High assurance programming in Cryptol [*]

Levent Erkök John Matthews

Galois, Inc.
421 SW 6th Ave. Suite 300
Portland, OR 97204
{levent.erkok,matthews}@galois.com

1. Introduction

Cryptol is a domain specific language tailored for cryptographic algorithms (`www.cryptol.net`). Explicit support for program verification is an indispensable part of the Cryptol toolset, due to the inherent high-assurance requirements of the application domain. To this end, Cryptol comes with a suite of formal-methods based tools, allowing users to perform various program verification tasks.

In this extended abstract, we provide an overview of the Cryptol language and its verification environment. The challenges in this domain are multifaceted: from the engineering concerns of providing an easy-to-use system for non-experts, to open research problems in program verification.

2. A taste of Cryptol

Cryptol is a pure functional language, built on top of a Hindley-Milner style polymorphic type system extended with size polymorphism and arithmetic type predicates (Lewis and Martin 2003). The size-polymorphic type system has been designed to capture constraints that naturally arise in cryptographic specifications. To illustrate, consider the following text from the AES definition (NIST 2001, Section 3.1):

> The **input** and **output** for the AES algorithm each consist of **sequences of 128 bits**. ... The **Cipher Key** for the AES algorithm is a **sequence of 128, 192 or 256 bits**. Other input, output and Cipher Key lengths are not permitted by this standard.

This description is captured precisely in Cryptol by the following type:

```
encrypt: {k} (k >= 2, 4 >= k)
              => ([128], [64*k]) -> [128]
```

Anything to the left of `=>` are quantified type-variables and predicates on them. In this particular case, the function `encrypt` is size-polymorphic, relying on the size-variable `k`. The predicates constrain what values the quantified size-variables can take on: Here, `k` is restricted to be between 2 and 4. To the right of `=>`, we see the actual type. The type of `encrypt` is a function, accepting tuples of type `([128], [64*k])` and returning `[128]`. The input to `encrypt` is a pair of two words, the first of which is precisely 128-bits wide; this is the plain-text. The second argument is a `64*k`-bit wide word (i.e., 128, 192, or 256 bits, depending on `k`), which corresponds to the key. The output of the function, the ciphertext, is another 128-bit word.

Note how this type precisely corresponds to the English description in the standard. For instance, it is statically ensured that the key-size will be one of 128, 192, or 256: the type system would not let a definition of `encrypt` be accepted that did not satisfy this requirement. (Similarly, all call-sites will be checked to satisfy this requirement as well.) Cryptol's expressive power in stating precise size relationships is one of its key advantages. In other languages, such constraints would either only appear in comments, or at best get dynamically checked at run-time, instead of being enforced statically at compile-time as in Cryptol.

3. Verification of Cryptol programs

Cryptol's program verification framework has been designed to address equivalence and safety checking problems.

The equivalence-checking problem asks whether two functions f and g agree on all inputs. In case f and g are not equivalent, we would like to find a particular value x such that $f\ x \neq g\ x$.

The safety-checking problem is about run-time exceptions. Given a function f, we would like to know if f's execution can perform operations such as division-by-zero or index-out-of-bounds. If so, we would like to know what particular input values will result in an exceptional behavior.

Users interact with Cryptol's verification system via `theorem` declarations. Here is an example Cryptol theorem, named `C`, stating that `dec` and `enc` are cryptographic inverses of each other:

```
theorem C: {key pt}.
      dec(key, enc(key, pt)) == pt;
```

(The notation "`{key pt}.`" reads "for all values of `key` and `pt`.")

Theorems in Cryptol are simply bit-valued functions, taking their quantified variables as inputs, returning `True` or `False` as their result. We call this the theorem-function correspondence. This unified view provides a consistent and familiar environment for end-users, avoiding the need for an extra language to express properties. As a nice consequence, proving a Cryptol theorem simply amounts to showing that it is equivalent to the constant function that always returns `True`. This is the main mechanism that connects program verification to equivalence checking in Cryptol.

The other weapon in Cryptol's verification toolbox is the automated verification capability that programs cannot cause run-time exceptions, such as division-by-zero or index-out-of-bounds. If such an exception is possible, Cryptol will display a particular input value that causes the exception.

4. Verification technology

Cryptol provides two main verification routes:

[*] Extended abstract.

- *Fully-automated:* For a restricted (but relatively large) subset of Cryptol, both equivalence and safety checking problems can be decided. For this subset, Cryptol provides fully automated tools to prove theorems with no further user involvement, using off-the-shelf SAT and SMT solvers as proof engines to complete the verification task (Barrett et al. 2008).
- *Semi-automated:* When automated solvers are not suitable, Cryptol provides a tool to translate Cryptol theorems to Isabelle/HOL automatically, where proofs can be manually constructed by the user.

It is a fair question to ask why Cryptol's formal-methods tools only focus on safety and functional-equivalence checking, instead of adapting a more general logic for expressing and proving properties. The reasons are quite practical:

- To do equivalence checking, users need not learn any new language in which to express properties. The term language of Cryptol is sufficient to express functional correctness properties concisely.
- Equivalence checking works well with an incremental development model; successive versions of an algorithm can be proven equivalent to a known "reference" specification, following the stepwise-refinement approach.
- Due to the theorem-function correspondence, all Cryptol theorems are executable. We take advantage of this correspondence by providing a "quick-check" facility, where theorems are checked automatically over randomly generated input data to aid in high-assurance development (Claessen and Hughes 2000).
- Counterexamples provided by equivalence checkers are extremely important. Having a concrete "bug" to look at makes all the difference in providing a practical verification environment for end-users.

Also, on the social side, we have found that the equivalence checking approach lowers the barriers for the adoption of formal methods. Since the users are not required to write program properties in an unfamiliar logic, equivalence checking is easier to introduce into daily development activities. The automated quick-check facility provides instant feedback, empowering users to try alternate ideas with a certain degree of confidence with no extra cost. Additionally, theorem declarations mixed with source-code serve as valuable documentation that is guaranteed not to become obsolete as the code around them evolves.

Equivalence checking applies not only to user programs, but also to compiler generated code as well. For intance, Cryptol's FPGA compiler backend performs extensive transformations to turn Cryptol programs into hardware primitives available on target FPGA platforms. The formal verification framework of Cryptol allows equivalence checking between Cryptol and netlist representations that are generated by various parts of the compiler. Therefore, any potential bugs in the compiler itself are also caught by the same formal verification framework. This is a crucial aspect of the system: proving the Cryptol compiler correct would be a prohibitively expensive task, if not impossible. Instead, Cryptol provides a *verifying* compiler, generating code together with a formal-proof that the output is functionally equivalent to the input program.

5. Verification via theorem proving

Cryptol's automated verification system works in a complete push-button manner, providing a seamless integration between daily programming and formal verification. Unfortunately, the fully automated system is not the panacea. Some important Cryptol theorems can not be proved in a reasonable amount of time (i.e. several days) using the currently available automated proving tools. These problems typically rely on high-level algebraic equivalences, and involve sub-problems that are notoriously hard for SAT/SMT solvers, such as multiplication (Kroening and Strichman 2008, Section 6.3.1).

To deal with these difficult theorems we have recently added a translator from Cryptol to the Isabelle proof assistant (Nipkow et al. 2002). Isabelle has a wide array of existing rewrite and proof search methods, and also allows users to certify their own custom proof strategies. The use of a proof assistant such as Isabelle allows one to integrate human insight into the proof process when fully automated tools do not provide a feasible alternative.

6. Conclusions

In this extended abstract, we have provided a very brief overview of the verification framework of Cryptol, a key ingredient of Cryptol's high-assurance development environment. The system has already proved itself useful in practice, especially in establishing the equivalence of reference and pipelined implementations of crypto-algorithms, essential in generating efficient FPGA representations. Since such transformations are done both manually by programmers and automatically by the compiler, it is absolutely essential to ensure that correctness is preserved through many structural transformations applied to obtain a time/space efficient realization.

Developing a practical programming language with formal verification in mind is an inherently difficult task. The needs for "ease of programming," and "ease of verification" are often at odds. Supporting more language constructs to simplify programming comes with an obligation to support those constructs precisely in the proof environment as well. (In the case of Cryptol, this is further complicated by the desire to support compilation down to FPGA based targets, where "run-time" resources are extremely limited.) Nonetheless, the rather general programming model of Cryptol based on size-types and pure functional programming, and an "integrated verification" based approach in the design of the toolset paid off nicely in practice, balancing the needs of both sides.

The web-site `www.cryptol.net` contains further resources related to Cryptol, including downloadable tools.

Acknowledgments

Many people have worked on Cryptol and its formal verification toolset over the years, including Jeff Lewis, Thomas Nordin, and Phil Weaver. We are also thankful to Magnus Carlsson, Lee Pike, John Launchbury, Joe Hurd, and Sally Browning for their feedback and comments on this work.

References

Clark Barrett, Silvio Ranise, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.smt-lib.org`, 2008.

Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proc. of International Conference on Functional Programming (ICFP)*. ACM SIGPLAN, 2000.

Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer, 2008.

J. R. Lewis and B. Martin. Cryptol: high assurance, retargetable crypto development and validation. In *Military Communications Conference 2003*, volume 2, pages 820–825. IEEE, October 2003.

T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, 2002.

NIST. Announcing the AES, November 2001. URL `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`. FIPS Publication 197.