A Verifying Core for a Cryptographic Language Compiler

Lee Pike (presenting)
Mark Shields[1] John Matthews

Galois Connections

August 15, 2006

[1]Presently at Microsoft.

Thanks

- Rockwell Collins Advanced Technology Center, especially David Hardin, Eric Smith, and Tom Johnson
- Konrad Slind, Bill Young, and our anonymous *ACL2* Workshop reviewers
- Matt Kaufmann and the other folks on the *ACL2*-Help list
- And of course, Pete Manolios and Matt Wilding for a heckuva workshop!

galois

Compiler Assurance: The Landscape

- Compilers are complex software systems.
  - Critical bugs are possible.
  - Compilers are targets for backdoors and Trojan horses.
- How do we get assurance for correctness?
  - Testing.
  - Long-term and widespread use (e.g., gcc).
  - Certification (e.g., Common Criteria, DO-178B).
  - Mathematical proof.

Proofs and Compilers: Two Approaches

1. A *verified compiler* is one associated with a mathematical proof.
   - One monolithic proof of correctness for all time.
   - Deep and difficult requiring parameterized proofs about the language semantics and the compiler transformations.
2. A *verifying compiler*[2] is one that emits both object code and a proof that the object code implements the source code.
   - Requires a proof for *each* compilation (the proof process must be automated).
   - But the proofs are only about concrete programs.

If you have a highly-automated theorem-prover (hmmm... where can I find one of those?), a verifying compiler is easier.

We take the verifying compiler approach.

[2] Unrelated to Tony Hoare's concept by the same name.

| galois |

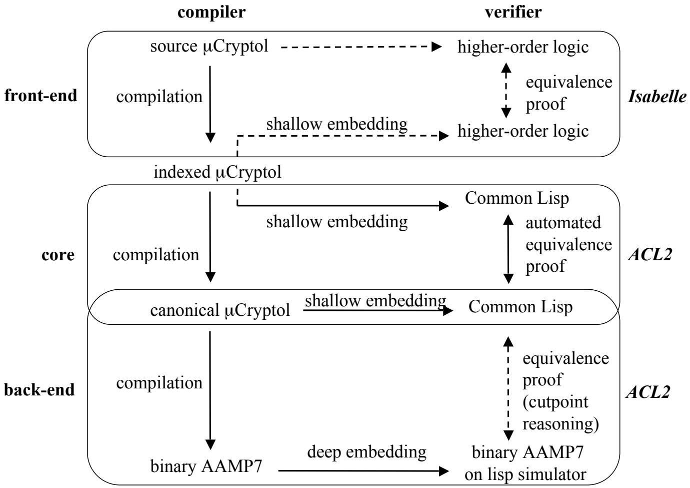
μ*Cryptol* in One Slide

```
fac : B^32 -> B^8;
fac i = facs @@ i
  where {
    rec
      index : B^8^inf;
      index = [0] ## [ x + 1 | x <- index];
    and
      facs : B^8^inf;
      facs = [1] ## [ x * y | x <- facs
                            | y <- drops{1} index];
  };
```

$$\begin{aligned} \texttt{index} &= 0, 1, 2, 3, 4, \ldots, 255, 0, 1, \ldots \\ \texttt{facs} &= 1, 1, 2, 6, 24, 120, 208, 176, \ldots \\ \texttt{fac}\,3 &= \texttt{facs @@}\,3 = 6 \end{aligned}$$

galois

Overall Infrastructure



galois

What We've Done: Snapshot

- A "semi-decision procedure" in *ACL2* for proving correspondence between *μCryptol* programs in "indexed form" and in "canonical form".
- A semi-decision procedure for proving termination in *ACL2* of *μCryptol* programs (including mutually-recursive cliques of streams).
- A *simple* translator for shallowly embedding *μCryptol* into *ACL2*.
- An *ACL2* book of executable primitive operations for specifying encryption protocols (including modular arithmetic, arithmetic in Galois Fields, bitvector operations, and vector operations).

These results are germane to

- Verifying compilers for other functional languages
- The verification of cryptographic protocols in *ACL2*
- Industrial-scale automated theorem-proving

Applications and Informal Metrics

Framework for *automated* translations, correspondence proofs, and termination proofs for, e.g.,

- Fibonacci, factorial, etc.
- TEA, RC6, AES

Caveat: `mcc` doesn't output the correspondence proof itself yet.

ACL2 "Condition of Nontriviality": for AES, *ACL2* automatically generates

- About 350 definitions
- 200 proofs
- 47,000 lines of proof output

Termination is decidable! (Thanks, Mark)

Let S be the set of stream names for a mutually-recursive clique of stream definitions. Then we say the clique is *well defined* if there exists a *measure function*

$$f : (\mathbb{N} \times S) \rightarrow \mathbb{N}$$

such that for each occurrence of a stream y in the body of the definition of stream x with delay d, we have

$$\forall k \in \mathbb{N}.\, k \geq d \Rightarrow f(k-d, y) < f(k, x)$$

The `mcc` compiler type system ensures well-definedness

- The compiler constructs a minimum delay graph for the clique of streams.
- N.B.: Only linearly-recursive programs can be written in *μCryptol*. This appears to be all you need for encryption protocols.

...But can we trust the compiler's type system?

galois

Termination is verifiable!

```
rec
  index : B^8^inf;
  index = [0] ## [ x + 1 | x <- index];
and
  facs : B^8^inf;
  facs = [1] ## [ x * y | x <- facs
                        | y <- drops{1} index];
```

```
(defun fac-measure (i s)
  (acl2-count
    (+ (* (+ i (cond ((eq s 'facs) 0)
                     ((eq s 'index) 0))) 2)
       (cond ((eq s 'facs) 1)
             ((eq s 'index) 0)))))
```

All termination proofs are *automatic* in *ACL2*.

galois

Contributed *ACL2* Book: Cryptographic Primitives

- **Arithmetic in $\mathbf{Z}_{2^n}$ (arithmetic modulo 2^n)**: addition, negation, subtraction, multiplication, division, remainder after division, greatest common divisor, exponentiation, base-two logarithm, minimum, maximum, and negation.
- **Bitvector operations**: shift left, shift right, rotate left, rotate right, append of arbitrary width bitvectors, extraction of *n* bitvectors from a bitvector, append of fixed-width bitvectors, split into fixed-width bitvectors, bitvector segment extraction, bitvector transposition, reversal, and parity.
- **Arithmetic in $\mathbf{GF}_{2^n}$ (the Galois Field over 2^n)**: polynomial addition, multiplication, division, remainder after division, greatest common divisor, irreducibility, and inverse with respect to an irreducible polynomial.
- **Pointwise extension of logical operations to bitvectors**: bitwise conjunction, bitwise disjunction, bitwise exclusive-or, and negation bitwise complementation.
- **Vector operations**: shift left, shift right, rotate left, rotate right, vector append for an arbitrary number of vectors, extraction of *n* subvectors extraction from a vector, flattening a vector vectors, building a vector of vectors from a vector, taking an arbitrary segment from a vector, vector transposition, and vector reverse.

|galois|

Correspondence Proof

- We prove that for a well-formed indexed *μCryptol* program, its canonical representation is observationally equivalent.
- Example: Factorial Proof

```
(make-thm :name |inv-facs-thm|
          :thm-type invariant
          :ind-name |idx_2_facs_2|
          :itr-name |iter_idx_facs_3|
          :init-hist ((0) (0))
          :hist-widths (0 0)
          :branches (|idx_2| |facs_2|))
```

This top-level macro call, with the appropriate keys, generates the necessary lemmas and correspondence theorem.

Two Problems for Automated Proof Generation

Two problems:

- The proof infrastructure must be general enough to automatically prove correspondence for arbitrary programs.
- The proof infrastructure must not fall over on real programs (getting factorial to work took a day; AES took a couple of months).
  - Type declarations hundreds of lines long (e.g., `B^8^4^4^11`).
  - Programs easily reaching more than a thousand lines (AES) in *ACL2*.

Some Mitigations: why *ACL2* was the right tool

The two difficulties are mitigated by *ACL2* (and its community):

- Generality:
  - *ACL2 user-books*: Use powerful *ACL2* books, particularly Rockwell Collins' `super-ihs` book for reasoning about arithmetic over bit-arrays (slated for public release).
  - *Macro language*: For any other "hard" lemmas, use macros. Instantiate macros with concrete values (usually making their proofs trivial) and prove them at "run-time" – these are usually bitvector theorems where we want to fix the width of the bitvectors.
- Scaling:
  - *Disabling*: Package up large conjunctions in recursive definitions to prevent gratuitous expensive rewrites. Disable expensive formulas.
  - *Hints*: "Cascading" computed hints that iteratively enable definitions after successive occurrences of being stable under simplification.

galois

What could have helped even more?

- A better way to find/search books (e.g., priorities on hints).
- Better integration with decision procedures/SMT (solvers)?
- Heuristics for searching for inconsistent hypotheses (e.g., induction step showing that the hyp. of the induction conclusion implies the hyp. of the induction hyp.). E.g.,

```
(implies (true-listp a)
         (equal (rev (rev a)) a))

Subgoal *1/2
(implies (and (not (endp A))
              (not (true-listp (cdr A)))
              (true-listp A))
         (equal (rev (rev A)) A))
```

Don't rewrite `(equal (rev (rev A)) A)`!

galois

Dirty (Clean?) Laundry

How hard was all this? Regarding the first author,

- Experience:
  - Some *Common Lisp* experience.
  - Little compiler experience.
  - Little *ACL2* experience.
  - No *μCryptol* experience.
  - No *AAMP7* experience.
- Effort:
  - Approx. 3 months to complete the core verifier.
  - About 2 months investigating back-end verification.

DSL verifying compilers *are* feasible!

What's Left?

- Front end: in *Isabelle* (because of higher-order language constructs); just a few transformations and pattern-matching.
- Back-end: more substantial: Galois helped do an initial cutpoint-proof of factorial on the *AAMP7*.

Without the *AAMP7* model, the back-end verification is infeasible: stay tuned for the *next* talk!

Additional Resources

Example μ*Cryptol* & *ACL2* specs and cryptographic primitives

`http://www.galois.com/files/core_verifier/`

μ*Cryptol* design and compiler overview (solely authored by M. Shields)

`http://www.cartesianclosed.com/pub/mcryptol/`

μ*Cryptol* Reference Manual (solely authored by M. Shields)

`http://galois.com/files/mCryptol_refman-0.9.pdf`

galois

Appendix.

Transformations: Source to Canonical

Front-End Transformations

1. Introduce safety checks
2. Simplify vector comprehensions
3. Eliminate patterns
4. Convert to indexed form

Indexed Form Generated

Begin Core Transformations

5. Push stream applications
6. Collapse arms
7. Align arms
8. Takes/segments to indexes
9. Convert to iterator form
10. Eliminate simple primitives
11. Eliminate zero-sized values
12. Inline and simplify
13. Introduce temporaries
14. Eliminate nested definitions
15. Share top-level value definitions
16. Box top-level definitions
17. Eliminate shadowing

Canonical Form Generated

What Made *ACL2* the Right Tool

Or... "How an *ACL2* novice can quickly do something useful."

- Powerful and easy *macros*:
  - Avoid (hard) general proofs by simple instantiation of parameters.
  - Simplifies creating a "proof framework" that is essential for an automated verifying compiler.
- "Industrial strength prover" – able to handle models as large as the *AAMP7* model and easily generate proofs tens of thousands of lines long.
- "First-order" language forces the user to consider specifications that have more automated proofs from the get-go.
- A large number of active expert users.
- Good documentation.
- Powerful user-defined books (e.g., `ihs` books).

Correspondence Proof

We prove the following property for the *core* transformations: for index-form program S and compiled canonical program C,

"If S has well-defined semantics (does not go wrong), then S and C are observationally equivalent."

– Xavier Leroy
Formal Certification of a Compiler Back-end
POPL 2006

Well-Definedness

The "*stream delay* from stream x to occurrence of stream y is d" means, for sufficiently large index $k \in \mathbb{N}$, that the k'th element of stream x depends on the value of the $(k - d)$'th element of stream y.

Let S be the set of stream names defined by a mutually-recursive clique of stream definitions. Then we say the clique is *well defined* if there exists a *measure function*

$$f : (\mathbb{N} \times S) \to \mathbb{N}$$

such that for each occurrence of a stream y in the body of the definition of stream x with delay d, we have

$$\forall k \in \mathbb{N}.\, k \geq d \Rightarrow f(k - d, y) < f(k, x)$$

galois

Shallow Embedding

`mcc` contains a small (1.2klocs, excluding libraries) translator from *μCryptol* to *Common Lisp* (the translator is unverified). Some highlights:

- *μCryptol* types as *ACL2* predicates: `B^32^2`,

```
(defund |$ind_0_typep| (x)
 (and (true-listp x)
      (natp (nth 0 x))
      (< (nth 0 x) 4294967296)
      (natp (nth 1 x))
      (< (nth 1 x) 4294967296)))
```

  `defund`ed because AES has types like `B^8^4^4^11`.

- *μCryptol* primitives: . . .

Proof Macros

Correspondence proofs are generated from a few macros:

- **Function correspondence theorems** of non-recursive definitions.
- **Type correspondence theorems** of type declarations.
- **Vector comprehension correspondence theorems**.
- **Stream-clique correspondence theorems** of recursive cliques of stream comprehensions.
- **Vector-splitting correspondence theorems** of type correspondence for vectors that have been `split` into a vector of subvectors.
- **Inlined `segments/takes` correspondence theorems** for inlined `segments` and `takes` operators over streams.

Factorial Correspondence Theorem

```
(defthm factorial-invariant
 (implies
  (and (natp i) (natp lim)
       (true-listp hist) (<= i (+ lim 1))
       (equal (nth (loghead 0 i) (nth 0 hist))
              (ind-facs i 'idx))
       (equal (nth (loghead 1 i) (nth 1 hist))
              (ind-facs i 'facs)))
  (and (equal (nth (loghead 0 lim)
              (itr-facs i lim hist)
              (ind-facs lim 'idx))
       (equal (nth (loghead 1 lim)
              (itr-facs i lim hist)
              (ind-facs lim 'facs)))))
```

Linear Recursion

Informally, a sequence

$$a_0, a_1, \ldots$$

is *linear recursive*[3] if

$$a_{n+k} = -\frac{c_{k-1}}{c_k} a_{n+k-1} - \cdots - \frac{c_1}{c_k} a_{n+1} - \frac{c_0}{c_k} a_n.$$

for constants $c_0, c_1, \ldots, c_k$, where $c_k \neq 0$.

[3] Obtained at `http://mathcircle.berkeley.edu/BMC3/Bjorn1/node3.html`.

|galois|