Explorations at the Intersection of Cryptography and Verification

Jared Roesch
jroesch@cs.ucsb.edu

Abstract— **We demonstrate a prototype framework for light weight reasoning about the correctness and performance of cryptographic algorithms. Our framework is informed by the state of the art in machine checked verfication, but goes beyond it in order to build a framework for the specification, verification, optimization, and execution of our algorithms ideally both in software and hardware.**

I. Introduction

Cryptographic algorithms and protocols are quickly becoming more prevelant, more special purpose, and more complex. In the face of this reasoning about their correctness is still not easy, and requires a combination of proof, testing, and peer review. Many of these essential algorithms are also key to our day to day lives and we require their implementations to not only be correct, but fast.

The question then becomes how do give cryptographers tools to specify, verify, test and optimize their algorithms. Ideally we would like to increase the rigor of our verification and allow for the use of a machine proof checker.

The programming language community has a long history with type theory, a computational proof theory that allows us to equate programs with proofs, and types with propositions. Type theory allows one to write down a specification (a type) satisfy it with a program (a proof) and then execute the program for a result.

Ours and many

II. Related Work

III. Lean

Lean is a new theorem proving environment from Microsoft Research.

- powerful elaboration engine
- small trusted kernel
- flexible kernel
- incremental compilation
- universe polymorphism
- mixed declarative and tactic proof style
- powerful automation

The core ideas are:

- type universes
- function spaces
- inductive types

A. Type Theory

Type theory is a foundational framework for mathematics. It was an approach that evolved out of Russel's work on setting mathematics on a consistent foundational system for mathematics. The early 20th century was spent developing type theory as a proof system. The key insight was the Curry-Howard Isomorphism, which roughly states that types can be associated with propositions and proofs with programs. This insight was adapted and it became apparent that typed variants of the lambda calculus can be used as proof systems, the most widely used of which is Martin-Lof type theory. You can then view types in this typed lambda calculus as various features of constructive logic.

If we begin with just an implicational fragment of logic we can use the simply typed lambda calculus which only has a single type constructor, the function type. This corresponds to implication. We can now view the typing judgment $a : A$ as statement that a is a proof of A, and we can interpret the typing judgment $f : A- > B$ as a computational implication. At the type level it allows us to produce a B given an A, and computationally it gives us a method for transforming a program of A into a program of B. If we remember that program's correspond to proofs we now have a computational method for proving.

We can of course extend our type system with richer features. We will quickly survey important additions to the Simply Typed Lambda Calculus that are needed to turn it into a dependently typed lambda calculus. The key idea is the erasure of phase distinction via the inclusion of a stratified type hierarchy, and Pi types.

Traditionally we maintain a phase distinction where types and values live in separate universes. Dependent types are about easing this distinction. Type systems can usually be made more expressive by a series of additions. The simplest of which is the introduction of quantifiers which allow us to abstract over types with type lambdas. We can then introduce kinds (the types of types), and kind polymorphism which allow us to specify type shapes or (type arities). We can continue to enrich our kind and type system, eventually reflecting types as kinds, and values as types, but these systems are actually more complicated then a dependently typed calculi. Each of these refines become more and more complex because it requires the language to maintain a discipline between values, kinds, sorts. Dependent types simply introduce the idea of a Pi Type of dependent function space. This type system generalization is the idea that we can make everything much simpler by creating a stratified hierarchy of types, and adding Pi and Inductive types. We then can collapse our language into a simple set of pseudo terms, and typing rules which are easy to verify for correctness.

We can then ignore all the complexity introduced by

types systems such as System F and generalizations of it. Instead we can focus on a simple language. Usually a dependently typed core only requires a few forms abstraction, variables, function application, and Pi Types. This is simply the lambda calculus that everyone knows enriched with Pi types. It is also useful to extend the calculus with inductive types, or recursive data types that may be parametrized by the values they are constructed with.

Pi Types can be simply viewed as a generalization of the traditional function type. The generalization allows the type of codomain to depend on value of the domain of the function. Given $B : P- > A; \Pi(x : A).B(x)$. This simple extension actually gives rise to a lot of power and will allow us to encode much of mathematics in type theory.

This is of course a very brief overview and more information can be found in [1] [2] and [3].

B. Theorem Proving

Automated Theorem Proving has been an area of active research for many decades. It is apparent that the verification of general properties is desirable to both high assurance software engineers, researches, and mathematicians. There have been various approaches to Automated Theorem Proving in the past ACL2, NuPRL, LF, LCF, Coq, Agda. The ones based on dependent type theories seem the most promising for various reasons (computational, program extraction, proof carrying code, ect). Automated Theorem Proving has been applied in many areas from the NTSB, NASA, JPL, and a variety of other places.

The current best way to leverage the proving power of Type Theory are Automated or Interactive Theorem Proving environments. These allow for the automatic satisfaction of some properties, and interactive proving of others. The state-of-the-art theorem proving environments in use for software verification are Coq, Agda, and Idris all of which provide the benefits of:

- dependent types (equivalent to a fragment of isolationistic logic)
- provides tools for automated proof search
- specification and implementation are one and the same
- executable "proofs" (i.e program extraction)

These are all based on versions of intensional type theory. Coq is base on on Coquand and Huet's Theory of Constructions, and Agda is an evolution Martin-Lof Type Theory [2].

Automated Theorem Proving has been an area of active research for many decades. It is apparent that the verification of general properties is desirable to both high assurance software engineers, researches, and mathematicians. There have been various approaches to Automated Theorem Proving in the past ACL2, NuPRL, LF, LCF, Coq, Agda. The ones based on dependent type theories seem the most promising for various reasons (computational, program extraction, proof carrying code, ect). Th current state-of-art is either based on Coquand and Huet's Theory of Constructions or the work done on Martin-Lof Type Theory [2]. Automated Theorem Proving has been applied in many areas from the NTSB, NASA, JPL, and a variety of other places.

It appears that ideas from ATP could be very useful in the domain of hardware specification. Most specification is performed separately from the implementation and suffers the common problems of pen and paper proof. In many cases Embedded Domain Specific Languages (EDSLs) have been used as a way to gain the power of a proof assistant and also write proof carrying code[4][5]. We want to leverage the ideas here, but ideally built a separate front-end that exposes the full power of proving, but also allows one to provide both an usable interface that also generates useful proof properties for free [4]. [6] [7]

There is a prototype implementation of ideas discussed in the final section available. It is currently written in Idris (which is one of the more practical dependently typed programming languages).

IV. Vrypto

V. Future Work

Figure 1: Hardware realization of $(u - v)/2$.

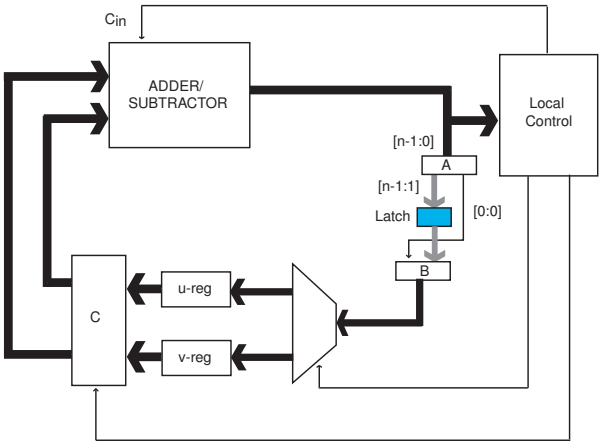
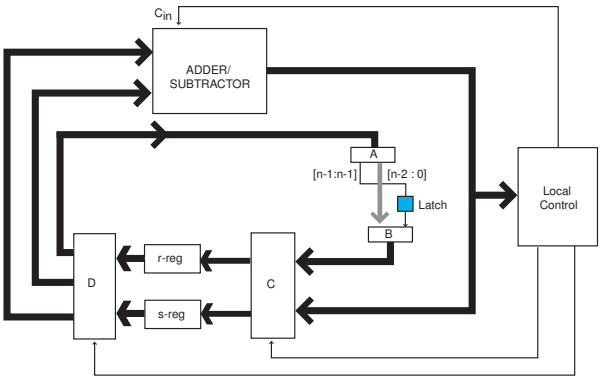


Figure 2: Hardware realization of $r + s$.



References

[1] Benjamin C. Pierce, *Types and Programming Languages*, MIT Press, 2002.
[2] B. Nordstrm, K. Petersson, and J. M. Smith, "Martin-lfs type theory," 2000.

[3] The Univalent Foundations Program, "Homotopy type theory: Univalent foundations of mathematics," Tech. Rep., Institute for Advanced Study, 2013.
[4] Daniel Ricketts, Valentin Robert, Dongseok Jang, Zachary Tatlock, and Sorin Lerner, "Automating formal proofs for reactive systems," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, 2014, PLDI '14, pp. 452–462, ACM.
[5] Thomas Braibant and Adam Chlipala, "Formal verification of hardware synthesis," *CoRR*, vol. abs/1301.4779, 2013.
[6] Adam Chlipala, *Certified programming with dependent types*, vol. 20, MIT Press, 2011.
[7] Benjamin C. Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjoberg, and Brent Yorgey, *Software Foundations*, Electronic textbook, 2014, http://www.cis.upenn.edu/ bcpierce/sf.