A language for symmetric-key cryptographic algorithms and its efficient implementation

Mark Shields
Galois Connections*
mbs@cartesianclosed.com

Abstract. The development of cryptographic hardware for classified data is expensive and time consuming. We present a domain-specific language, *μCryptol*, and a corresponding compiler, `mcc`, to address these costs. *μCryptol* supports the capture of mathematically precise specifications of algorithms, while also allowing those specifications to be compiled to efficient imperative code able to execute on embedded microprocessors.

1 Introduction

In the civilian realm, a handful of symmetric-key cryptographic algorithms such as DES[1], AES[2, 3] and RC5[4] are sufficient to cover the majority of needs for data secrecy. As new hardware arises, it is feasible to reimplement, test and inspect each implementation for correctness. Software implementation on general purpose microprocessors are the norm, with hardware implementations typically following only for high-end users.

In the military realm, the number of cryptographic algorithms is much larger. Typically, a country will have a small set of contemporary algorithms which are approved for use in tunneling classified data over unclassified channels. However, a country may also have many legacy algorithms (which may not be very well documented), and may be required to communicate (perhaps as part of a newly formed coalition) with another country using any of *that* country's algorithms. Hardware implementations are the norm, and algorithms do not necessarily use the power-of-two word sizes supported by general purpose microprocessors.

Users in the military realm thus suffer three significant costs. The most obvious cost is in development time needed to reimplement algorithms on each new generation of hardware. Developers must typically reconstruct usable specifications from official "specifications", code, test, and supply all the supporting documentation to demonstrate that their implementation does indeed conform to the specification.

The second cost is in security evaluation time. Implementations which are to be certified for use with classified data must undergo rigorous security *evaluation* by a third-party (the National Security Agency has this role in the United States). Typically there are a small number of specialist evaluators and a large backlog of implementations awaiting evaluation.

The third cost is that, during the long delay in developing, certifying and deploying new cryptographic hardware (typically on the order of years), parties which should be communicating have to just make do with legacy, low-bandwidth equipment, or simply not communicate at all. It is difficult to quantify the cost of lack of relevant information.

From this we draw out three key requirements:

1. We need a way of precisely specifying algorithms which is neutral as to the underlying implementation. For example, the specification should not depend on a particular word size, or assume a software *vs.* hardware implementation.
2. We need a way of quickly going from specification to implementation.
3. We need a way of quickly demonstrating correspondence of implementation with specification.

This paper presents a domain specific language, *μCryptol* (pronounced "micro Cryptol"), for specifying symmetric-key algorithms. *μCryptol* is a variant of the *Cryptol* [5] language. Both languages lend themselves to precise and readable *declarative specifications* of algorithms while remaining neutral as to underlying word size and hardware *vs.* software implementation. *μCryptol* refines *Cryptol* so that it is feasible to compile *μCryptol* programs to efficient, imperative programs which may execute in embedded environments where dynamic memory is generally considered too costly in both performance and code complexity. Indeed, as a proof-of-concept, this paper also presents a compiler, `mcc`, from *μCryptol* programs to binary images for the *AAMP7* [6, 7] embedded microprocessor of Rockwell Collins, Inc.

* Author's present address: Microsoft, One Microsoft Way, Redmond WA, 98052-6399, USA.

μCryptol is also carefully restricted so that it is feasible for an automated theorem prover such as *ACL2* [8] to prove input/output equivalence of the code generated by `mcc` with the original *μCryptol* program. (Indeed we intend `mcc` to be a *certifying* compiler. That is, we intend the proof of correspondence to be fully automated, with `mcc` supplying all the necessary correspondence assertions and supporting lemmas needed for the proof to proceed without human intervention. However, this aspect of `mcc` shall be covered in a future paper.)

To summarize, we address our three requirements as follows:

1. For specification, we use the language *μCryptol*.
2. For rapid development, we use the compiler `mcc`.
3. For rapid security evaluation, we use the theorem prover *ACL2* (in conjunction with proof scripts emitted by `mcc`).

In the following we introduce *μCryptol* by a worked example (Section 2), then present the language more systematically (Section 3). No prior knowledge of *Cryptol* is assumed. For a complete presentation, however, we refer the reader to the reference manual[9]. We then show how to compile *μCryptol* into imperative code, first as a target-neutral series of front-end transformations (Section 4), then as a target-specific back-end transformation (Section 5). The interface between the front and back ends uses a simple *μCryptol* virtual machine, called *CrAM*. Again, for a complete description of *CrAM* we refer the reader to the machine specification[10]. We briefly discuss our experience implementing `mcc` in Section 6, and conclude in Section 8.

2 *μCryptol* by example

In this section we introduce *μCryptol* using the Tiny Encryption Algorithm of Wheeler and Needham[11]. For comparison, we include a *C* implementation in Figure 1.

TEA is a 32-round Feistel-style algorithm with 64-bit block size and 128-bit key. Figure 2 presents the algorithm in *μCryptol*.

2.1 Basic constructs

The program begins with an *`exports` declaration* indicating which top-level definitions of the program shall be visible from external modules. (The compiler uses this declaration to avoid inlining away declarations which must be callable at runtime.)

```
typedef unsigned long word;
void code(word* v, word* k) {
  word y=v[0], z=v[1];
  word sum=0, delta=0x9e3779b9;
  int n=32;
  while (n-- > 0) {
    sum += delta;
    y += (z<<4)+k[0] ^ z+sum ^ (z>>5)+k[1];
    z += (y<<4)+k[2] ^ y+sum ^ (y>>5)+k[3];
  }
  v[0]=y; v[1]=z;
}
```

Fig. 1: TEA encryption in *C*

```
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 ];
};
```

Fig. 2: TEA encryption in *μCryptol*

Next come six *type abbreviation definitions*. A type name must begin with an upper-case letter. The definition `Word = B^W;` makes `Word` an abbreviation of the type `B^W`. The symbol `B` denotes the type of bits, and `^` is the type constructor for *vectors*, with element type to the left and vector width to the right. `W` is itself a type abbreviation for the natural number `32`, hence `Word` is a 32-bit word. However, note that there is nothing special

about 32-bit words in *μCryptol*. We could just as easily have used 31 or 3177-bit words. Furthermore, a single *μCryptol* program may manipulate words, and indeed vectors of any element type, of any mix of widths.

μCryptol is a *strongly-typed* language: the type of every value manipulated by a *μCryptol* program must be known at compile-time, and it is not possible for a *μCryptol* program to fail at run-time because a value has an unexpected type (for example, attempting to add 3- and 4-bit numbers). Since the width of a vector is part of its type, strong typing requires that the type system be able to perform arithmetic on vector widths.

To ensure types are well formed (and, for example, to reject nonsensical "types" such as `32^5`), *μCryptol* uses a *kind system*. Types which have corresponding values have kind `Type`. Natural numbers have kind `Nat`. We'll later introduce three additional kinds to further refine the construction of types. Kinds may be made explicit by the programmer by including kind signatures. For example, we could add the signature `W : Nat;` immediately before the definition for W.

After the type abbreviations come two *value definitions*. Value names must begin with a lower case letter, type and value definitions may be arbitrarily intermixed, and the scope of a top-level definition is all of its following definitions.

The value `delta` is given a type signature `delta : Word`, indicating its body evaluates to a 32-bit word. In this case the body of `delta` is already the literal value `0x9e3779b9`. In general, a value definition body may be any well-typed expression.

Literals numbers in *μCryptol* denote vectors of bits from most-to-least significant bit order. Indeed, a literal denotes the family of all bit vectors with sufficient width to represent the literal. For example, the literal `2` denotes `[true, false]`, `[false, true, false]`, `[false, false, true, false]`, and so on. A literal in a particular context denotes the unique vector of width appropriate for the context's type. Most of *μCryptol*'s built-in primitives represent families similarly indexed by widths and/or types.

After `delta` comes the definition of `code`, the TEA encryption function. The line `code : (Block, Key) -> Block;` declares `code` to be a function accepting a tuple of a `Block` (two 32-bit words) and `Key` (four 32-bit words) and returning a `Block`. (Note that, unlike a vector, a tuple may contain values of different types in each of its elements.) On the next line is the definition of (the value of) `code` itself. *μCryptol* is a first-order language and does not have λ-expressions, hence all functions must have named definitions, and all function arguments must appear in the definition header. The fragment `code ([y0, z0], k)` deconstructs `code`'s sole argument using a tuple *pattern*. All patterns in *μCryptol* are irrefutable (cannot fail to match any value of appropriate type). The tuple pattern in turn deconstructs its first element into a vector of two elements. Tuple and vector patterns, and values, are written using standard `(...)` and `[...]` notation.

The body of `code` is the vector `[ysN, zsn]`, which is in the scope of three nested definitions introduced by a `where` clause. Definitions in *μCryptol* may be arbitrarily nested. Just as for the top-level, each definition in a `where` clause is in scope of all following definitions in the same clause, as well as the term to the left of the `where`.

In this case, the `where` clause contains a single clique of *mutually-recursive definitions*. (Both `rec` and `and` are keywords, and indicate the beginning and continuation of a clique.) It is here that the *μCryptol* representation of TEA departs most significantly from the *C* representation.

2.2 Streams

Taking a closer look at the *C* program of Figure 1, we see that (argument initialization, the loop counter, and result initialization aside) only three variables are updated imperatively: `sum`, `y` and `z`. Furthermore, these variables are updated in lock-step (exactly once per loop iteration). We could have made *μCryptol* an imperative language (for example, by introducing mutable references in the style of ML). However, recall from Section 1 that one of our design goals was to remain neutral as to whether *μCryptol* programs should execute on a general purpose microprocessor, or be synthesized to hardware. So what does TEA look like in hardware?

Figure 3 presents an idealized representation of TEA in synchronous (single-clocked) hardware. Each arrow represents a 32-bit datapath. The boxes labeled `sum`, `y` and `z` are intended to be latches (with shared latched signal). All remaining circuits are combinatorial, representing the corresponding operators in *C* on 32-bit words. The latch `sum` starts with zero, `y` with v_0 and `z` with v_1. Clocking the circuit 32 times leaves the encrypted block in latches `y` and `z`.

A long tradition[12–14] in hardware modeling is to represent latches by the infinite sequence of values

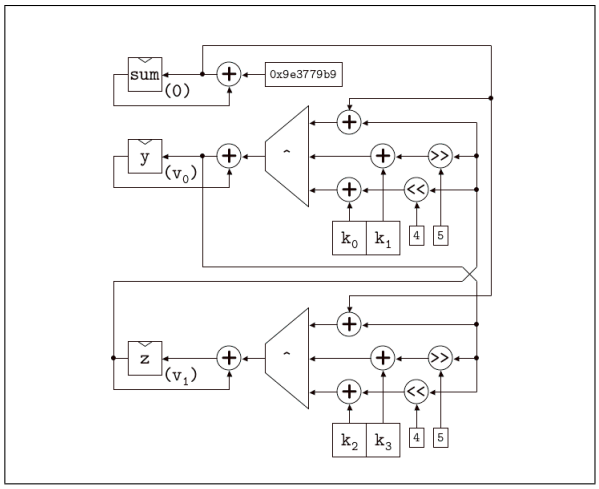

Fig. 3: TEA represented in a latched circuit style.

they latch. We call infinite sequences *streams*, to distinguish them from vectors, whose width must always be finite. Returning to the *μCryptol* program of Figure 2, we have three stream definitions corresponding to the three latches of Figure 3, and the three imperative variables of Figure 1. By using streams, *μCryptol* avoids needing any imperative features, which simplifies both its formal specification and compilation, and (we believe) makes the behavior of programs more predictable by their authors.

The easiest place to start is with `sums`. Its type signature is `sums : Word^inf`, indicating `sums` is a stream of 32-bit words. We use `^inf` as the type constructor for streams, which is suggestive of streams being vectors only with infinite width. (`inf` is not, however, a type.) The body of `sums` defines it to be the singleton vector `[0]` followed by (the `#` primitive appends a vector to a stream) the stream `[ sum + delta | sum <- sums ]`. This term is a *stream comprehension*, and mimics the set comprehensions familiar from set theory. In particular, the comprehension states "construct a stream which, for each successive element of `sums` (call it `sum`), has the element `sum + delta`."

Of course, this is a recursive reference to `sums`, hence care must be taken to ensure the definition is *well-defined*. In this case, the second element of `sums` depends on only the first element, which is known to be zero. Similarly, the third element depends only on the second, which we just calculated, and so on. A novelty of *μCryptol* is its use of the type system to decide well-definedness of streams. Section 3 covers this aspect in detail.

Continuing with the example, we consider the stream `ys`. Like `sums`, it has a type signature indicating its element type, and a body indicating its first element is `v0`, the first 32-bit word of the 64-bit block. Also like `sums`, its successive elements are defined by a stream comprehension, but this time with three *arms*. A multi-arm stream comprehension steps through each arm in parallel. The second and third arms draw successive elements from streams `ys` and `zs`. The first arm, however, draws elements from the stream `drops{1} sums`. This term is the stream `sums`, but with the first element removed. Looking at the *C* program of Figure 1, we see the calculation for `y` uses the value of `sum` updated on the previous line. Similarly, looking at the circuit of Figure 3 we see the combinatorial circuit updating latch `y` uses the newly-computed value for latch `sum`. Hence in the *μCryptol* program, we must "reach ahead" one stream element when drawing a value for `sum`.

The body of the comprehension for `ys` mimics the *C* expression used to update `y`. The term `k @ 0` indexes the first (leftmost) element of the vector `k`. The type system requires that the width of an index be just enough bits to index into any of the elements of the left-hand side. In this case, since `k` is of width 4 and the index is of width 2, the type system effectively prohibits indexing out of range into `k`. However for non power-of-two sized vectors, it is possible to attempt to index out-of-range at run time, a point we return to later.

The operator `^^` is bitwise exclusive-or on words, and `+` is addition modulo 2^n when operating on words of width n. The `<<` and `>>` operators are analogous to their counterparts in *C*, albeit generalized to arbitrary vectors of arbitrary width. Every simple type in *μCryptol* has a corresponding value for "zero", hence the shift operators always know how to shift appropriate zeroes into the left or right of a vector.

The stream `zs` is defined similarly for `ys`, but with a `drops` on both `sums` and `ys`. Again, both the *C* program and the circuit diagram justify the reaching ahead.

With the three streams out of the way, we now return to explain the body term `[ysN, zsn]` of `codes`. The `@@` operator is analogous to the `@` operator, except indexing into a stream instead of a vector. *μCryptol* allows type abbreviation names for types of kind `Nat` to appear in terms, where they are equivalent to a literal number of the same value. Thus the two occurrences of `N` are equivalent to `32`, and the vector consists of the 33'rd elements of the streams `ys` and `zs`. These values correspond to the final values of `y` and `z` in the *C* program, and the final contents of latches `y` and `z` in the circuit after 32 clocks.

In the beginning of this section we stated TEA uses 64-bit block size and 128-bit key, yet the *μCryptol* program uses exclusively 32-bit words. We can easily repair this artifact of the transliteration from *C* with the following definition:

```
code' : (B^64, B^128) -> B^64;
code' (v, k) = join (code (split v, split k));
```

The primitive `split` breaks a vector of $n \times m$ elements into a vector of n vector of m elements, and `join` does the reverse.

2.3 Polymorphism

How do `split` and `join` know these values of n and m? And for that matter, how does `+` know the modulus for addition, and literal numbers know how many leading `false` bits are required? Most primitives in *μCryptol* are *constrained parametric polymorphic*[15]. For example, the literal n has the type:

`forall w : Nat . B^w where w >= width` n

Here `width` is a type operator (of kind `Nat -> Nat`) yielding the number of bits required to represent its argument. This type states that n may have any width `w` (a natural number), *provided* it is greater or equal to `width` n. Similarly, `+` has the (unconstrained) type:

```
forall w : Nat . (# B^w, B^w #) -> B^w
```

Consider the expression `3 + 2`. If we instantiate the `w` type parameter of each of these three primitives to `2`, the expression evaluates to `1`, since the addition is modulo 4. However, instantiating them to `3` yields `5`. Hence in *μCryptol the meaning of a term may depend on the instantiation of type parameters within it.* Yet looking again at the *μCryptol* program of Figure 2 we see no mention of type instantiation for the 77 type parameters of all the primitive instances.

μCryptol uses a *type completion* algorithm to propagate known type information from type signatures and type annotations down into primitives at the leaves of the syntax tree. The result of the algorithm is either:

1. A "type error" message if there is no way to instantiate type parameters consistent with existing types; or
2. An "insufficient type information" message if there is no unique choice for a type parameter; or
3. An equivalent program with all type parameters explicitly instantiated.

Since we can only give meaning to programs *after* type completion, we feel it is critical to make the type completion algorithm as simple as possible. As a first step, we restrict polymorphism to primitives only, and require all programmer definitions to be monomorphic. We then use a form of "local type inference"[16, 17] in which terms are checked in a context of a partially-determined type, the result of which must be either a fully-determined type or an error. The algorithm proceeds left-to-right on primitives, pushing partial types for each argument down, and using the resulting fully-determined types to further inform the types of remaining arguments to the right. Constraint simplification is used locally per primitive occurrence to propagate type information, but constraints from distinct primitive occurrences *never* interfere with each other (compare with Hindley/Damas/Milner style type inference[18] which uses a global constraint set). The type completion algorithm has the property that it will never guess how to instantiate a type parameter if there is not a unique choice, hence we can be sure that type completion will never require a *μCryptol* compiler to guess the intended meaning of a program.

It is possible for the programmer to manually instantiate any type parameter. For example, `3 + 2` in a context of type `B^2` may be written as `3 + {2} 2` or `3{2} + 2` or even (for the truly pedantic) `3{2}+{2} 2{2}`. Primitives with multiple type parameters may have them instantiated by name or position. Sometimes a type instantiation is required. For example, the type completion algorithm can never determine the number of elements of a stream to drop in an occurrence of `drops`, hence that number must be given in a type instantiation. Hence the term `drops{1} sums` appearing in Figure 2.

2.4 Arbitrary values

Earlier we mentioned the possibility of indexing out-of-range of a vector at run time. A number of other things can go wrong at run time: division, remainder (on numbers or polynomials), or log of zero, shift or rotate out of range, and attempting to find the inverse of a polynomial with respect to a reducible polynomial. We must now explain what happens in these erroneous situations. The problem has two tensions.

For the first tension, recall our requirement of remaining neutral as to a hardware *vs.* software implementation of a *μCryptol* program. Consider for

the moment how index out-of-range would "naturally" be handled. Microprocessors with memory management units may raise a processor exception should the access fall outside of a readable page, while those without may happily address any byte of memory without complaint, possibly wrapping around segment boundaries. A hardware lookup table, without additional comparator logic, would most likely wrap around, or may yield an indeterminate result. On the one hand we wish to be precise as to the meaning of every *μCryptol* program, yet on the other hand we don't want to over-constrain implementations to use additional cycles/gates to enforce a particular behavior.

The second tension arises from the divide between cryptographic algorithm author and implementor. The author may know, by careful reasoning or deep understanding of the algorithm, that no erroneous situation would ever arise. In this case, our semantics for *μCryptol* should not force an implementor (of which the compiler is but one) from inserting unnecessary safety checks for a situation which will never occur. However, during development the author may wish to know if anything has gone awry, and would welcome such safety checks.

Our compromise is to allow primitives to return *arbitrary yet well-typed values* in erroneous situations, and to be *lazy* on arbitraries. For example, an implementation of `[0{8}, 1, 2]@3` is free to return any 8-bit word. Furthermore, an implementation of `[0{8}, 1, 2]@3 - [0{8}, 1, 2]@3` need not be `0`, since the implementation is free to choose each arbitrary value independently. Since *μCryptol* is pure, the implementation cannot perform any observable side-effect in this case (such as mailing the current memory image to `hacker@mafioso.com`). Since *μCryptol* is lazy on arbitrary values, the result may actually have no impact on the remainder of the program. To aid debugging, our interpreter for *μCryptol* represents arbitrary values explicitly as "arbitrary", and reports the program location they first arose.

Though our examples didn't show it, *μCryptol* also supports `if-then-else` conditionals, enumerations, and two additional methods of expressing streams which are mostly of use to the compiler.

3 Language

We refer the reader to the *μCryptol* Reference Manual[9] for a complete description of the language syntax, kind- and type-system rules, and the kind and type completion algorithms. This section will give an overview of the above, then focus on streams.

3.1 Syntax

Figure 4 presents *μCryptol* syntax. We write $\overline{t}$ to detote the sequence $t_0 ... t_{n-1}$ for implicit n. The syntax has been slightly simplifed (some kind and type annotation posibilities are not handled) and idealized (we are not specific about separators and empiness constraints on sequences of syntactic items).

In Section 1 we said all terms have type of kind `Type`. In fact, there are three kinds relevant to terms:

`Simp` is the kind of types of simple values which may be freely passed and returned from functions, and stored in tuples and vectors. In other words, values of types of kind `Simp` are first-class values.

`FnSt` adds to `Simp` the types of functions and streams, which are not first class. Streams may only be passed as arguments and returned by a just a few primitives.

`Type` adds to `FnSt` the types of stream transformers, which are primitives manipulating streams, such as `drops`.

We have that `Simp` is a sub-kind of `FnSt`, which is a sub-kind of `Type`.

Delay analysis of stream definitions requires the type system to keep track of delays, which are integers. We use the kind `Int` for these types.

The kind `?` represents the unknown kind. We call a kind *partially determined* if it contains one or more occurrences of `?`, and *fully determined* otherwise. Partially determined kinds are used during kind completion.

Types may be built from the application of a type operator, *type_op*. The type operators of *μCryptol* are given in Figure 5, along with their kinds.

The type τ`^inf` is shorthand for the type τ`^inf[32,?]`.

Types such as `5 + 3` and `width 14` are subject to simplification, in this case to `8` and `4`. We call all other types, such as `8`, `B^8` and `(B^8,B)`, *structural*.

Paralleling the situation for kinds, types may also include `?`, the unknown type. We use the same partially- and fully-determined qualifiers for types. Partially determined types are used during type completion.

Naturals	n	Integers	z		
Type variable names	*tyvar*	Type abbreviation names	*tyabbr*	Term variable names	*tmvar*

Kinds	κ	::=	`Simp` | `Nat` | `Int` | `FnSt` | `Type` | κ `->` κ | `?`
Types	τ	::=	*tyvar* | *tyabbr* | `(` τ `:` κ `)` | `?` | τ *type_op* τ | *type_op* $\overline{\tau}$
Primitive constraint	*pconstr*	::=	τ `>=` τ | τ `=` τ | τ `==` τ
Type schemes	σ	::=	`forall` $\overline{\textit{tyvar} : \kappa}$ `.` τ `where` $\overline{\textit{pconstr}}$
Patterns	p	::=	*tmvar* | `_` | `(` $\overline{p}$ `)` | `[` $\overline{p}$ `]` | `(` p `:` τ `)`
Unboxed tuple patterns	*ut_pat*	::=	`(#` $\overline{p}$ `#)`
Type binding sequences	*tbs*	::=	`{` $\overline{\tau}$ `}` | `{` $\overline{\textit{tyvar} := \tau}$ `}`
Arm conditions	*cond*	::=	`<` *type_atom* | `true`
Comprehension binding	*bind*	::=	p `<-` t
Case split arm	*arm*	::=	*cond* `->` t
Terms	t	::=	*tmvar* | *tyabbr* $(tbs)_{\text{opt}}$ | `(` $\overline{\textit{term}}$ `)` | `(` t `:` τ `)` | `[` $\overline{\textit{term}}$ `]` | `[` t `(,` $t)_{\text{opt}}$ `..` t `]`
		|	`[` t `|` $\overline{\textit{bind}}$ `]` | `[|` t $\overline{\textit{\| bind}}$ `|]` | *term_op* $(tbs)_{\text{opt}}$ $\overline{t}$ | t *term_op* $(tbs)_{\text{opt}}$ t | t $\overline{t}$
		|	t `where {` $\overline{\textit{locdef}}$ `}` | `if` t `then` t `else` t | `assume` t `in` t
		|	`\` *tmvar* `.` `{` $\overline{\textit{arm}}$ `}` | t `.` *tmvar* τ
Stream definition	*recdef*	::=	*tmvar* `:` τ `;` *tmvar* `=` t `;`
Local definition	d	::=	p `:` τ `;` p `=` t `;`
		|	*tmvar* `:` τ `;` *tmvar* *ut_pat* `=` t `;`
		|	`rec` *ut_pat* $\overline{\textit{recdef}}$
		|	`itr` *tmvar* `:` τ `;` `itr` *tmvar* `(#` $\overline{p}$ *tmvar* `#)` p `=` `{` $\overline{\textit{arm}}$ `}` `;`
		|	*tyabbr* `:` κ `;` *tyabbr* `=` τ `;`
Programs	*prog*	::=	`exports` $\overline{\textit{tmvar}}$ `;` $\overline{d}$

Fig. 4: Simplified and idealized syntax of *μCryptol*.

Type schemes are only used internally by the type system, and never appear within *μCryptol* programs.

Examples of most term forms have already been presented in Section 1. Terms may be built by the application of term operators, *term_op*. (We also call these term primitives, or just primitives). These operators, along with their type schemes, are presented in Figure 6.

An enumeration is either of the form `[`*first*`,` *next* `..` *last*`]` or `[`*first* `..` *last*`]` (in which *next* is taken to be *first* `+` 1). Enumerations are shorthand for the vector `[`$\overline{n}$`]` where $n_i = \textit{first} + (\textit{next} - \textit{first}) * i$.

The term `assume` t_0 `in` t_1 is t_1 if t_0 is `true`, otherwise is an arbitrary (implementation defined) value of the same type as t_1.

3.2 Type completion

The type completion algorithm is given four inputs:

1. A possibly partially-determined *expected* type. This type is "pushed" down into the term by the algorithm, and collects all type information from type signatures, type annotations, and the term context.
2. A type context giving the fully-determined types for all in-scope term variables. The algorithm requires all function arguments types to be declared, either by a type signature on the function definition or type annotations on the function's argument patterns.
3. The definitions for all in-scope type abbreviations.
4. The term to be type completed.

The algorithm either fails with a type error message, or succeeds and returns a fully-determined type and type completed term. The fully determined type will match, modulo type simplification, the partially determined type, except that each occurrence of `?` will have been instantiated by a structural type of the appropriate kind.

The type completion algorithm invokes a similar kind completion algorithm to determine the kinds of types appearing in programs. The kind completion algorithm also accepts a possibly partially-determined kind, and returns a fully-determined kind and rewritten type.

The use of `?` within partially determined types cannot express type sharing. For example, in the type `(?,?)`, each occurrence of `?` may be instantiated independently. However, the algorithm uses type constraints while checking application of primitive operators based on the primitive's known type scheme. For example, the type `(a,a)` can express intended equality between the tuple's element types, and such types may arise from the specialization of a type scheme. In this way,

Constants

Boolean	`false, true : B`	
Natural	n : `forall w : Nat . B^w where w >= width` n	
General $0/-1$	`zero, limit : forall t : Simp . t`	

Arithmetic in $\mathbf{Z}_{2^n}$

Unary op	op _ : `forall w : Nat . B^w -> B^w`	for $op \in \{$`-, lg2`$\}$
Binary op	_ op _ : `forall w : Nat . (# B^w, B^w #) -> B^w`	for $op \in \{$`+, -, *, /, %, **`$\}$
Binary func	f _ _ : `forall w : Nat . (# B^w, B^w #) -> B^w`	for $f \in \{$`min, max, gcd`$\}$

Arithmetic in $\mathbf{GF}_{2^n}$

Addition	`padd _ _ : forall wl : Nat, wr : Nat . (# B^wl, B^wr #) -> B^(max wl wr)`
Multiplication	`pmult _ _ : forall wl : Nat, wr : Nat . (# B^wl, B^wr #) -> B^(wl + wr - 1) where wl + wr >= 1`
Division	`pdiv _ _ : forall wl : Nat, wr : Nat . (# B^wl, B^wr #) -> B^wl`
Remainder	`pmod _ _ : forall wl : Nat, wr : Nat . (# B^wl, B^wr #) -> B^(wr - 1) where wr >= 1`
Greatest common divisor	`pgcd _ _ : forall wl : Nat, wr : Nat . (# B^wl, B^wr #) -> B^(max wl wr)`
Inversion	`pinv _ _ : forall wl : Nat, wr : Nat . (# B^wl, B^wr #) -> B^(wl - 1) where wl >= 1`

General equality

Binary rel	_ rel _ : `forall t : Simp . (# t, t #) -> B`	for $rel \in \{$`==, !=`$\}$

Inequalities in $\mathbf{Z}_{2^n}$

Binary rel	_ rel _ : `forall w : Nat . (# B^w, B^w #) -> B`	for $rel \in \{$`<, <=, >, >=`$\}$

Logical operators (extended pointwise to words)

Bit binary op	_ bop _ : `(# B, B #) -> B`
Word binary op	_ wop _ : `forall w : Nat . (# B^w, B^w #) -> B^w` for $(bop, wop) \in \{$(`/\`, `&&`)(conjunction), (`\/`, `||`)(disjunction), (`^`, `^^`)(exclusive-or)$\}$
Bit complement	`not _ : B -> B`
Word complement	`~_ : forall w : Nat . B^w -> B^w`

Vector operations

Shift/rotate	_ op _ : `forall w : Nat, t : Simp . (# t^w, B^ width(w - 1) #) -> t^w where w >= 1` for $op \in \{$`<<`(shift left), `>>`(shift right), `<<<`(rotate left), `>>>`(rotate right)$\}$
Index/reverse index	`_@_, _!_ : forall w : Nat, t : Simp . (# t^w, B^ width(w - 1) #) -> t where w >= 1`
Zeroth element	`zeroth _ : forall t : Simp . t^1 -> t`
Append	`append` n $\overbrace{\texttt{\_ \ldots \_}}^{n}$: `forall w0 : Nat, ..., w`$_{n-1}$ `: Nat, t : Simp . (# t^w0, ..., t^w`$_{n-1}$ `#) -> t^(w0 + ... + w`$_{n-1}$`)`
Extract	`extract` n _ : `forall w0 : Nat, ..., w`$_{n-1}$ `: Nat, t : Simp . t^(w0 + ... + w`$_{n-1}$`) -> (t^w0, ..., t^w`$_{n-1}$`)`
Join	`join _ : forall wi : Nat, wo : Nat, t : Simp . t^wi^wo -> t^(wi * wo)`
Split	`split _ : forall wi : Nat, wo : Nat, t : Simp . t^(wi * wo) -> t^wi^wo`
Suffix	`drop _ : forall wl : Nat, wr : Nat, t : Simp . ^(wl + wr) -> t^wr`
Prefix	`take _ : forall wl : Nat, wr : Nat, t : Simp . t^(wl + wr) -> t^wl`
Segment	`segment _ _ : forall ws : Nat, wr : Nat, t : Simp . (# t^(ws + wr), B^ width wr #) -> t^ws where ws >= 1`
Transpose	`transpose _ : forall wi : Nat, wo : Nat, t : Simp . t^wi^wo -> t^wo^wi`
Reverse	`reverse _ : forall w : Nat, t : Simp . t^w -> t^w`

Other operations

`true` if odd parity	`parity _ : forall w : Nat . B^w -> B`
`true` if irreducible	`pirred _ : forall w : Nat . B^w -> B`
Project	`project` n n' _ : `forall t0 : Simp, ..., t`$_{n-1}$ `: Simp . (t0, ..., t`$_{n-1}$`) -> t`$_{n'}$ for $n' < n$

Stream operations

Suffix	`drops _ : forall wl : Nat, wi : Nat, t : Simp, d : Int, l : Nat . t^inf{wi, d -- wl, l} => t^inf{wi, d, l}`
Prefix	`takes _ : forall wl : Nat, wi : Nat, t : Simp, wh : Nat . t^inf[wi, wh] => t^wl`
Segment	`segments _ _ : forall ws : Nat, wi : Nat, t : Simp, wh : Nat . (# t^inf[wi, wh], B^wi #) => t^ws`
Index	`_@@_ : forall wi : Nat, t : Simp, wh : Nat . (# t^inf[wi, wh], B^wi #) => t`
Append	`_##_ : forall wl : Nat, wi : Nat, t : Simp, d : Int, l : Nat . (# t^wl, t^inf{wi, d ++ wl, l} #) => t^inf{wi, d, l}`
Cycle	`cycle _ : forall w : Nat, wi : Nat, t : Simp, d : Int, l : Nat . t^w => t^inf{wi, d, l} where w >= 1`

Fig. 6: Term operators and their types.

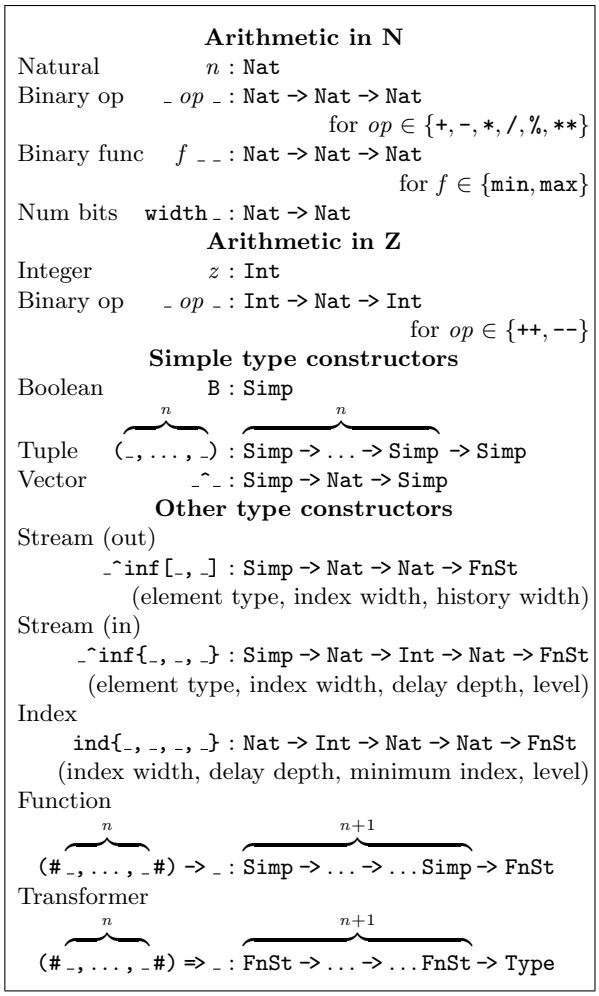

Fig. 5: Type operators and their kinds.

type information gathered from each primitive argument may influence the expected type of arguments to its right. In other words, the algorithm uses partially determined types to propagate type information between nodes, and type constraints to propagate type information between arguments.

We have found the type completion algorithm to work well on the *μCryptol* programs we have written. Only literal numbers, and primitives such as `split` and `drops`, require systematic type annotations. The behavior of the type completion algorithm is predictable from the program syntax. Error messages generally refer to the source of the programming error, rather than a remote program location suffering from the consequence of an error made elsewhere.

3.3 Well-defined streams

For the remainder of this section, we shall focus on the key aspect of the type system: Ensuring all cliques of stream definitions are *well-defined*.

In the following we shall distinguish between program contexts *inside* a particular clique of mutually-recursive stream definitions, and those *outside* of it. For example, in

```
xs @@ 3
  where {rec xs = [1] ## [x + 1|x <- xs]; }
```

the occurrence of `##` is inside, while `@@` is outside, the definition of `xs`.

We also use of the notion of an integer *stream delay*. We say the "delay from stream x to occurrence of stream y is d" to mean, for all sufficiently large indexes $k \in \mathbf{N}$, that the k'th element of stream x depends on the value of the $(k-d)$'th element of stream y at that occurrence. The "minimum delay from x to y" is the least delay over all occurrences of y, and dually for "maximum delay". To illustrate, consider:

```
rec (xs : B^8^inf) = [0, 1, 2] ##
  [x + y | x <- ys /*1*/
         | y <- drops{2} ys /*2*/];
and (ys : B^8^inf) = xs;
```

There are two occurrences of `ys` in `xs`, the first with delay 3, the second with delay 1. There is one occurrence of `xs` in `ys`, with delay 0. In general, an instance of `##` increases the delay from the stream under definition to any streams appearing in the primitive's right hand side, while an instance of `drops` decreases the delay. Delays may be negative.

The definition of well-defined streams is given in Figure 7.

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*

$$f \in (\mathbf{N} \times S) \to \mathbf{N}$$

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

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

Fig. 7: Well-defined streams.

3.4 Deciding well-definedness

With the definition of well-definedness in hand, we now choose an intended model for streams. In the following we use **N** to denote the set of natural numbers, **Z** the set of integers, **V** the partially ordered set (poset) of *μCryptol* values, and $T_\perp$ the poset T with new bottom element $\perp$. Figure 8 distinguishes three classes of models based on denotational and operational interpretations.

The function class is the most expressive. Implementing streams using simple functions from indexes to elements is technically possible, but would be too slow in practice for streams which reference themselves multiple times. This problem can be avoided using a dynamically-constructed memoization table, but that in turn requires dynamic memory management.

The co-inductive class is familiar to users of the lazy functional programming language *Haskell* [19], and may be implemented using updating closures[20]. However, again they require dynamic memory management. *Cryptol* implements this class.

The iterative class is the least expressive, but has the virtue of requiring only a finite amount of memory which can be statically allocated at compile time. In this class, the computation of a new stream element is guaranteed to require only up to h previously computed elements from the stream. *μCryptol* implements this class (suitably generalized to multiple streams).

Next is to choose how flexible the language should be in defining streams, which will in turn influence the difficulty of deciding well-definedness. Again, there is a spectrum of approaches.

Least flexible is to restrict the language syntax so that every stream definition is guarded by a use of `##`, forbid the use of `@@`, `takes` and `segments` within the body of any stream definition, and forbid the use of `drops` altogether. In this case every delay is strictly positive, and every clique of definitions has a measure function.

At the other extreme, most flexible is to allow all stream primitives to be used in all contexts. However the problem of determining a delay for each stream occurrence is most likely undecidable, since it is now possible to express a data-dependent index into a stream inside its own definition. Indeed, we must even refine our notion of measure function to allow delays to be index dependent (that is, we must replace d in Figure 7 with the application $d(k)$). For example, the definition

```
rec xs = [0 .. 3]##
    [(xs @@ (3 - (x % 4))) + 1|x <- xs];
```

is indeed well-defined, and even has an iterative implementation, but this fact requires knowledge of the properties of modular arithmetic. *Cryptol* takes this approach, but does not attempt to detect ill-defined streams at compile time.

(Note that *Cryptol* goes even further and allows definitions which may defer the choice as to whether they manipulate streams or vectors. Most *Cryptol* primitives operate on arbitrary sequences whose width is a type parameter which may be instantiated to a finite number (thus yielding a vector) or the distinguished number `inf` (thus yielding a stream).)

In *μCryptol* we take a middle approach. We restrict the use of primitives on a particular stream according to whether they are inside or outside that stream's definition. More precisely, we forbid `@@`, `takes` and `segments` to be used on a stream inside its own definition, and forbid `drops` to be used outside. We also forbid the application of user-defined functions to streams, and forbid stream comprehensions with multiple generators in a single arm. With these restrictions, the delays can be determined by the type system by tracking uses of `##` and `drops`.

To see the system in action, consider the clique of definitions in Figure 9.

```
rec (xs : B^8^inf) =
  [3] ## [ x+y | x<-xs | y<-[0{8}]##ys ];
and (ys : B^8^inf) =
  [5] ## [ x+y+z | x<-drops{2} xs
                 | y<-ys | z<-zs ];
and (zs : B^8^inf) =
  [7] ## [ x+z | x<-drops{3} xs | z<-zs ];
```

Fig. 9: An example clique of streams to illustrate well-definedness.

The minimum delays for these definitions is presented as a directed weighted graph in Figure 10. (Later we shall also require a maximum delay graph.)

These definitions are indeed well-defined, using the measure function:

$$f(k,x) = 3 \times (k + \mathit{offset}(x)) + \mathit{order}(x)$$
$$\text{where } \mathit{offset} = \mathtt{xs} \mapsto 0, \mathtt{ys} \mapsto 1, \mathtt{zs} \mapsto 2$$
$$\mathit{order} = \mathtt{xs} \mapsto 0, \mathtt{ys} \mapsto 2, \mathtt{zs} \mapsto 1$$

Of course any measure function will do, but the above measure has a form which applies to any

Class	Denotational	Operational	Example
Iterative	$\left(\sum_{0 \le i \le h} \mathbf{V}^i\right) \to \mathbf{V}$	Iterators over h history elements	`rec xs = [0, 1]## [x + y | x <- xs | y <- drops{1} xs];`
Co-inductive	$\nu a.\mathbf{V} \times a$	Lazy lists	`rec xs = [0]## [x + y | x <- xs, y <- [0, 1]];`
Function	$\mathbf{N} \to \mathbf{V}_\perp$	Memoized functions	`rec xs = takes{4} (drops{4} xs)## ([0 .. 3] ## xs);`

Fig. 8: Three classes of recursive stream definitions.

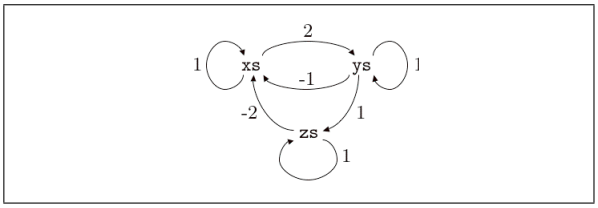

Fig. 10: Minimum delay graph corresponding to streams in Figure 9

clique of definitions. Indeed, Figure 11 gives an algorithm to construct a measure function from any minimum delay graph.

1. Let G be the delay graph, and n the number of vertices of G.
2. Let G' be the all-pairs shortest path closure of G.
3. If any vertex of G' has an edge to itself with zero or negative weight, reject the streams as ill-defined.
4. Initialize $\mathit{offset} \in \mathit{vertices}(G) \to \mathbf{N}$ to be everywhere 0.
5. While there exists a vertex x in G with an outgoing edge of negative weight $-d$:
   5.1. Add d to the weight of each outgoing edge of x in G.
   5.2. Subtract d from the weight of each incoming edge of x in G.
   5.3. Update $\mathit{offset}(x)$ to be $\mathit{offset}(x) + d$.
6. At this point G has no negative weighted edges. Delete from G all strictly positive weighted edges.
7. Transpose G.
8. Find a topological ordering O of the vertices of G.
9. Let $\mathit{order} \in \mathit{vertices}(G) \to \mathbf{N}$ be the function from each vertex to its position in O.
10. Return the measure function

$$f(k, x) = n \times (k + \mathit{offset}(x)) + \mathit{order}(x)$$

Fig. 11: Calculating a measure function from a minimum delay graph.

Thanks to the above algorithm, we may reduce the problem of deciding well-definedness to that of finding the delay graph G. (We defer showing soundness and completeness of this algorithm to future work.)

Recall from Figure 8 that we wish to restrict each stream in *μCryptol* to the class of iterators over a known finite number, h, of history elements. We call the smallest such number the *history width* of the stream. (There's no need for all streams in the same program or even same clique to share the same history width.) It turns out that every well-defined stream is serendipitously also finite without the need for any additional restrictions. However, the actual history widths are needed when compiling streams (see Section 4).

Thankfully, with a small enhancement the algorithm of Figure 11 may also determine history widths. First, we switch from weighted graphs to bi-weighted graphs, where each edge now holds both the minimum and maximum delay between the streams of its vertices. The above algorithm continues to work exclusively on minimum weights, except for steps 5.1 and 5.2, which must adjust both minimum and maximum weights of an edge together. Just before step 6, the minimum history width for each stream x may be read off as the greatest maximum delay of all incoming edges to x. (If x has no incoming edges, the history width is taken to be 1.)

So far our examples of streams have used types of the simple form τ`^inf`, which we stated was an abbreviation for τ`^inf[32, ?]`. In order to construct G for each clique during type completion, the type system has two forms of stream types:

τ`^inf{`w`,` d`,` l`}` The type of a stream as it appears inside of the clique it is being defined within.

τ`^inf[`w`,` h`]` The type of a stream as it appears outside of the clique it has been defined within.

These types use the following parameters:

- τ The type (of kind `Simp`) of stream elements.
- w The width (of kind `Nat`) of stream index words. All streams in a clique must share the same width.
- d The delay (of kind `Int`) from the stream under definition to a program context of this

type. This number is used to track delays though uses of `##` and `drops`.

l The nesting level number (of kind `Nat`) of the stream under definition. This number is used to prevent a clique from becoming mutually-recursive with a nested clique within it.

h The history width (of kind `Nat`) of the stream.

Recall from Figure 6 the type schemes for the stream primitives. The primitives `takes`, `segments` and `@@` are the only consumers of streams outside of definitions. These primitives are polymorphic on the parameters τ, w and h. The primitives `##` and `drops` are the only transformers of streams inside definitions. These primitives are also polymorphic on τ, w, d and l. However, notice how the delay type parameter is adjusted between the input and output types of these primitives. In particular, `##` adds to the delay of its argument, while `drops` subtracts. Thanks to these types, the majority of the delay analysis is done during routine type completion of primitive application.

To complete the analysis, the type completion algorithm proceeds as in Figure 12 when encountering a clique of stream definitions.

1. Construct a graph G with a vertex for each stream name in the clique, and initially no edges.
2. For each stream definition of the clique with name x and body t:
   2.1. Invoke the algorithm recursively with:
   - An expected type of τ_x`^inf{`w_x`,0,`l`}`, where τ_x is given in x's signature, w_x is either given or defaults to `32`, and l is the stream's (syntactic) nesting level.
   - The current type context extended to map each stream name y in the clique to a special type of the form τ_y`^inf{`w_y`,` *minRef* `,` *maxRef* `,` l`}`, where τ_y is as given in y's signature, w_y and l are defined as above, and *minRef* and *maxRef* are initially empty *mutable references* to delays.
   - The term t.

   2.2. At a stream variable y in the stream body, the algorithm may find itself with:
   - An expected type of the form τ_{exp}`^inf{`w`,`d`,`l`}`.
   - A type reported by the type context for y of the form τ_{act}`^inf{`w`,` *minRef* `,` *maxRef* `,` l`}`.

   In this case, check τ_{exp} and τ_{act} are compatible, and update either *minRef* or *maxRef* to minimally extend the interval between them such that $minRef \leq d \leq maxRef$. (If *minRef* and *maxRef* are empty, set them to be d.)

   2.3. When the algorithm returns from checking t, for each y such that the entry for y in the type context has non-empty *minRef* and *maxRef*, add an edge to G from x to y with minimum weight *minRef* and maximum weight *maxRef*.
3. Use the augmented algorithm of Figure 11 to determine a measure (if any) for the clique, and the history width h_x for each stream x.
4. For each stream x in the clique:
   4.1. If the type signature for x gave a known history width h, check that $h_x \leq h$, and revise h_x to be h.
   4.2. If required, revise h_x upwards to the next power of two.
5. Check that, for every pair of streams x and y, $w_x = w_y$.
6. Continue the algorithm on the definitions and terms in scope of the clique, with a type context extended to map each stream x to the type τ_x`^inf[`w_x`,`h_x`]`.

Fig. 12: The type completion algorithm for cliques of stream definitions.

4 Implementation: Front-end

Figure 13 shows the organization of the front-end passes of `mcc`. The type completion algorithm is run on the source program, and if successful, the type completed program is the starting point for the remaining compiler passes. The most important path is that running from the type completed program to flat *CrAM*.

4.1 Canonical *μCryptol*

Canonical *μCryptol* is a language subset with a direct operational interpretation, and is intended to be as close to a final implementation as possible while still remaining neutral as to hardware *vs.* software implementation. (Compare it with the STG code of the GHC *Haskell* compiler[20].) In the following, we shall concentrate mostly on software targets, since that has been the focus of our work, with the understanding that much of the following also applies to hardware targets.

In Canonical *μCryptol*, every value arising during execution is either small enough to pass and construct on a value stack (we call such values *unboxed*), or has an explicitly named variable bound

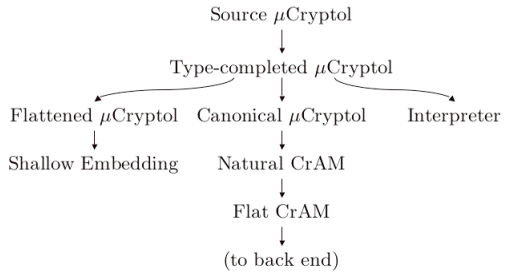

Fig. 13: Organization of front end `mcc` passes.

to its value in a `where` clause. All function and primitive arguments are ready to be passed without the need for additional storage to evaluate them. All definitions are at the top-level, eliminating the need for closures. Vector comprehensions are of the form $(\texttt{join})_{\text{opt}}$ `[`$term$`|`x`<-[0,1..`$n-1$`]]`, which can be implemented as a simple for-loop. All partial operations (those which may yield an arbitrary result) are wrapped by an explicit `assume` clause, which makes explicit the implied partiality. No patterns remain, and all function arguments are either wildcards (ignored) or simple variables (which name the argument location). A number of primitives, such as `reverse`, are eliminated since they have simple definitions as comprehensions. No values of zero-sized type, such as `()`, `()^3` or `B^0` are ever constructed, passed as arguments, or returned from functions (the absence of side-effects means such values are uniquely determined from their type alone, and so need no run-time representation). No top-level definition is shadowed (so that back-end tools don't need to respect scoping).

One final restriction of Canonical *μCryptol* is key. We could interpret recursive streams as lazy lists, and give them a direct operational interpretation using (for software) updating closures[20], or (for hardware) latched dataflow networks. However, for software, this would require the implementation to support dynamic memory allocation, while for hardware, it would distribute latches throughout the circuit, rather than grouping them into shift registers. To avoid this problem, Canonical *μCryptol* does not support streams, but instead has a notion of *iterators*. (Actually, iterators are also a part of *μCryptol*, and can appear in source programs, though it seems unlikely any human would wish to use them.) To illustrate, consider Figure 14, which defines the stream of Fibonacci numbers (modulo 2^8) and extracts the fifth such number.

```
rec (fibs : B^8^inf) = [0, 1] ##
  [x + y | x <- drops{1} fibs
         | y <- fibs];
fifth = fibs @@ 4;
```

Fig. 14: Fibonacci sequence in *μCryptol*.

Recall from Section 3 that every *μCryptol* stream is guaranteed to require only a finite number of earlier elements to calculate the current element, and that we introduced the term "history width" to refer to (an upper bound on) that number. Of course for `fibs`, only two previous elements are required. A *C* implementation of `fibs` exploiting this fact is given in Figure 15. (Though a programmer would greatly optimize this program, this stylized form generalizes to multiple streams and long history widths.)

```
typedef unsigned char word8;
void fibs(int i, word8 hist[2]) {
  int j;
  for (j = 0; j <= i; j++) {
    if      (j < 1) hist[0] = 0;
    else if (j < 2) hist[1] = 1;
    else hist[j&1] = hist[(j-2)&1] +
                     hist[(j-1)&1];
  }
}
word8 tmp[2];
word8 fifth() {
  fibs(4, tmp);
  return tmp[0];
}
```

Fig. 15: Fibonacci sequence in *C*.

The *C* `fibs` function accepts an index `i` of the desired element of the original *μCryptol* stream `fibs`, and a small *history buffer*, `hist`. The `i`'th stream element is left within `hist` on exit from the function, from which the caller may extract it. It calculates this element by iterating upwards from index 0 to `i`, calculating the current index's stream element based (if needed) on the earlier elements already evaluated and stored in `hist`. We implement the history buffer "lazily" using modular arithmetic on its index rather than explicitly shuffling along older elements as new ones are calculated. Since `hist` is of width $2 = 2^1$, we write `j&1` instead of `j%2`. We must use the same modulo indexing to extract the result: In this case, `4%2` is just `0`.

A hardware implementation is shown in Figure 16. Again, the finite history width property is exploited, this time to represent the history buffer as a shift register of width 8 and depth two. The shift register is initialized with `0` and `1`, then the circuit is clocked `i` − 2 times, after which the de-

sired element will be left at the head (rightmost end) of the shift register.

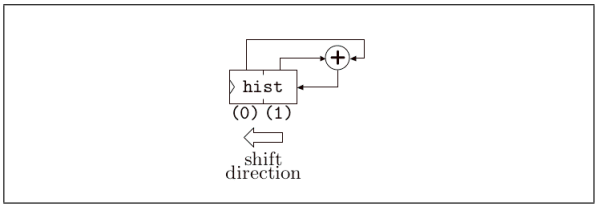

Fig. 16: Fibonacci sequence in hardware.

Figure 14 can be rewritten to Canonical *μCryptol* as in Figure 17.

```
itr fibs (# (i : B^32) #)
            (hist : B^8^2) =
  { < 1 -> 0
  | < 2 -> 1
  | true -> hist@(drop (i - 1)) +
            hist@(drop (i - 2)) };
(tmp : B^8^2) = fibs 4;
(fifth : B^8) = tmp@0;
```

Fig. 17: Fibonacci sequence in Canonical *μCryptol*.

Here `itr` is a keyword introducing a new top-level definition, `fibs`, which appears to the rest of the program as a function of type `B^32->B^8^2` taking an index to a history buffer. The canonical `fibs` function is intended to mimic that of the *C* version: Iterate from 0 to `i` updating `hist`. However the iteration and updating is *not* expressed explicitly within Canonical *μCryptol* (doing so would make *μCryptol* too expressive, and defeat our desire to remain side-effect free). Instead, the body of `fibs` actually has the type `(#B^32,B^8^2#)->B^8`. Given the current stream index `i` and the history buffer `hist` holding the elements $\texttt{i} - 1$ and $\texttt{i} - 2$ (or arbitrary should `i` be too small), the body of `fibs` simply does a case split on `i`, and calculates the appropriate element. That is to say, the body of `fibs` is just a code fragment which the compiler should wrap with the appropriate iterator loop and history buffer updating. The `itr` keyword is intended to alert the reader that the "function" being defined has this dual-nature: described by a fragment and completed by the compiler to be a function. (Compare with the function passed to a `fold` operator to recurse over a recursive datatype.)

Iterators generalize easily to the case of multiple mutually-recursive streams. The history buffer becomes a tuple of history buffers, one for each stream, and successive stream elements are computed in lock-step.

4.2 Transformation to canonical form

Figure 18 presents the sequence of source-to-source transformations which take any *μCryptol* program into canonical form. Most steps are straightforward or standard from the compilation of functional programming languages.

1. Introduce safety checks
2. Simplify vector comprehensions
3. Eliminate patterns
4. Eliminate streams
   - 4.1. Convert to indexed form
   - 4.2. Push stream applications
   - 4.3. Collapse arms
   - 4.4. Align arms
   - 4.5. Takes/segments to indexes
   - 4.6. Convert to iterator form
5. Eliminate primitives
6. Eliminate zero-sized values
7. Inline and simplify
8. Introduce temporaries
9. Eliminate nested definitions
10. Share top-level value definitions
11. Box top-level definitions
12. Eliminate shadowing

Fig. 18: Steps to convert type complete *μCryptol* programs into canonical form.

We shall focus on the six sub-steps to step 4—the translation which actually takes streams to the iterators described above.

Sub-step 1 Stream constructs are replaced with an equivalent term in *indexed form*.

For this step, *μCryptol* supports two additional term forms which (a) define streams as functions from indexes to values, and (b) apply such a function to a particular index. For example, the term `[0{8},1]##ys` (for stream `ys`) appears in indexed form as `\i.{<2 -> [0{8},1] @ (drop i)|true -> ys.i -2}`. The `\` indicates abstraction, while the `.` after `ys` indicates application.

The indexed form constructs are highly stylized, and *μCryptol does not* support general λ-abstractions and application. Abstraction is only over values of a special abstract index type `ind{`w`,` d`,` m`,` l`}`, where:

- w is the width of a concrete index into the stream. When a value of index type is used in a context other than application to a stream, it is coerced to type `B^`w. In the above example, `(drop i)` has type `B^1`, and discards the initial 31 bits of `i` as a 32-bit word.
- d is the delay depth of the context of the abstraction, and is used by the type system

to detect ill-defined stream definitions as described in Section 3.

m is the minimum value for the index. In the above example, this value would be 0 for `i` in the body to the right of `<2`, and 2 to the right of `true`.

l is the level for the stream definition containing this abstraction (again, see Section 3).

An abstraction is always paired with a case split on the index. An application must be of the form t `.` *tmvar* τ where:

t is a stream expression.

tmvar is a stream index variable.

τ is a type of kind `Int`, in other words, a constant integer, indicating the offset to apply to *tmvar* when indexing into t. The above parameter m prevents subtracting too large an offset from the index.

Just as for iterators, indexed form for streams is also available to programmers, and indeed may be freely intermixed with the combinator style introduced in Section 1.

Sub-step 2 Stream applications are pushed into `if-then-else` and `where` terms, and each stream definition within a clique is η-expanded[21] so as to begin with an explict index abstraction. (We use a fresh index variable name which is shared by all streams in a clique.)

Sub-step 3 Nested stream abstractions are collapsed into a single level. For example, given the abstraction:

```
\i.{ <2->0
   | <4->(\j.{ <1->1 | <2->2 | true->3 })
          . i -1
   | true -> (\j.{ <5->4 | true->5 })
              . i -2 }
```

we may correct for the application offsets `-1` and `-2` to yield:

```
\i.{ <2->0
   | <4->(\j.{ <2->1 | <3->2 | true->3 })
          . i +0 }
   | true -> (\j.{ <7->4 | true->5 })
              . i +0 }
```

which in turn may be collapsed into the single abstraction:

```
\i.{ <2->0 | <3->2 | <4->3
   | <7->4 | true -> 5 }
```

Notice how some abstraction cases (such as that yielding `1`) can be dropped since they will never fire.

Thanks to the η-expansion of step 2, each recursive stream definition will at this point be a single abstraction and case split.

Sub-step 4 Arms are aligned to eliminate strictly positive index offsets. Recall from Section 3 the termination measure for a clique of mutually-recursive stream definitions consists of an offset for each stream and an ordering amongst the streams. We now "slide" each stream according to its offset. This will allow the new elements for each stream to be constructed together.

Consider again the example of Figure 9, which at this point of compilation resembles Figure 19. Recall the offset for `zs` was determined by the type completion algorithm to be 2. We thus wish to slide the stream `zs` forward by 2 elements, inserting two dummy values (which we take to be `0`) at its head. All references to `zs`, both inside and outside the clique of streams, must be updated to take account of this slide. That is, we must add 2 to the offsets for indexes into `zs` from within the definition of `xs` and `ys`, and subtract 2 from the offsets for indexes into `xs` and `ys` from within the definition of `zs`. After doing this, and sliding `ys` by its offset 1, the result is as in Figure 20.

```
rec (xs : B^8^inf) = \i .
  { < 1 -> 3
  | < 2 -> (xs . i -1) + 0
  | true -> (xs . i -1) + (ys . i -2) };
and (ys : B^8^inf) = \i .
  { < 1 -> 5
  | true -> (xs . i +1) + (ys . i -1) +
            (zs . i -1) };
and (zs : B^8^inf) = \i .
  { < 1 -> 7
  | true -> (xs . i +2) + (zs . i -1) };
```

Fig. 19: Streams of Figure 9 before sub-step 4.

```
rec (xs : B^8^inf) = \i .
  { < 1 -> 3
  | < 2 -> (xs . i -1) + 0
  | true -> (xs . i -1) + (ys . i -1) };
and (ys : B^8^inf) = \i .
  { < 1 -> 0
  | < 2 -> 5
  | true -> (xs . i +0) + (ys . i -1) +
            (zs . i +0) };
and (zs : B^8^inf) = \i .
  { < 2 -> 0
  | < 3 -> 7
  | true -> (xs . i +0) + (zs . i -1) };
```

Fig. 20: Streams of Figure 9 after sub-step 4.

Another way to see sliding in action is to consider the same example presented in dataflow form. The

left of Figure 21 shows the dataflow before sliding, while the right shows it after. Notice all dataflow arrows point (slightly!) to the right after sliding. (This diagram also uses the ordering of the clique's measure to slightly offset each stream, albeit at a sub-element width. This aspect is realized in sub-step 6.)

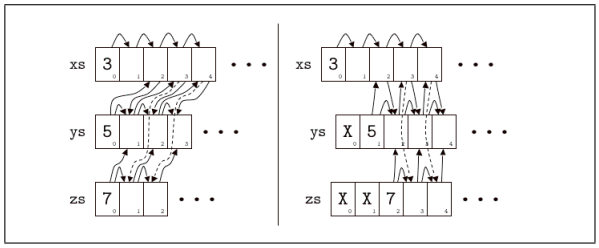

Fig. 21: The dataflow between elements for the streams of Figure 10 as written (left) and after alignment by insertion of dummy initial values (right).

Sub-step 5 Some primitives, such as `takes` and `segments`, are replaced with uses of `@@` if the sub-vector being extracted is wider than the history with of the stream. For example, `takes{3} xs` will be left as is if `xs` has history width of four or greater, but converted to `[xs@@0, xs@@1, xs@@2]` otherwise.

Sub-step 6 Each clique of stream definitions is rewritten as a single iterator definition. A history buffer for each stream is tupled according to the stream ordering in the clique measure. The arms of each stream in the clique are merged together, and the body term for each arm is tupled, also according to the stream ordering. Each reference to a stream within the clique is of the form $x\,.\,i\ -n$ for stream name x, index variable i (which is now shared by all streams in the clique) and natural number n. Each such term is replaced by `project` m j `@ (drop(`i `-` n`))`, where m is the number of streams in the clique and j the position of stream x in the stream ordering. References to a stream from outside the clique are replaced with a call to the iterator function, a project, and an index.

Figure 22 presents the streams of Figure 9 after this step.

A term such as

```
takes xs == [ 3, 3, 8, 28, 111 ]
```

in the scope of these definitions would be translated to

```
[ project 3 0 (iter_xs_ys_zs i)@drop i
```

```
itr iter_xs_ys_zs (# (i : B^32) #)
  (hist : ( B^8^1, B^8^1, B^8^1 )) =
  { < 1 -> ( 3, 0, 0 )
  | < 2 -> (project 3 0 hist@0 + 0, 0, 5)
  | < 3 -> ( xs, zs, ys )
    where {
      (xs : B^8) = (project 3 0 hist@0) +
                   (project 3 2 hist@0);
      (zs : B^8) = 7;
      (ys : B^8) = xs+(project 3 2 hist@0)+zs;
    }
  | true -> ( xs, zs, ys )
    where {
      (xs : B^8) = (project 3 0 hist@0) +
                   (project 3 2 hist@0);
      (zs : B^8) = xs+(project 3 1 hist@0);
      (ys : B^8) = xs+(project 3 2 hist@0)+zs;
    }
```

Fig. 22: Streams of Figure 9, after sub-step 6.

```
| (i : B^32) <- [0, 1 .. 4] ] ==
    [ 3, 3, 8, 28, 111 ]
```

4.3 Translation to *CrAM*

For software targets, there are still five aspects of compilation which, though common across all targets, cannot be cleanly expressed in *μCryptol*:

1. *μCryptol* uses names for all object locations, but we must resolve these to addresses, either into a stack for argument locations, or into a heap for boxed objects.
2. *μCryptol* assumes a stack discipline for the allocation of temporary space. For example, consider:

   t_3 `where {y =` t_2`; } where {x =` t_1`; }`

   μCryptol assumes temporary space is allocated for the result of t_1, then t_2, and then reclaimed after evaluation of t_3.
   However, because targets may have no dynamic memory management, we wish instead to statically allocate space for all temporary results, but at the same time share space between temporaries whose lifetimes never intersect. For example, in the above example t_1 and t_2 cannot safely share space, however they may do so in:

   `(`t_3 `where {y =` t_2`; }) + (`t_4 `where {x =` t_1`; })`
3. *μCryptol* makes no distinction between unboxed (may be passed and returned by value on the argument stack) and boxed (must be passed by reference) values. However, manipulation of boxed and unboxed values is very different in machine code.

4. For effeciency, it is vital for values to be constructed directly in place of their final destination, rather than in a temporary location which must later be copied into place. For example, consider:

   ```
   join [t|x <- [0..7]]
   ```

   This term will evaluate t eight times (with a different binding for `x` of course), to yield a sub-vector. These eight sub-vectors are then joined into a single vector. A naive compilation of this term would construct the eight sub-vectors in temporary spaces, then copy them all into the final vector. We prefer to, if possible, construct each sub-vector directly in place.
5. A number of primitives (such as division of boxed words) require additional temporary space.

We use an intermediate language, the *μCryptol Abstract Machine* (*CrAM*), and a translation from Canonical *μCryptol* to *CrAM* to effect the above five aspects. *CrAM* is a stack-based machine with built-in primitives corresponding to the majority of *μCryptol*'s arithmetic, logical and polynomial primitives. *CrAM* also makes memory allocation, address de-referencing, and the distinction between boxed and unboxed values explicit.

We actually use two flavors of *CrAM*. Natural *CrAM* does not assume a flat code address space, but instead allows instruction sequences to be nested. For example, `BRANCH`, which tests the stack top for `true` or `false`, has as parameters two instruction sequences. Natural *CrAM* also uses labels for all memory objects, regardless of whether they are temporaries or global objects. More information is available in the reference manual[10].

Flat *CrAM* uses conventional flat code address space, with labels and jump instructions. It also requires all temporary objects to have concrete addresses.

The translation from natural to flat forms involves the usual control flow encoding, but also resolves temporaries to their final addresses in the heap. Of course doing so requires knowledge of the size of memory objects, and any constraints on memory, such as segment boundaries and valid data address ranges. Both of these aspects are target dependent, hence the translation from natural to flat forms is parameterized by this small aspect of the target.

Temporary layout is a generalization of the graph coloring problem, which is NP-complete[22]. Our algorithm procceeds in two phases. The first phase constructs a conflict graph between temporaries based on the scope of their lifetimes, which is conveniently made explicit in the natural form of *CrAM*. The second phase assigns temporaries to memory locations to avoid conflict, using the heuristic of working from larger to smaller objects. We've found this algorithm to work very well on our example programs, despite its simplicity.

In order to support verification, `mcc` may translate any *μCryptol* program into a semantically equivalent set of *ACL2* definitions—a so called "shallow embedding" of *μCryptol* into *ACL2*. Unfortunately *ACL2* is not expressive enough for this translation to be compositional. In particular, *ACL2* does not allow nested definitions (needed for *μCryptol*'s `where` clauses), does not support anonymous λ-abstractions (needed for *μCryptol*'s stream expressions), and has a very weak notion of pattern matching (insufficient for *μCryptol*'s patterns). Hence a subset of the above transformations must be applied to a *μCryptol* program before it is in a "flattened" form suitable for translation to *ACL2*. A consequence of this flattening is that, using *ACL2* alone, we cannot perform a front-to-back verification of a compiled program. A future paper will describe how we address this problem using a hybrid of theorem provers.

We mention that `mcc` also contains an interpreter usable from a command-line interface. The interpreter works over any type-completed term (and even, though not shown in Figure 13, any intermediate stage of compilation). We found having a built-in interpreter to be tremendously useful while debugging the transformations.

5 Implementation: Back-end

Figure 23 shows the organization of the back-end passes of `mcc`.

We target the Rockwell Collins *AAMP7* embedded microprocessor[6, 7]. The *AAMP7* is a stack-based machine with 32-bit segmented addressing, 16/32-bit integer and 32/48-bit floating-point operands. The lack of registers improves code density (most instructions are a single byte), which is significant in embedded applications where code typically executes directly from slow ROM. Though the *AAMP7* does not have any particular hardware support for cryptographic algorithms *per se*, it does support running multiple concurrent process *partitions* which are rigorously separated in time and space by hardware[23]. This

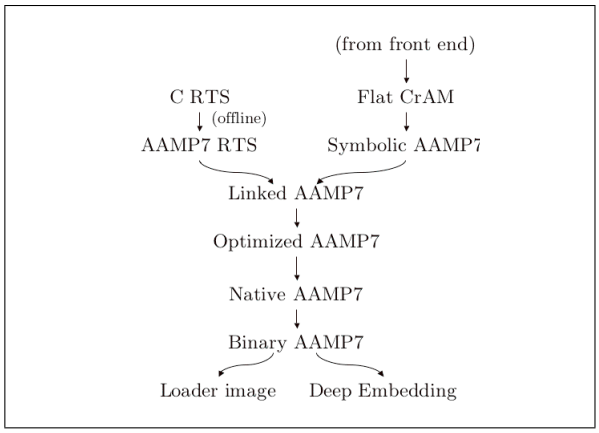

Fig. 23: Organization of back end `mcc` passes.

separation makes the *AAMP7* attractive for use in in-line encrypters which support multiple concurrent communication channels, since it allows each channel to carry data at different classification levels without risk of cross-channel leaking from a high to low level. In all other respects the *AAMP7* is a conventional microprocessor.

The back-end passes are routine as the majority of heavy-lifting has been done by the front-end. Most flat *CrAM* instructions can be translated to, at most, a handful of *AAMP7* instructions. There are only two complications. The first is that the order of operands on the *CrAM* stack is often mismatched with the order required by the corresponding instruction on the *AAMP7*. We deal with this by introducing a few "virtual" *AAMP7* instructions which rearrange the stack top. (The *AAMP7* has instructions to exchange the top two stack elements, but otherwise enforces a strict stack access order. In particular, the *AAMP7* stack is not randomly addressable.) During peephole optimization, the compiler attempts to parse, for each operand subject to a rearrangement, the maximal sequence of sequential side-effect free *AAMP7* instructions which create it. If each operand is successfully parsed to completion, the compiler simply reorders the groups of instructions and eliminates the virtual instruction. Otherwise, locals are used to effect the rearrangement. Thankfully this optimization fires in the majority of cases.

The second complication in translating *CrAM* to *AAMP7* is with the arithmetic and logical primitives on boxed words, and the polynomial primitives. These are sufficiently complicated to warrant encoding as a separate run-time system (RTS) of around 2,700 lines of *C*. We used an extant *C* to *AAMP7* compiler to compile the RTS to *AAMP7* assembly. Instances of these primitives in *CrAM* are translated to calls out to the RTS, and the relevant parts of the RTS are linked into the final program. Note, however, that the RTS is nothing more than these primitives. In particular, we don't need (or want) any RTS support for memory allocation and garbage collection, since this has all been statically determined by the compiler.

Verification is simplified if we know the final physical memory address of every heap object. Rather than emit relocatable *AAMP7* assembly language and try to reconstruct physical addresses from the assembler output, we decided to implement both the assemble and link phases directly within `mcc`. Hence `mcc` may transform a *μCryptol* program all the way to a binary file ready to be loaded by the *AAMP7* monitor into a freshly reset microprocessor. For simplicity, we only generate binary images which use a single *AAMP7* process partition.

For verification, the compiler may also emit the final binary *AAMP7* program as a set of *ACL2* "proof events". These events place an *ACL2* model of the *AAMP7* in a state mimicking the real *AAMP7* about to come out of the reset state and begin execution.

6 Implementation experience

We implemented `mcc` and supporting tools using the functional programming language *OCaml* [24]. It's difficult to overstate the usefulness of a higher-order functional language for symbolic programs such as compilers. As a small example, the stack rearrangement optimization mentioned in Section 5 was trivially programmed in a few dozen lines using higher-order parser combinators[25].

We also exploited the "camlp4" meta-programming facility of *OCaml* to simplify the compiler in three ways. Firstly, we extended *OCaml* with support for *monadic expressions*[26] (which we based on those of *Haskell* [19]) using a very simple syntax extension. This helped neatly structure the more complicated parts of the compiler, such as type completion. Secondly, we embedded *Common Lisp* as a quotable sub-language within *OCaml*. This embedding was tremendously useful when programming the shallow embedding of *μCryptol* into *ACL2* [8] (*ACL2* stands for "A computational logic for applicative *Common Lisp*"), as it allowed *Lisp* templates to be written directly within `mcc`'s source code. Finally, and most ambitiously, we embedded *μCryptol* itself as a quotable language within *OCaml*. This embedding made the code

for source-to-source transformations (the heart of the compiler) both concise and readable, as both the left-hand side pattern to rewrite, and the right-hand side replacement for each transformation rule could be written directly in *μCryptol* rather than in the "representation of *μCryptol* abstract syntax as *OCaml* data constructor expressions". The compiler came in at around 43,000 lines, which we think is much smaller than if the camlp4 facility were not available. We heartily recommend *OCaml* to all compiler writers.

7 Related work

Our design for *μCryptol* of course took *Cryptol* [5] as its starting point. *Cryptol*, in turn, drew from three areas of prior work:

- The "look and feel" of the language overall is heavily influenced by Haskell[19].
- The use of width information in the type system is based on work done in the functional programming community on sized types[27–29]. However, since we do not have to deal with arbitrary user-defined recursive data types, but rather a single built in data type, our situation is much simpler.
- The approach to modelling streams borrows from the hardware modeling languages Lava[12], the synchronous dataflow language Signal[30], and work on hardware synthesis from functional languages[31].

μCryptol's type completion algorithm adapts the local type inference algorithms developed for languages with expressive subtyping[16, 17]. Our approach to the compilation of iterative streams exploits the stylized form of stream definitions[32], and thus avoids the complexities of the general recursion-to-iteration transformation[33].

8 Conclusions and future work

To summarize, *μCryptol* is a strongly typed, pure first-order functional language in which the only recursion is in the form of streams. The type system supports booleans, tuples, vectors, streams and functions, as well as arithmetic on natural numbers. A rich set of polymorphic primitives are provided for modular and binary polynomial arithmetic and bitwise logical operators on words of arbitrary bit width, and for manipulating vectors and streams. All user value definitions are monomorphic. The type system rejects ill-defined streams, hence all programs yield a result of the appropriate type. The burden of type annotations is eased by a type completion algorithm.

We have used the *μCryptol* compiler, `mcc`, to compile AES[2] to *AAMP7* code, and have successfully run the result on both an *AAMP7* simulator and evaluation board. Though the generated code could do with more optimization, it is at least imperative, uses no dynamic memory allocation, and does not require excessive temporary space.

There are three areas of future work. The most obvious is to continue to improve the generated code of `mcc`. Many deeper optimizations, such as function inlining, vector fusion, and stream fusion remain to be implemented. Furthermore, the code generation for stream iterators could also be further specialized to handle simple cases (such as a singleton history vector) without the overhead needed to support the general case (such as modular indexing). Generated code could also be improved by allowing more values to have unboxed representations (such as small tuples), and mapping small *μCryptol* words to small machine words (such as 8- and 16-bit machine words instead of just 32-bit). Beyond this, we speculate that a form of representation analysis[34] could reduce the time spent copying temporary objects into their final containing object.

The second area is to widen the scope of *μCryptol* to cover asymmetric-key cryptographic algorithms, which would involve the addition of primitives for groups such as Elliptic Curves over various finite fields[35], and prime fields[36]. This development has been carried out for *Cryptol*, and it would be straightforward to carry this work over to *μCryptol*.

The third, and most critical area of future work is again to widen the scope of *μCryptol* from algorithm to *cryptographic equipment specification*. In the military cryptography community, "equipment" refers to the entire cryptographic device, including:

1. The protocols used for interacting with the user (which can be as simple as the insertion of a crypto-ignition key, or require a built-in display and keyboard).
2. The protocols used for key establishment, rekeying and key zeroing (for example, IKE[37]).
3. The protocols used to encode data and voice traffic, possibly with error detection and correction (for example, Project 25 public safety digital radio[38]).

4. The protocols used to establish and maintain an encrypted tunnel (for example, IPSec[39]).
5. The protocols used to transmit packets over a medium (for example, modulation of a radio carrier).

The collection of these protocols is often called a "waveform", by a hangover from the early days of analog in-line voice encrypters. The above list represents a daunting spread of functionality, and it is likely to require a family of inter-operating languages rather than a single general-purpose language. However, the costs outlined in the introduction are really more a consequence of the complexity of the equipment specification problem rather than just the cryptographic algorithm specification. Hence our vision can only be fully realized by taking on the equipment problem.

9 Acknowledgments

The author thanks John Matthews and Lee Pike for their work on the automated correspondence aspects of the project, Jeff Lewis for his work on the original *Cryptol* design and implementation, Sigbjorn Finne for his work on the original *Cryptol* to *C* compiler, Dave Hardin of Rockwell Collins for his help with understanding the constraints of embedded military cryptography and the *AAMP7*, and Fergus Henderson for his help understanding hardware synthesis from *Cryptol* specifications.

This work was performed (under subcontract SC-307-04 with Rockwell Collins) under contract MDA904-03-C-1778 with the Department of Defense.

References

1. Federal Information Processing Standards Publication: Specifications for data encryption standard. Technical Report 46-2, National Institute of Standards and Technology (1993) Available at `http://www.itl.nist.gov/fipspubs/fip46-2.htm`.
2. Federal Information Processing Standards Publication: Specification for the advanced encryption standard (AES). Technical Report 197, National Institute of Standards and Technology (2001) Available at `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.
3. Daemen, J., Rijmen, V.: The Design of Rijndael: AES—The Advanced Encryption Standard. Springer (2002)
4. Rivest, R.L.: The RC5 encryption algorithm. In Preneel, B., ed.: Proceedings of the 1994 Workshop on Fast Software Encryption (FSE), Belgium. Volume 1008 of LNCS., Springer (1995) 86–96 Revised 1997. Available at `http://theory.lcs.mit.edu/~rivest/Rivest-rc5rev.pdf`.
5. Lewis, J.R., Martin, W.B.: Cryptol: High assurance, retargetable crypto development and validation. In: Proceedings of the IEEE/AFCEA Conference on Military Communications (MILCOM), Boston, MA. (2003) Available at `http://www.galois.com/files/milcom.pdf`.
6. Best, D., Kress, C., Mykris, N., Russell, J., Smith, W.: An advanced-architecture CMOS/SOS microprocessor. IEEE Micro (1982) 11–26
7. Rockwell Collins, Inc.: AAMP7r1 reference manual. Available on request (2003)
8. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers (2000) ISBN 0792377443.
9. Shields, M.B.: μCryptol reference manual. Technical report, Galois Connections, Inc. (2005) Available at `http://www.galois.com/files/mCryptol_refman-0.9.pdf`.
10. Shields, M.B.: CrAM reference manual. Technical report, Galois Connections, Inc. (2005) Available at `http://www.galois.com/files/CrAM_refman-0.9.pdf`.
11. Wheeler, D.J., Needham, R.M.: TEA, a tiny encryption algorithm. In Preneel, B., ed.: Proceedings of the 1994 Workshop on Fast Software Encryption (FSE), Belgium. Volume 1008 of LNCS., Springer (1995) 363–366 Available at `http://www.cl.cam.ac.uk/ftp/users/djw3/tea.ps`.
12. Bjesse, P., Claessen, K., Sheeran, M., Singh, S.: Lava: Hardware design in Haskell. In: Proceedings of the 1998 International Conference on Functional Programming (ICFP), Baltimore, MA, ACM Press (1999) 174–184 Available at `http://www.cs.chalmers.se/~bjesse/lava.ps`.
13. Sheeran, M.: Designing regular array architectures using higher order functions. In Jouannaud, J.P., ed.: Proceedings of the International Conference on Functional Programming Languages and Computer Architecture (FPCA), Nancy, France. Volume 201 of Lecture Notes in Computer Science., Springer (1985) 220–237
14. Jones, G., Sheeran, M.: The study of butterflies. In Britwistle, G.M., ed.: Proceedings of the IVth Banff Higher Order Workshop, Alberta, Canada. (1990)
15. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad hoc. In: Conference record of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), Austin, TX, ACM Press (1989) 60–76 Available at `http://homepages.inf.ed.ac.uk/wadler/papers/class/class.ps`.

16. Pierce, B.C., Turner, D.N.: Local type inference. In: Conference record of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), San Diego, CA, ACM Press (1998) 252–265 Available at http://www.cis.upenn.edu/~bcpierce/papers/lti-popl.ps.
17. Odersky, M., Zenger, C., Zenger, M.: Colored local type inference. In: Conference record of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), London, UK, ACM Press (2001) 41–53 Available at http://lampwww.epfl.ch/papers/clti-colored.ps.gz.
18. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: Conference record of the Ninth ACM Symposium on Principles of Programming Languages (POPL), Albuquerque, NM, ACM Press (1982) 207–212
19. Peyton-Jones, S.L., ed.: Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press (2003) ISBN 0521826144. Available at http://www.haskell.org/definition/haskell98-report.ps.gz.
20. Peyton Jones, S.L.: Implementing lazy functional languages on stock hardware: The Spineless Tagless G-machine. Journal of Functional Programming **2**(2) (1992) 127–202 Available at http://research.microsoft.com/users/simonpj/papers/spineless-tagless-gmachine.ps.gz.
21. Barendregt, H.P.: The Lambda Calculus. revised edn. Number 103 in Studies in Logic and the Foundations of Mathematics. North-Holland (1991) ISBN 0444875085.
22. Garey, M.R., Johnson, D.S.: The complexity of near-optimal graph coloring. Journal of the ACM (JACM) **23**(1) (1976) 43–49
23. Greve, D., Richards, R., Wilding, M.: A summary of intrinsic partitioning verification. In: In Proceedings of the Fifth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2), Austin, TX. (2004) Available at http://www.cs.utexas.edu/users/moore/acl2/workshop-2004/contrib/greve-richards-wilding/acl2-paper.pdf.
24. Leroy, X., Doligez, D., Garrigue, J., Rémy, D., Vouillon, J.: The Objective Caml system: Documentation and user's manual. Available at http://caml.inria.fr/pub/docs/manual-ocaml/index.html (2005)
25. Hutton, G., Meijer, E.: Functional pearl: Monadic parsing in Haskell. Journal of Functional Programming **8**(4) (1998) 437–444 Available at http://www.cs.nott.ac.uk/~gmh/pearl.pdf.
26. Wadler, P.: Comprehending monads. Mathematical Structures in Computer Science, Special issue of selected papers from the 6th Conference on Lisp and Functional Programming **2** (1992) 461–493 Available at http://homepages.inf.ed.ac.uk/wadler/papers/monads/monads.ps.
27. Hughes, J., Pareto, L., Sabry, A.: Proving the correctness of reactive systems using sized types. In: Conference record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, FL, ACM Press (1996) 410–423 Available at http://www.md.chalmers.se/~rjmh/Papers/popl-96.ps.
28. Hughes, J., Pareto, L.: Recursion and dynamic data-structures in bounded space: Towards embedded ML programming. In: Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP), ACM Press (1999) 70–81 Available at http://www.md.chalmers.se/~pareto/icfp99.ps.
29. Xi, H.: Dependently typed data structures. In: Proceedings of the Workshop on Algorithmic Aspects of Advanced Programming Languages (WAAAPL), Paris, France. (1999) 17–32 Available at http://www.cs.bu.edu/~hwxi/academic/papers/waaapl99.pdf.
30. Bournai, P., Chéron, B., Gautier, T., Houssais, B., Guernic, P.L.: SIGNAL manual. Technical Report PI-745, Institute de Recherche en Informatique et Systèmes Aléatoires (IRISA), France (1993) Available at ftp://ftp.irisa.fr/techreports/1993/PI-745.ps.gz.
31. Frankau, S., Mycroft, A.: Stream processing hardware from functional language specifications. In: Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS), Big Island, HI, IEEE (2003) 278 Available at http://www.cl.cam.ac.uk/~am/papers/hicss_2003.pdf.
32. Halbwachs, N., Raymond, P., Ratel, C.: Generating efficient code from data-flow programs. In Maluszynski, J., Wirsing, M., eds.: Proceedings of the Third International Symposium on Programming Language Implementation and Logic Programming (PLILP), Passau, Germany. Volume 528 of Lecture notes in Computer Science., Springer (1991) 207–218 Available at http://www-verimag.imag.fr/PEOPLE/Nicolas.Halbwachs/PS/plilp.ps.gz.
33. Liu, Y.A., Stoller, S.D.: Program optimization using indexed and recursive data structures. In Norris, C., Fenwick, J.J.B., eds.: Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM), ACM Press (2002) 108–118 Available at http://www.cs.sunysb.edu/~stoller/PEPM02.html.
34. Shao, Z.: Flexible representation analysis. In: Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP), Amsterdam, The Netherlands, ACM Press (1997) 85–98 Available at http://flint.cs.yale.edu/flint/publications/flex.pdf.

35. Hankerson, D., Menezes, A., Vanstone, S.: Guide to Elliptic Curve Cryptography. Springer-Verlag, New York (2004) ISBN 038795273X.
36. RSA Laboratories: PKCS #1: RSA encryption standard (version 2.1) (2002) Available at ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf.
37. Kaufman, C.: Internet key exchange (IKEv2) protocol (2005) RFC 4306. Available at http://www.ietf.org/rfc/rfc4306.txt.
38. Telecommunications Industry Association: Project 25: Standards for public safety radio communications (2006) Web page at http://www.tiaonline.org/standards/project_25.
39. Kent, S., Atkinson, R.: Security architecture for the internet protocol (1998) RFC 2401. Available at http://www.ietf.org/rfc/rfc2401.txt.