| galois |

ACL2 Challenge Problem: Formalizing BitCryptol

April 20th, 2005

John Matthews

Galois Connections

matthews@galois.com

Roadmap

- SHADE verifying compiler
- Deeply embedding Cryptol semantics in ACL2
- Challenge problem
- A possible long-term solution

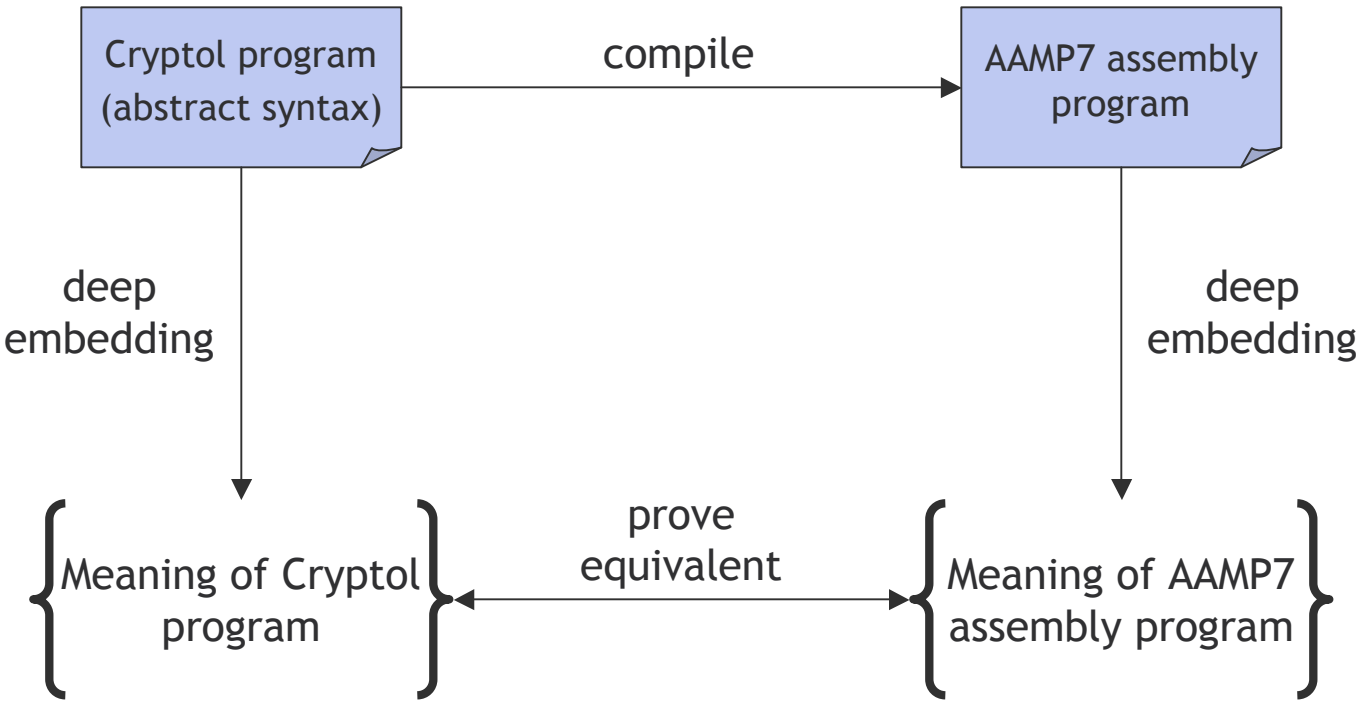
Motivation

- Programming errors can significantly weaken the strength of a cryptographic algorithm
- Flawed crypto implementations can have grave consequences
- Difficult to have crypto experts perform code-to-spec reviews of increasingly numerous crypto implementations
  - Trusted experts are scarce
  - Review process is expensive
  - Optimized designs are complex, easy to overlook corner cases

High Assurance AAMP7 Crypto software

- SHADE project goals
  - *Secure, High Assurance Development Environment* for AAMP7 software
  - Develop secure software in multiple languages, including Cryptol
  - Runs on AAMP7 microprocessor
    - Supports data and timing separation in hardware
  - Collaboration with Rockwell Collins, UT Austin
- Galois SHADE goals:
  - Crypto algorithms written in a high-level domain-specific declarative programming language (Cryptol)
  - SHADE Cryptol compiler generates efficient AAMP7 code
  - Compiler also generates formal proof that AAMP7 code is correct
  - Proof is automatically certified by ACL2. The SHADE compiler is **not** trusted.

SHADE Correctness property

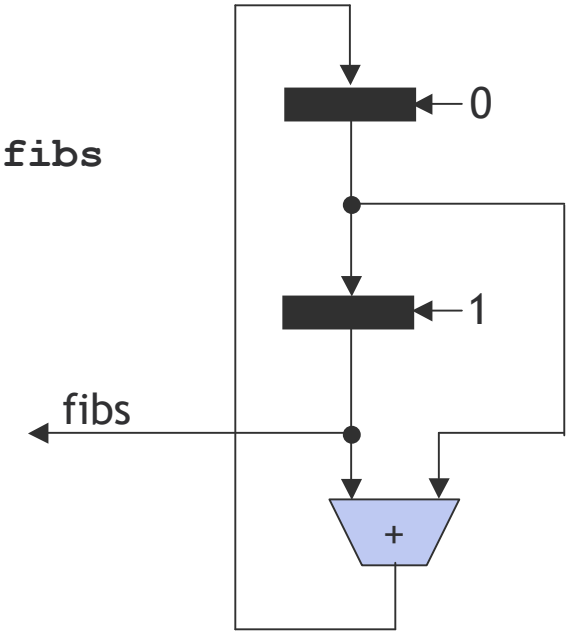


Example

- Simple Cryptol program for calculating Fibonacci numbers (mod 2^{32})

```
Word : Type;
Word = B^32;

rec
  fibs : Word^inf[32, 2];
  fibs = [0, 1] ## [x + y | x <- drops{1} fibs
                          | y <- fibs];
```



- Elements calculated by fibs:

```
fibs = [0, 1, 1, 2, 3, 5, 8, 13, 21, …]
```

Step 1: Formalize Fib spec in ACL2

```
(defun fib-spec (n)
  (cond
    ((not (integerp n))   0)
    ((< n 1)              0)
    ((equal n 1)          1)
    (t                    (logext *word-size* (+ (fib-spec (- n 1))
                                                 (fib-spec (- n 2)))))))
```

- Currently: **`fibs`** is shallowly embedded in ACL2
- Goal: Deeply embed **`fibs`** using semantics function for entire Cryptol language (i.e. a Cryptol interpreter)
  - Inputs:
    - Abstract syntax for a Cryptol expression
    - Association list mapping free variables to their values
  - Output:
    - Value for Cryptol expression
- Prove that shallow embedding is equivalent to deep embedding

Remaining verification steps

2. Compile **`fibs`** to AAMP7 assembly code
3. Generate cutpoint assertions
   - entrypoint assertion: top-of-stack is input value **`n`**
   - exitpoint assertion: top-of-stack is **`(fib-spec n)`**
   - cutpoint assertions: current stack, memory locations correspond to particular **`fib-spec`** subexpressions

     …plus lots of environmental assumptions and frame assertions
4. Symbolically simulate AAMP7 from each cutpoint to verify assertions
5. Use cutpoint measures to show AAMP7 code terminates

For more details, see

A Symbolic Simulation Approach to Assertional Program Verification, J. Matthews, J Moore, S. Ray, D. Vroon

Problem: we can't get past step 1!

- Issue: Cryptol has lazy streams (infinite sequences) as data values. But, the type system ensures that
  - top-level Cryptol program only looks at a finite prefix of any stream
  - all stream definitions must be well-founded, with finite history.
  - Result: Any (μ)Cryptol program can be compiled to efficient code by implementing streams as finite circular buffers
- We've tried several ways to deeply embed lazy streams in ACL2, without success
- So, I've created an ACL2 challenge problem
  - Stripped-down subset of Cryptol, called *BitCryptol*
  - Contains just enough features to demonstrate the problem

BitCryptol types

- A BitCryptol value can be only one of three types
  - Bit (i.e. Boolean)
  - Stream index (i.e. a natural number)
  - A stream of bits (i.e. a function taking a natural number as input and returning a bit as output)

BitCryptol semantics

- Next, I'll give some equations the BitCryptol semantics function must satisfy for each well-formed BitCryptol expression

```
((sem * *) => *)
```

  - First argument: expression
  - Second argument: environment of free variable values (as an association list)
  - Result: value of expression in the environment
- These equations will be heavily used in the equivalence proof of the deep-to-shallow embedding

Bits and stream indexes

Bit constants

```
(sem nil env) = nil
(sem t env) = t
```

Stream index constant

```
(sem n env) = n
```
[*Assuming* `(natp n)`]

Exclusive-or

```
(sem `(Xor ,b1 ,b2) env)
  = (not (equal (sem ,b1 env)
                (sem ,b2 env)))
```

Variables (of any type)

Variable

```
(sem v env)
 = (cdr (assoc-equal v env))
```

Streams

Comprehension

```
(sem `(SLam ,v ,b) env)
 = (lambda (n)
     (let ((newenv `((,v . ,n) ,@env)))
       (sem b newenv)))
```

Application

```
(sem `(At ,s ,n) env)
 = (apply (sem s env) (sem n env))
```

Streams

Cons

```
(sem `(SCons ,b ,s) env)
 = (lambda (n)
     (if (zp n)
         (sem b env)
       (apply (sem s env) (- n 1))))
```

Tail

```
(sem `(STail ,s) env)
 = (lambda (n)
     (apply (sem s env) (+ n 1)))
```

Local stream definitions

- Can be mutually-recursive, and nested
  - HOL semantics also enforces well-foundedness

```
(sem `(SWhere ,s
         ((,v1 . ,s1)
          (,v2 . ,s2)
          ...))
     env)
 = (LETREC ((recenv `((,v1 . ,f1)
                      (,v2 . ,f2)
                      ...
                      ,@env))
            (f1 (lambda (n) (apply (sem s1 recenv) n)))
            (f2 (lambda (n) (apply (sem s2 recenv) n)))
            ...)
     (sem s recenv))
```

HOL semantics of a recursive stream

- To get an idea of the technique we use to formalize recursive stream definitions in Isabelle (without needing domain theory), I'll show the semantics of a **`SWhere`** expression containing a single recursive stream definition:

```
(sem `(SWhere ,s ((,v . ,d))) env)
= (let* ((d_fcn (lambda (s') (sem d `((,v . s') ,@env))))
         (d_strm (lambda (n) (wf_fix_stream d_fcn n))))
    (sem s `((,v . d_strm) ,@env)))
```

- The definition of **`wf_fix_stream`** is well-founded on stream index **`n`**. Parameter **`f`** is a function from streams to streams.

```
(defun wf_fix_stream (f n)
  (f (lambda (k)
       (if (< k n)
           (wf_fix_stream f k)
         nil)))
     n))
```

BitCryptol example program

- Calculates the least significant bit of the N^{th} Fibonacci number

```
`(SWhere (At "fib" ,N)
   (("fib"       . (SCons nil "fib-tail"))
    ("fib-tail" . (SCons t (SLam "n"
                             (Xor (At (STail "fib") "n")
                                  (At "fib" "n")))))))
```

- Assuming **N** is a natural number greater than 1, then

```
(apply (sem "fib" recenv) N)
= (apply (sem "fib-tail" recenv) (- N 1))
= (apply (sem `(SLam "n" …) recenv) (- N 2))
= …
= (xor (apply (sem "fib" recenv) (- N 1))
       (apply (sem "fib" recenv) (- N 2)))
```

Our deep embedding approach so far

- Problem: no function objects in ACL2
- Idea: represent streams syntactically in the environment
  - Add an *observation parameter* to the input
    - If expression is a finite vector, then observation is value itself
    - If expression is a function, then observation is value to apply function to
    - If expression is a stream, then observation is stream index
  - Add *high-water* indexes to stream bindings in environment
    - Set to ω when stream variable is defined
  - When looking up stream variable in environment:
    1. Check that observation parameter is less than high-water mark
    2. Update stream variable's high-water index to be the observation parameter
    3. Evaluate expression that the stream variable was bound to (with the current observation parameter)

And the measure is…?

- Intuitively, it seems like the semantics should terminate. We've tried a series of increasingly complex measures, but haven't found one that works:
  - Expression measure
  - Expression+environment measure
  - ω-polynomial measure

Expression measure

- Uses **`acl2-count`** on size of expression
- Problem: looking up stream variables

Expression+environment measure

- Also takes the size of the environment into account
  - Size of each each stream binding is ω * (high-water mark) + (size of stream expression)
  - Size of expression is ω * (observation parameter) + (size of expression)
- Problem 1: **`STail`** expressions increment the observation parameter
- Problem 2: **`SWhere`** expressions increase size of environment by adding new environment entries

ω-polynomial measure

- Makes **`SWhere`** expressions larger than the stream bindings they introduce
  - Define a recursive measure function on expressions, that returns an *ω-polynomial*
  - Size of most expressions based on **`acl2-count`**
  - Size of a stream comprehension is ω * (size of stream body)
  - Size of a **`SWhere`** expression is sum of stream definitions, plus the size of the body
  - Size of stream variable binding in environment is (high-water index) * (size of stream)
  - Use this measure function in expression+environment measure
- Had to build an ACL2 book for ω-polynomial arithmetic
- Problem: What is the measure of a stream variable in an expression, or in a set of recursive stream definitions?

HOL semantics

- At this point we gave up
- Instead, we developed a core subset of language, called femtoCryptol
  - bitvectors
  - tuples
  - mutully recursive, nested stream definitions
  - stream comprehensions
- Able to give a deep embedding for femtoCryptol in Isabelle
  - Defined a fixpoint operator for environments of well-founded stream transformers

```
types 'a stream  = nat => 'a
      'a senv    = string => ('a stream) option

fix_senv :: 'a => string set => ('a senv => 'a senv)
            => 'a senv"
```

- Validated the semantics on simple femtoCryptol programs, like fib

A modest proposal

- *Warning*: 1/2-baked ideas from here on out

- Formalize higher order functions, as a new ACL2 theory
  - N.B: higher order functions ≠ higher order logic!
  - But do need to avoid logical paradoxes
- Goals
  - First-order ACL2 theory
  - No types
  - Minimal (or maybe no) changes to ACL2 core.
  - Simple correspondence between ACL2 functions and function objects
  - Powerful enough to deeply embed Cryptol
    - Ideally: prove any theorem of classical higher order logic
  - Function objects are executable

Function objects

- Add function objects as a new kind of atom to ACL2 universe
- "Good" atoms and function objects are stratified into *ranks*
  - Existing ACL2 objects have rank zero
  - A function object has rank *n* + 1 if it returns a rank *n* object when applied to a rank *n* object, and returns **`nil`** otherwise
  - Any rank *n* object also has rank *n* + 1
- **`apply`** operator applies a function object to a value
  - May not give the value you expect if rank of the function's argument is too large
  - Should be able to prove that **`apply`** can't be ranked

Function comprehension axiom schema

- If this formula is provable about ACL2 expressions *e* and *n*

```
(and (natp n)
     (forall (x) (if (has_rank x n)
                     (has_rank e n)
                   (null e))))
```

- Then this formula is valid in the theory

```
(let ((f (lambda (x) e)))
   (and (has_rank f (+ n 1))
        (equal (apply f x) e )))
```

Properties about `apply`

- These properties seem useful for writing *rank-polymorphic* ACL2 functions
- Extensionality

```
(implies (and (has_rank f n)
              (has_rank g n)
              (forall (x)
                (equal (apply f x)
                       (apply g x)))
         (equal f g))
```

- `has_rank` is stratified

```
(implies (has_rank f (+ n 1))
         (has_rank (apply f x) n))
```

Questions about function objects

- Is this theory a conservative extension of ACL2?
- Does it work in the presence of local events and functional instantiation?
- Is it powerful enough?
- What changes have to be made to the core of ACL2?

Even more modest proposals

- Matt Kaufmann: Use ideas from domain theory and **`defchoose`** to give semantics of Cryptol in current ACL2 framework
  - Model streams as a series of increasingly precise finite approximations
  - Use **`defchoose`** to pick a good enough approximation for the desired stream index observation
- Thomas Nordin: Change semantics of streams to be finite, but large, sequences.
  - Key idea: SCons, SDrop do not change the size of the sequence
  - Allows for recursive stream definitions
  - Many laws would hold as is, some would have to be weakened a bit
- Warren Hunt: Remove **`STail`** expressions from Cryptol
  - Would make semantics much easier to formalize in ACL2
  - Any given Cryptol program can be rewritten to not use **`STail`**

Questions