 ko || -0m00.03s ||     -236 ko |   -3.70% |
    0m00.51s |  456816 ko | Util/ZRange.vo                                  |  0m00.52s
|  456608 ko || -0m00.01s ||      208 ko |   -1.92% |
    0m00.49s |  457600 ko | Util/ZRange/Operations.vo                              |
0m00.49s |  457296 ko || +0m00.00s ||      304 ko |   +0.00% |
    0m00.36s |  421340 ko | Util/ZRange/Show.vo                                    |
```

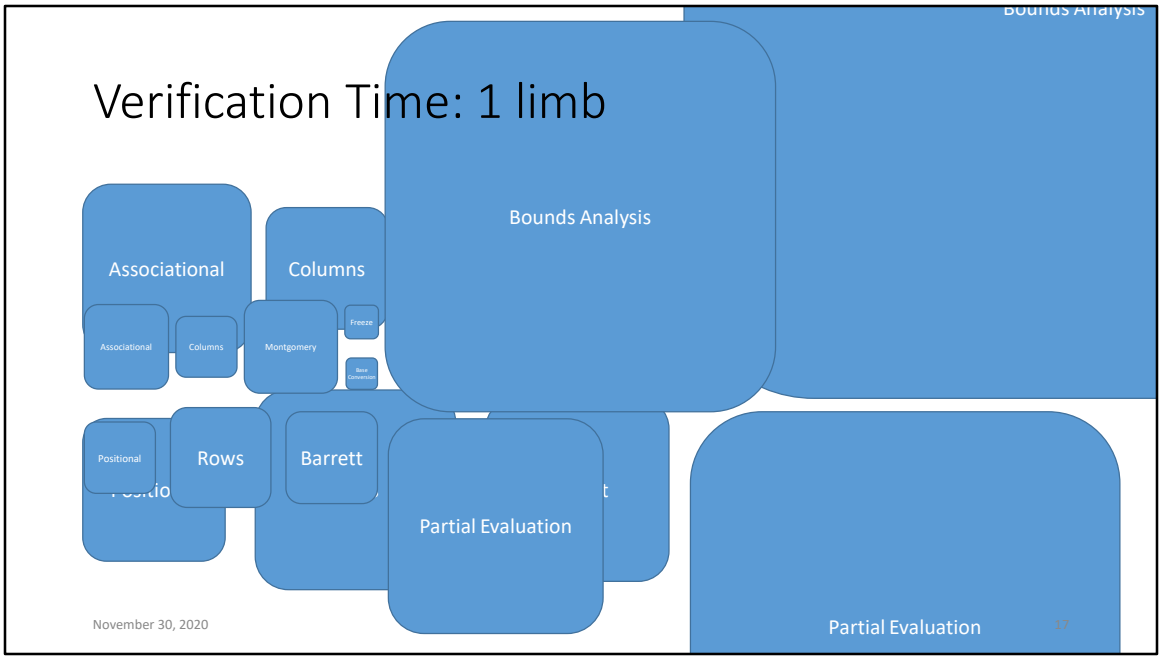
0m00.39s | 421284 ko || -0m00.03s || 56 ko | -7.69% |

1m14.22s+2m17.93s+1m14.14s+0m02.17s+0m01.96s+0m01.12s+0m15.65s+0m09.34s+0m06.10s+0m01.80s+0m01.76s
python -c 'print(sum(m*60+s for m, s in (map(float,i.replace("s","").split("m")) for i in "1m14.22s+2m17.93s+1m14.14s+0m02.17s+0m01.96s+0m01.12s+0m15.65s+0m09.34s+0m06.10s+0m01.80s+0m01.76s".split("+"))))'
326.19

Arithmetic/Saturated.vo	34.62s	5.883876273	
Arithmetic/BarrettReduction.vo		22.54s	4.747630988
Arithmetic/WordByWordMontgomery.vo		22.48s	4.741307836
Arithmetic/Core.vo	18.83s	4.339354791	
Arithmetic/FancyMontgomeryReduction.vo		11.96s	3.458323293
Arithmetic/MontgomeryReduction/Proofs.vo			5.71s
2.389560629			
Arithmetic/UniformWeight.vo		4.69s	2.165640783
Arithmetic/BarrettReduction/Generalized.vo			4.62s
2.149418526			
Arithmetic/BarrettReduction/HAC.vo		4.10s	2.024845673
Arithmetic/Freeze.vo	3.04s	1.743559577	
Arithmetic/Primitives.vo	3.04s	1.743559577	
Arithmetic/BarrettReduction/RidiculousFish.vo			2.75s
1.658312395			
Arithmetic/BaseConversion.vo		2.51s	1.584297952
Arithmetic/ModOps.vo	2.44s	1.562049935	
Arithmetic/ModularArithmeticTheorems.vo		2.28s	1.509966887
Arithmetic/Partition.vo	1.75s	1.322875656	
Arithmetic/PrimeFieldTheorems.vo		1.74s	1.319090596
Arithmetic/BarrettReduction/Wikipedia.vo		1.20s	1.095445115
Arithmetic/ModularArithmeticPre.vo		0.55s	0.741619849
Arithmetic/MontgomeryReduction/Definition.vo			0.37s
0.608276253			

TODO: scale by time taken, include estimates for rewriting-based
TODO: how to time-estimate bounds analysis?

When we're using the proof engine for partial evaluation, the time the proof engine takes to run the proof script just keeps growing and has unacceptable asymptotics



Size of partial eval on ex size 1: 98.616s, 495 rewrites
Size of partial eval on ex size 2: 607.765s, 1269 rewrites

1/2nd to 1/4th size
Sizes:
$ python3 -c 'import math; sizes=[("Bounds",326.19), ("Associational",9.79+5.49), ("Rows", 21.61), ("Base Conversion", 2.16), ("BarrettReduction",18.06), ("WordByWordMontgomery", 18.44), ("Positional",10.91), ("Columns",7.94),("Freeze",2.44)]; print("\n".join(f"{r:<20} {math.sqrt(v)/2.0}" for r, v in sizes))'
Bounds 9.030365441110343
Associational 1.9544820285692064
Rows 2.324327859833892
Base Conversion 0.7348469228349535
BarrettReduction 2.124852936087578
WordByWordMontgomery 2.147091055358389
Positional 1.6515144564913744
Columns 1.408900280360537
Freeze 0.7810249675906654

```
git show 4844fa07f958215bbb30bdca58d0dd0c9d927575 | grep 'After \|Total Time\|Arithmetic/\|----' | less -S
(export COQPATH=/home/jgross/Documents/repos/fiat-crypto/coqprime/src:/home/jgross/Documents/repos/fiat-crypto/rupicola/src:/home/jgross/Documents/repos/fiat-crypto/rupicola/bedrock2/bedrock2/src:/home/jgross/Documents/repos/fiat-crypto/rupicola/bedrock2/deps/coqutil/src:/home/jgross/Documents/repos/fiat-crypto/rewriter/src; for i in src/Arithmetic/*.v; do command time -f "${i}o (real: %e, user: %U, sys: %S, mem: %M ko)" "coqc"   -q -w +implicit-core-hint-db,+implicits-in-term,+non-reversible-notation,+deprecated-intros-until-0,+deprecated-focus,+unused-intro-pattern,+variable-collision,-deprecated-hint-constr,-fragile-hint-constr,+omega-is-deprecated,+deprecated-instantiate-syntax,+non-recursive -w -notation-overridden,-undeclared-scope  -R src Crypto $i; done) 2>&1 | tee log

$ grep -o 'Arithmetic/.*user[^,]*' log | sed s'/^\([^ ]*\).*user: \([^ ,]*\).*/\2\t\1/g' | sort -h
0.50    Arithmetic/ModularArithmeticPre.vo
1.24    Arithmetic/CoreExtra.vo
1.33    Arithmetic/Partition.vo
1.56    Arithmetic/PrimeFieldTheorems.vo
1.69    Arithmetic/ModOps.vo
1.80    Arithmetic/ModularArithmeticTheorems.vo
2.16    Arithmetic/BaseConversion.vo
2.44    Arithmetic/Freeze.vo
2.53    Arithmetic/Primitives.vo
4.06    Arithmetic/UniformWeight.vo
5.49    Arithmetic/SaturatedAssociational.vo
5.83    Arithmetic/BYInv.vo
7.94    Arithmetic/SaturatedColumns.vo
9.46    Arithmetic/FancyMontgomeryReduction.vo
9.79    Arithmetic/CoreAssociational.vo
10.91   Arithmetic/CorePositional.vo
17.72   Arithmetic/Core.vo
18.06   Arithmetic/BarrettReduction.vo
18.44   Arithmetic/WordByWordMontgomery.vo
21.61   Arithmetic/SaturatedRows.vo
33.76   Arithmetic/Saturated.vo

 1m14.22s |  967412 ko | AbstractInterpretation/Proofs.vo                    |
1m12.40s |  967344 ko || +0m01.81s ||       68 ko |   +2.51% |
```

2m17.93s | 960828 ko | AbstractInterpretation/Wf.vo |
2m18.00s | 960820 ko || -0m00.06s || 8 ko | -0.05% |
1m14.14s | 1131868 ko | AbstractInterpretation/ZRangeProofs.vo |
1m14.12s | 1124580 ko || +0m00.01s || 7288 ko | +0.02% |
0m02.17s | 752944 ko | AbstractInterpretation/ZRange.vo |
0m02.05s | 750316 ko || +0m00.12s || 2628 ko | +5.85% |
0m01.96s | 563392 ko | AbstractInterpretation/AbstractInterpretation.vo
| 0m02.11s | 562712 ko || -0m00.14s || 680 ko | -7.10% |
0m01.12s | 556432 ko | AbstractInterpretation/WfExtra.vo |
0m01.18s | 555212 ko || -0m00.05s || 1220 ko | -5.08% |

1m14.22s | 967412 ko | AbstractInterpretation/Proofs.vo |
1m12.40s | 967344 ko || +0m01.81s || 68 ko | +2.51% |
2m17.93s | 960828 ko | AbstractInterpretation/Wf.vo |
2m18.00s | 960820 ko || -0m00.06s || 8 ko | -0.05% |
1m14.14s | 1131868 ko | AbstractInterpretation/ZRangeProofs.vo |
1m14.12s | 1124580 ko || +0m00.01s || 7288 ko | +0.02% |
0m15.65s | 880424 ko | Util/ZRange/LandLorBounds.vo |
0m15.02s | 880216 ko || +0m00.63s || 208 ko | +4.19% |
0m09.34s | 715912 ko | Util/ZRange/CornersMonotoneBounds.vo
| 0m09.37s | 715836 ko || -0m00.02s || 76 ko | -0.32% |
0m06.10s | 707744 ko | Util/ZRange/BasicLemmas.vo |
0m06.23s | 708104 ko || -0m00.13s || -360 ko | -2.08% |
0m02.17s | 752944 ko | AbstractInterpretation/ZRange.vo |
0m02.05s | 750316 ko || +0m00.12s || 2628 ko | +5.85% |
0m01.96s | 563392 ko | AbstractInterpretation/AbstractInterpretation.vo
| 0m02.11s | 562712 ko || -0m00.14s || 680 ko | -7.10% |
0m01.80s | 697128 ko | Util/ZRange/SplitRangeBounds.vo |
0m01.80s | 697036 ko || +0m00.00s || 92 ko | +0.00% |
0m01.76s | 731112 ko | Util/ZRange/SplitBounds.vo |
0m01.90s | 731080 ko || -0m00.13s || 32 ko | -7.36% |
0m01.12s | 556432 ko | AbstractInterpretation/WfExtra.vo |
0m01.18s | 555212 ko || -0m00.05s || 1220 ko | -5.08% |
0m00.78s | 493584 ko | Util/ZRange/OperationsBounds.vo |
0m00.81s | 493820 ko || -0m00.03s || -236 ko | -3.70% |
0m00.51s | 456816 ko | Util/ZRange.vo | 0m00.52s
| 456608 ko || -0m00.01s || 208 ko | -1.92% |
0m00.49s | 457600 ko | Util/ZRange/Operations.vo |
0m00.49s | 457296 ko || +0m00.00s || 304 ko | +0.00% |
0m00.36s | 421340 ko | Util/ZRange/Show.vo |
0m00.39s | 421284 ko || -0m00.03s || 56 ko | -7.69% |

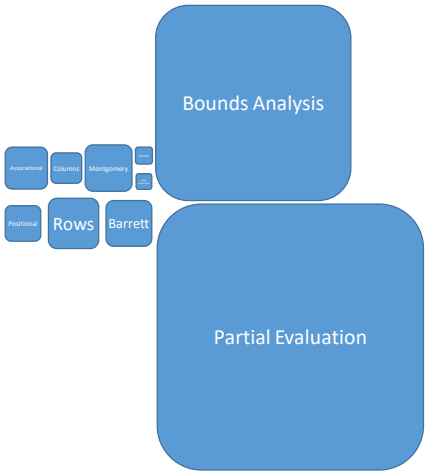
1m14.22s+2m17.93s+1m14.14s+0m02.17s+0m01.96s+0m01.12s+0m15.65s+0m09.34s+0m06.10s+0m01.80s+0m01.76s
python -c 'print(sum(m*60+s for m, s in (map(float,i.replace("s","").split("m")) for i in "1m14.22s+2m17.93s+1m14.14s+0m02.17s+0m01.96s+0m01.12s+0m15.65s+0m09.34s+0m06.10s+0m01.80s+0m01.76s".split("+"))))'
326.19

Arithmetic/Saturated.vo	34.62s	5.883876273	
Arithmetic/BarrettReduction.vo		22.54s	4.747630988
Arithmetic/WordByWordMontgomery.vo		22.48s	4.741307836
Arithmetic/Core.vo	18.83s	4.339354791	
Arithmetic/FancyMontgomeryReduction.vo		11.96s	3.458323293
Arithmetic/MontgomeryReduction/Proofs.vo			5.71s
2.389560629			
Arithmetic/UniformWeight.vo		4.69s	2.165640783
Arithmetic/BarrettReduction/Generalized.vo			4.62s
2.149418526			
Arithmetic/BarrettReduction/HAC.vo		4.10s	2.024845673
Arithmetic/Freeze.vo	3.04s	1.743559577	
Arithmetic/Primitives.vo	3.04s	1.743559577	
Arithmetic/BarrettReduction/RidiculousFish.vo			2.75s
1.658312395			
Arithmetic/BaseConversion.vo		2.51s	1.584297952
Arithmetic/ModOps.vo	2.44s	1.562049935	
Arithmetic/ModularArithmeticTheorems.vo		2.28s	1.509966887
Arithmetic/Partition.vo	1.75s	1.322875656	
Arithmetic/PrimeFieldTheorems.vo		1.74s	1.319090596
Arithmetic/BarrettReduction/Wikipedia.vo		1.20s	1.095445115
Arithmetic/ModularArithmeticPre.vo		0.55s	0.741619849
Arithmetic/MontgomeryReduction/Definition.vo			0.37s
0.608276253			

TODO: scale by time taken, include estimates for rewriting-based
TODO: how to time-estimate bounds analysis?

When we're using the proof engine for partial evaluation, the time the proof engine takes to run the proof script just keeps growing and has unacceptable asymptotics

Verification Time: 2 limbs



November 30, 2020 18

Size of partial eval on ex size 1: 98.616s, 495 rewrites
Size of partial eval on ex size 2: 607.765s, 1269 rewrites

1/8th size
Sizes:
$ python3 -c 'import math; sizes=[("Bounds",326.19), ("Associational",9.79+5.49), ("Rows", 21.61), ("Base Conversion", 2.16), ("BarrettReduction",18.06), ("WordByWordMontgomery", 18.44), ("Positional",10.91), ("Columns",7.94),("Freeze",2.44)]; print("\n".join(f"{r:<20} {math.sqrt(v)/2.0}" for r, v in sizes))'
Bounds 9.030365441110343
Associational 1.9544820285692064
Rows 2.324327859833892
Base Conversion 0.7348469228349535
BarrettReduction 2.124852936087578
WordByWordMontgomery 2.147091055358389
Positional 1.6515144564913744
Columns 1.408900280360537
Freeze 0.7810249675906654

```
git show 4844fa07f958215bbb30bdca58d0dd0c9d927575 | grep 'After \|Total Time\|Arithmetic/\|----' | less -S
(export COQPATH=/home/jgross/Documents/repos/fiat-crypto/coqprime/src:/home/jgross/Documents/repos/fiat-crypto/rupicola/src:/home/jgross/Documents/repos/fiat-crypto/rupicola/bedrock2/bedrock2/src:/home/jgross/Documents/repos/fiat-crypto/rupicola/bedrock2/deps/coqutil/src:/home/jgross/Documents/repos/fiat-crypto/rewriter/src; for i in src/Arithmetic/*.v; do command time -f "${i}o (real: %e, user: %U, sys: %S, mem: %M ko)" "coqc"   -q -w +implicit-core-hint-db,+implicits-in-term,+non-reversible-notation,+deprecated-intros-until-0,+deprecated-focus,+unused-intro-pattern,+variable-collision,-deprecated-hint-constr,-fragile-hint-constr,+omega-is-deprecated,+deprecated-instantiate-syntax,+non-recursive -w -notation-overridden,-undeclared-scope  -R src Crypto $i; done) 2>&1 | tee log

$ grep -o 'Arithmetic/.*user[^,]*' log | sed s'/^\([^ ]*\).*user: \([^ ,]*\).*/\2\t\1/g' | sort -h
0.50    Arithmetic/ModularArithmeticPre.vo
1.24    Arithmetic/CoreExtra.vo
1.33    Arithmetic/Partition.vo
1.56    Arithmetic/PrimeFieldTheorems.vo
1.69    Arithmetic/ModOps.vo
1.80    Arithmetic/ModularArithmeticTheorems.vo
2.16    Arithmetic/BaseConversion.vo
2.44    Arithmetic/Freeze.vo
2.53    Arithmetic/Primitives.vo
4.06    Arithmetic/UniformWeight.vo
5.49    Arithmetic/SaturatedAssociational.vo
5.83    Arithmetic/BYInv.vo
7.94    Arithmetic/SaturatedColumns.vo
9.46    Arithmetic/FancyMontgomeryReduction.vo
9.79    Arithmetic/CoreAssociational.vo
10.91   Arithmetic/CorePositional.vo
17.72   Arithmetic/Core.vo
18.06   Arithmetic/BarrettReduction.vo
18.44   Arithmetic/WordByWordMontgomery.vo
21.61   Arithmetic/SaturatedRows.vo
33.76   Arithmetic/Saturated.vo

 1m14.22s |  967412 ko | AbstractInterpretation/Proofs.vo                        |
1m12.40s |  967344 ko || +0m01.81s ||       68 ko |   +2.51% |
```

2m17.93s | 960828 ko | AbstractInterpretation/Wf.vo |
2m18.00s | 960820 ko || -0m00.06s || 8 ko | -0.05% |
1m14.14s | 1131868 ko | AbstractInterpretation/ZRangeProofs.vo |
1m14.12s | 1124580 ko || +0m00.01s || 7288 ko | +0.02% |
0m02.17s | 752944 ko | AbstractInterpretation/ZRange.vo |
0m02.05s | 750316 ko || +0m00.12s || 2628 ko | +5.85% |
0m01.96s | 563392 ko | AbstractInterpretation/AbstractInterpretation.vo
| 0m02.11s | 562712 ko || -0m00.14s || 680 ko | -7.10% |
0m01.12s | 556432 ko | AbstractInterpretation/WfExtra.vo |
0m01.18s | 555212 ko || -0m00.05s || 1220 ko | -5.08% |

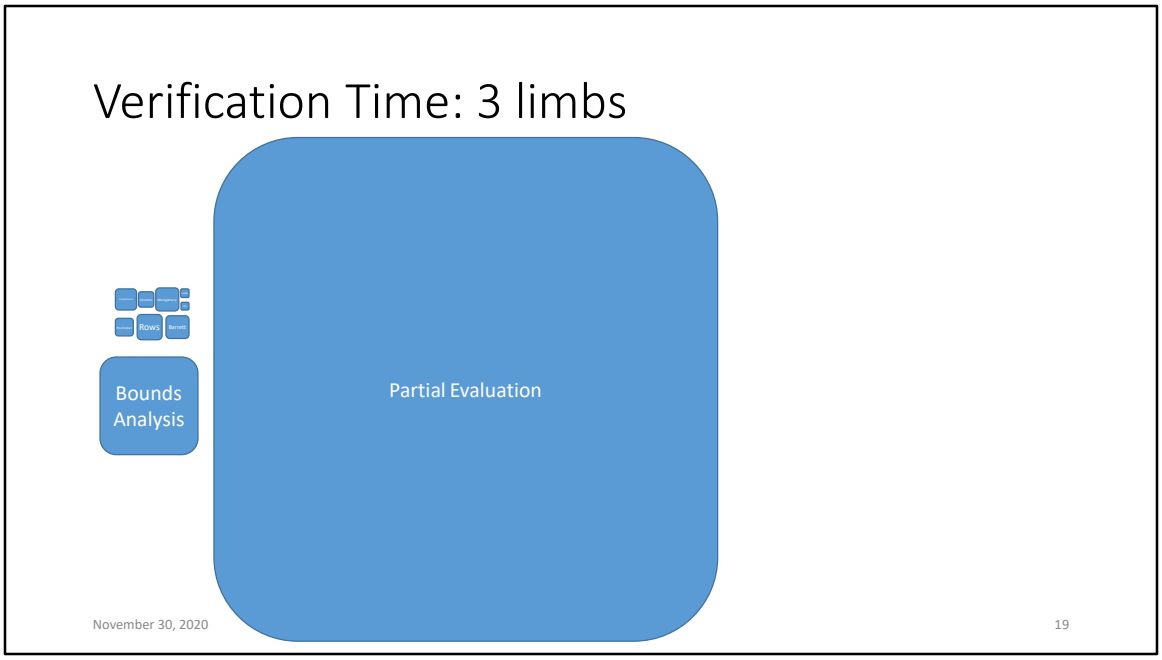
1m14.22s | 967412 ko | AbstractInterpretation/Proofs.vo |
1m12.40s | 967344 ko || +0m01.81s || 68 ko | +2.51% |
2m17.93s | 960828 ko | AbstractInterpretation/Wf.vo |
2m18.00s | 960820 ko || -0m00.06s || 8 ko | -0.05% |
1m14.14s | 1131868 ko | AbstractInterpretation/ZRangeProofs.vo |
1m14.12s | 1124580 ko || +0m00.01s || 7288 ko | +0.02% |
0m15.65s | 880424 ko | Util/ZRange/LandLorBounds.vo |
0m15.02s | 880216 ko || +0m00.63s || 208 ko | +4.19% |
0m09.34s | 715912 ko | Util/ZRange/CornersMonotoneBounds.vo
| 0m09.37s | 715836 ko || -0m00.02s || 76 ko | -0.32% |
0m06.10s | 707744 ko | Util/ZRange/BasicLemmas.vo |
0m06.23s | 708104 ko || -0m00.13s || -360 ko | -2.08% |
0m02.17s | 752944 ko | AbstractInterpretation/ZRange.vo |
0m02.05s | 750316 ko || +0m00.12s || 2628 ko | +5.85% |
0m01.96s | 563392 ko | AbstractInterpretation/AbstractInterpretation.vo
| 0m02.11s | 562712 ko || -0m00.14s || 680 ko | -7.10% |
0m01.80s | 697128 ko | Util/ZRange/SplitRangeBounds.vo |
0m01.80s | 697036 ko || +0m00.00s || 92 ko | +0.00% |
0m01.76s | 731112 ko | Util/ZRange/SplitBounds.vo |
0m01.90s | 731080 ko || -0m00.13s || 32 ko | -7.36% |
0m01.12s | 556432 ko | AbstractInterpretation/WfExtra.vo |
0m01.18s | 555212 ko || -0m00.05s || 1220 ko | -5.08% |
0m00.78s | 493584 ko | Util/ZRange/OperationsBounds.vo |
0m00.81s | 493820 ko || -0m00.03s || -236 ko | -3.70% |
0m00.51s | 456816 ko | Util/ZRange.vo | 0m00.52s
| 456608 ko || -0m00.01s || 208 ko | -1.92% |
0m00.49s | 457600 ko | Util/ZRange/Operations.vo |
0m00.49s | 457296 ko || +0m00.00s || 304 ko | +0.00% |
0m00.36s | 421340 ko | Util/ZRange/Show.vo |
0m00.39s | 421284 ko || -0m00.03s || 56 ko | -7.69% |

1m14.22s+2m17.93s+1m14.14s+0m02.17s+0m01.96s+0m01.12s+0m15.65s+0m09.34s+0m06.10s+0m01.80s+0m01.76s
python -c 'print(sum(m*60+s for m, s in (map(float,i.replace("s","").split("m")) for i in "1m14.22s+2m17.93s+1m14.14s+0m02.17s+0m01.96s+0m01.12s+0m15.65s+0m09.34s+0m06.10s+0m01.80s+0m01.76s".split("+"))))'
326.19

Arithmetic/Saturated.vo	34.62s	5.883876273	
Arithmetic/BarrettReduction.vo		22.54s	4.747630988
Arithmetic/WordByWordMontgomery.vo		22.48s	4.741307836
Arithmetic/Core.vo	18.83s	4.339354791	
Arithmetic/FancyMontgomeryReduction.vo		11.96s	3.458323293
Arithmetic/MontgomeryReduction/Proofs.vo			5.71s
2.389560629			
Arithmetic/UniformWeight.vo		4.69s	2.165640783
Arithmetic/BarrettReduction/Generalized.vo			4.62s
2.149418526			
Arithmetic/BarrettReduction/HAC.vo		4.10s	2.024845673
Arithmetic/Freeze.vo	3.04s	1.743559577	
Arithmetic/Primitives.vo	3.04s	1.743559577	
Arithmetic/BarrettReduction/RidiculousFish.vo			2.75s
1.658312395			
Arithmetic/BaseConversion.vo		2.51s	1.584297952
Arithmetic/ModOps.vo	2.44s	1.562049935	
Arithmetic/ModularArithmeticTheorems.vo	2.28s		1.509966887
Arithmetic/Partition.vo	1.75s	1.322875656	
Arithmetic/PrimeFieldTheorems.vo		1.74s	1.319090596
Arithmetic/BarrettReduction/Wikipedia.vo		1.20s	1.095445115
Arithmetic/ModularArithmeticPre.vo		0.55s	0.741619849
Arithmetic/MontgomeryReduction/Definition.vo			0.37s
0.608276253			

TODO: scale by time taken, include estimates for rewriting-based
TODO: how to time-estimate bounds analysis?

When we're using the proof engine for partial evaluation, the time the proof engine takes to run the proof script just keeps growing and has unacceptable asymptotics



Now at 2h25m, nearly 60 GB RAM
This “partial evaluation” piece of fiat-crypto had inadequate asymptotics when done in the proof engine.
Size of partial eval on ex size 1: 98.616s, 495 rewrites
Size of partial eval on ex size 2: 607.765s, 1269 rewrites
Size of partial eval on ex size 3: 8706.089s, 2398 rewrites, mem: 58,206,840 ko, (2^130-5, x64)

1/16th size
Sizes:
$ python3 -c 'import math; sizes=[("Bounds",326.19), ("Associational",9.79+5.49), ("Rows", 21.61), ("Base Conversion", 2.16), ("BarrettReduction",18.06), ("WordByWordMontgomery", 18.44), ("Positional",10.91), ("Columns",7.94),("Freeze",2.44)]; print("\n".join(f"{r:<20} {math.sqrt(v)/2.0}" for r, v in sizes))'
Bounds 9.030365441110343
Associational 1.9544820285692064
Rows 2.324327859833892

Base Conversion 0.7348469228349535
BarrettReduction 2.124852936087578
WordByWordMontgomery 2.147091055358389
Positional 1.6515144564913744
Columns 1.408900280360537
Freeze 0.7810249675906654

git show 4844fa07f958215bbb30bdca58d0dd0c9d927575 | grep 'After \|Total Time\|Arithmetic/\|----' | less -S
(export COQPATH=/home/jgross/Documents/repos/fiat-crypto/coqprime/src:/home/jgross/Documents/repos/fiat-crypto/rupicola/src:/home/jgross/Documents/repos/fiat-crypto/rupicola/bedrock2/bedrock2/src:/home/jgross/Documents/repos/fiat-crypto/rupicola/bedrock2/deps/coqutil/src:/home/jgross/Documents/repos/fiat-crypto/rewriter/src; for i in src/Arithmetic/*.v; do command time -f "${i}o (real: %e, user: %U, sys: %S, mem: %M ko)" "coqc" -q -w +implicit-core-hint-db,+implicits-in-term,+non-reversible-notation,+deprecated-intros-until-0,+deprecated-focus,+unused-intro-pattern,+variable-collision,-deprecated-hint-constr,-fragile-hint-constr,+omega-is-deprecated,+deprecated-instantiate-syntax,+non-recursive -w -notation-overridden,-undeclared-scope -R src Crypto $i; done) 2>&1 | tee log

$ grep -o 'Arithmetic/.*user[^,]*' log | sed s'/^\([^]*\).*user: \([^ ,]*\).*/\2\t\1/g' | sort -h
0.50 Arithmetic/ModularArithmeticPre.vo
1.24 Arithmetic/CoreExtra.vo
1.33 Arithmetic/Partition.vo
1.56 Arithmetic/PrimeFieldTheorems.vo
1.69 Arithmetic/ModOps.vo
1.80 Arithmetic/ModularArithmeticTheorems.vo
2.16 Arithmetic/BaseConversion.vo
2.44 Arithmetic/Freeze.vo
2.53 Arithmetic/Primitives.vo
4.06 Arithmetic/UniformWeight.vo
5.49 Arithmetic/SaturatedAssociational.vo
5.83 Arithmetic/BYInv.vo
7.94 Arithmetic/SaturatedColumns.vo
9.46 Arithmetic/FancyMontgomeryReduction.vo
9.79 Arithmetic/CoreAssociational.vo
10.91 Arithmetic/CorePositional.vo
17.72 Arithmetic/Core.vo
18.06 Arithmetic/BarrettReduction.vo

18.44 Arithmetic/WordByWordMontgomery.vo
21.61 Arithmetic/SaturatedRows.vo
33.76 Arithmetic/Saturated.vo

1m14.22s | 967412 ko | AbstractInterpretation/Proofs.vo |
1m12.40s | 967344 ko || +0m01.81s || 68 ko | +2.51% |
2m17.93s | 960828 ko | AbstractInterpretation/Wf.vo |
2m18.00s | 960820 ko || -0m00.06s || 8 ko | -0.05% |
1m14.14s | 1131868 ko | AbstractInterpretation/ZRangeProofs.vo |
1m14.12s | 1124580 ko || +0m00.01s || 7288 ko | +0.02% |
0m02.17s | 752944 ko | AbstractInterpretation/ZRange.vo |
0m02.05s | 750316 ko || +0m00.12s || 2628 ko | +5.85% |
0m01.96s | 563392 ko | AbstractInterpretation/AbstractInterpretation.vo
| 0m02.11s | 562712 ko || -0m00.14s || 680 ko | -7.10% |
0m01.12s | 556432 ko | AbstractInterpretation/WfExtra.vo |
0m01.18s | 555212 ko || -0m00.05s || 1220 ko | -5.08% |

1m14.22s | 967412 ko | AbstractInterpretation/Proofs.vo |
1m12.40s | 967344 ko || +0m01.81s || 68 ko | +2.51% |
2m17.93s | 960828 ko | AbstractInterpretation/Wf.vo |
2m18.00s | 960820 ko || -0m00.06s || 8 ko | -0.05% |
1m14.14s | 1131868 ko | AbstractInterpretation/ZRangeProofs.vo |
1m14.12s | 1124580 ko || +0m00.01s || 7288 ko | +0.02% |
0m15.65s | 880424 ko | Util/ZRange/LandLorBounds.vo |
0m15.02s | 880216 ko || +0m00.63s || 208 ko | +4.19% |
0m09.34s | 715912 ko | Util/ZRange/CornersMonotoneBounds.vo
| 0m09.37s | 715836 ko || -0m00.02s || 76 ko | -0.32% |
0m06.10s | 707744 ko | Util/ZRange/BasicLemmas.vo |
0m06.23s | 708104 ko || -0m00.13s || -360 ko | -2.08% |
0m02.17s | 752944 ko | AbstractInterpretation/ZRange.vo |
0m02.05s | 750316 ko || +0m00.12s || 2628 ko | +5.85% |
0m01.96s | 563392 ko | AbstractInterpretation/AbstractInterpretation.vo
| 0m02.11s | 562712 ko || -0m00.14s || 680 ko | -7.10% |
0m01.80s | 697128 ko | Util/ZRange/SplitRangeBounds.vo |
0m01.80s | 697036 ko || +0m00.00s || 92 ko | +0.00% |
0m01.76s | 731112 ko | Util/ZRange/SplitBounds.vo |
0m01.90s | 731080 ko || -0m00.13s || 32 ko | -7.36% |
0m01.12s | 556432 ko | AbstractInterpretation/WfExtra.vo |
0m01.18s | 555212 ko || -0m00.05s || 1220 ko | -5.08% |
0m00.78s | 493584 ko | Util/ZRange/OperationsBounds.vo |
0m00.81s | 493820 ko || -0m00.03s || -236 ko | -3.70% |
0m00.51s | 456816 ko | Util/ZRange.vo | 0m00.52s

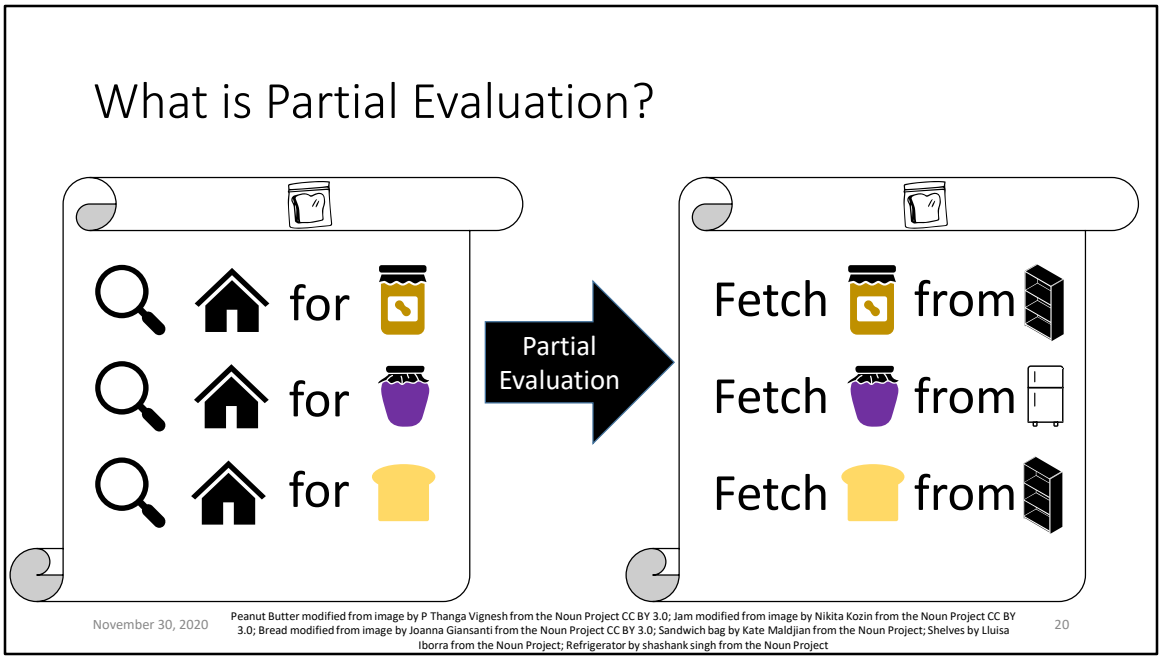
| 456608 ko || -0m00.01s || 208 ko | -1.92% |
 0m00.49s | 457600 ko | Util/ZRange/Operations.vo |
0m00.49s | 457296 ko || +0m00.00s || 304 ko | +0.00% |
 0m00.36s | 421340 ko | Util/ZRange/Show.vo |
0m00.39s | 421284 ko || -0m00.03s || 56 ko | -7.69% |

1m14.22s+2m17.93s+1m14.14s+0m02.17s+0m01.96s+0m01.12s+0m15.65s+0m09.34s+0m06.10s+0m01.80s+0m01.76s
python -c 'print(sum(m*60+s for m, s in (map(float,i.replace("s","").split("m")) for i in "1m14.22s+2m17.93s+1m14.14s+0m02.17s+0m01.96s+0m01.12s+0m15.65s+0m09.34s+0m06.10s+0m01.80s+0m01.76s".split("+"))))'
326.19

Arithmetic/Saturated.vo	34.62s	5.883876273
Arithmetic/BarrettReduction.vo	22.54s	4.747630988
Arithmetic/WordByWordMontgomery.vo	22.48s	4.741307836
Arithmetic/Core.vo	18.83s	4.339354791
Arithmetic/FancyMontgomeryReduction.vo	11.96s	3.458323293
Arithmetic/MontgomeryReduction/Proofs.vo	5.71s	2.389560629
Arithmetic/UniformWeight.vo	4.69s	2.165640783
Arithmetic/BarrettReduction/Generalized.vo	4.62s	2.149418526
Arithmetic/BarrettReduction/HAC.vo	4.10s	2.024845673
Arithmetic/Freeze.vo	3.04s	1.743559577
Arithmetic/Primitives.vo	3.04s	1.743559577
Arithmetic/BarrettReduction/RidiculousFish.vo	2.75s	1.658312395
Arithmetic/BaseConversion.vo	2.51s	1.584297952
Arithmetic/ModOps.vo	2.44s	1.562049935
Arithmetic/ModularArithmeticTheorems.vo	2.28s	1.509966887
Arithmetic/Partition.vo	1.75s	1.322875656
Arithmetic/PrimeFieldTheorems.vo	1.74s	1.319090596
Arithmetic/BarrettReduction/Wikipedia.vo	1.20s	1.095445115
Arithmetic/ModularArithmeticPre.vo	0.55s	0.741619849
Arithmetic/MontgomeryReduction/Definition.vo	0.37s	0.608276253

TODO: scale by time taken, include estimates for rewriting-based
TODO: how to time-estimate bounds analysis?

When we're using the proof engine for partial evaluation, the time the proof engine takes to run the proof script just keeps growing and has unacceptable asymptotics



Let me tell you what this partial evaluation actually is.

Partial evaluation is like: if you have instructions for making a peanut butter and jelly sandwich that start with “search the house for the peanut butter, the jelly and the bread”, and if you always keep them in the same place, rather than writing out instructions that start with “search the house”, you can instead write out instructions that start with “fetch the peanut butter from the top shelf of the pantry on the right”.

Notably, partial evaluation is partial; you can partially evaluation equations like x + 2 + y – x + 6 into y + 8 without knowing what y is.

What is Partial Evaluation?

$x + 2 + y - x + 6$ → Partial Evaluation → $y + 8$

Notably, partial evaluation is partial; you can partially evaluation equations like x + 2 + y – x + 6 into y + 8 without knowing what y is.

Partial Evaluation in Fiat Cryptography

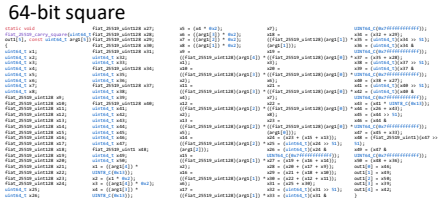
Template Code:

```
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.
```

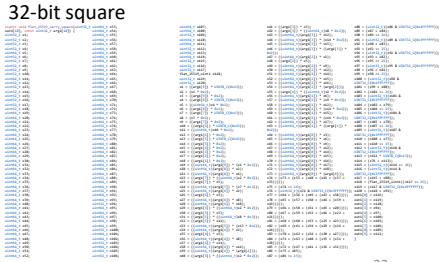
Partial Evaluation

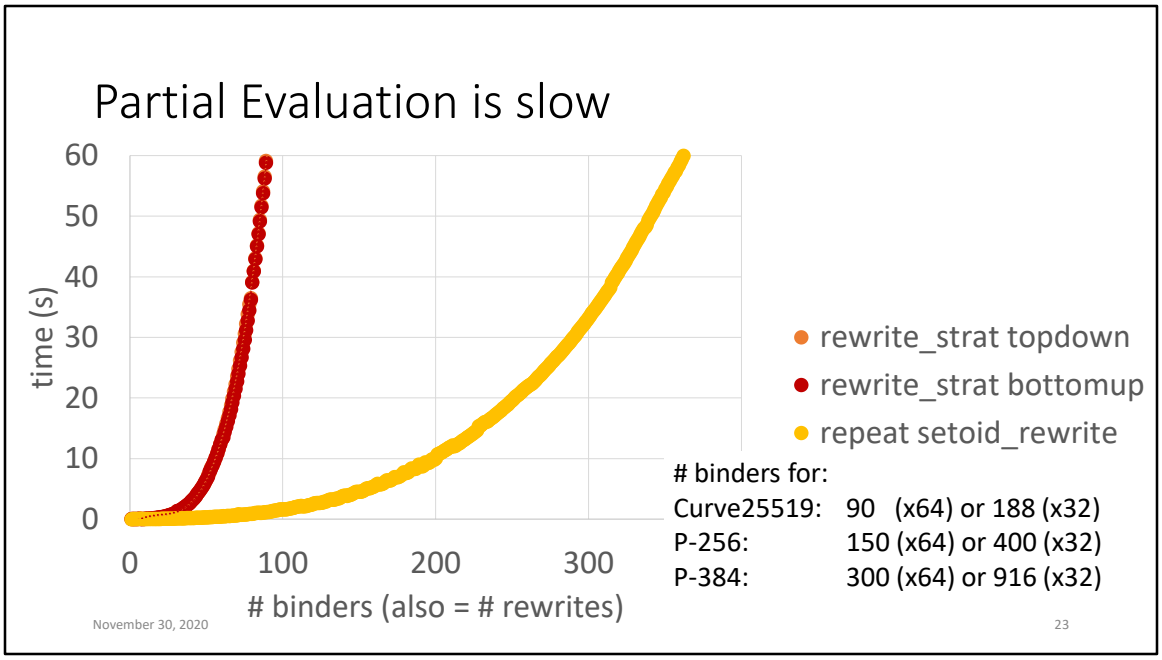
Partial Evaluation

64-bit square



32-bit square





Here are the tools to do partial evaluation in the proof engine

Note topdown and bottomup are almost identical
The underlying reason for this piece being hard is that all of the abstraction barriers that we introduced to carve the problem up into manageable pieces are broken here, so that we get fast low-level code out (this is a general pattern around performant code)
A large piece of my PhD work was making this possible in a way that scales

UnderLetsPlus0

What is a proof engine?

- Declare a goal to prove
- Issue instructions to make partial progress on proving
- Can write scripts to automate issuing of instructions
- Tracks the progress and current state
- Can issue a trail (proof certificate) to be checked by a small checker ("kernel")

A proof engine lets you declare a goal to prove; issue instructions to make partial progress on proving (interactive; lets you insert ingenuity);
can write scripts to automate issuing of instructions
tracks the progress made and where you are; can issue a trail to be checked by a small kernel / TCB

Our Approach

- Dig deep to find the places of asymptotic blowup
- Understand the precise source of the blowup
- Fuse the different compiler passes deeply

Let me now tell you our approach to solving performance issues.
… As we discovered, the key ingredient to the solution in the case of partial evaluation in fiat crypto was to fuse the various reduction and compiler passes on a deep level.

Requirements for Partial Evaluation

- β-reduction
- ιδ-reduction + rewrites
- code sharing preservation

β-reduction

- Useful for eliminating function call overhead in the generated code, which is important for output code performance
- Example: ((λ x. x + 5) 2) ⇝ 2 + 5

Change slide
TODO: drop termination

Achieving termination is interesting
Handling binders is interesting
Efficiency is interesting

ιδ-reduction + rewrites

- Useful for precomputation and eliminating function call overhead
- Arithmetic simplification necessary for getting right asymptotics of generated lines of code in fiat-crypto (quadratic vs. quartic)
- Example:
  map (λ x. x + 5) [0; 1; z] ⇝ [(λ x. x + 5) 0; (λ x. x + 5) 1; (λ x. x + 5) z]
  - Note that this leaves β redexes
  - Without β-reduction, this can blow up code size
- Fusing rewriting with β-reduction in a way that scales

Before/after example:
ιδ-reduction is about inlining function bodies and precomputing the results of case analyisis and loops or recursion.
Rewriting is about arithmetic or equational simplification.

Efficiency of rewriting is interesting

Code Sharing Preservation

- Necessary for avoiding exponential blowup in generated code size
- Example:
  map f (**let** y := x + x **in let** z := y + y **in** [z; z; z])
  ⇝ **let** y := x + x **in let** z := y + y **in** map f [z; z; z]
  ⇝ **let** y := x + x **in let** z := y + y **in** [f z; f z; f z]
- Fusing this with β- and ι- reduction

Compiler passes

- β-reduction
  - eliminating function call overhead
- ιδ-reduction + rewrites
  - inlining definitions to eliminate function call overhead
  - arithmetic simplification
- code sharing preservation
  - to avoid exponential blowup in code size

These requirements can each be thought of as a kind of compiler pass

Extra Requirements

- Verified
  - Without extending the TCB
- Performant
  - Should not introduce extra super-linear factors

On performant: note that we don't quite manage this one, but we do a lot better than the interactive solutions

The compiler passes need to be fused

- Needed to achieve adequate asymptotic performance!
- Separating out rewriting results in quartic rather than quadratic loc
- Separating out ι-reduction (constant propagation) results in *enormous* code-size blowup
- Separating out code-sharing-preservation results in *enormous* code-size blowup

Maybe:
ιδ+rewriting exposes new β-redexes
Can't do code-sharing preservation before δ, but δ without βι results in terms that are too big

2o mins

Implementation

- Reflective for performant and verified
- Normalization by Evaluation (NbE) (for β)

  + let-lifting monad (code-sharing)

  + rewriting (ιδ+rewrite)

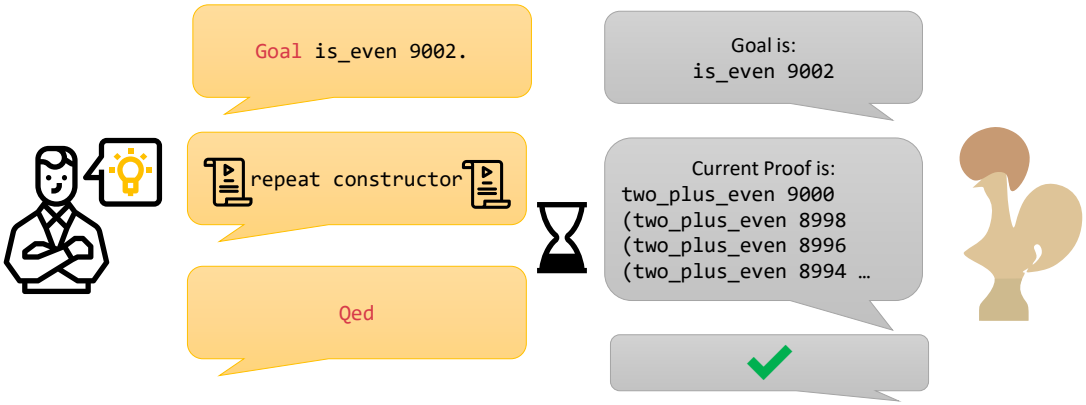
Proof by Reflection

- Most steps in the proof engine make partial progress towards a goal and leave behind a trail
- Coq's proof engine has a highly optimized primitive step for validating the output of a computation
- Phrasing the goal so that we can just validate the output of a computation
  - Verifying the process, rather than having an ad-hoc process that leaves behind a trail verifying the output

Example: compute the parity of a number; prove evenness by validating the computed parity

Non-Reflection Example

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



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.
```

Thinking modified from image by ArmOkay from the Noun Project CC BY 3.0; Coq logo from https://calebstanford.com/2019/01/15/coq-vector-image/

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)`

Why reflective rewriting?

- Reflective rewriting is asymptotically faster
- The trail left by proof-engine-based rewriting is super-linear in the size of the code being transformed
- Tracking the goal incurs super-linear overhead in the number of binders
- Recursively computing only the output is asymptotically faster
- Side benefit: we can extract it to OCaml to run as a nifty command-line utility

Trail can be avoided with a great deal of cleverness

Normalization by Evaluation

Goal: Reuse substitution in Gallina for substitution in ASTs

Example: Turn "(λ z p n x. z + (x + (p + n))) 0 1 (-1)" into

```
(λ z p n x. (λ a b. rewrite("+", a, b))
  z ((λ a b. rewrite("+", a, b))
      x ((λ a b. rewrite("+", a, b))
          p n)))) (rewrite("0")) (rewrite("1")) (rewrite("-1"))
```

Expression application ⇝ Gallina application

Expression abstraction ⇝ Gallina abstraction

Expression constants ⇝ rewriter invocations on η-expanded forms

Say: everything not in quotes is Gallina, quoted things are AST in a deeply embedded language
TODO: colorize quoted text or change font

In standard NbE, we just insert constant application at the leaves, rather than rewriter invocations. (Maybe emphasize this more)

Normalization by Evaluation

Goal: Reuse substitution in Gallina for substitution in ASTs

Example: Turn “(λ z p n x. z + (x + (p + n))) 0 1 (-1)” into

```
(λ z p n x. (λ a b. rewrite(“+”, a, b))
  z ((λ a b. rewrite(“+”, a, b))
       x ((λ a b. rewrite(“+”, a, b))
            p n)))) (rewrite(“0”)) (rewrite(“1”)) (rewrite(“-1”))
```

Then reduce!

Normalization by Evaluation

Goal: Reuse substitution in Gallina for substitution in ASTs

Example: Turn "(λ z p n x. z + (x + (p + n))) 0 1 (-1)" into

```
(λ z p n x. (λ a b. rewrite("+", a, b))
  z ((λ a b. rewrite("+", a, b))
       x ((λ a b. rewrite("+", a, b))
            p n)))) (rewrite("0")) (rewrite("1")) (rewrite("-1"))
```

Then reduce!

↝ (λ x. rewrite("+", "0", rewrite("+", "x", rewrite("+", "1", "-1"))))

Let-Lifting

- Let-Lifting monad for code-sharing-preservation
- Assignment + return; bind is derived
- Rewrote NbE in this Let-Lifting monad
- Haven't seen it in the literature, but it's not too tricky
- Automatic ι-reduction was too tricky to figure out, so I hard-coded the cases we needed for fiat-crypto

Note: alternative to CPS monad

Rewriting

- For ιδ+rewrite
- Using Parametric Higher-Order Abstract Syntax (PHOAS) to deal with binders allows delaying rewriting
- We thus achieve complete rewriting in a single pass when the rewrite rules form a DAG
  - We have extra magic for when they don't. The magic is called "fuel" and "try again".

TODO: fix Example:
"map (λ x. x + y) [0; 1]"
⇝ [(λ x. rewrite("x + y")) "0"; (λ x. rewrite("x + y")) "1"]

More Features

- Select rewrite rule based on Coq's pattern matching so we don't need to walk the entire list of rewrite rules at every identifier/constant node just to see which ones apply
- On-the-fly emission of a type of codes for relevant constants
- Partial evaluation on the generated rewriter (further 2x efficiency)

Using extracted code: pattern-matching compilation gives approximately 4x speedup over naive strategy, + another 2x if we pre-reduce the rewriter (which OOMs if we don't use pattern matching compilation (quadratic code size in # of rewrite rules? (approximately due to encoding artifacts (ι-expansion of head symbol)))

We select which rewrite rule to use based on Coq's pattern matching, which means that we don't need to walk the entire list of rewrite rules at every identifier/constant node just to see which ones apply
We enable this efficiency by on-the-fly emission of a type of codes for the constants we care about (seems like a new way of doing things not present elsewhere in the literature)
We further gain efficiency (about 2x) by doing partial evaluation on the generated rewriter itself (using Coq's built-in mechanisms)

Implementation

- Reflective for performant and verified
- Normalization by Evaluation (NbE) (for β)

  + let-lifting monad (code-sharing)

  + rewriting (ιδ+rewrite)

  + more features

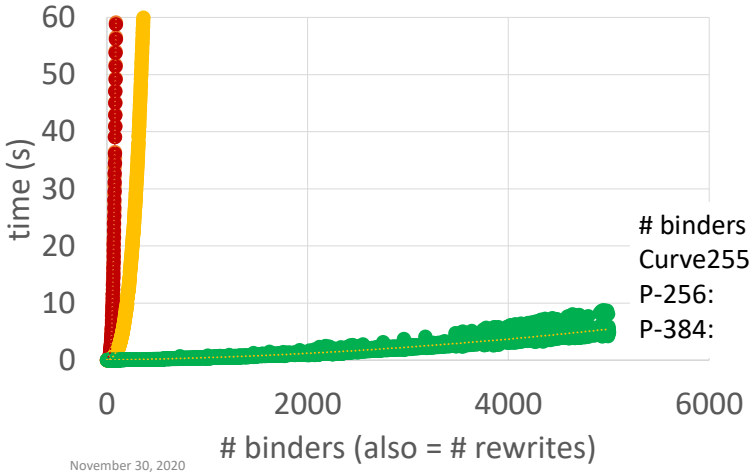
Note that most of the individual components have been done before, and many are even standard compiler passes.
However, in most domains, the compiler passes don't need to be fused.
Furthermore, reflective procedures can't be modularly fused at a deep level because they're outside the proof engine; the work needs to be redone from scratch

Evaluation

- It works!
- It's performant!

Alright, everyone who I sent to sleep can wake up now.

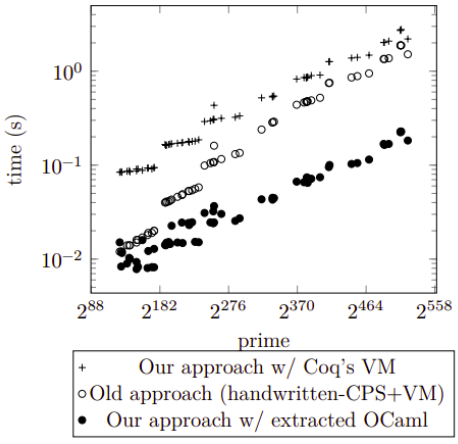
Performance



binders for:

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

Performance on Fiat Cryptography



Timing of different partial-evaluation implementations for Fiat Cryptography as prime modulus grows (only unsaturated Solinas x64)

Much better than 4,000 millennia too!

It seems like it would also solve one of the two performance issues that killed the parser-synthesizer that I worked on for my masters.

Our Approach

- Dig deep to find the places of asymptotic blowup
- Understand the precise source of the blowup
- Fuse the different compiler passes deeply

Takeaways

- Opportunity: Automate Verification to Enable Innovation
- Big Problem: Asymptotic Performance
- My Contribution: Reflective Partial Evaluation
- Important Next Steps

Takeaways from this talk: there is an opportunity, there is a problem, there are state of the art methodologies and our project, and we'll share the next steps

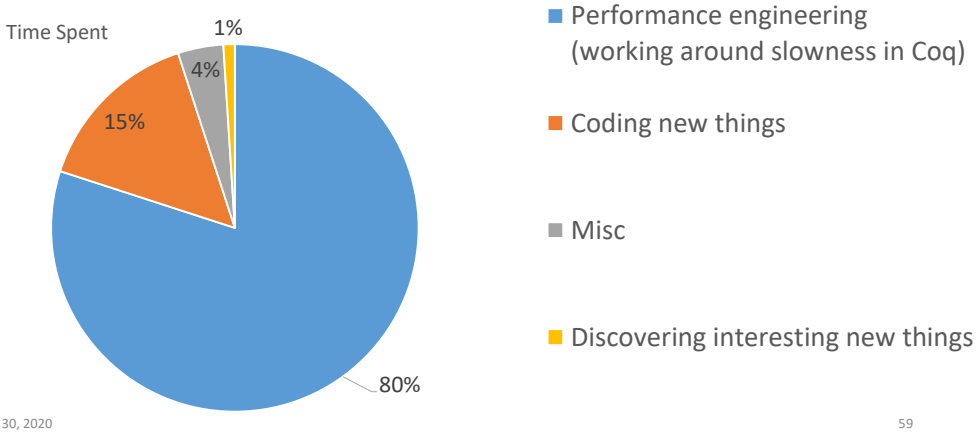
Note that I'll be talking in the context of interactive dependently typed tactic driven proof assistants, because human ingenuity is important.

Let’s take a step back

- We succeeded, but this was very hard
- All of this to work around inadequate asymptotic performance of the proof engine
- This is typical!

TODO: Stats about weeks of work and lines of code changed in rewriter

What I did in my PhD



In fact, I spent about 80% of my PhD working around slowness in the proof engine!

Our current approach to performance

- Using abstraction to prevent excessive unfolding
- Carving out the proof engine…
- …and replacing it with reflection

Use abstraction with an eye towards managing the burden on the proof assistant. Take any pieces that are too big…

Abstraction is not enough

- Systems code is often written in an adversarial context
- Symmetric crypto code is often written empirically
- Performant code breaks abstraction barriers

You can’t use abstraction in several places because using abstraction…

Reflection will not save us

- Using a proof assistant is for easily inserting human ingenuity to prove a broad range of things
- Using reflection is essentially giving up “easy” part
- As problems get bigger and harder and we need more ingenuity, it won’t be cost-effective to do it reflectively
- Already in the partial evaluator I hit the same performance-scaling issues that I was trying to avoid by writing it in the first place (albeit at a smaller and surmountable scale)

Only 3 pieces of ingenuity

Can we avoid carving out the proof engine?

- Where is the performance issue?
- Turns out that it's pretty far from the problem we're solving
  - (This should be obvious, because if it wasn't, reflection wouldn't help.)
  - Example: evar instance allocation has nothing to do with correctness of a given C algorithm
- In my experience, it's not about generating a proof trail and it's not even really about individual steps being slow
  - It's about asymptotics of accessing and updating data being tracked
  - Sometimes just walking the term repeatedly is too much overhead

This lends an answer Why reflection help at all?

Reflection helps *because* it's solving a more limited problem
This means reflective proof engine isn't enough

Not just an engineering challenge

- "Don't make stupid choices" isn't enough to get good asymptotic performance
- Try writing `rewrite_strat`
  - inside the tactic engine
  - every step considered as progress towards proving something
  - linear in # of binders + # of rewrite locations + size of term
  - really hard, maybe impossible!
- We need to systematically study proof engines with an eye towards asymptotic performance!

As a field, we need to study proof engines with an eye towards asymptotic performance

Next Questions about Proof Engines

- Where does the performance overhead really come from?
- What things are people not currently doing due to performance overhead?
- What is an adequate set of primitives?
- What are acceptable thresholds on asymptotic behavior?
- Is it possible to achieve adequate performance simultaneously on all the primitives?

Barebones research agenda

Starting by systematically understanding the field and then asking deep design questions.

What are the requirements on something to be a “proof engine”?

My current take is “every step makes partial progress towards proving something” and “error messages about proof validity are local”

My hope

I think solving this problem—getting the basics of proof engines right, asymptotically—will drastically accelerate the scale of what we as a field can handle, and bring verification closer to its promise and potential of enabling innovation in industry.

Thank you for your time and attention!

Questions?