From Type Theory to End-to-End Proof of Realistic Systems - Lecture 1

Adam Chlipala
Notes by Cheng Zhang, Molly Pan, Pratap Singh

June 2022

1 Introduction

Prof Chlipala's lectures at OPLSS discuss the use of the Coq proof assistant to build real-world systems with formal proofs of correctness. The type theory discussed in other OPLSS lectures provides enough of a formalism to give such correctness proofs for significant real-world programs. Lecture 1 focuses on high-level software, in particular the Fiat Cryptography project which automatically generates verified implementations of finite-field arithmetic operations that are used in modern cryptographic protocols such as TLS.

Current plan for the lectures:

- Today: High-level software—elliptic curve cryptography;
- Next lecture: Low-level software;
- Final lecture: Hardware.

2 Architecture of a verified Coq development

When using a proof assistant to build a verified system, it is important to keep track of which components are trusted and which are untrusted. The trusted component here does not mean "trust-worthy", but means "assumed to be correct". In other word, our system can deduce errours result only when the trusted component goes wrong. Hence we want to minimize the amount of code that must be trusted (since this is where bugs can enter the system). Typically, we have a trusted specification for our program that describes what the program must do; we then write an untrusted program that implements that specification. In Coq, we also construct a proof script (written in a high-level tactic language) which provides guidance to the Coq proof engine on how to generate a proof term (written in a dependently typed λ-calculus). Importantly, all of these proof components are untrusted, and so can be as complex as desired. The only component of the proof system that must be trusted is a typechecker

for proof terms, which checks that the type of the proof term is the logical formula corresponding to the user-provided specification. Equivalently (by Curry-Howard), the proof term provides a constructive proof that the program obeys its specification. See diagram 1.

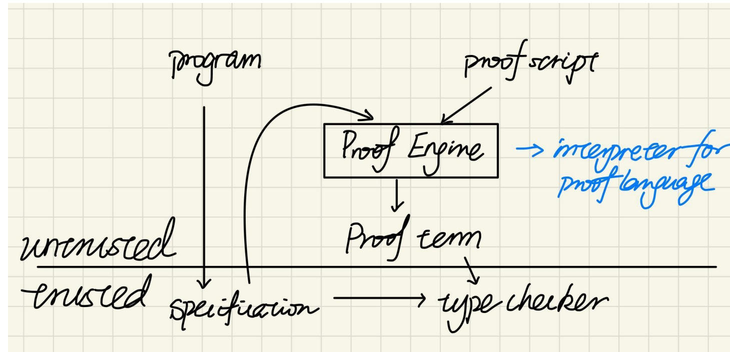

Figure 1: The trusted and untrusted parts of the Coq proof system.

When designing complex verified systems, we typically encounter a tradeoff between the trustworthiness of theorems and how easy they are to prove.

In many pieces of software, to improve performance, people would use optimization encoding for common operations. This requires us to prove the correctness of these optimizations. However there are many challenges:

- The performance of proof engine
- Low level implementation might change

We need a proof system that is *fast*, *efficient* and *implementation-generic.*

3 Finite-field arithmetic and its implementation on processors

One common tool in cryptography is arithmetic modulo a large prime number. Consider the following:

$$m = 2^{255} - 19$$
$$T = \mathbb{N} \mod m$$
$$x_T \cdot y_T = x \cdot y \mod m$$

above defines the operation in T which is the modular arithmetic with a specific prime $m = 2^{255} - 19$. m is a Pseudo-Mersenne Prime which is chosen to optimize the performance of the multiplication algorithm, as we will show later.

In a 64-bit computer, a large number can be represented as a array of 64-bit chunks:

$$n = n_1 + 2^{64} n_2 + 2^{128} n_3 + 2^{256} n_4 + \dots$$

However, multiplying numbers in this representation is slow, since modular multiplication is a slow operation.

Alternatively, if we change the base from 2^{64} to 2^{255}: $r_0 + 2^{255} r_1$ for some r_0 and r_1, we then have:

$$\begin{aligned} & r_0 + 2^{255} r_1 \mod (2^{255} - 19) \\ &= r_0 + (2^{255} - 19 + 19) r_1 \mod (2^{255} - 19) \\ &= r_0 + 19 r_1 \mod (2^{255} - 19) \\ &= r_0 + 19 r_1 \mod m \end{aligned}$$

This implementation can be computed much faster since the numbers being multiplied are much smaller. This example demonstrates the tradeoff between ease of reasoning and understanding versus real-world performance.

Currently the commonly chosen integer representations for operations in this field are:

- 51-bit chunks on a 64-bit systems or,
- alternating 25-bit chunks and 26-bit chunks on a 32-bit systems.

We therefore want a mechanism to reason about these algorithms that is independent of the integer representation used. (Following code can be found in `https://github.com/mit-plv/fiat-crypto` in `/src/demo.v`)

The integer is represented as a list of integers each with a weight:

```
Definition mul (p q:list (Z*Z)) : list (Z*Z) :=
flat_map (fun t =>
    map (fun t' =>
        (fst t * fst t', (snd t * snd t')%RT))
q) p.
```

In previous examples:

$$n = n_1 + 2^{64} n_2 + 2^{128} n_3 + 2^{256} n_4 + \dots$$

The n_i are the integers and the $2^{64}, 2^{128}, \dots$ is the weight. This way we can represent integers independent of their bases.

Here is an example for multiplication in base 10. We omit the tactic scripts used to generate the proofs; readers are encouraged to step through the proofs and observe how the proof state changes.

```
Example base10_2digit_mul (a0:Z) (a1:Z) (b0:Z) (b1:Z) :
    {ab| eval ab = eval [(10,a1);(1,a0)] * eval [(10,b1);(1,b0)]}.
```

We can read the proof goal as follows: let a be the two-digit number with tens digit a_1 and ones digit a_0, and let b be the two-digit number with tens digit b_1 and ones digit b_0. We then want to find an expression ab such that evaluating ab always produces the result as evaluating a, evaluating b, then taking the product of the result.

And here is an example for modular multiplication with $2^{255} - 19$:

```
Example base_25_5_mul (f g:tuple Z 10) :
  { fg : tuple Z 10 | (eval w fg) mod (2^255-19)
                    = (eval w f * eval w g) mod (2^255-19) }.
```

The result of each proof contains two parts: a term containing the expression implementing each operation, and a proof that the term satisfies the desired specification.

4 The Type Theory

Besides the normal dependent function type:

$$\textsc{Π Intro}\quad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . e : \Pi_{x:\tau_1} \tau_2} \qquad \textsc{Π Elim}\quad \frac{\Gamma \vdash e_1 : \Pi_{x:\tau_1} . \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 [e_2/x] : \tau_2}$$

We also need a set of equational rules in order to prove the optimized version will output the same result as the unoptimized version.

Thus, we introduce the rules for definitional equality:

$$\textsc{$\equiv$ Elim}\quad \frac{\Gamma \vdash e : \tau_1 \qquad \tau_1 \equiv \tau_2}{\Gamma \vdash e : \tau_2} \qquad \textsc{reflexitivity}\quad \frac{}{\vdash \text{refl} : e \equiv e} \qquad \textsc{β equivalence}\quad \frac{}{(\lambda x : \tau . e_1) e2 \equiv e_1[e_2/x]}$$

$$\textsc{transitivity}\quad \frac{e_1 \equiv e_2 \qquad e_2 \equiv e_3}{e_1 \equiv e_3}$$

The above rules specify a natural dependent type theory, but unfortunately it is not well suited to building a proof assistant due to the reliance on the type equivalence judgement $\tau_1 \equiv \tau_2$. The proof assistant would need to somehow generate and store all the equivalences used in the proof, which is intractable.

Instead, we can embed a strategy: *strat* for generating equivalences, which allows the proof to be represented compactly. The strategy embeds the way the proof assistant generates type equivalences. We then have the following proof rules:

$$\textsc{$\equiv$ Elim}\quad \frac{\Gamma \vdash e : \tau_1 \qquad \tau_1 \equiv_{strat} \tau_2}{\Gamma \vdash e : \tau_2} \qquad \textsc{reflexitivity}\quad \frac{}{\vdash \text{refl} : e \equiv e}$$

$$\textsc{β equivalence}\quad \frac{}{(\lambda x : \tau . e_1) e2 \equiv_{strat} e_1[e_2/x]} \qquad \textsc{transitivity}\quad \frac{e_1 \equiv_{strat} e_2 \qquad e_2 \equiv_{strat} e_3}{e_1 \equiv_{strat} e_3}$$

This design gives fast and deterministic reasoning about type equalities, but the rewriting strategies must be added to the typechecker, growing the trusted code base.

To produce a small system to prove equalities, we can embed *rewriting* into our proof rules. Then, instead of generating a proof tree, we can just generate the rewrites that get us from one type to another. We have the following rules:

REFLEXIVITY

$$\frac{}{\Gamma \vdash \text{refl}_e : e = e}$$

REWRITE

$$\frac{\Gamma \vdash e_1 : e_2 = e_3 \qquad \Gamma \vdash e_4 : \tau}{\Gamma \vdash \text{rewrite}(e_1, e_4)[e_3/e_2] : \tau}$$

This induces a new structure for the proof system that embeds a rewriting strategy without additional rules in the typechecker. However the downside of this approach is that rewriting-based proofs are typically slow and too large to be stored: the system needs to explicitly store each rewriting step used, to justify that their composition is correct.

We instead use the approach of performing the reduction in the term language itself, translating proof goals into terms which can then be reduced by the usual reduction strategies of the term language. We maintain a set of rewrite rules that are specialized to the particular system of interest, which (along with the goals) is passed to a *reifier* that translates them into abstract syntax trees of the term language. These ASTs can then be simplified according to the normal reduction rules. The resulting AST then corresponds to a new proof goal, which can be recovered by reversing the reification transformation. The system is illustrated by Figure 2.

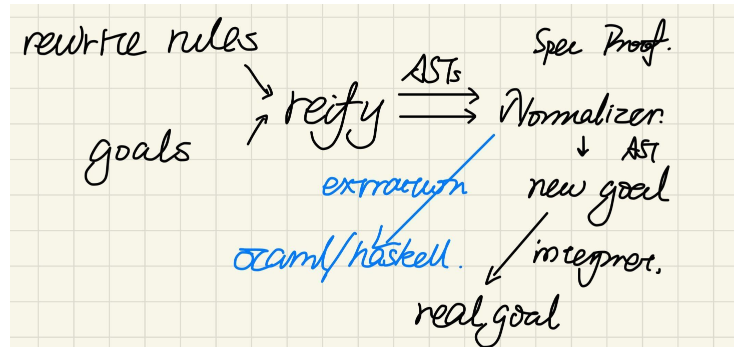

Figure 2: Translating and normalizing the "native" goals into terms.

The benefit of this strategy over the explicit rewriting approach is that the proof term does not have to store explicit applications of each rewrite rule used in the proof. Instead, the rewriting rules are translated into a normalizer that is proven to only perform safe reductions on the term. Since we know that every reduction performed by the normalizer is safe, we know that the overall computation was safe as well.

5 Implementation of Fiat Cryptography

The current version of Fiat Cryptography uses the approach of reifying the proof goal and performing reductions in the term language. As a result, the system generates both correct implementations and proofs of correctness for the cryptographic primitives.

Additionally, Coq provides an `Extraction` feature that extracts executable code to languages such as OCaml and Haskell. Fiat Cryptography uses this to produce a standalone executable that can generate optimized C code for these cryptographic primitives without needing to run Coq. The performance of these generated implementations is comparable to the previous hand-optimized implementations.