Embedding Cryptol in Higher Order Logic

Joe Hurd
Computer Laboratory
Cambridge University
joe.hurd@cl.cam.ac.uk

10 March 2007

Abstract

This report surveys existing approaches to embedding Cryptol programs in higher order logic, and presents a new approach that aims to simplify as much as possible reasoning about the embedded programs.

1 Introduction

One way to formally specify a cryptographic algorithm is to express its operation in a logic. Higher order logic is an expressive specification logic that is well-supported by verification tools, including a verifying compiler from an executable subset of the logic to a model of the ARM instruction set.

Cryptol [2] is a domain specific language for symmetric-key cryptographic algorithms, using a sophisticated type system to ensure consistency of the 'bit twiddling' and vector operations typical of this class of program.

To reason about cryptographic algorithms it would thus be beneficial to develop a method for embedding Cryptol programs into higher order logic. This would allow programmers to implement cryptographic algorithms in a convenient programming language, using a Cryptol interpreter to catch type errors and generate prototypes for testing deeper properties. Once these checks have passed, the algorithm can be embedded into higher order logic where it can be formally verified to satisfy its specification.

Any such embedding must be closely scrutinized to ensure that the Cryptol semantics are faithfully transferred to higher order logic. Cryptol mutually recursive sequences are particularly challenging, because the syntax allows sequences to be defined where evaluating one element requires an infinite execution. Since all functions in higher order logic are total, special care must be taken when embedding these 'nonterminating sequences'.

Naturally, the fidelity of the embedding is of primary importance, but a secondary consideration is how easy it is to prove properties of Cryptol programs embedded in higher order logic. One useful indicator of this is the degree to which higher order logic types are employed to represent properties of Cryptol

programs. For example, suppose a Cryptol program contained the finite sequences [1 1] and [1 1 1]. These have different Cryptol types (to reflect their different lengths), but if finite sequences were embedded as lists they would end up with the same higher order logic type. If the specification of the program relied on the lengths of the sequences, additional proof obligations would be generated to compensate for the type information that was lost in the embedding.

Section 2 surveys existing approaches to embedding Cryptol into higher order logics, and Section 3 presents a new embedding of a Cryptol subset that uses finite Cartesian products to capture the Cryptol type system as precisely as possible.

2 Related Work

The most relevant work is Li and Slind's shallow embedding of Cryptol into HOL4 [3]. Cryptol sequences are modelled with HOL lazy lists, where the elements of the lazy lists can either be booleans or lazy lists. This allows Cryptol sequence operations (split, join, etc.) to be defined smoothly as polymorphic functions. Arithmetic operations are defined by converting finite subsequences of boolean lazy lists to HOL words and using word arithmetic, and there is syntactic sugar for sequences containing finite and infinite number ranges. The embedding includes a definition principle for mutually recursive sequence comprehensions, although only supports a certain form of definition that guarantees termination.

Matthews formalized a deep embedding of fCryptol (pronounced "femto-Cryptol", it being a subset of μCryptol) in the Isabelle/HOL theorem prover [4]. Separate higher order logic types are used for finite and infinite sequences of signed bitvectors. The embedding defines a higher order logic datatype for the abstract syntax of fCryptol, including mutually recursive (infinite) sequence comprehensions, together with a denotational semantics for fCryptol expressions and some semantics-preserving transformation rules. In the denotational semantics nonterminating sequence expressions are assigned a default value of the meaning type.

In mid-2006 Matthews also formalized a shallow embedding of μCryptol into Isabelle/HOLCF: an extension of higher order logic with support for first-class partial functions.[1] This makes it easy to faithfully embed nonterminating sequence expressions in the logic, but when reasoning about embedded Cryptol programs there are additional proof obligations that expressions terminate. The same theory contains a shallow embedding of the ACL2 logic, and can be used to verify the initial compilation stages of the verifying μCryptol compiler `mcc` [5].

[1] This is unpublished work.

3 Embedding a Cryptol Subset

The goal of this section is to present an embedding of Cryptol into higher order logic that simplifies as much as possible reasoning about embedded programs. This goal leads to two main design choices:

1. Cryptol programs are embedded as higher order logic functions; and
2. higher order logic types are used to convey as much information as possible about the embedded programs.

Both these design choices restrict the set of possible Cryptol programs that may be embedded. The first restricts the embedding to terminating Cryptol programs, since functions in higher order logic are total. Although there are several techniques for modelling partial functions in higher order logic, they all involve more complicated reasoning than total higher order logic functions and so this approach is rejected.

The second design choice above restricts the embedding to Cryptol programs where the expressions in the program do not map to any higher order logic type. For example, Section 3.1 shows how the length of a Cryptol sequence can be encoded in its higher order logic type. Thus, any Cryptol program that works over sequences of variable lengths cannot be embedded in higher order logic. An example of this is the Cryptol implementation of the SHA-1 secure hash algorithm, which produces a 160-bit digest of an arbitrary length input, and therefore has Cryptol type $[N] \to [160]$.

3.1 Cryptol Types in Higher Order Logic

The Cryptol bit type naturally maps to higher order logic booleans, and Cryptol tuples and higher order logic tuples behave in exactly the same way. The only problematic types to embed are the types of Cryptol sequences. Cryptol sequences can have finite or infinite length, and their length is encoded in their Cryptol type. For example, [4] is the Cryptol type of 4-bit numbers, and [inf][4] is the type of infinite sequences of 4-bit numbers.

The Cryptol type $[\text{inf}]\alpha$ of infinite sequences can be simply modelled in higher order logic with the type

$$\alpha \text{ inf} \equiv \mathbb{N} \to \alpha \ .$$

The Cryptol type $[n]\alpha$ of sequences of length n can be modelled with the higher order logic vector type

$$\alpha \text{ vector} \equiv \tau_n \to \alpha \ ,$$

where τ_n is a specially created higher order logic type that contains exactly n elements. It is possible to define higher order logic functions that are polymorphic over all finite α-vectors [1].

3.2 Cryptol Sequences in Higher Order Logic

As demonstrated in the previous section, it is possible to represent the length of a sequence as part of the higher order logic type, as it is done in Cryptol. Moreover, functions can be polymorphic over all finite sequences, so the higher order logic versions of the Cryptol sequence operations need only distinguish two cases: one handling infinite sequences and one handling finite sequences. For example, here are the two higher order logic definitions for appending a finite sequence to the front of a finite or infinite sequence.

$$\begin{aligned}&\mathsf{seq\_append\_finite}\ (x : [m]\alpha)\ (y : [n]\alpha) \equiv \\ &\quad \mathsf{FCP}\ i.\ \mathsf{if}\ i < m\ \mathsf{then}\ x\ \%\%\ i\ \mathsf{else}\ y\ \%\%\ (i - m)\end{aligned}$$

$$\begin{aligned}&\mathsf{seq\_append\_infinite}\ (x : [n]\alpha)\ (y : [\mathrm{inf}]\alpha) \equiv \\ &\quad \lambda i.\ \mathsf{if}\ i < n\ \mathsf{then}\ x\ \%\%\ i\ \mathsf{else}\ y\ (i - n)\end{aligned}$$

For ease of reading Cryptol syntax is used for the argument types, and the notation $\mathsf{FCP}\ i.\ \phi(i)$ and $v\ \%\%\ i$ is used to construct and select elements of vectors. In practice both of these append functions can be parsed and printed using the standard Cryptol symbol # (the parser uses the argument type to disambiguate).

In a similar way finite and infinite higher order logic versions of the map, drop and zip sequence operations can be defined. There is an elegant way to define sequence comprehensions using the map and zip operators. For example, consider the following Cryptol implementation of the Fibonacci sequence:

```
fib = [0 1] # [| x + y || x <- drop (1,fib) || y <- fib |]
```

The sequence comprehension can be embedded into higher order logic as

$$\mathsf{map}\ (\lambda(x, y).\ x + y)\ (\mathsf{zip}\ (\mathsf{drop}\ 1\ \mathsf{fib})\ \mathsf{fib})\ .$$

Using the Cryptol symbol | for zip and a new binder syntax for map, this is parsed and pretty printed as

$$(\mathsf{seq}\ (x, y).\ x + y)\ (\mathsf{drop}\ 1\ \mathsf{fib} \mid \mathsf{fib})$$

which looks similar to the Cryptol syntax for sequence comprehension.

3.3 Cryptol Programs in Higher Order Logic

Cryptol programs consist of a series of sequence and function definitions, each of which may contain nested definitions. There are two problems with embedding this scheme into higher order logic:

1. although there are proof tools in HOL4 to assist in the definition of recursive functions, they do not extend to Cryptol's recursive sequence definitions; and
2. defined constants in higher order logic exist in a global scope, so true nested definitions cannot be made.

To overcome the first problem, the actual definition of a recursive sequence in higher order logic is expressed in terms of its embedding as a function from $\mathbb{N}$. For example, here is the higher order logic definition of the Fibonacci sequence from Section 3.2:

$$\mathsf{fib}\ i \equiv \mathsf{if}\ i < 2\ \mathsf{then}\ \mathsf{V}[0w;\ 1w]\ \%\%\ i\ \mathsf{else}\ \mathsf{fib}\ (i-1) + \mathsf{fib}\ (i-2)\ .$$

This version of the recursive sequence definition can be handled by the recursive function definition proof tool in HOL4.

Once the required sequence has been defined, the definition is proved equivalent to a sequence expression closer to standard Cryptol syntax. Continuing with the Fibonacci sequence, here is the final theorem:

$$\vdash\ \mathsf{fib} = \mathsf{V}[0w;\ 1w]\ \#\ (\mathsf{seq}\ (x, y).\ x + y)\ (\mathsf{drop}\ 1\ \mathsf{fib}\ |\ \mathsf{fib})\ .$$

The notation $\mathsf{V}[0w;\ 1w]$ is used to construct a 2-vector from a list of two words (each of which is represented as a 32-vector of bits).

Note: This two stage approach to defining recursive sequences is just an extension of the recursive function definition package, where a primitive non-recursive definition is proved to satisfy the given recursion equations.

In the present work there is no attempt to overcome the second problem mentioned at the start of this section: the global scope for constants in higher order logic prevents true nested definitions. Nested definitions are simply λ-lifted to the top level, pending a standard HOL syntax for nested definitions and an extension to the function definition package to handle them.

3.4 Example: TEA Encryption

This section demonstrates the higher order logic embedding on a Cryptol implementation of Wheeler and Needham's TEA (Tiny Encryption Algorithm). Figure 1 is a verbatim copy of the μCryptol implementation of TEA on page 2 of Shields' paper [6]. Figure 2 shows the embedding into higher order logic.

Certainly there are differences in the concrete syntax, but the structure is identical in both implementations. The only significant change is the λ-lifting of the nested definitions tea_sums, tea_ys and tea_zs. These are defined using the technique of Section 3.2 by mutual recursion and equivalence proof. Naturally, since the function tea depends on these mutually recursive sequences, they must be defined first.

4 Summary

This report has surveyed existing approaches to embedding Cryptol programs in higher order logic, and presented a new approach that aims to simplify as much as possible reasoning about the embedded programs, even at the cost of not being able to handle all Cryptol programs. The 'right embedding' will surely depend on the particular reasoning task to be performed, and will borrow ideas from all approaches.

```
exports code;

N = 32;
W = 32;
Word = B^W;
Block = Word^2;
Key = Word^4;
Index = B^W;

delta : Word;
delta = 0x9e3779b9;

code : (Block, Key) -> Block;
code ([v0, v1], k) = [ys@@N, zs@@N] where {
  rec sums : Word^inf;
  sums = [0] ##
         [ sum + delta | sum <- sums ];
  and ys : Word^inf;
  ys = [v0] ##
       [ y+((z<<4)+k@0 ^^ z+sum ^^ (z>>5)+k@1)
       | sum <- drops{1} sums
       | y <- ys
       | z <- zs ];
  and zs : Word^inf;
  zs = [v1] ##
       [ z+((y<<4)+k@2 ^^ y+sum ^^ (y>>5)+k@3)
       | sum <- drops{1} sums
       | y <- drops{1} ys
       | z <- zs ];
};
```

Figure 1: A μCryptol implementation of TEA.

$\mathsf{tea\_N} \equiv 32$

$\text{tea\_W} \equiv 32$

$\text{tea\_Word} \equiv \mathbb{B} \ast\ast \text{tea\_W}$

$\text{tea\_Block} \equiv \text{tea\_Word} \ast\ast 2$

$\text{tea\_Key} \equiv \text{tea\_Word} \ast\ast 4$

$\text{tea\_Index} \equiv \mathbb{B} \ast\ast \mathsf{tea\_W}$

$\mathsf{tea\_delta} : \text{tea\_Word} \equiv 0x9e3779b9w$

$\vdash$ $(\mathsf{tea\_sums}\ v\ k : \text{tea\_Word inf}) =$
$\quad \mathsf{V}[0w]\ \#\ (\mathsf{seq}\ x.\ x + \mathsf{tea\_delta})\ (\mathsf{tea\_sums}\ v\ k)$

$\vdash$ $(\mathsf{tea\_ys}\ v\ k : \text{tea\_Word inf}) =$
$\quad \mathsf{V}[v\ \%\%\ 0]\ \#$
$\quad (\mathsf{seq}\ (sum, y, z).$
$\quad\quad y + (((z \ll 4) + (k\ \%\%\ 0))\ ??\ (z + sum)\ ??\ ((z \gg 5) + (k\ \%\%\ 1))))$
$\quad (\mathsf{drop}\ 1\ (\mathsf{tea\_sums}\ v\ k) \mid \mathsf{tea\_ys}\ v\ k \mid \mathsf{tea\_zs}\ v\ k)$

$\vdash$ $(\mathsf{tea\_zs}\ v\ k : \text{tea\_Word inf}) =$
$\quad \mathsf{V}[v\ \%\%\ 1]\ \#$
$\quad (\mathsf{seq}\ (sum, y, z).$
$\quad\quad z + (((y \ll 4) + (k\ \%\%\ 2))\ ??\ (y + sum)\ ??\ ((y \gg 5) + (k\ \%\%\ 3))))$
$\quad (\mathsf{drop}\ 1\ (\mathsf{tea\_sums}\ v\ k) \mid \mathsf{drop}\ 1\ (\mathsf{tea\_ys}\ v\ k) \mid \mathsf{tea\_zs}\ v\ k)$

$(\mathsf{tea} : \text{tea\_Block} \times \text{tea\_Key} \to \text{tea\_Block})\ (v, k) \equiv$
$\quad \mathsf{V}[\mathsf{tea\_ys}\ v\ k\ \mathsf{tea\_N};\ \mathsf{tea\_zs}\ v\ k\ \mathsf{tea\_N}]$

Figure 2: A higher order logic embedding of TEA.

Acknowledgements

Thanks to John Matthews and Konrad Slind for helping me to understand the features of their embeddings, and Jeff Lewis for answering my Cryptol questions. Thanks also to Anthony Fox for reimplementing HOL4 index types to make my embedding of finite sequences work properly, and Michael Norrish for integrating them into the parser and pretty-printer.

References

[1] John Harrison. A HOL theory of Euclidean space. In Joe Hurd and Tom Melham, editors, *18th International Conference on Theorem Proving in Higher Order Logics: TPHOLs 2005*, volume 3603 of *Lecture Notes in Computer Science*, pages 114–129, Oxford, UK, August 2005. Springer.

[2] 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.

[3] Guodong Li and Konrad Slind. An embedding of Cryptol in HOL-4. Unpublished draft, 2005.

[4] John Matthews. fCryptol semantics. Available from the author on request, January 2005.

[5] Lee Pike, Mark Shields, and John Matthews. A verifying core for a cryptographic language compiler. In Panagiotis Manolios and Matthew Wilding, editors, *Proceedings of the Sixth International Workshop on the ACL2 Theorem Prover and its Applications*, pages 1–10, Seattle, Washington, USA, August 2006. HappyJack Books.

[6] Mark Shields. A language for symmetric-key cryptographic algorithms and its efficient implementation. Available from the author's website, March 2006.