Formally Verified Encryption of High-Level Datatypes

Konrad Slind
Joe Hurd

School of Computing, University of Utah
Computer Lab, University of Cambridge

History

- Wanted nice functional programming example for class (2001)
- Picked AES, translated to ML
- Had a go at verifying functional correctness in HOL-4 (2002)
- Lessons: pretty easy ... and that's GOOD!

This Talk

- Review AES formalization, proofs
- Lessons of proofs
- Extension to a wide variety of datatypes

General Perspective

- HOL has internal FP language with all the usual stuff (pattern matching, type inference, polymorphism)
- Use that to formalize crypto algorithms at an abstract level
- Similar to Cryptol
- Use logic to verify functional correctness

Higher Order Logic

- Simple, powerful logic (Church 1943)
- Predicate logic + typed λ-calculus
- Quantification over predicates, functions, and sets
- Supports formalization of (near) arbitrary mathematics
- ‘Core’ ML and HOL have similar type systems
- We’ll ignore the differences

Higher Order Logic

- Reasoning about hardware and software can require fairly sophisticated mathematics
- IEEE floating point requires the real numbers and analysis
- Correctness of randomized algorithms requires probability
- We won’t use anywhere near this amount of power

Functional Specification

- Encryption followed by decryption should be the identity

$$Decrypt(key, Encrypt(key, input)) = input$$

- Should be easy to prove, even formally!
- We have done this for 128 bit keys
- Did not prove: encryption should be effectively impossible to invert without the key

Specification details

- The specification is phrased in terms of the finite field GF(2^8)
- This never enters into the proofs
- A block (the plaintext input) is a 16-tuple of 8-bit bytes
- A key is the same size as a block
- A state is a 4×4 block of bytes (notionally)

The state

- Accessed by byte, by row, and by column
- Modelled by a 16-tuple.

```
type state = word8 * word8 * word8 * word8 *
             word8 * word8 * word8 * word8 *
             word8 * word8 * word8 * word8 *
             word8 * word8 * word8 * word8
```

- Sometimes written to look like a matrix

```
(b00, b01, b02, b03,
 b10, b11, b12, b13,
 b20, b21, b22, b23,
 b30, b31, b32, b33)
```

Moving into and out of the state

- Into (stripe)

```
to_state (b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15)
                =
            (b0,b4,b8,b12,
             b1,b5,b9,b13,
             b2,b6,b10,b14,
             b3,b7,b11,b15)
```

- Back out (unstripe)

```
from_state (b0,b4,b8,b12,
            b1,b5,b9,b13,
            b2,b6,b10,b14,
            b3,b7,b11,b15)
       =
  (b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15)
```

Implementation—High Level View

- Before encryption, the key is used to produce a key schedule $[k_0, \ldots, k_{10}]$. These are xor-ed with the state in each round.

- Encryption

$$plaintext \xrightarrow{k_0} state_0 \xrightarrow{k_1} \ldots \xrightarrow{k_9} state_9 \xrightarrow{k_{10}} ciphertext$$

- Decryption

$$plaintext \xleftarrow{k_0} state_0 \xleftarrow{k_1} \ldots \xleftarrow{k_9} state_9 \xleftarrow{k_{10}} ciphertext$$

What happens in a round?

- Encryption

$$\xrightarrow{k} = (\mathbf{xor}\ k) \circ \mathsf{MixCols} \circ \mathsf{ShiftRows} \circ \mathsf{SubBytes}$$

- Decryption

$$\xleftarrow{k} = \mathsf{MixCols}^{-1} \circ (\mathbf{xor}\ k) \circ \mathsf{SubBytes}^{-1} \circ \mathsf{ShiftRows}^{-1}$$

Sboxes

- The Sbox and its inverse are permutations on bytes
- Can be thought of as a 16×16 matrix indexed by the two halves of a byte
- Instead, modelled as a total function $word8 \longrightarrow word8$

Sboxes (cont'd)

- Snippet from HOL source (256 cases) :

$$\begin{aligned}
&\mathsf{Sbox}(F,F,F,F,T,F,T,F) = (F,T,T,F,F,T,T,T)\ \wedge\\
&\mathsf{Sbox}(F,F,F,F,T,F,T,T) = (F,F,T,F,T,F,T,T)\ \wedge\\
&\mathsf{Sbox}(F,F,F,F,T,T,F,F) = (T,T,T,T,T,T,T,F)\ \wedge\\
&\mathsf{Sbox}(F,F,F,F,T,T,F,T) = (T,T,F,T,F,T,T,T)\ \wedge
\end{aligned}$$

- $\vdash \mathsf{InvSbox} \circ \mathsf{Sbox} = \mathsf{I}$
- Proved by case analysis and evaluation (trivial)
- Found transcription bug

SubBytes (non-linear byte substitution)

```
genSubBytes S (b00,b01,b02,b03,
               b10,b11,b12,b13,
               b20,b21,b22,b23,
               b30,b31,b32,b33)
      =
   (S b00, S b01, S b02, S b03,
    S b10, S b11, S b12, S b13,
    S b20, S b21, S b22, S b23,
    S b30, S b31, S b32, S b33)

 SubBytes    = genSubBytes Sbox
 InvSubBytes = genSubBytes InvSbox
```

$$\vdash \forall s.\ \mathsf{InvSubBytes}(\mathsf{SubBytes}\ s) = s$$

ShiftRows

- Shift left the first row by 0, the second row by 1, the third by 2 and the fourth by 3.
- Code:

```
ShiftRows (b00,b01,b02,b03,
           b10,b11,b12,b13,
           b20,b21,b22,b23,
           b30,b31,b32,b33)
                  =
          (b00,b01,b02,b03,
           b11,b12,b13,b10,
           b22,b23,b20,b21,
           b33,b30,b31,b31)
```

$\vdash \mathsf{InvShiftRows}(\mathsf{ShiftRows}(s)) = s$ is trivial

MixCols

- Operates on each column
- This can be reduced to matrix multiplication and then to $\mathsf{MultCol}(a,b,c,d) = (a',b',c',d')$ where

$$
\begin{array}{rcccccccc}
a' & = & 02\bullet a & \mathtt{xor} & 03\bullet b & \mathtt{xor} & c & \mathtt{xor} & d \\
b' & = & a & \mathtt{xor} & 02\bullet b & \mathtt{xor} & 03\bullet c & \mathtt{xor} & d \\
c' & = & a & \mathtt{xor} & b & \mathtt{xor} & 02\bullet c & \mathtt{xor} & 03\bullet d \\
d' & = & 03\bullet a & \mathtt{xor} & b & \mathtt{xor} & c & \mathtt{xor} & 02\bullet d
\end{array}
$$

InvMixCols

- The inverse operation does more work. $\mathsf{InvMultCol}(a,b,c,d) = (a',b',c',d')$ where

$$
\begin{aligned}
a' &= \texttt{0E}\bullet a \texttt{ xor } \texttt{0B}\bullet b \texttt{ xor } \texttt{0D}\bullet c \texttt{ xor } \texttt{09}\bullet d \\
b' &= \texttt{09}\bullet a \texttt{ xor } \texttt{0E}\bullet b \texttt{ xor } \texttt{0B}\bullet c \texttt{ xor } \texttt{0D}\bullet d \\
c' &= \texttt{0D}\bullet a \texttt{ xor } \texttt{09}\bullet b \texttt{ xor } \texttt{0E}\bullet c \texttt{ xor } \texttt{0B}\bullet d \\
d' &= \texttt{0B}\bullet a \texttt{ xor } \texttt{0D}\bullet b \texttt{ xor } \texttt{09}\bullet c \texttt{ xor } \texttt{0E}\bullet d
\end{aligned}
$$

MixCols contd.

- Code (higher order)

```
genMixCols MC (b00,b01,b02,b03,
               b10,b11,b12,b13,
               b20,b21,b22,b23,
               b30,b31,b32,b33)
  =
let val (b00’, b10’, b20’, b30’) = MC (b00,b10,b20,b30)
    val (b01’, b11’, b21’, b31’) = MC (b01,b11,b21,b31)
    val (b02’, b12’, b22’, b32’) = MC (b02,b12,b22,b32)
    val (b03’, b13’, b23’, b33’) = MC (b03,b13,b23,b33)
in
   (b00’, b01’, b02’, b03’,
    b10’, b11’, b12’, b13’,
    b20’, b21’, b22’, b23’,
    b30’, b31’, b32’, b33’)
end
```

MixCols final.

- Instantiations

```
MixCols    = genMixCols MultCol
InvMixCols = genMixCols InvMultCol
```

$$\vdash \forall s : \mathit{state}.\ \mathsf{InvMixCols}(\mathsf{MixCols}\ s) = s$$

- Computationally hard to prove
- Interesting to see how SAT or BDDs would do

Inversion Lemmas

- Lemmas

$$\vdash \forall s.\ \textbf{from\_state}(\textbf{to\_state}\ s) = s$$

$$\vdash \forall s.\ \textbf{to\_state}(\textbf{from\_state}\ s) = s$$

$$\vdash \forall w.\ \textbf{InvSbox}(\textbf{Sbox}\ w) = w$$

$$\vdash \forall s.\ \textbf{InvSubBytes}(\textbf{SubBytes}\ s) = s$$

$$\vdash \forall s.\ \textbf{InvShiftRows}(\textbf{ShiftRows}\ s) = s$$

$$\vdash \forall x\, y\, z.\ x \bullet (y \;\mathsf{xor}\; z) = (x \bullet y) \;\mathsf{xor}\; (x \bullet z)$$

$$\vdash \forall s : \mathit{state}.\ \textbf{InvMixCols}(\textbf{MixCols}\ s) = s$$

Verification

Formalization

- First wrote purely functional version in SML
- Then transcribed to HOL-4 (easy)

Validation

- Have to make sure that AES properly implemented (*i.e.*, encryption and decryption not just the identity function)
- By comparing with data in specification document

Statement of Correctness

$$\vdash \forall key\ plaintext.$$

$$\texttt{let}\ (encrypt, decrypt) = \mathsf{AES}\ key$$

$$\texttt{in}$$

$$decrypt(encrypt\ plaintext) = plaintext$$

AES just sets up the key schedule and gives it to the encryptor and decryptor

AES

$$\mathsf{AES} : \mathit{key} \rightarrow ((\underbrace{\mathit{block} \rightarrow \mathit{block})\times(\mathit{block} \rightarrow \mathit{block}}))$$

$$\mathsf{AES} : \mathit{key} \rightarrow (\underbrace{(\mathit{block} \rightarrow \mathit{block})}_{\mathit{encrypt}} \times \underbrace{(\mathit{block} \rightarrow \mathit{block})}_{\mathit{decrypt}})$$

$$
\begin{array}{l}
\mathsf{AES}\ \mathit{key} = \\
\qquad \texttt{let}\ \mathit{sched} = \mathsf{mk\_keysched}\ \mathit{key}\ \texttt{in} \\
\qquad \texttt{let}\ \mathit{isched} = \mathsf{reverse}\ \mathit{sched}\ \texttt{in} \\
\qquad\quad ((\mathsf{from\_state} \circ \mathsf{Round}\ 9\ (\mathsf{tl}\ \mathit{sched}) \\
\qquad\qquad\qquad \circ\ \mathsf{xor}\ (\mathsf{hd}\ \mathit{sched}) \circ \mathsf{to\_state}), \\
\qquad\quad\ (\mathsf{from\_state} \circ \mathsf{InvRound}\ 9\ (\mathsf{tl}\ \mathit{isched}) \\
\qquad\qquad\qquad \circ\ \mathsf{xor}\ (\mathsf{hd}\ \mathit{isched}) \circ \mathsf{to\_state}))
\end{array}
$$

Proof

- Enumeration of states seems infeasible (at least 2^{128} states)
- Exhaustive testing not apparently possible
- What to do?
- Idea: symbolically execute the algorithms

Symbolic Execution

- Evaluation with variables
- In $decrypt(encrypt\ plaintext)$, $plaintext$ is a 16-tuple of bytes.

- Choices

$decrypt(encrypt\ v)$

$decrypt(encrypt\ (v_0, \ldots, v_{15}))$

$decrypt(encrypt\ ((v_{0,0}, v_{0,1}, v_{0,2}, v_{0,3}, v_{0,4}, v_{0,5}, v_{0,6}, v_{0,7}), \ldots$

$(v_{15,0}, v_{15,1}, v_{15,2}, v_{15,3}, v_{15,4}, v_{15,5}, v_{15,6}, v_{15$

Proof (cont'd)

- Let input plaintext be an arbitrary tuple of variables $(v_0, \ldots, v_{15})$ and just let the algorithms run. Decryption should undo the effects of encryption.
- Not feasible either!
- Exponential-sized formulas from variables occurring in conditions of if—then–else expressions
- Improved Idea: controlled symbolic execution plus rewriting with inversion lemmas

Proof (cont'd)

- Problem: key schedule generation is complex.
- If non-trivial properties of it are needed, then proof is no longer easy
- Fortunately, all that is necessary to know about the keyschedule is that its length is 11.
- Easy to prove by symbolically executing keyschedule generator
- Once this is proved, we can use a list of variables $[k_0, k_1, k_2, \ldots, k_{10}]$ as keyschedule value.

Proof Outline

After controlled symbolic evaluation:

$$\ldots \circ \mathsf{MixCols}^{-1} \circ \mathsf{xor}(k_8) \circ \mathsf{SubBytes}^{-1} \circ \mathsf{ShiftRows}^{-1}$$
$$\circ \mathsf{MixCols}^{-1} \circ \mathsf{xor}(k_9) \circ \mathsf{SubBytes}^{-1} \circ \mathsf{ShiftRows}^{-1}$$
$$\circ \mathsf{xor}(k_{10}) \circ \mathsf{to\_state}$$
$$\circ \mathsf{from\_state}$$
$$\circ \mathsf{xor}(k_{10}) \circ \mathsf{ShiftRows} \circ \mathsf{SubBytes} \circ \mathsf{xor}(k_9) \circ \mathsf{MixCols}$$
$$\circ \mathsf{ShiftRows} \circ \mathsf{SubBytes} \circ \mathsf{xor}(k_8) \circ \mathsf{MixCols} \ldots$$

Proof Outline

$$\ldots \circ \mathsf{MixCols}^{-1} \circ \mathsf{xor}(k_8) \circ \mathsf{SubBytes}^{-1} \circ \mathsf{ShiftRows}^{-1}$$
$$\circ\, \mathsf{MixCols}^{-1} \circ \mathsf{xor}(k_9) \circ \mathsf{SubBytes}^{-1} \circ \mathsf{ShiftRows}^{-1}$$
$$\circ\, \mathsf{xor}(k_{10}) \circ \mathsf{to\_state}$$
$$\circ\, \mathsf{from\_state}$$
$$\circ\, \mathsf{xor}(k_{10}) \circ \mathsf{ShiftRows} \circ \mathsf{SubBytes} \circ \mathsf{xor}(k_9) \circ \mathsf{MixCols}$$
$$\circ\, \mathsf{ShiftRows} \circ \mathsf{SubBytes} \circ \mathsf{xor}(k_8) \circ \mathsf{MixCols} \ldots$$

Proof Outline

$$\ldots \circ \mathsf{MixCols}^{-1} \circ \mathsf{xor}(k_8) \circ \mathsf{SubBytes}^{-1} \circ \mathsf{ShiftRows}^{-1}$$
$$\circ\, \mathsf{MixCols}^{-1} \circ \mathsf{xor}(k_9) \circ \mathsf{SubBytes}^{-1} \circ \mathsf{ShiftRows}^{-1}$$
$$\circ\, \mathsf{xor}(k_{10}) \circ \mathsf{to\_state}$$
$$\circ\, \mathsf{from\_state}$$
$$\circ\, \mathsf{xor}(k_{10}) \circ \mathsf{ShiftRows} \circ \mathsf{SubBytes} \circ \mathsf{xor}(k_9) \circ \mathsf{MixCols}$$
$$\circ\, \mathsf{ShiftRows} \circ \mathsf{SubBytes} \circ \mathsf{xor}(k_8) \circ \mathsf{MixCols} \ldots$$

Proof Outline

$$\ldots \circ \mathsf{MixCols}^{-1} \circ \mathsf{xor}(k_8) \circ \mathsf{SubBytes}^{-1} \circ \mathsf{ShiftRows}^{-1}$$
$$\circ \mathsf{MixCols}^{-1} \circ \mathsf{xor}(k_9) \circ \mathsf{SubBytes}^{-1} \circ \mathsf{ShiftRows}^{-1}$$
$$\circ \mathsf{xor}(k_{10}) \circ \mathsf{to\_state}$$
$$\circ \mathsf{from\_state}$$
$$\circ \mathsf{xor}(k_{10}) \circ \mathsf{ShiftRows} \circ \mathsf{SubBytes} \circ \mathsf{xor}(k_9) \circ \mathsf{MixCols}$$
$$\circ \mathsf{ShiftRows} \circ \mathsf{SubBytes} \circ \mathsf{xor}(k_8) \circ \mathsf{MixCols} \ldots$$

Proof Outline

$$\ldots \circ \mathsf{MixCols}^{-1} \circ \mathsf{xor}(k_8) \circ \mathsf{SubBytes}^{-1} \circ \mathsf{ShiftRows}^{-1}$$
$$\circ \mathsf{MixCols}^{-1} \circ \mathsf{xor}(k_9) \circ \mathsf{SubBytes}^{-1} \circ \mathsf{ShiftRows}^{-1}$$
$$\circ \mathsf{xor}(k_{10}) \circ \mathsf{to\_state}$$
$$\circ \mathsf{from\_state}$$
$$\circ \mathsf{xor}(k_{10}) \circ \mathsf{ShiftRows} \circ \mathsf{SubBytes} \circ \mathsf{xor}(k_9) \circ \mathsf{MixCols}$$
$$\circ \mathsf{ShiftRows} \circ \mathsf{SubBytes} \circ \mathsf{xor}(k_8) \circ \mathsf{MixCols} \ldots$$

Proof Outline

$\ldots \circ \mathsf{MixCols}^{-1} \circ \mathsf{xor}(k_8) \circ \mathsf{SubBytes}^{-1} \circ \mathsf{ShiftRows}^{-1}$

$\circ\, \mathsf{MixCols}^{-1} \circ \mathsf{xor}(k_9) \circ \mathsf{SubBytes}^{-1} \circ \mathsf{ShiftRows}^{-1}$

$\circ\, \mathsf{xor}(k_{10}) \circ \mathsf{to\_state}$

$\circ\, \mathsf{from\_state}$

$\circ\, \mathsf{xor}(k_{10}) \circ \mathsf{ShiftRows} \circ \mathsf{SubBytes} \circ \mathsf{xor}(k_9) \circ \mathsf{MixCols}$

$\circ\, \mathsf{ShiftRows} \circ \mathsf{SubBytes} \circ \mathsf{xor}(k_8) \circ \mathsf{MixCols} \ldots$

Discussion

- Model of AES in theorem prover
- Can be executed by (deductive) evaluation on ground or symbolic terms
- Can also be proved correct, in same logical system, with same proof tools.
- Correctness proved with little difficulty

Discussion (contd)

- Unwound 10 rounds then inversion lemmas used to collapse from the inside
- A few lemmas needed interaction, but perhaps most of that can be automated?
- Proof in Cryptol used to related high-level algorithmic spec. to more concrete algorithmic spec.
- Our approach is complementary, in that it proves a correctness property (sanity check) of the original high-level spec.

Part II: Dealing with High Level Types

- AES deals with byte blocks
- But one wants to deal with elements of high-level types
- Example: one might wish to encrypt a database of medical patients (*e.g.*, tree of records)
- Lots of programming needed to bridge the gap
- Boring and error-prone.
- Polytypism to the rescue!

Types and Algorithms

- Types often tend to come before algorithms
- One defines a type, then defines algorithms over the type
- Example: ADTs (Abstract Data Types)
- Very successful, provides a solid underpinning for software engineering
- But not the only game in town!

Polytypism

- Polytypic algorithms are so general that they apply to a wide range of types
- In a sense, they come before types and get instantiated when a new type is defined
- NB. Not the same idea as polymorphism

Polytypism in FP

- Datatype declaration introduces a particular shape of tree
- A polytypic algorithm operates uniformly, modulo the shape of the data
  - Equality
  - Substitution
  - Printing
  - Mapping into bitstrings (and back out)

Polytypism and Encryption

- Idea: Automatically map high level data to bitstrings then use AES.

- Allows correctness of AES to be factored out and re-used for encryption of all datatypes.

Types formally

- Type signature Ω holds the arities of type operators

- Types are defined inductively:
  - Countable set of type variables: $\alpha, \alpha_1, \alpha_2, \ldots, \beta, \beta_1, \ldots$
  - If c in Ω has arity n, and each of $\tau_1, \ldots, \tau_n$ is a type, then $(\tau_1, \ldots, \tau_n)c$ is a type

- New types are added to Ω when they are defined

Basic Types

- Booleans: bool.
  Values are true and false.

- Pairs: $(\alpha, \beta)\mathsf{prod}$. Written $\alpha * \beta$.
  Values constructed with $(-, -)$.

- Sums: $(\alpha, \beta)\mathsf{sum}$. Written $\alpha + \beta$.
  Values constructed with $\mathsf{INL} : \alpha \to \alpha + \beta$ and $\mathsf{INR} : \beta \to \alpha + \beta$.

- Functions: $(\alpha, \beta)\mathsf{fun}$. Written $\alpha \to \beta$.
  Values constructed via lambda abstraction $\lambda v.\ M$

Datatypes

- Mechanism for introducing user-defined types
- Recursive $\quad \mathsf{num} = 0 \mid \mathsf{Suc}\ \texttt{of}\ \mathsf{num}$
- Polymorphic
  - Partial functions : $\alpha\ \mathsf{option} = \mathsf{None} \mid \mathsf{Some}\ \texttt{of}\ \alpha$
  - Homogeneous lists : $\alpha\ \mathsf{list} = [\,] \mid ::\ \texttt{of}\ \alpha * \alpha\ \mathsf{list}$
- Example value:

$$[\mathsf{None}, \mathsf{Some}\ (\mathsf{Suc}\ 0)] : \mathsf{num\ option\ list}$$

More Datatypes

- Nested under existing member of Ω. The following is a type of finitely branching trees:

$$\alpha\,\mathsf{tree} = \mathsf{Node}\ \mathtt{of}\ \alpha * \alpha\,\mathsf{tree}\ \mathsf{list}$$

- Mutually recursive (and nested)

$$\begin{array}{rcl}
(\alpha,\beta)\mathsf{exp} & = & \mathsf{Var}\ \mathtt{of}\ \alpha \\
& \mid & \mathsf{Cond}\ \mathtt{of}\ (\alpha,\beta)\mathsf{bexp} * (\alpha,\beta)\mathsf{exp} * (\alpha,\beta)\mathsf{exp} \\
& \mid & \mathsf{App}\ \mathtt{of}\ \beta * (\alpha,\beta)\mathsf{exp}\ \mathsf{list} \\
(\alpha,\beta)\mathsf{bexp} & = & \mathsf{Less}\ \mathtt{of}\ (\alpha,\beta)\mathsf{exp} * (\alpha,\beta)\mathsf{exp} \\
& \mid & \mathsf{And}\ \mathtt{of}\ (\alpha,\beta)\mathsf{bexp} * (\alpha,\beta)\mathsf{bexp} \\
& \mid & \mathsf{Not}\ \mathtt{of}\ (\alpha,\beta)\mathsf{bexp}
\end{array}$$

Polytypism Sketch : Coding

- Given an environment Γ of encoders and decoders for types
- Synthesize encoders/decoders for a compound type by mimicking the structure of the type:
- Example: The type $(\mathsf{num} * \mathsf{bool\ option})\mathsf{list}$.
- Suppose Γ is an encoder context containing at least encoders for the types num, list, option, and bool

Coding example

- Synthesized encoder:

$$\text{encode\_list}\ (\text{encode\_prod}\ \text{encode\_num}\\ \qquad\qquad (\text{encode\_option}\ \text{encode\_bool}))$$

- Decoder (using a decoder context):

$$\text{decode\_list}\ (\text{decode\_prod}\ \text{decode\_num}\\ \qquad\qquad (\text{decode\_option}\ \text{decode\_bool}))$$

Interpretation

- Our approach is based on an interpretation $[\![\_]\!]_{\Theta,\Gamma}$ of HOL types into terms.

$$
\begin{array}{rcll}
[\![v]\!]_{\Theta,\Gamma} & = & \Theta(v) & \text{if } v \text{ a tyvar} \\
[\![\,(\tau_1,\ldots,\tau_n)c\,]\!]_{\Theta,\Gamma} & = & \Gamma(c)\;\; [\![\tau_1]\!]_{\Theta,\Gamma} \cdots [\![\tau_n]\!]_{\Theta,\Gamma} & \text{otherwise}
\end{array}
$$

- The interpretation is parameterized by two maps: Θ, which maps type variables; and Γ, which maps type operators.

- Lifted to terms: $I_{\Theta,\Gamma}(M:\tau) = [\![\tau]\!]_{\Theta,\Gamma}(M)$

Interpretation

- Interpretations common in formal semantics: translate syntax into informal mathematics (models)
- Support meta-theoretic exercises (soundness, completeness)
- In contrast, we interpret HOL types (syntax) into HOL terms (syntax)
- Allows proof support to be automatically defined each time a datatype is declared

Polytypism and Proof Automation

- When a new type τ is introduced, then automatically define new functions:
  - Size : $\tau \longrightarrow \mathsf{num}$
  - Lifting : $\underbrace{\tau}_{\text{ML}} \longrightarrow \underbrace{\tau}_{\text{HOL}}$
  - Coding
    - Encode : $\tau \longrightarrow \mathsf{bool\ list}$
    - Decode : $\mathsf{bool\ list} \longrightarrow \tau$
- These are used to support proof automation

Bringing it all together

- Given capability to synthesize encoders and decoders for τ (see paper for details)
- Algorithms for padding bitstrings to exact multiples of the block size (pad), and unpadding to the original length (unpad); and
- Use mode of operation (CBC) to lift block encryptor E and decryptor D to CBC_E and CBC_D, which work on sequences of blocks.

Correctness statement

$$\mathsf{AES}\ key = (E, D) \supset$$

$$\underbrace{\mathsf{decode}_\tau \circ \mathsf{unpad} \circ \mathsf{CBC}_D}_{decryption} \circ \underbrace{\mathsf{CBC}_E \circ \mathsf{pad} \circ \mathsf{encode}_\tau}_{encryption} = \mathbf{I}$$

Easy to show when we know

- $D \circ E = \mathbf{I}$ (AES correctness)
- $\mathsf{CBC}_D \circ \mathsf{CBC}_E = \mathbf{I}$ (Trivial)
- $\mathsf{unpad} \circ \mathsf{pad} = \mathbf{I}$ (Trivial)
- $\mathsf{decode}_\tau \circ \mathsf{encode}_\tau = \mathbf{I}$ (Easy)

Conclusions

- Sanity-checking style proofs for encryption and decryption of high-level types are not hard

- Possibilities for full automation

Future Work

- Code (or hardware) generation from these high-level specs
- Ongoing work with Mike Gordon at Cambridge
- Done via equivalence preserving steps in theorem prover
- Goal: a (longer) unbroken assurance chain through to final artifact