Pragmatic Equivalence and Safety Checking in Cryptol

Levent Erkök John Matthews
{levent.erkok,matthews}@galois.com

PLPV'09; Savannah, GA

January 2009

| galois |

The Cryptol Project

Goal: To reduce the cost of developing, certifying, and deploying cryptographic applications

The Cryptol Project

Goal: To reduce the cost of developing, certifying, and deploying cryptographic applications

- A Domain Specific Language
  - High level design exploration
  - Fully executable

The Cryptol Project

Goal: To reduce the cost of developing, certifying, and deploying cryptographic applications

- A Domain Specific Language
  - High level design exploration
  - Fully executable
- Automated Synthesis down to FPGAs

The Cryptol Project

Goal: To reduce the cost of developing, certifying, and deploying cryptographic applications

- A Domain Specific Language
  - High level design exploration
  - Fully executable
- Automated Synthesis down to FPGAs
- Verification tool-chain
  - SAT/SMT based property checking
  - Safety checking
  - QuickCheck
  - Translation to Isabelle/HOL

Cryptol Type System

- Captures bit-precise size-type relations
- Hindley-Milner + arithmetic constraints
    - Both linear and non-linear operations

Cryptol Type System

- Captures bit-precise size-type relations
- Hindley-Milner + arithmetic constraints
  - Both linear and non-linear operations
- Numeric literals are one source of constraints:

```
13 : {a} (a >= 4) => [a]
```

"The literal 13 is represented by a bit vector that requires at least 4 bits to represent"

Cryptol Type System

- Captures bit-precise size-type relations
- Hindley-Milner + arithmetic constraints
  - Both linear and non-linear operations
- Numeric literals are one source of constraints:

```
13 : {a} (a >= 4) => [a]
```

"The literal 13 is represented by a bit vector that requires at least 4 bits to represent"

- Arbitrary arithmetic expressions as constraints:

```
split : {a b c} [a*b]c -> [a][b]c
```

Cryptol Type System

- Captures bit-precise size-type relations
- Hindley-Milner + arithmetic constraints
  - Both linear and non-linear operations
- Numeric literals are one source of constraints:

```
13 : {a} (a >= 4) => [a]
```

  "The literal 13 is represented by a bit vector that requires at least 4 bits to represent"

- Arbitrary arithmetic expressions as constraints:

```
split : {a b c} [a*b]c -> [a][b]c
```

- NB. Size types; **not** dependent types!

Capturing Cryptography

- From AES standard definition:

  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.*

Capturing Cryptography

- From AES standard definition:

  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.*

In Cryptol:

Capturing Cryptography

- From AES standard definition:

  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.*

In Cryptol:

[128]

Capturing Cryptography

- From AES standard definition:

  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.*

In Cryptol:

$[128] \rightarrow [128]$

Capturing Cryptography

- From AES standard definition:

  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.*

In Cryptol:

$$([128]\ ,\ [64*k]) \rightarrow [128]$$

Capturing Cryptography

- From AES standard definition:

  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.*

In Cryptol:

$$\texttt{(k >= 2, 4 >= k)} \Rightarrow \texttt{([128] , [64*k])} \rightarrow \texttt{[128]}$$

Capturing Cryptography

- From AES standard definition:

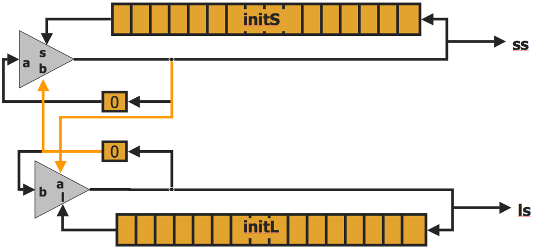
  The **input** *and* **output** *for the AES algorithm each consist of* **sequences of 128 bits**. *... The* **Cipher Key** *for the AES algorithm is a* **sequence of 128, 192 or 256 bits**. *Other input, output and Cipher Key lengths are not permitted by this standard.*

In Cryptol:

```
{k} (k >= 2, 4 >= k) ⇒ ([128] , [64*k]) → [128]
```

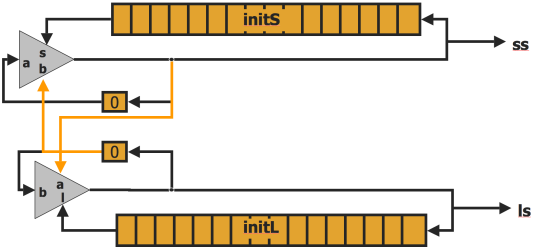
A taste of Cryptol expressions

Informal circuit diagrams are often used by cryptographers:



A taste of Cryptol expressions

Informal circuit diagrams are often used by cryptographers:



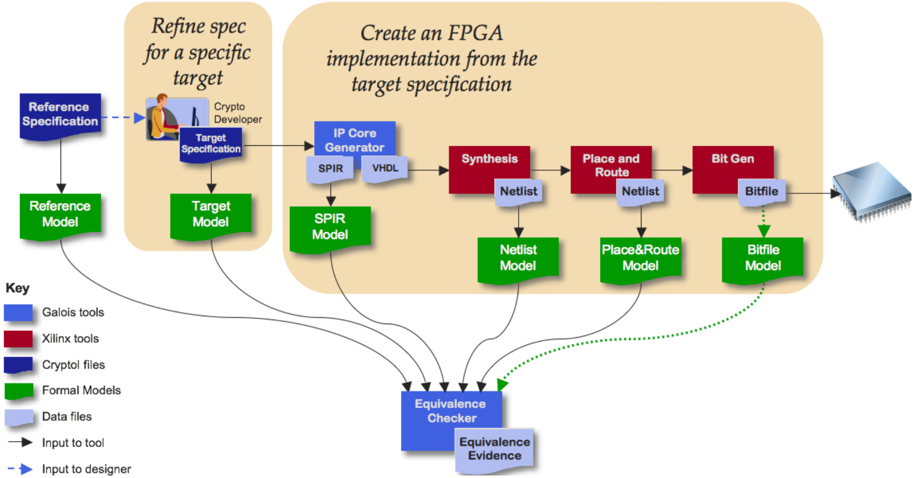
Code (Cryptol implementation)

```
ss = [| (s+a+b) <<< 3 || s <- initS # ss
                      || a <- [0] # ss
                      || b <- [0] # ls |];

ls = [| (l+a+b) <<< (a+b) || l <- initL # ls
                          || a <- ss
                          || b <- [0] # ls |];
```


galois

Cryptol verification flow



galois

High Assurance Cryptol

- "The" original motivation
- Equivalence checking at various levels:
  - Cryptol vs. Cryptol
  - Cryptol vs. generated VHDL/Netlist
  - Cryptol vs. hand-written VHDL
  - [Future] Cryptol vs. bit-file
- Key component in crypto-evaluation
- "Verifying" compiler approach
  - Found several Cryptol-FPGA compiler bugs already!
- Stepwise refinement with confidence

Verification desiderata

- Push button
- Full coverage of Cryptol
- Fast

Verification desiderata

- Push button
- Full coverage of Cryptol
- Fast
- What we have
  - Push button (but manual option available when needed)
  - Good coverage of Cryptol
  - Fast enough (most of the time)

galois

Verification desiderata

- Push button
- Full coverage of Cryptol
- Fast
- What we have
  - Push button (but manual option available when needed)
  - Good coverage of Cryptol
  - Fast enough (most of the time)
- Theoretical limits
  - Full problem is undecidable
  - Equivalent to solving the halting problem

Compromise

- Restrict the language subset
  - Monomorphic
  - Finite
  - First-order
  - Symbolically terminating

Compromise

- Restrict the language subset
  - Monomorphic
  - Finite
  - First-order
  - Symbolically terminating
- Bad news: The problem remains NP-Complete!
  - Easy reduction to 3-SAT
- Good news: Most practical instances are feasible
  - Thanks to the advances in SAT/SMT technologies

Outline

Equivalence checking

- Given two Cryptol functions f, g
  - Either prove they agree on all inputs
  - Or, provide a counter-example
- Typically:
  - f: Spec, written for clarity
  - g: Implementation, optimized for speed/space/FPGA etc.

Boolean functions are theorems!

Let

```
f, g, h : [8] -> [8];
f x = (x-1)*(x+1);
g x = x*x - 1;
h x = x*x + 1;
theorem FG: {x}. f x == g x;
theorem FH: {x}. f x == h x;
```

Boolean functions are theorems!

Let

```
f, g, h : [8] -> [8];
f x = (x-1)*(x+1);
g x = x*x - 1;
h x = x*x + 1;
theorem FG: {x}. f x == g x;
theorem FH: {x}. f x == h x;
```

- No need to learn a new language!

Boolean functions are theorems!

Let

```
f, g, h : [8] -> [8];
f x = (x-1)*(x+1);
g x = x*x - 1;
h x = x*x + 1;
theorem FG: {x}. f x == g x;
theorem FH: {x}. f x == h x;
```

- No need to learn a new language!

Prover in action

```
Cryptol> :prove FG
Q.E.D.
Cryptol> :prove FH
Falsifiable.
FH 60
         = False
```


galois

Safety checking

- Given a function f
  - Either prove that nothing bad will happen at run-time
  - Or, provide a counter-example
- Statically catch:
  - Index out-of-bounds
  - `ASSERT`ion failures
  - Uses of `error` and `undefined`
  - Division/modulus by 0
  - Polynomial division/modulus by 0
  - Logarithm of zero

Safety checking

- Given a function *f*
  - Either prove that nothing bad will happen at run-time
  - Or, provide a counter-example
- Statically catch:
  - Index out-of-bounds
  - `ASSERT`ion failures
  - Uses of `error` and `undefined`
  - Division/modulus by 0
  - Polynomial division/modulus by 0
  - Logarithm of zero
- Safe programs *really* don't crash!

Checking safety - Index out of bounds - I

Let

```
lkup1 : ([4][2], [2]) -> [2];
lkup1 (xs, i) = xs @ i;
```

Checking safety - Index out of bounds - I

Let

```
lkup1 : ([4][2], [2]) -> [2];
lkup1 (xs, i) = xs @ i;
```

We have

```
Cryptol> :safe lkup1
"lkup1" is safe; no safety violations exist.
```


galois

Index out of bounds - II

Let

```
lkup2 : ([6][2], [3]) -> [2];
lkup2 (xs, i) = xs @ i;
```


galois

Index out of bounds - II

Let

```
lkup2 : ([6][2], [3]) -> [2];
lkup2 (xs, i) = xs @ i;
```

We have

```
Cryptol> :safe lkup2
*** Violation detected:
lkup2 ([0 0 0 0 0 0], 6)
   = index of 6 is out of bounds (valid range is 0 thru 5).
```

Index out of bounds - III

Let

```
lkup3 : ([6][2], [3]) -> [2];
lkup3 (xs, i) = if i >= 6 then 0 else xs @ i;
```

Index out of bounds - III

Let

```
lkup3 : ([6][2], [3]) -> [2];
lkup3 (xs, i) = if i >= 6 then 0 else xs @ i;
```

We have

```
Cryptol> :safe lkup3
*** 1 safety condition to be checked.
*** line 2, col 42: index out of bounds
*** Verified safe.
*** All safety checks pass, safe to execute.
```

Index out of bounds - IV

Let

```
lkup4 : ([6][2], [3]) -> [2];
lkup4 (xs, i) = if i > 6 then 0 else xs @ i;
```


galois

Index out of bounds - IV

Let

```
lkup4 : ([6][2], [3]) -> [2];
lkup4 (xs, i) = if i > 6 then 0 else xs @ i;
```

We have

```
Cryptol> :safe lkup4
*** Violation detected:
lkup4 ([0 0 0 0 0 0], 6)
   = index of 6 is out of bounds (valid range is 0 thru 5).
```

Summary so far

- Fully automated
- No separate verification language
- Properties are first class

Summary so far

- Fully automated
- No separate verification language
- Properties are first class
- Other tools available:
    - Checking satisfiability
    - Check against VHDL
    - Check against C
    - QuickCheck
    - Automatic translation to Isabelle/HOL
        - Custom "Cryptol" theory for aiding in manual proof

galois

Outline

Verification strategy

- Given a Cryptol function `f`
  - Run *f* symbolically on its input
  - Generate “code” as execution proceeds
  - Generate “verification conditions” for checking safety
- The residual thus generated is the “formal model” of `f`
- Translate the “formal model” to AIG/SMT-Lib

Verification strategy

- Given a Cryptol function `f`
  - Run *f* symbolically on its input
  - Generate “code” as execution proceeds
  - Generate “verification conditions” for checking safety
- The residual thus generated is the “formal model” of `f`
- Translate the “formal model” to AIG/SMT-Lib
- To show `f` and `g` equivalent:
  - Show that their formal models are equivalent

Verification strategy

- Given a Cryptol function `f`
  - Run *f* symbolically on its input
  - Generate "code" as execution proceeds
  - Generate "verification conditions" for checking safety
- The residual thus generated is the "formal model" of `f`
- Translate the "formal model" to AIG/SMT-Lib
- To show `f` and `g` equivalent:
  - Show that their formal models are equivalent
- To prove a theorem:
  - Exploit the fact that theorems are boolean-functions
  - Show that it is equivalent to the constant function that always returns `True`

Example

Cryptol Program:

```
f : [8] -> [2][8];
f x = [y z]
 where {
     y = g (x+1);
     z = h (x, y);
 };

g : [8] -> [8];
g x = 2 * x;

h : ([8], [8]) -> [8];
h (x, y) = if x > y
           then x
           else y+1;
```


galois

Example

Cryptol Program:

```
f : [8] -> [2][8];
f x = [y z]
 where {
    y = g (x+1);
    z = h (x, y);
 };

g : [8] -> [8];
g x = 2 * x;

h : ([8], [8]) -> [8];
h (x, y) = if x > y
           then x
           else y+1;
```

Formal Model for f:

```
INPUT s0:[8]
```

Notes:

s0 ← x

Example

Cryptol Program:

```
f : [8] -> [2][8];
f x = [y z]
 where {
    y = g (x+1);
    z = h (x, y);
 };

g : [8] -> [8];
g x = 2 * x;

h : ([8], [8]) -> [8];
h (x, y) = if x > y
           then x
           else y+1;
```

Formal Model for f:

```
INPUT s0:[8]
s1:[8] = s0 + 1
```

Notes:

s0 $\leftarrow$ x

Example

Cryptol Program:

```
f : [8] -> [2][8];
f x = [y z]
 where {
    y = g (x+1);
    z = h (x, y);
 };

g : [8] -> [8];
g x = 2 * x;

h : ([8], [8]) -> [8];
h (x, y) = if x > y
           then x
           else y+1;
```

Formal Model for f:

```
INPUT s0:[8]
s1:[8] = s0 + 1
s2:[8] = s1 * 2
```

Notes:

s0 ← x

s2 ← y {= g (x+1)}

Example

Cryptol Program:

```
f : [8] -> [2][8];
f x = [y z]
 where {
    y = g (x+1);
    z = h (x, y);
 };

g : [8] -> [8];
g x = 2 * x;

h : ([8], [8]) -> [8];
h (x, y) = if x > y
           then x
           else y+1;
```

Formal Model for f:

```
INPUT s0:[8]
s1:[8] = s0 + 1
s2:[8] = s1 * 2
OUTPUT s2
```

Notes:

s0 ← x

s2 ← y {= g (x+1)}

Example

Cryptol Program:

```
f : [8] -> [2][8];
f x = [y z]
 where {
     y = g (x+1);
     z = h (x, y);
 };

g : [8] -> [8];
g x = 2 * x;

h : ([8], [8]) -> [8];
h (x, y) = if x > y
           then x
           else y+1;
```

Formal Model for f:

```
INPUT s0:[8]
s1:[8] = s0 + 1
s2:[8] = s1 * 2
OUTPUT s2
s3:[1] = s0 > s2
```

Notes:

```
s0 ← x
s2 ← y {= g (x+1)}
s3 ← is x > g (x+1)?
```

Example

Cryptol Program:

```
f : [8] -> [2][8];
f x = [y z]
 where {
    y = g (x+1);
    z = h (x, y);
 };

g : [8] -> [8];
g x = 2 * x;

h : ([8], [8]) -> [8];
h (x, y) = if x > y
           then x
           else y+1;
```

Formal Model for f:

```
INPUT s0:[8]
s1:[8] = s0 + 1
s2:[8] = s1 * 2
OUTPUT s2
s3:[1] = s0 > s2
s4:[8] = s2 + 1
```

Notes:

```
s0 ← x
s2 ← y {= g (x+1)}
s3 ← is x > g (x+1)?
s4 ← else branch
```

Example

Cryptol Program:

```
f : [8] -> [2][8];
f x = [y z]
 where {
     y = g (x+1);
     z = h (x, y);
 };

g : [8] -> [8];
g x = 2 * x;

h : ([8], [8]) -> [8];
h (x, y) = if x > y
           then x
           else y+1;
```

Formal Model for f:

```
INPUT s0:[8]
s1:[8] = s0 + 1
s2:[8] = s1 * 2
OUTPUT s2
s3:[1] = s0 > s2
s4:[8] = s2 + 1
s5:[8] = ite s3 s0 s4
```

Notes:

```
s0 ← x
s2 ← y {= g (x+1)}
s3 ← is x > g (x+1)?
s4 ← else branch
```

Example

Cryptol Program:

```
f : [8] -> [2][8];
f x = [y z]
 where {
     y = g (x+1);
     z = h (x, y);
 };

g : [8] -> [8];
g x = 2 * x;

h : ([8], [8]) -> [8];
h (x, y) = if x > y
           then x
           else y+1;
```

Formal Model for f:

```
INPUT s0:[8]
s1:[8] = s0 + 1
s2:[8] = s1 * 2
OUTPUT s2
s3:[1] = s0 > s2
s4:[8] = s2 + 1
s5:[8] = ite s3 s0 s4
OUTPUT s5
```

Notes:

```
s0 ← x
s2 ← y {= g (x+1)}
s3 ← is x > g (x+1)?
s4 ← else branch
```

Notes on the formal model

- The only data-type is fixed-size bit-vectors
    - Types are serialized
- Original program completely unrolled
    - No functions, no loops
    - Essentially one huge expression per output!
        - With subexpression sharing..
- Easy to map to SMT-Lib or generate AIG

Outline

galois

Supported subset of Cryptol

- The automated verifier supports Cryptol functions that are:
  1. Monomorphic,
  2. Finite,
  3. First-order,
  4. Symbolically terminating.

Supported subset of Cryptol

- The automated verifier supports Cryptol functions that are:
  1. Monomorphic,
  2. Finite,
  3. First-order,
  4. Symbolically terminating.
- First three restrictions directly deduced from type

Supported subset of Cryptol

- The automated verifier supports Cryptol functions that are:
  1. Monomorphic,
  2. Finite,
  3. First-order,
  4. Symbolically terminating.
- First three restrictions directly deduced from type
- Last one is undecidable in general
  - But many instances are easily detectable..

Supported subset of Cryptol

- The automated verifier supports Cryptol functions that are:
  1. Monomorphic,
  2. Finite,
  3. First-order,
  4. Symbolically terminating.
- First three restrictions directly deduced from type
- Last one is undecidable in general
  - But many instances are easily detectable..
- This is still a very large and useful subset for Cryptol
  - Especially for block ciphers

The “monomorphism” restriction

- Symbolic simulator needs to have a fixed size input
- Underlying logic is fixed-size bit vectors
- Unfortunate: Most Crypto-algorithms are size-polymorphic
    - Luckily, only a few instances are typically important

The “monomorphism” restriction

- Symbolic simulator needs to have a fixed size input
- Underlying logic is fixed-size bit vectors
- Unfortunate: Most Crypto-algorithms are size-polymorphic
  - Luckily, only a few instances are typically important

Doubling a number

```
twice, twice' : {a} (a >= 2) => [a] -> [a];
twice  x = x+x;
twice' x = 2*x;
```

Equivalence at a = 4

```
Cryptol> :eq (twice : [4] -> [4]) (twice' : [4] -> [4])
True
```

The "monomorphism" restriction

- Symbolic simulator needs to have a fixed size input
- Underlying logic is fixed-size bit vectors
- Unfortunate: Most Crypto-algorithms are size-polymorphic
  - Luckily, only a few instances are typically important

Doubling a number

```
twice, twice' : {a} (a >= 2) => [a] -> [a];
twice  x = x+x;
twice' x = 2*x;
```

Equivalence at a = 4

```
Cryptol> :eq (twice : [4] -> [4]) (twice' : [4] -> [4])
True
```

- Can we just generalize?

Properties might rely on size!

A simple function

```
f : {a} (fin a) => [a] -> Bit;
f x = x != 0;
```


galois

Properties might rely on size!

A simple function

```
f : {a} (fin a) => [a] -> Bit;
f x = x != 0;
```

Use the satisfiability checker

```
Cryptol> :sat (f : [0] -> Bit)
No variable assignment satisfies this function
```

Properties might rely on size!

A simple function

```
f : {a} (fin a) => [a] -> Bit;
f x = x != 0;
```

Use the satisfiability checker

```
Cryptol> :sat (f : [0] -> Bit)
No variable assignment satisfies this function

Cryptol> :sat (f : [1] -> Bit)
((f : [3] -> Bit)) 1
      = True
```

- Satisfiable at any type except when `a = 0`
- Wanted: A theory of size-parametricity for Cryptol!

The "finiteness" restriction

- Symbolic simulator cannot represent infinite input/output
  - The formal model would have to be infinite..
- Such proofs typically require induction

The "finiteness" restriction

- Symbolic simulator cannot represent infinite input/output
    - The formal model would have to be infinite..
- Such proofs typically require induction
- Need to settle for finite prefixes:
    - Equivalence for the first K clock-cycles..

The "hidden counter" attack

Spec and implementation

```
pts : [inf][128];
pts = [0 ..];

spec, imp : [128] -> [inf][128];
spec k = [| pt + k || pt <- pts |];
imp  k = take(100, spec k) # pts;
```

- imp follows spec for the first 100 outputs
- Then it starts leaking the plain text!

Approval granted!

Equivalence tester in action

```
Cryptol> :eq spec imp
ERROR: "spec" has an infinite number of outputs
ERROR: "imp" has an infinite number of outputs
Cryptol> :eq (\k -> take(50, spec k)) (\k -> take(50, imp k))
True
Cryptol> :eq (\k -> take(100, spec k)) (\k -> take(100, imp k))
True
```

- Will be approved if equivalence checked up to first 100 cycles!

Check the 101st element!

Looking deeper..

```
Cryptol> :eq (\k -> spec k @ 100) (\k -> imp k @ 100)
False
(\k -> spec k @ 100) 87729047721804447611651265978502737985
        = 87729047721804447611651265978502738085
(\k -> imp k @ 100) 87729047721804447611651265978502737985
        = 0
```

- There is no general way to know how deep we need to look..
- Wanted: Induction capabilities in the equivalence checker!

The "first-order" restriction

- Only data type available is fixed-size bit vectors
- Tuples, records, finite sequences are all expanded away
- No way to represent functions..

The "first-order" restriction

- Only data type available is fixed-size bit vectors
- Tuples, records, finite sequences are all expanded away
- No way to represent functions..
- Luckily: Higher order functions are rare in Cryptol!
- Infrequent uses can mostly be rewritten away
- Wanted: (Maybe) Automatic firstification for Cryptol

The “symbolic termination” restriction

- Only applies to recursive functions
    - Uses are discouraged: Use streams instead
    - Recursive stream definitions are fine
- Bad news: Cannot tell ahead..
    - All other restrictions are detectable by just looking at the type
    - The prover will loop itself
- Good news: Typically easy to deal with once spotted..
    - See paper for an example

Outline

Conclusions

- Formal verification is not a luxury for Cryptol
- Basic pillar of Cryptol's high assurance approach
- Verified vs. Verifying compiler
  - Already found several bugs
- Compilation to non-standard targets
  - FPGAs
  - GPUs
  - Verification against hand-written VHDL
  - Formal equivalence is paramount
- Programming as if correctness mattered..
  - Encourages stepwise refinement
  - Maintain equivalence at each step

Thank you!

- Academic licenses available for the Cryptol interpeter
    - www.cryptol.net
    - (NB. No support for verification except for QuickCheck)
- Full version evaluation licenses expected soon