CHALMERS

Case Studies in Cryptol

A study of domain specific languages for DSP algorithms

Master of Science Thesis in Computer Science and Engineering

MUHAMMAD ASIF

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Göteborg, Sweden, November 2009

Case Studies in Cryptol
A study of domain specific languages for DSP algorithms

MUHAMMAD ASIF

Examiner: PATRIK JANSSON

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden October 2009

Abstract

Digital Signal Processing (DSP) has become part of many electronic applications these days; confluent to this, Domain Specific Languages (DSLs) are prevailing among practitioners of many engineering domains. Cryptol is a domain specific language for cryptography, it was designed by Galois and it has been used by NSA from the start. Cryptol is designed for cryptographic specifications; however, this project aims at evaluating Cryptol as a domain specific language for digital signal processing algorithms. The report also includes a proposal for extensions to Cryptol to make it applicable to DSP algorithms. This thesis derives its motivation from the DSL for DSP research that the Functional Programming group has started with Ericsson. This project involved implementing a set of DSP algorithms in Cryptol and analyzing applicability of Cryptol from the experiences in this new domain. This study shows that Cryptol is too specialised to cryptographic algorithms and that it is not possible to specify, run or verify many DSP algorithms in Cryptol. However there is a special class of DSP algorithms that are far easier to code in Cryptol than in C or Java. Small overlap with DSP algorithms was expected since DSLs are customized for a certain problem area. The evaluation of Cryptol in the DSP domain revealed that some enhancements are necessary to make it applicable to DSP algorithms. The outcome of this study is a set of extensions to Cryptol which are discussed towards the end of this report.

Keywords: Cryptol, Galois, NSA, domain specific language (DSL), digital signal processing (DSP).

Foreword

This document is a masters thesis report in Functional programming written for Chalmers University of Technology. This thesis was done by Muhammad Asif at the Department of Computer Science and Engineering.

The project has its foundation in the functional programming group as Mary Sheeran from this group proposed the idea of this project. The purpose was to investigate the possibility of using Cryptol as a domain specific language for DSP algorithms. A number of case studies were conducted to evaluate Cryptol as a tool for DSP algorithms and experiences from the case studies were used to propose some extensions to Cryptol.

The project was accomplished under the supervision of Joel Svensson while Patrik Jansson was examiner. I would like to thank Joel for his valuable guidance and I am grateful to Patrik for helping me take the right decisions during the course of the project. I am pleased to mention Mary Sheeran who provided me an opportunity to perform this thesis work. I appreciate Emil Axelsson for sharing his research and providing valuable case studies. I also express my gratitude to Levent Erkök from Galois who provided answers to plenty of my questions regarding the specific details of Cryptol.

Table of Contents

Chapter 1

Introduction

Digital signal processing has become a vital aspect of every digital system that interfaces with the physical world. Digital signal processing dates back to the 1960s but it was then not so prevalent because of the lack of powerful computers. During the 1980s the invention of dedicated and customized number crunching hardware helped this technology to proliferate [5]. This new hardware was named Digital Signal Processor; it was more specialized and powerful than a general purpose processor. Since then, DSPs have replaced analog circuits with a reconfigurable and robust piece of hardware in many of the electronics systems around us. Reconfiguration in DSPs comes from their programmability, which requires effective software development for this platform. This thesis investigates an approach for easier, efficient and correct software development in this domain.

Domain specific languages are not a recent invention; they are part of every field that involves programming either as a language or as a graphical tool. Domain specific languages are designed keeping in view a certain domain and include vocabulary which is more consistent with a domain expert's vocabulary. Using a DSL can reduce the learning effort and provide a single platform for specification and verification of DSP algorithms. DSLs are often created in an ad-hoc fashion, often leading to high development costs and implementations of varying quality but they are fruitful in the sense that they increase the productivity as well as reduce the effort to map a certain domain to a general purpose language [18,19].

Cryptol is a domain specific language for cryptography. It was designed by Galois Inc. and it is a language of choice for cryptographers at NSA [10]. Cryptol comes with a complete toolset which can be used to specify a cryptographic algorithm, formally verify the specification and generate hardware or software from the specification. Inspiration for this project is to explore how suitable Cryptol is for specification and verification of DSP algorithms. This problem is interesting since it involves study of a domain specific language, learning how to state requirements of a DSL and exploring characteristics of DSP algorithms. For Galois it is of interest because this may extend the application of Cryptol to DSP which is a flourishing and attractive area to explore.

This thesis involves analyzing Cryptol and its toolset for suitability as a DSL for DSP algorithms. This research requires knowledge of Cryptol as well as mature understanding of DSP algorithms. A DSL for DSP in the first place should be coherent with the vocabulary of a DSP domain expert; secondly, modern DSPs generally include hardware to provide certain tasks such as floating point arithmetic, CRC, multiply-accumulate etc; the generated code should also use this hardware efficiently. This report can serve as a preliminary analysis for any research related to the design of a domain specific language for the DSP domain. This report outlines the selected approach for evaluation of Cryptol as a domain specific language for DSP and gives details of the procedure. It also discusses the results of the case studies and finally it concludes with a proposal for necessary extensions to Cryptol.

1.1 Research Goal and Problem Statement

The purpose of this research is to perform some case studies in Cryptol involving implementation of DSP algorithms in Cryptol and derive judgments from the case studies. The case studies can provide good reasons and motivation for decisions about whether Cryptol can be used for implementing DSP algorithms or not. This research can prove fruitful for Galois in extending Cryptol for DSP domains.

This research is based on the question that: "Is it possible to use Cryptol as a DSL for DSP?" This question is important because it is interesting to find merits of an existing DSL which can likely serve as a DSL for DSP.

1.2 Limitations

As stated earlier the project derives its inspiration from the DSL for DSP project which Chalmers has started with Ericsson. This study is related to but independent from that project. That project is targeted at inventing a new domain specific language for DSP. Instead, the idea here is to measure capabilities of Cryptol in the DSP domain. This study is more focused towards evaluating Cryptol as a domain specific language for the DSP algorithms.

1.3 Organisation of the Report

The first few weeks of this project were dedicated to the study of Cryptol and its toolset as well as practicing it with some medium sized cryptographic algorithms. I started studying DSP algorithms after having enough understanding of Cryptol. In the beginning there were some meetings with my supervisor regarding the initial findings about Cryptol and its features. The result of this work is presented in this chapter and the next chapter. Chapter 3 outlines the approach that has been used to analyze Cryptol for DSP algorithms. It delineates the set of case studies selected and their brief description. In the later chapters, implementation details of various categories of DSP algorithms are discussed. Chapter 4 covers digital filters which are the most commonly used DSP algorithms; it starts with implementing very simple filters and then discusses a few complex ones. Chapter 5 contains spectrum analysis algorithms which are also quite common category of DSP algorithms; it outlines how DFT and its improved version FFT can be modeled and specified in Cryptol. Chapter 6 includes channel coding and digital modulation algorithms while Chapter 7 is about matrix and vector operations in Cryptol. In Chapter 8 the cumulative result of the case studies done in previous chapters is presented and extensions are suggested to make Cryptol suitable for DSP algorithms. Report ends with chapter 9 which consists of related work and the conclusion of the work.

Word	*Definition*
Galois	A company in USA that has developed many domain specific languages, it also deals with formal verification and functional languages.
DSP	Digital signal processing. It deals with converting an analog signal to digital, applying some mathematical operation on the digital signal and then converting the resulting digital signal back to analog form.
VHDL	VHSIC (Very High Speed Integrated Circuits) Hardware Description Language. It is the most widely used domain specific language to specify digital circuits.
Haskell	A popular functional programming language [26].
NSA	National Security Agency. It is an intelligence agency of USA. It is responsible for the collection and analysis of foreign communications and foreign signals intelligence, which involves cryptanalysis. It is also responsible for protecting U.S. government communications and information systems from similar agencies elsewhere, which involves cryptography.
DSL	Domain Specific Language. It is a special purpose programming language targeted to be used in a specific problem domain.

Chapter 2

Background

This section forms a background of Domain Specific Languages, outlines features of Cryptol and introduces Digital Signal Processing.

2.1 Domain Specific Languages

A domain specific language is one that is customized to meet the requirements of a particular problem domain. Unlike DSLs, general purpose languages (GPLs) are not designed for a specific domain and it is often difficult to express concepts of a domain in a GPL. A DSL is specialized for one type of specifications and provides a small comprehensive syntax for users. DSLs allow specification of aspects quickly and easily by providing a more expressive and less complex language interface [4]. The idea behind DSLs is to make a simpler and easier platform for the domain experts so that they can free themselves from the details of any general purpose programming language and can specify and verify their implementations using a simplified set of abstractions [10]. DSLs are generally easier to learn and less ambiguous to understand even for non-programmers. Generally a domain expert is not a proficient programmer and therefore, use of a DSL to state dynamics of his domain sounds attractive. DSLs free users from inventing tricky ways to achieve a goal; they are rather equipped with well defined constructs to model a domain concept. Using DSLs specifications are written on a higher level of abstraction without knowledge of the underlying platform. Specifications written in DSLs are translated to some general purpose language using a tool called a DSL compiler; translated code is compiled to target architecture that actually realizes the goal. Specification writers need not know about the library calls or the target platform. They use a DSL to write their ideas in unambiguous, readable and verifiable format. This approach increases the productivity in two ways; first by letting domain expert codify their ideas easily and second by automatic code generation for the target platform.

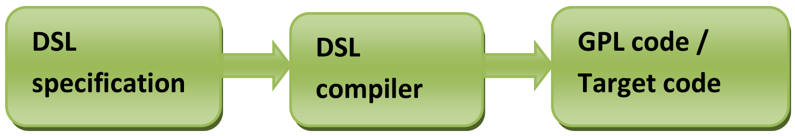

Figure 2.1: A Typical DSL Scheme

DSLs allow solutions to be expressed in the idiom and at the level of abstraction of the problem domain. Consequently, domain experts themselves can understand, validate, modify, and often even develop DSL programs. DSLs are small languages that make the programs compact and self-documenting due to simplicity of syntax. DSLs ensure software engineering principles like productivity, reliability, maintainability and portability. DSLs allow validation at the domain level. Examples of DSLs include SQL for database queries, LaTeX for document generation, VHDL for hardware description. Even Excel for spread sheets, BNF for grammars and the list goes on.

2.2 Cryptol

To make this report more self-contained, here is an introduction to Cryptol which is a summary of details in [1, 2].

2.2.1 Files and Modules

- A Cryptol program consists of a collection of definitions. This contrasts with a typical imperative language like C, which is a sequence of assignments. The order of definitions is unimportant, and there are no side effects of a definition. Avoiding side effects means that a specification can be understood by understanding each of its parts, without having to understand its context [1]. This characteristic makes the definition compositional i.e. we can divide a problem to small parts, solve the parts separately and solution to each part can be combined to form the solution of actual problem.

- There is no module structure of files in Cryptol; there can be any number of functions in a file. Functions cannot be grouped on the basis of what kind of operation they perform or what data they operate on; i.e. it is not possible to define classes.

- A Cryptol file can be included in another file to reuse the functionality.

- Files can have any extension but .cry is recommended. By default Cryptol searches for .cry files when a load command is issued in the interpreter.

2.2.2 Functions and Parameter Passing

- Cryptol is a functional language and functions are first class citizen. Functions can be passed or returned by other functions.

- Functions can be in-lined within other functions using the `where` clause, there can be any level of nested in-lined functions. These are also called local functions as they are only visible in the enclosing definition.

```
example x = result
    where { result = 2*x };
```

In contrast to Haskell the indentation of `where` clause does not matter; the braces signify the beginning and end of local definitions.

- There can be many functions in a file; they must all have unique names.
- It is possible to define recursive functions.

  For instance

  ```
  fun x = if x==0 then 0 else x+(fun (x-1))
  ```

- Functions are typed. See section 2.2.3 for a description.
- Higher order functions are supported. A function such as `highord1` takes one 32-bit number and returns a function of type `[32]->[32]`.

  ```
  highord1:[32]->([32]->[32]);
  highord1 x = (\y->x*y);
  ```

  while a function such as `highord2` takes a function of type `[32]->[32]` and gives a 32-bit number.

  ```
  highord2: ([32]->[32])->[32];
  highord2 f = f 100;
  ```

- In Haskell a function does not have any fixed number of arguments; For example, for a function `fun:Int->Int->Int` all of the following serve as a valid definitions:

  ```
  fun x = (\y->2*(x + y));

  fun = \(x->(\y->2*(x + y)));

  fun x y = 2*(x + y);
  ```

  But, in Cryptol a function can have either 0 or 1 argument and that is why third definition above is not valid in Cryptol. To avoid lambda expressions, arguments can be passed comma separated. For instance a function `fun` can also be defined with type `fun: ([8],[8])->[8]`.

  ```
  fun (x, y) = 2*(x+y);
  ```

2.2.3 Cryptol Type System

- The type system of Cryptol is the actual power for cryptography; the basic type is Bit which is either True or False. Numbers and characters are sequence of bits. A sequence of numbers or characters is also a sequence of bits. There are constructs that provide very convenient way to rearrange different sequence of bits to perform useful computations in cryptographic applications.

- The type of a number is the count of the number of bits in it. Types in Cryptol are written by juxtaposing square brackets [] containing a number representing size of sequence at that level for example the type [8] represents a literal consisting of 8-bits while the type [4][2] represents a sequence of size 4 each of whose elements is 2 bits wide. Note that an n-bit wide number has a type [n] and not [n]Bit since Bit is considered as implicit in every type.

  For instance the numeric literal 6 is of type [3] which means that it is a sequence of 3 bits, since 3 bits are enough to encode 6 in binary (110_2). Most significant bit of the number is the right most in the sequence.

- Sequences can also contain sequences for instance: `[2 3 4 5]` has a type `[4][3]` because it is a sequence of 4 numeric literals and 3 is the minimum number of bits required to store each of them. It is possible for a sequence to have any level of nested sequences but, sequences at each level of nesting must have the same type. For example: `[[2 3] [4 5]]` is a sequence each of whose elements is also a sequence.

- There are many views of a sequence. If we have a number of width 32-bits we can either see it as a sequence of 32 bits [32] or we can split it up into 4-bytes that will have a type [4][8] or we can split each of those 4 bytes to two 4-bit nibbles to get a value of type [4][2][4]. This block manipulation is very common in cryptographic algorithms.

- Strings are sequence of 8-bit characters. For example string “abc” has a type [3][8] because it is a sequence of three 8-bit characters.

- It is possible to define tuples. For instance, `(6,"abc",[2 3 4 5])` has a type `([3],[3][8],[4][3])`.

- In this report size and width refer to the same concept. In a type such as `[n]a`, `n` is the size or width and `a` is the shape of the sequence. The `width` primitive can be used to find the width of a sequence.

  ```
  Cryptol> width 32
  6
  ```

  Since 6 bits are suffice to store 32.

  ```
  Cryptol> width [17 2 32 14]
  4
  ```

  The sequence consists of 4 numbers.

  ```
  Cryptol> width [[1] [2] [3]]
  3
  ```

There are 3 elements in the sequence which are each a sequence of length 1.

- If a function is declared to accept a 16-bit number then no matter how small argument is provided 16 bits will be used to store it. But if the size of an argument is not constrained, the minimum number of bits required to code the literal will be used. For instance, 32 will have a type `(a>=6)=>[a]`. Type Constraints are given to the left of =>.

- Definitions in Cryptol can be given a type signature; signatures are optional but sometimes useful and even necessary. Functions can be given types restricting the size of inputs and outputs. If no type is given the type is inferred from the definition of the function and the minimum required sizes of inputs to the function.

  The following defines a 32-bit constant:

  ```
  x : [32];
  x = 13;
  ```

  Type variables are declared by wrapping the collection of them in braces in front of the type, as in the following example:

  ```
  f : {a} [a] -> [a];
  f  x = x;
  ```

  The type of `f` is read as: for all sizes `a`, `f` takes an `a` sized word and returns a word of the same size. A function that takes an 8-bit number and returns a number that is at least 32 bit wide can be coded as:

  ```
  test1 :{a} (a>=32) =>[8]->[a];
  ```

  Where `a` denotes the size (number of bits) of the output, `(a >=32)` is the type constraint; it is also possible to use `width` function to define the type constraint. >= is a valid operator but >, < or =< are not valid operators to appear in the type constraint.

  ```
  test2: {a b} [a][b]->[b];
  ```

  `test2` takes a list which is of length `a` and each element of the list is of width `b`, and returns a number which is `b` bits wide.

  If a function is not given any type then its type will be inferred. For example Cryptol infers from the definition that the type of following function is `([a]b,[a]b)->[a]b`.

  ```
  typelessfunc (x,y) = x+y;
  ```

  If `typelessfunc` is called with arguments 4 and 10 its inferred type will be `([4],[4])->[4]`. It happens because minimum of 4 bits are required to store 10 and the operator '+' requires both the inputs and the results must have the same sizes; therefore, the inferred size of 4 is [4] and also of the result.

- In a function of type `fun:{a}(a,a)->a,` `a` can stand for any Cryptol type. While for a function such as, `fun:{a}([a],[a])->[a]` `a` can only be a number.

- It is possible to add type annotations to expressions. For example, 2:[16] represents a 2 which is 16 bits wide; similarly in expression x+4, providing a type annotation such as this x+4:[10] can be used to force the width of x to be 10 bits. The size specified in the type annotation should be enough to store the literal. A type annotation for example, 4:[2] will result in an error because 3 bits is the minimum size to store 4.

- It is also possible to define records in Cryptol; records are a way to package related functions together. Records are first class citizens as well. Record types are nameless: they are identified only by their field names. Following is an example of a record type:

  ```
  {real:[32]; imag:[32]}
  ```

  The above record type represents a record having two elements: `real` and `imag`. Both of them are 32-bit numbers. Following are some examples of records:

  `{foo = True; bar = (False, True)}` is a record with two fields 'foo' and 'bar'.
  `{foo = True; bar : [2]; bar = 0}` here type annotations are also added to the field.
  `{foo = True; bar = {baz = False}}` is an example of nested record

  Field names should be unique in a record but nested records may use a field name that has been used in an outer record.

  Here is how records fields can be accessed:

  ```
  Cryptol> r.foo where r = {foo = True}
  True
  Cryptol> {foo = True}.foo
  True
  Cryptol> {foo = {bar = False}}.foo.bar
  False
  Cryptol> (f 4).foo where f x = {foo = x}
  0x4
  ```

- Records can be pattern matched in an argument to a function in various ways to access fields and nested records.

  Here is an example of how fields in a record can be accessed using projections:

  ```
  test : {a b c} (fin c) =>({dec: (a,b) -> c; enc: (a,c) -> b},a,c) -> Bit;
  ```

```
test (alg, key, pt) = alg.dec (key, alg.enc (key, pt)) == pt;
```

And here is the corresponding code using record patterns:

```
test : {a b c}(fin c) =>({dec: (a,b) -> c; enc: (a,c) -> b},a,c) -> Bit;
test ({dec = d; enc = e}, key, pt) = d (key, e (key, pt)) == pt;
```

- Record fields may also be polymorphic for example, in the following:

```
test : {b} {enc : {a} (a >= 2) => [a] -> b} -> [2]b;
test alg = [(alg.enc t1) (alg.enc t2)]
    where {
            t1 : [32]; t1 = 0;
            t2 : [64]; t2 = 0
            };
```

Here `enc` is a function which is polymorphic and it is provided as an argument to function test.

2.2.4 Features of Cryptol

- All arithmetic operations are also available to sequences (of any level of nesting) provided the operands are of exactly same sizes and shapes. For instance, '+' can add 2 numbers as well as corresponding elements of two sequences of numbers or a sequence of sequence of numbers. Hence point-wise matrix addition/subtraction becomes very easy.

- Cryptol is very fond of parentheses. Parentheses are necessary in most of places that causes the definitions to become a bit clumsy.

  In Haskell following serves as a correct definition but, in Cryptol `func y` should be parenthesized.

```
func x = \t->x*t;
test y = func y 2;
```

- Type synonyms are available to make it easy to name and reuse types.

- Individual element of a sequence or a group of elements of a sequence can be accessed via @, @@,! And !! operators.

  `[2 3 4 6]@2` gives element number 2.
  `[2 2 3 5 6 78 8]@@[2 5 1]` gives a sequence of element corresponding to indices 2,5 and 1.
  ! and !! do the same, but index 0 starts from the end of the sequence.

- Shift (<<,>>) and rotate (<<<,>>>) operators can be used to shift or rotate a sequence or a number.

For example:

`[2 3 4 4]<<2` will produce [4 4 0 0] while `[2 3 4 4]<<<2` will produce [4 4 2 3].

- "#" is used to append a sequence to another sequence.
- It is possible to transform a sequence of width n=k*j into a sequence of k sub-sequences of width j each.
  Operations `split`, `join`, `splitBy` and `groupBy` manipulate a sequence in different ways.

  `split` splits a list into sub-lists based on the sizes of specified sub-list. For example:

  ```
  (split [2 4 5 6 5 6]):[2][3][10]
  ```

  will produce `[[2 4 5] [6 5 6]]` which is a list of two lists each of whose elements are 10 bits wide.

  `join` concatenates sub-lists of a list. For example, `join [[2 3] [4 5]]` will produce `[2 3 4 5]`. While `join [1 0 0]` will produce 1, since in this case each sub-list is a literal which is 1 bit wide(since 0 and 1 require 1 bit).

  `splitBy` takes a number and a list and breaks the list into as many parts as specified by the number. But, the size of the list should be divisible by the number otherwise, it is an error.

  `groupBy` does a little different, it also takes a number and a list and breaks list into sub-lists of size specified by the number. It also requires that, the size of list should be divisible by the number provided.

  `tail` and `drop` are also available and they work just like in Haskell.

- Sequences can be finite or infinite. A function that requires an infinite sequence as input cannot accept a finite sequence and vice versa. Technically, a finite sequence does not unify to an infinite sequence.
- Sequences can be constructed using a construct similar to Haskell's list comprehension. For instance

  ```
  [|2*x||x<-[0 1 2 3] |]
  ```

  multiplies each element of [0 1 2 3] by 2. In contrast to Haskell, there are (|) at the start and end of list while (||) is used instead of (|) in Haskell.

  For instance

```
[|(x,y) ||x<-[0 1 2],y<-[3 4]|]
```

will produce

```
[(0,3) (0,4) (1,3) (1,4) (2,3) (2,4)]
```

this uses x and y in sequence i.e. each element of [0 1 2] will be paired with each element of [3 4]. The resulting list will have a size equal to the product of sizes of [0 1 2] and [3 4]. In contrast to this, the following:

```
[|(x,y) ||x<-[0 1 2] || y<-[3 4 5]|]
```

will produce

```
[(0,3) (1,4) (2,5)]
```

this uses x and y in parallel. That is, corresponding element of [0 1 2] will be paired with corresponding element of [3 4 5]. The resulting list will have the size of minimum of the list used in generator sequences.

- Numbers are represented in 2's complement form. For instance if -5 is represented in 4 bits it is coded as 0b1011. For this reason, if you type 0>(-1) in Cryptol interpreter it will return False since it first converts -1 to 2's complement and then compares with 0; in other words it ignores the signs while comparing. In order to use the correct sign of a number it is necessary to know the size of number. Because it is only possible to check the most significant bit (MSB) if the size of the number is known. If the MSB is true then the number is negative otherwise, positive or zero.

- The only conditional construct is if-then-else.

- There is a function called `zero.` It is a polymorphic function that can be used to generate zeros of arbitrary shape and size. For instance the following function:

  ```
  appzeros x = (x#zero):[20][8]
  ```

  takes a sequence and appends as many 8-bit 0s as necessary to make it of size 20.

- It is also possible to define a sequence of bits as a polynomial. Such as, <|x^3+x+1|> represents a bit sequence 1011 and thus, has a type [4]. Polynomials are used in many cryptographic applications. There are operations like `pmul`, `pdiv` and `pmod` that multiply, divide and find remainder of two polynomials.
- Similar to Haskell, anonymous functions[1] are possible to define using lambda expressions.

[1] An anonymous function is a function that is defined or called, without being bound to an identifier. In functional languages this is done using a lambda expression.

2.2.5 Limitations of Cryptol

DSLs are also called *little languages* [12] and they have their limitations and so has Cryptol. Interestingly, there are some features which are primitives in other functional languages but not available in Cryptol.

- There is no syntactic sugar for accommodating side effects.
- It is not possible to define custom operators.
- Infix notation is not supported.
- Lengths of all the sequences should be known at compile time. That is it is not possible to construct a list whose size is determined by a variable i.e. an argument to a function.
- `tail` and `drop` require that the length of resulting list must be known at compile time.
- It is difficult to code divide-and-conquer type of algorithms i.e. an algorithm which is defined in terms of the same algorithm applied on smaller size of any of its argument. The same is true if the size of the argument is increasing in recursive calls; for instance, you cannot build a list by accumulating elements to the list in recursive calls.
- Cryptol does not support sum types. That is, it is not possible for a type to have more than one constructor.
- Unlike in Haskell, functions and types cannot be grouped into modules and there is no way to control their visibility outside of the file that contains them. This limits the possibility of an abstract data type definition.
- It is sometimes required to use definitions written in some other language; in Cryptol this is possible only with VHDL. A component written in VHDL can be integrated and used with a Cryptol specification but this is possible only in VHDL mode. Cryptol modes will be discussed next.

2.2.6 Cryptol Modes

Cryptol comes with a number of tools that assist specification and verification. There is a Cryptol interpreter for type checking Cryptol files and evaluating functions. There are certain modes [2] in which the interpreter runs; when an expression is entered at the interpreter prompt, it is translated to the intermediate form associated with the current mode and then evaluated. Regardless of the mode all concrete syntax is first translated into an abstract syntax tree called the IR[2]. For some modes, the IR is their intermediate form, while other modes translate from the IR to another intermediate form [2] specific to that mode.

2.2.6.1 Cryptol Modes for Hardware Design

Bit

This mode performs interpretation on the IR; useful for prototyping circuits. It supports all features of the Cryptol language.

Word

The word mode in Cryptol models data as sequences of computer words. Instead of treating each bit independently as does the bit mode. It is not significantly faster than the bit mode in practice (especially for interactive development); but, it is sometimes useful because it has a smaller memory footprint on some programs.

Symbolic

Performs symbolic interpretation on the IR, useful for prototyping circuits, supports equivalence checking

Besides these, there are some other modes for hardware generation described in [2].

2.2.6.2 Cryptol Modes for Test Code Generation

C mode (C-backend)

In C mode, programs and expressions are translated to C. When a load[3] command is issued, the Cryptol file will be translated to C and compiled. Top-level expressions will be translated to C, linked with the currently loaded file, and then executed to produce the result. C mode can only compile monomorphic definitions; any function which is polymorphic in size of their arguments is ignored.

SBV mode

Cryptol also has an SBV (symbolic bit vector) mode where it uses the SBV compiler [3] to generate C, C++ and Haskell. This mode is used for generating C/C++/Haskell that can be verified using a SAT[4] or SMT[5] solver. SBV is an intermediate representation which is in static single assignment form and has only bit-vector operations.

SBV vs C-mode

C code generated in SBV mode is much faster than what is generated using the C mode of the interpreter. The approach in the SBV backend is different. In this mode, programs are first compiled into an intermediate form called SBV (symbolic bit-vector). The C-backend was mainly designed for integration with external projects, while SBV was designed for formal-verification using SMT solvers. The simplicity of the SBV representation allows Cryptol to

[2] Intermediate Representation

[3] Load command is used to load a Cryptol code file into interpreter. In the interpreter we write :load filename.

[4] Satisfiability solver

[5] Satisfiability Modulo Theories

generate really fast C code. But it comes at a cost: Only monomorphic, first-order, symbolically terminating, and finite functions can be translated in this way; so only a subset of Cryptol is supported. The other difference between C and SBV modes is that the code generated by SBV does not do memory allocation/de-allocation at run-time (as opposed to the C-mode); so it might be more suitable for embedded devices with limited memory sources.

2.2.7 An Example Cryptographic Algorithm

Having seen the features of Cryptol it is interesting to see an example of cryptographic algorithm in Cryptol. The following example of AES algorithm has been extracted from the Cryptol programming guide [2].

The AES algorithm is an iterated block cipher[6] with a block length of 128 and a variable key length defined to be one of 128, 192 or 256 bits. As per the AES specification, the definition of the algorithm is parameterized by two symbolic constants, Nb and Nk. These two parameters define the number of 32-bit words per block and number of 32-bit words per key respectively. The block size is fixed at 128, and we will set the key size to be 128 for the purposes of this example [2].

Next is an excerpt from the AES definition in Cryptol:

```
Nb = 128 / 32; // number of 32-bit words per block
Nk = 128 / 32; // number of 32-bit words per key
Nr = max(Nb, Nk) + 6;// number of rounds
```

Following is the top-level function `encrypt`, which takes two arguments: the round keys; called `RK`, and the plaintext to be encrypted; called `PT`.

```
encrypt (RK, PT) = unstripe (Rounds (State, RK))
        where
        {
                State : [4][Nb][8];
                State = stripe PT;
        };
```

In the code above sequence of transformation is done in following steps:

1. Transform the input plaintext into the State using the `stripe` function
2. Apply the `Rounds` function on the State and round keys to yield the final State
3. Transform this final state into an array of bytes using `unstripe`.

Definitions of `stripe` and `unstripe` are given below:

```
        stripe : [4*Nb][8] -> [4][Nb][8];
```

[6] An iterated block cipher applies a fixed round of computation to a block of data many times.

```
        stripe block = transpose (split block);

        unstripe : [4][Nb][8] -> [4*Nb][8];
        unstripe state = join (transpose state);
```

`Rounds` specifies the data flow within the cipher itself; following is a Cryptol definition of the `Rounds` function:

```
Rounds (State, (initialKey, rndKeys, finalKey)) = final
        where
        {
                istate = State ^ initialKey;
                rnds = [istate] # [| Round (state, key)
                                     || state <- rnds
                                     || key <- rndKeys |];
                final = FinalRound (last rnds, finalKey);
        };
```

The Cryptol listing above, implements iteration using list comprehension: it constructs a sequence of intermediate values, the first of which is the initial value, then one for each step of the iteration, and the last of which is the result of the iteration. The prelude for the AES cipher consists simply of XORing each byte of the incoming state with the initial key. In the definition above, the result of the XOR is called `istate`, because it is the initial state for the rounds iterations. The value `rnds` is defined as a recursive sequence. The first element of the sequence is the value of `istate`. The second element is the result of applying the `Round` function on the first element of the `rnds` sequence (i.e. `istate`) and the first element of the `rndKeys` sequence. The third element of `rnds` is the result of applying the `Round` function on the second element of `rnds` (which is the result of the previous iteration) and the second element of rndKeys. This pattern continues until we exhaust `rndKeys`, at which point the sequence `rnds` is complete and its last element is the result of iterating the `Round` function once for each element of `rndKeys`.
Definitions of `Round, rndKeys` and `FinalRound` are omitted to avoid digression.

2.3 Digital Signal Processing

2.3.1 Digital Signal Processor

Digital signal processing has become part of every major electronic system; it is concerned with converting analog signals into digital signals and manipulating the signals in many ways. The idea behind DSP is to use of advances in computer hardware and replace the analog circuit by a processor. Digital signal processing could either be done on a general purpose machine or on a specialized hardware[7] or a digital signal processor. A digital signal processor is specialized for DSP applications by the use of dedicated hardware and provision of instructions for complicated tasks. DSP architectures are designed to make execution of DSP algorithms efficient; refer to

[7] An Application Specific Integrated Circuit for DSP applications.

[17] for details. Digital signal processors run software as a general purpose computer does and can be programmed with their own instruction set. A DSP processor comes on an evaluation board that includes a number of IO ports to interface with the world and hardware units specialized for efficient execution of complex algorithms.

2.3.2 DSP Applications

Digital signal processing applications include biomedicine, sonar, radar, seismology, speech and music processing, imaging and communications. There are numerous devices that make use of digital signal processors; audio/video cards, digital cameras, fax machines, modems, base-stations, optical mice, cellular phones, high-capacity hard disks and digital TVs use a DSP in some form. Their ubiquity is because of lower cost than analog systems and their capability to perform operations that are not possible in the analog domain [15]. An estimate [13] by Texas Instruments[8] says that DSPs are used as the engine in 70% of the world's digital cellular phones, and with the increase in wireless applications this number will proliferate.

2.3.3 DSP Software

DSP software is used to program a DSP chip for an application. A certain digital signal processor has an instruction set which is fairly complex in comparison to a general purpose CPU’s instruction set. A DSP instruction can be rather large and might involve intensive computations. Programs targeted at a DSP are not often written in assembly language. Instead the programmer uses a higher level language to realize the algorithm: for example C, C++, BASIC, Java or MATLAB[9]. Algorithms written in the higher level language are compiled and linked with some specific libraries for the underlying DSP, producing an executable for the target DSP. Libraries are necessary to use IO routines and programmable hardware units. There are many benefits of using higher level languages in this domain. However, higher level languages are not a clear winner; because, sometimes it advantageous to code some functionality in assembly. Refer to [11] for a thorough discussion on this. DSP boards are quite sophisticated and come with accessories to assist the DSP software development. A modern DSP board is shipped with a complete development kit that includes a development environment consisting of drivers, simulator, debugger etc.

General purpose languages cannot easily express DSP algorithms; and few DSP programmers are experts of a general purpose language. Therefore, the idea of a DSL for DSP sounds appealing. A domain specific language for DSP is a platform where a DSP programmer can describe and test a DSP algorithm at a higher level of abstractions and translate the abstract specification to target DSP architecture. Another challenge, when programming DSP algorithms

[8] An American company based in Dallas, Texas, United States, renowned for developing and commercializing semiconductor and computer technology.

[9] A famous program for scientific modeling and simulation.

is to extract the optimization out of the architecture as much as possible which is achieved by optimizing the generated code for a certain DSP processor. SPIRAL[16] outlines an optimization technique that generates optimized code from higher level specifications of DSP algorithms.

2.3.4 DSP Algorithms - an overview

A DSP algorithm generally involves a numerical computation or formula specified symbolically or by a series of steps. Each step of an algorithm has a clearly defined meaning and it should have finite number of steps. However, many embedded system applications are wrapped in a 'while(1)' loop which runs forever but the algorithms inside these loops are finite [14]. Most of these algorithms also have analog counter-parts because digital signal processing is just an efficient and flexible replacement of analog signal processing. An analog circuit takes a continuous signal and produces a continuous signal while a DSP algorithm takes samples of an analog signal, performs some operations on them and produces output samples; which can be converted to analog signals. In this way digital processor remains indistinguishable from the outside world.

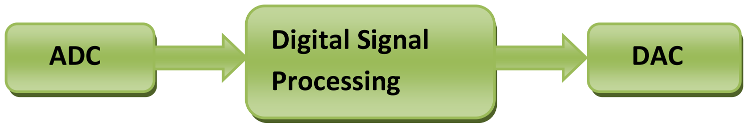

Fig 2.2: How DSPs Interact with the World

Most commonly used DSP algorithms fall in these categories; derived from [19].

- *Filtering*- Removing unwanted frequencies from a signal such as finite impulse response, infinite impulse response, moving average, autoregressive filters
- *Spectrum Analysis* - Determining the frequencies in a signal like Discrete Fourier Transform, Fast Fourier Transform.
- *Synthesis* - Generating complex signals such as speech Linear Predictive Coding.
- *System Identification* - Identifying the properties of a system by numerical analysis.
- *Compression* - Reducing the memory or bandwidth it takes to store a signal, such as audio or video such as MPEG[10], JPEG[11].

This was an overview of DSP algorithms. Now let's look a little deeper into DSP algorithms and their semantics.

[10] Moving Picture Expert Group, a video compression standard.
[11] Joint Photographic Expert Group, an efficient picture compression.

2.3.5 DSP Algorithms- digging deeper

DSP algorithms operate on samples of an analog signal: they get a finite sequence of such samples operate on them and produce resulting samples. Converting analog signals to samples does not influence the DSP algorithms at all. In fact DSP algorithms are mostly defined via a numerical relation between the input samples and the output samples.

Analog signals are continuous time and continuous values; for example a sine wave or a DC voltage etc. To get digital signals, analog signals are sampled at fixed intervals and then quantized; these fixed intervals divide the continuous time into intervals which is called discrete time. A digital signal has a value at each interval of time but its value remains fixed during two sampling intervals. A digital signal is represented as x[n] where "n" denotes the sample number, "n" could be 0, 1, 2…

Following are some simple DSP algorithms that have been taken from [5]; we study them to develop an understanding of inputs, outputs and semantics of such algorithms.

2.3.5.1 Unity Gain Filter:

A unity gain filter has following relation between input and output:

$$y[n] = x[n]$$

Each output sample y[n] is exactly the same as the corresponding input sample x[n].

$y[0] = x[0]$
$y[1] = x[1]$
$y[2] = x[2]$
……….......

It can be said that this filter has a gain of unity since every output sample is 1 times the input signal.

2.3.5.2 Pure Delay Filter:

A pure delay filter delays the input samples by 1 time interval. Following equation relates input and output of this filter:

$$y[n] = x[n-1]$$

It means that output value at time t = nh is simply the input at time t = (n-1)h, or the signal is delayed by time h: where h is sampling interval and n is the sample number.

$y[0] = x[-1]$

y[1] = x[0]
y[2] = x[1]
y[3] = x[2]
…………

Note that as sampling is assumed to commence at t = 0, the input value x[-1] at t = -h is undefined. Samples prior to t=0 are taken as 0 therefore in this case x[-1] is 0.

2.3.5.3 Two Term Average Filter:

y[n] = (x[n] + x[n-1]) / 2

The output is equal the average of the input signal at current and previous time intervals. Following are first few terms of this filter:

y[0] = (x[0] + x[-1]) /2
y[1] = (x[1] + x[0]) / 2
y[2] = (x[2] + x[1]) / 2
y[3] = (x[3] + x[2]) / 2
....

2.3.5.4 Discrete Cosine Transform:

Discrete cosine transform is used in many DSP applications. It has the following form:

$$Y[n] = \sum_{k=0}^{N-1} x[k]*\cos[n*pi*(k+1/2)/N] \quad \text{where } n = 0,1,\ldots,N-1$$

In the above formula, x and Y denote the input and output samples respectively. x is in time domain while Y is in frequency domain i.e. k denotes an interval in time while n is an interval in frequency. N is the total number of samples used to calculate DCT (Discrete Cosine Transform). This equation suggests that, for calculating one output sample all N samples from 0 to N-1 are multiplied with cosine of a number and summed.

Chapter 3

Analysis approach- DSP algorithms in Cryptol

This chapter describes the evaluation procedure that will be used in the following chapters to analyze Cryptol in the domain of DSP algorithms.

3.1 Evaluation Method

As stated in section 1.1 the goal of the project is to examine how effective Cryptol is for specification of DSP algorithms. Cryptol is renowned for its effectiveness in the Cryptographic domain but the problem at hand involves exploring the usefulness of this language in the DSP domain. One approach is to study the language and the domain algorithms and find out ways to map the domain operations to the language under evaluation. To evaluate Cryptol the same approach is used: we implement some DSP algorithms in Cryptol and derive results from ease or difficulty of implementations; this will shed light on capabilities of Cryptol as a language usable in DSP domain. This approach will conclude in either success when most of DSP algorithms can be specified using Cryptol or in a partial success when only some of them could be specified or in a failure when none of them could be specified. In case of success Cryptol should be amended to meet the requirements of DSP algorithms and in case of failure a new language with feature set tailored to DSP applications will be required. A partial success case is more interesting because, in this case Cryptol will require some extensions to become applicable to DSP programs.

3.2 Selection of DSP Algorithms

Next step in the analysis was to select DSP algorithms to be used in the evaluation. For this I and my supervisor met with Emil Axelsson, who is a postdoc at the Department of Computer Science and Engineering and working on a project for developing a domain specific language for digital signal processing. He helped us to prepare a list of commonly used DSP algorithms. We name those algorithms as “Case Studies” in this report. The case studies were mostly basic and they form building blocks of large DSP algorithms and applications. Targeting simple algorithms first was natural as well as effective since a simple algorithm is expected to uncover aspects of the language quickly and easily.

3.3 A Selection of Case Studies

Here is a list of case studies we have selected along with a brief description of where they are applied.

1. Filtering

   - FIR filter (Finite Impulse Response):
   - AR filter (Auto Regressive):
   - ARMA filters (Auto Regressive Moving Average):
   - IIR (Infinite Impulse Response)

   Filtering is used in many applications such as frequency selection, signal demodulation, removal of noise etc [6].

2. Spectrum analysis

   - FFT (Fast Fourier Transform)

   FFT is used in many DSP applications such as voice recognition system, biomedical signals etc [6].

3. Error correction and detection

   - Interleaving/De-interleaving
     A block of bits or samples are reordered to prevent burst errors. It is an important subsystem[12] in many communication systems such as CDMA (Code Division Multiple Access).
   - Convolution coding
     It is a popular channel coding techniques for detecting and correcting transmission errors.
   - Viterbi Trellis
     This performs the decoding of convolution coded data.
   - Cyclic Redundancy Check
     CRC is used to detect transmission errors in a block of data.

4. Modulation

   - BPSK (Bipolar Phase Shift Keying)
   - QPSK (Quadrature Phase Shift Keying)

[12] A communication system consists of a number of subsystems.

The two algorithms above are famous digital modulation techniques used in communication systems like CDMA.

5. Vector manipulation

- Applying a function to all elements of a vector.
- Applying a function to corresponding elements of two vectors.
- Calculating a result from all elements of a vector.

Vector operations enumerated above are simple but they are used frequently in larger DSP algorithms.

6. Matrix manipulation

- Determinant
- Matrix inverse
- Cholesky Factorization
- QR-decomposition

These operations are used in many digital signal processing applications so they are important indirectly.

7. Miscellaneous

- Spreading/De-spreading
  Spreading is to multiply a high speed digital signal with a slower one; it is used in DSSS (Direct Sequence Spread Spectrum) CDMA.
- Scrambling/De-scrambling
  Scrambling is like a stream cipher where one sequence of bits is multiplied with another stream which acts as a key stream to scramble the result. It is also used in CDMA forward and reverse link.
- Up-sampling/Down-sampling
  Up-sampling is to increase the sample rate of a sampled data by interpolating new samples between two samples, down sampling is skipping some of the samples to reduce the sample rate.
- Modulo 2 long division

3.4 Assumptions

In the following chapters a category-wise implementation of DSP algorithms is done in Cryptol. Almost all DSP algorithms required floating point computations but Cryptol does not support floating point operations. However, our interest is in abstract definition of algorithms; therefore, we assume that floating point operations are available. DSP algorithms also require many scientific functions and there is no library of those functions in Cryptol therefore, whenever we need them, we define them as a function that takes some arguments and returns a constant of correct type to make our algorithm type check successfully. After each implementation a set of limitations are also stated and even if it does not include floating point limitation described here it should be assumed implicit.

Cryptol can generate C or VHDL from the specification. However, for specification of DSP algorithms we shall not worry about whether it is possible to generate the code because, we want to evaluate Cryptol on the basis of how easy it makes the definition of DSP algorithms.

3.5 Some Handy Abstractions

The Case studies revealed many characteristics of DSP algorithms, based on this some commonly used patterns and types are coded in Cryptol for use as libraries. The Case studies were then re-implemented using these libraries.

3.5.1 Arithmetic Library

Most common operations in DSP algorithms involve arithmetic data type therefore; this library contains a template for specifying operations on an arithmetic data type (e.g. Complex numbers). The library can be edited to include operations for other arithmetic types (e.g. Float).

The Arithmetic Library is represented by a record type which serves as a template for any arithmetic type. Following is how it is defined in Cryptol:

```
type Arithmetic(a) =
                {
                    add:(a,a)->a;
                    sub:(a,a)->a;
                    mul:(a,a)->a;
                    div:(a,a)->a;
                    expi:a->a;            // exponentiation of type ‘a’
                    fromInt:[32]->a;     // conversion of a 32-bit integer
                                         // to type ‘a’
                    null:a;             // identity element w.r.t addition.
                };
```

An arithmetic type can be introduced by providing implementations for all the operations in this record type. One of the most useful types in DSP algorithms is complex number which is defined in the library as follows.

```
type Complex a= {real:a; imag:a};
```

It is a record having two components real and imaginary both of type `a`. Now we define a new type `ComplexType` which is a specialization of the Arithmetic type.

```
type ComplexType a = Arithmetic(Complex (a));
```

Now it is possible to define an implementation of Arithmetic template: described earlier for `ComplexType`. Following is an implementation for a complex number having real and imaginary parts as 32-bit integers. If +,-,* etc are defined for other data types, then [32] can be replaced with corresponding type name to invent the implementation for complex of that type.

```
ComplexArith:ComplexType([32]);
ComplexArith =
            {
                add (a,b) = {real=a.real+b.real;imag=a.imag+b.imag};
                sub (a,b) = {real=a.real-b.real;imag=a.imag-b.imag};
                mul (a,b) = {real=a.real*b.real-a.imag*b.imag;
                             imag=a.imag*b.real+a.real*b.imag};
                div (a,b) = undefined;       //omitted for simplicity
                expi n    = {real=1;imag=1}; //fake definition
                fromInt n = {real=n;imag=0};
                null = {real=0:[32];imag=0:[32]};
            };
```

Similarly we can define operations for other arithmetic types using this template. Int32Arith is another implementation of this library (but not included in this report) for 32-bit integers. Currently floating point numbers are not available in Cryptol but we assume that they can be implemented using the same template. This library will be used in DSP algorithms which are covered in the following chapters. These algorithms accept an extra argument of type `Arithmetic`, which has the set of operations for respective type of samples.

3.5.2 List Library

It was also found during the case studies that DSP algorithms most of the time work with lists. They often compute a value from the list; such as, transform a list to a different shape or apply a function on the list a certain number of times. Therefore, we generalized the commonly found patterns into this library. Following is a Cryptol specification and brief description of them.

3.5.2.1 Fold

This performs the function of Haskell's fold operation. That is, it takes a function, an initial value and a list; starting with the initial value, it accumulates the result of application of the function to successive elements of the list. Following is how it is done in Cryptol:

```
fold:{a b c}(fin a)=>(b->c->c,c,[a]b)->c;
fold (f,init,list) = run (0,init)
        where
        {
            N = width list;
            run (i,result) = if i<N then run (i+1,(f (list@i)) result)
                                      else result;

        };
```

3.5.2.2 Map

Map is similar to Haskell's map i.e. it takes a function and a list and applies the function on each element of the list. It can be done in Cryptol easily as follows.

```
map:{a b c}(b->c,[a]b)->[a]c;
map (f,input) = [|f v||v<-input|];
```

Similar to this is an index-map, it generalizes a pattern in which the function `f` is also dependent on the index of element to which it is applied.

```
imap:{a b c d}(fin d,d>=1)=>([(d)]->b->c,[a]b)->[a]c;
imap (f,input) = [|((f i) v) ||v<-input||i<-[0..]|];
```

3.5.2.3 Transform

This pattern takes a function and a value and applies the function given number of times to that value. For the case studies this pattern in used with a list type and a function from list to a list.

```
transform:{a b c}(fin c,c>=1)=>(a->a,a,[c])->a;
transform (f,input,times) = apply(0,input)
        where
        {
            apply (i,result) = if i<times then apply(i+1,f result) else result;
        };
```

A variant of this is a pattern in which the function 'f' is also dependent on the number of time it is applied.

```
itransform:{a b c}(fin c,c>=1)=>([c]->a->a,a,[c])->a;
itransform (f,input,times) = apply(0,input)
        where
        {
            apply (i,result) = if i<times then apply(i+1,((f i)result)) else result;
        };
```

3.5.2.4 Zipwith

This is corresponds to Haskell's `zipWith` which takes a function of two arguments and two lists and applies the function on corresponding elements of the lists.

```
zipwith:{m b c d n}(b->c->d,[m]b,[n]c)->[min(m,n)]d;
zipwith (f,xs,ys) = [|((f x) y)||x<-xs||y<-ys|];
```

3.5.2.5 Sum

This corresponds to haskell's sum, which sums a list. `sum` takes an arithmetic library of a certain type and a sequence and sums the sequence using the addition operation available in the library. Following is how it is defined in Cryptol:

```
sum:{a b}(fin b)=>(Arithmetic (a),[b]a)->a;
sum (lib,xs) = fold (\x->\y->lib.add(x,y),lib.null,xs);
```

3.5.2.6 Sumprod

This function takes two sequences, multiplies corresponding elements of the two sequences and sums the resulting sequence. For addition and multiplication it uses operations available in the library passed to it as an argument. Here is the Cryptol code for this:

```
sumprod:{a b c}(fin b,fin c)=>(Arithmetic(a),([b]a,[c]a))->a;
sumprod (lib,(xs,ys)) = fold(\m->\n->lib.add(m,n),lib.null,zipwith(\x->\y->lib.mul(x,y),xs,ys));
```

As visible from the type signature widths of the two sequences can be different, but the resulting list of products has a length equal to the smaller one (because of `zipwith`).

All of the above patterns will be used in performing the case studies described in the following chapters.

Chapter 4

Filtering Algorithms

The previous chapter presented a list of famous filtering algorithms; in this chapter a Cryptol implementation of some of them are presented. Some simple filtering algorithms are also discussed before going to more practical and a bit complex ones.

4.1 Modeling Filtering Algorithms in Cryptol

In order to model a digital filter in Cryptol it is required to look into how an algorithm is coded for DSP processor. Analysis of some of DSP algorithms [6] revealed that a DSP algorithm (coded in C), first retrieves a sample from an ADC on the DSP board operates on the sample produces an output sample and sends it to the output port of DSP board. To accomplish the operation it usually requires memory to remember previous input or/and output samples. A DSP algorithm often involves the same processing on every sample; this helps in modeling a DSP algorithm in Cryptol. It is not possible to declare a memory area in Cryptol but we can model the part of DSP algorithm that actually calculates the output sample.

A DSP algorithm can be modeled as a step function:

(state, input) → (newstate, output)

The state is not saved anywhere but it can be passed, modified and returned from a Cryptol function. In a DSP algorithm written in C such a memory is declared as a global array but in Cryptol this is not possible. Therefore, it is assumed that a driver function can be added that declares and initializes this memory and passes it back and forth to the C translation of this function.

4.2 Implementing Filtering Algorithms in Cryptol

We start with very simple filtering algorithms in Cryptol.

4.2.1 Simple Non-recursive Filters

4.2.1.1 Gain Filter

Equation relating input and output of this filter is:

$$y[n] = k*x[n]$$

where k is the filter gain. This can be specified in Cryptol in the following way:

```
K = 3; //just an example value
gfilter: [32]->[32];
gfilter xn = K*xn;
```

here `K` is the filter gain which could also be passed to `gfilter` as a parameter. No state information is needed in this implementation.

`gfilter` has been modeled as a function that takes a 32-bit number and produces a 32-bit number. The reason for using 32-bit numbers is to make sure that sufficiently large numbers can be used for gain and samples. This definition works fine as long as the result of multiplication of `K` and any of the samples does not exceed 32 bit. Otherwise, it will overflow and truncate the result to 32 bits.

In Cryptol "*" has the following type signature[13]

```
* :{a b} ([a]b,[a]b)->[a]b
```

which means it takes two sequences of same the width and shape and produces a resulting sequence of same width and shape. It however is true in other languages for example, the product of two integers is integer and product of two floats is float. But size inference causes problems here. If a number is not given a type annotation it will be considered as big as number of bits sufficient for storing it. The following definition reveals that problem:

```
gfilter':{b}([b],[b])->[b];
gfilter' (K,xn) = K*xn;
```

Here `gfilter′` takes gain as an argument. It is customary for `K` to have same type as `xn` otherwise it violates the type requirement of "*". But, the actual problem happens when this is used. If `gfilter′` is called with argument (2, 3) it gives 2, while the expected output was 6, this happens because of Cryptol size inference. When `gfilter′` is called with 2 and 3 a size is inferred for both 2 and 3 is 2 bits Now result of multiplication should also be 2 bits wide, this is 2*3=6(110_2) is 2 in two bits.

While in the previous version 32 bits were used to accommodate numbers as wide as 32 bits. But, it works as long as the result of the output does not exceed 32 bits. Same argument holds for the rest of filtering algorithms outlined in this chapter.

4.2.1.2 Delay Filter

This filter delays the input samples by one time unit as can be seen from the equation:

[13] Signature of * suggest that b can be anything including a record, a tuple or a function. However, if two lists of tuples are multiplied (using *) then it multiplies corresponding elements of the tuples. It compiles if it is used for multiplying two lists of records or functions but, then it gives runtime error. It seems to be a bug in Cryptol there should be some constraint on the shape of sequences * makes sense and can work with.

y[n] = x[n-1]

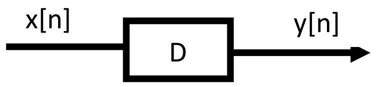

Figure 4.1: A Delay Filter

In the figure D represents a delay of one sample. Here is a Cryptol representation of the delay filter:

```
dfilter:([32],[32])->([32],[32]);
dfilter (s,xn) = (ns,yn)
        where
        {       yn = s;
                ns = xn;
        };
```

In the above Cryptol code, state is in argument `s` and input is `xn` while, `yn` is output sample and ns is new state; it simply outputs the previous sample and updates the state with new sample. For calculating the first output sample state `s` is initialized with 0 and for the subsequent samples the next state is used. This scheme for variable naming holds for all algorithms.

4.2.1.3 Difference Filter

Following is an equation of a difference filter is:

y[n] = x[n] – x[n-1]

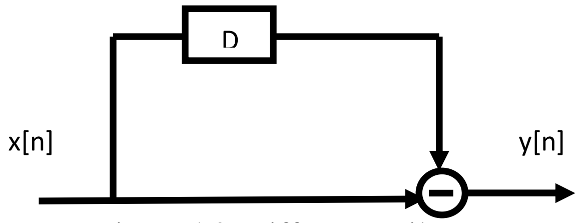

Figure 4.2: Difference Filter

i.e. output at each discrete time instance is difference of input between this and previous time instance. Following is a Cryptol representation of this:

```
diff_filter:([32],[32])->([32],[32]);
diff_filter (s,xn) = (ns,ys)
        where
        {
                yn = xn-s;
                ns = xn;
        };
```

This is quite similar to the delay filter; `diff_filter` takes previous sample and new sample as input and produces an output by taking a difference of new and old sample. Now, the new state is the new input sample.

4.2.1.4 Averaging Filters

An averaging filter takes average of a number (implementation dependent) of input samples to produce the output samples. Two term average filter has the following equation:

y[n] = (x[n]+x[n-1]) / 2

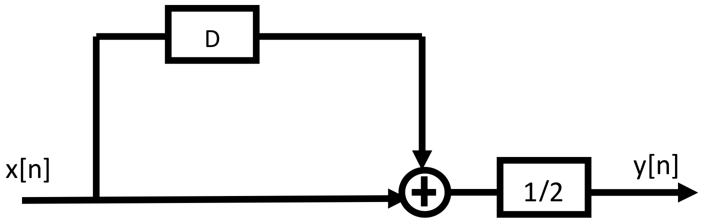

Fig. 4.3: An Averaging Filter

which can be specified in Cryptol as:

```
avg_filter:([32],[32])->([32],[32]);
avg_filter (s,xn) = (ns,yn)
        where
        {
                yn = (xn+s)/2;
                ns = xn;
        };
```

A three-term average filter averages last three samples to produce the next sample. It has following input output representation:

y[n] = (x[n]+x[n-1]+x[n-2])/3

A Cryptol specification corresponding to the above equation looks like this:

```
avg3_filter:([2][32],[32])->([32],[2][32]);
avg3_filter (s,xn) = (ns,ys)
        where
        {
                yn = (xn+(s@0)+(s@1))/3;
                ns = [xn]#(drop (1,s>>1));
        };
```

In the above case, state is represented by a list of size 2: since output depends on previous two samples and the current samples. Output is an average of state and new sample, while new state is calculated by shifting the previous state to right and replacing the first element by the new sample.

So far, filters only depend on the input sequence not on previous outputs. If output samples depends both on input and previous output samples; it is called a recursive filter.

4.2.2 Simple Recursive Filters

4.2.2.1 A Basic Recursive Filter

A simple recursive filter has the following equation:

y[n] = x[n] + y[n-1]

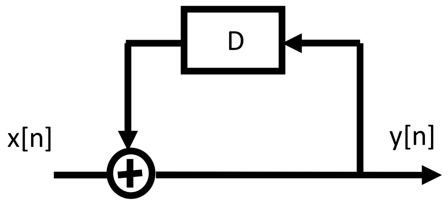

Fig. 4.4: A Basic Recursive Filter

in this filter output depends on input sample and only the last output sample.

```
recsimple:([32],[32])->([32],[32]);
recsimple (s,xn) = (ns,yn)
        where
        {
                yn = xn+s;
                ns = yn;
        };
```

In the implementation above, state consists of previous output, the next output is the sum of state and new sample while next state is the same as new output. For calculating the first output sample `s` should be `0`.

4.2.2.2 First Order Recursive Filter

A first order recursive filter has the following form

y[n] = (a[0]*x[n]+a[1]*x[n-1]-b[1]*y[n-1]) / b[0] where a[0],a[1],b[0] and b[1] are filter coefficients, these are used to adjust the response of filter. Coefficients are calculated based on the desired characteristics of the filter; however, values we have used are just examples. This filter can be coded in Cryptol like this:

```
record1: ([32],[32],[32])->([32],[32],[32]);
record1 (sin,sout,xn) = (sin',sout',yn)
        where
        {
                a0 = 12;
                a1 = 2;
                b0 = 1;
                b1 = 9;
                yn = (a0*xn+sin*a1-b1*sout)/b0;
                sin' = xn;
                sout' = yn;
        };
```

In this filter, output is not only a function of current and previous input but also of the previous output, which requires state to include that as well. Therefore, state consists of two arguments `sin` and `sout`. Next states are `sin'` and `sout'` while `yn` is output sample.

4.2.2.3. Second Order Recursive Filter

A second order recursive filter has the following form:

y[n] = (a[0]*x[n]+a[1]*x[n-1]+a[2]*x[n-2] – b[1]*y[n-1] – b[2]*y[n-2]) / b[0]

The above relation is implemented in Cryptol like this:

```
record2:([2][32],[2][32],[32])->([2][32],[2][32],[32]);
record2 (sin,sout,xn) = (sin',sout',yn)
        where
        {
                a0 = 2;
                a1 = 4;
                a2 = 5;
                b0 = 1;
                b1 = 53;
                b2 = 3;
                yn=(a0*xn+a1*(sin@0)+a2*(sin@1)-
                        b1*(sout@0)-b2*(sout@1))/b0;
                sin' = [xn]#(drop (1,sin>>1));
                sout' = [yn]#(drop (1,sout>>1));
        };
```

It is much like first order recursive filter except, the state consists of a sequence instead of a number.

Limitations
All of the above implementations work only with integer data.

Now we discuss some more practical filter algorithms. The Cryptol definition for them is also quite simple.

4.2.3 Finite Impulse Response (FIR) filter

FIR is a filter with no feedback in equation. It is used in applications where the system requires guaranteed stability or linear phase [14]; it is represented with the following formula:

y[n] = b[0]*x[n]+b[1]*x[n-1]+ …..+ b[N]*x[n-N]

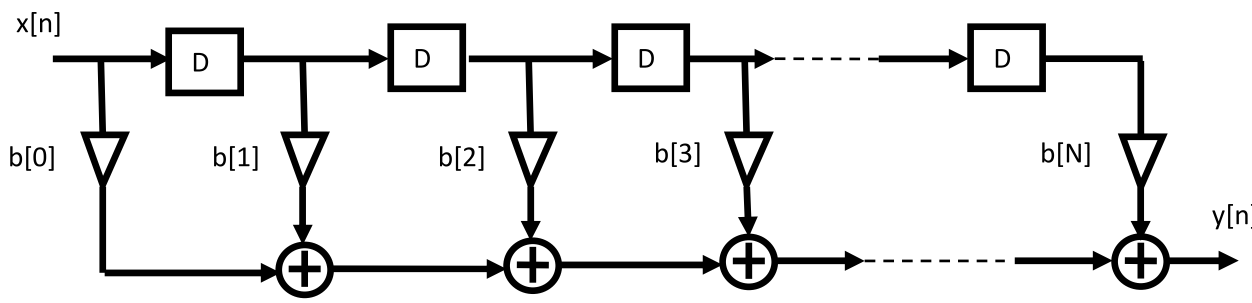

Fig. 4.5: FIR Filter

where N is order of the filter: order is the number of previous input samples required to calculate the next output sample. In the figure triangular blocks represent coefficients which are multiplied by delayed input signal. All the products are finally summed to produce the output. Following is a Cryptol version of this filter algorithm, which is derived from C implementation given in [6]:

```
N = 20;
fir: ([N][32],[32])->([N][32],[32]);
fir (s,xn) = (ns,yn)
        where
        {

                yn = sop;
                a = [1 2..20]; //example value only.
                ns = [xn]#(drop(1,s>>1));
                sop = sumprod(Int32Arith,(a,ns));
        };
```

Description of algorithm

- N is defined to be a constant which represents the number of taps in the filter.

- `a` stores the filter coefficients.
- `ns` is the new state.
- `sop` makes use of `sumprod` defined in list library (section 3.5.2). It multiplies corresponding elements of `a` and `ns` and sums the resulting list. Int32Arith[14] is the implementation of arithmetic library for 32-bit integers.
- `yn` is the output for the current discrete time instance.

4.2.4 Infinite Impulse Response (IIR) Filter

An IIR filter is a practical example of simplified recursive algorithms described before. An IIR is used in a number of signal processing applications. Fig. 4.6 shows a block diagram connecting input and output of this filter.

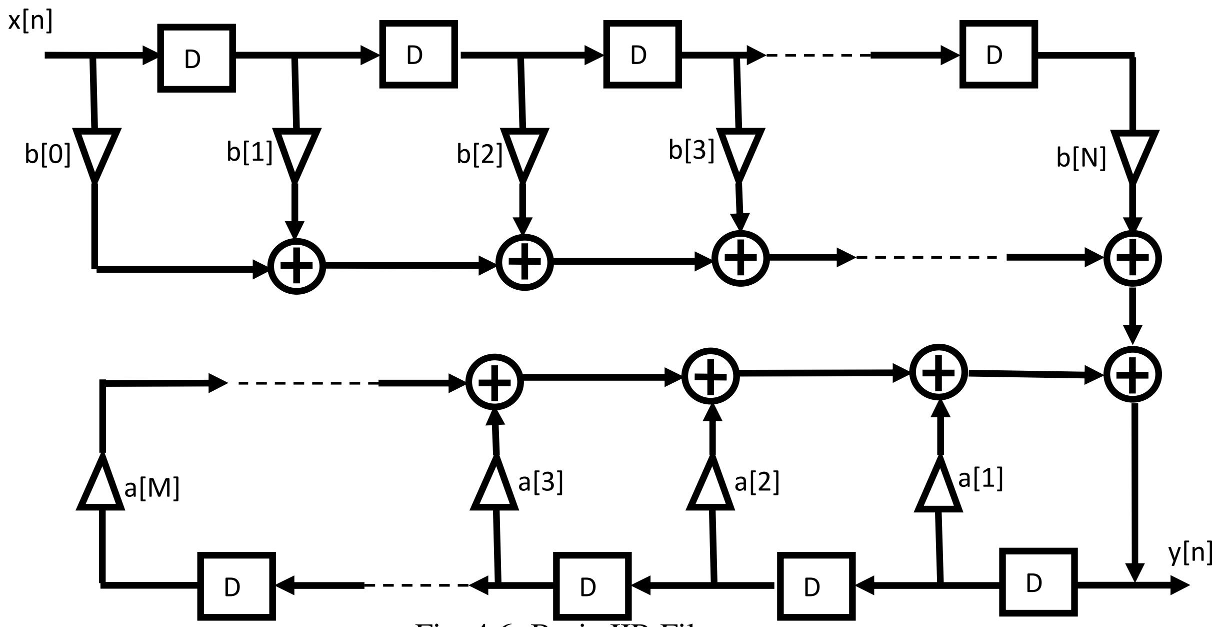

Fig. 4.6: Basic IIR Filter

Following formula relates input and output of this filter:

y[n]= [(b[0]*x[n]+b[1]*x[n-1]+b[2]*x[n-2]+…+b[M]*x[n-M]) –
(a[1]*y[n-1]+a[2]*y[n-2]+… +a[N]*a[n-N])]/ a[0]

This can be implemented in Cryptol using the following Cryptol code:

[14] Int32Arith is also defined in arithmetic library, definition is not included in the report.

```
N = 10;
M = 15;
iir :([N][32],[M][32],[32])->([N][32],[M][32],[32]);
iir (outputs,inputs,xn) = (outputs',inputs',yn/a@0)
       where
       {
              a = [1..N]; // example values
              b = [1..M]; //  example values
              sumout = sumprod(Int32Arith,(tail a ,outputs));
              sumin = sumprod(Int32Arith,(b,tail inputs'));
              yn = sumout-sumin;
              outputs' = [yn]#(drop (1,outputs>>1));
              inputs' = [xn]#(drop (1,inputs>>1));
       };
```

Description of algorithm

- `outputs` and `inputs` are input state and `outputs'` and `inputs'` are new state.
- `N` and `M` are assigned fixed values of 10 and 15 respectively.
- `a` and `b` are coefficients, given values are only for example and not calculated from MATLAB or otherwise.
- `sumout` calculates sum of products of `tail a` (because of only a[1] to a[n-N] in the formula) and the previous N outputs.
- `sumin` evaluates the sum of products of b and the previous M inputs.
- Difference of `sumout` and `sumin` is stored in `yn`.
- `yn/a@0` is the resulting output sample, because there is a division by a[0] in the formula.
- `inputs` and `outputs` constitute the new state.

All of the algorithms described in this chapter can easily be parameterized to accept an argument of arithmetic library of some type, and perform all the computations using the operations in that library.

Chapter 5

Spectrum Analysis

Spectrum analysis includes a frequency domain transform of a time domain signal. First we present a model of such an algorithm that can be mapped to Cryptol, and then we list and describe the Cryptol specifications for them.

5.1 Modeling Spectrum Analysis in Cryptol

Spectrum analysis algorithms transform a sequence of input samples and produce a sequence of output samples. To come up with a model, a few algorithms from [6] were studied for reference. In these algorithms a buffer accumulates a sequence of input samples and produces a sequence of output samples when a certain amount of samples get accumulated. The actual computation is done on a sequence of samples and the result is another sequence of samples. Therefore for such an algorithm a Cryptol definition should accept a sequence of samples and produce a sequence of samples. Such a function will have this as type:

```
[N]a -> [N]a
```

i.e. a function from a sequence of N samples of some arithmetic type a to N samples of the same type. Examples of such algorithms include DFT (Discrete Fourier Transform) DCT (Discrete Cosine Transform), FFT (Fast Fourier Transform) etc.

5.2 Spectrum Analysis in Cryptol

5.2.1 Discrete Fourier Transform in Cryptol

DFT has following relationship between input and output:

$$X[k] = \sum_{n=0}^{n=N-1} x[n]*\exp^{(-2*\pi*j*k*n)/N} \qquad \text{for } k = 0,\ldots.,N-1$$

where N is the number of samples on which the transformation is calculated; it is also called N-point DFT. x[n] is an input sample at discrete time n, X[k] denotes output sample (fourier transformed) at frequency interval k, exp is mathematical constant e(=2.7182..), j is √-1 (square root of -1) which is an imaginary number.

In the library we defined in section 3.5.1, there was a function called `expi` which computes exp (=2.71828…) raise to power any j (=√-1) times an arithmetic data type we shall use it in this algorithm. π being a floating point cannot be defined in Cryptol. To cater this, we assume the existence of a constant named `pi`. In our algorithms we give it an integer value to make our algorithms type check successfully.

Following is the Cryptol specification for the DFT algorithm, which is just a translation of the above DFT equation stated above.

```
pi = 3;
type size=64; // type constant, can only be used in the type signature.
type dft_type b =(Arithmetic(b),[size]b)->[size]b;

dft:{b}dft_type b;
dft (lib,xs) = [|summation (products k)||k<-[0..(N-1)]|]
        where
        {
                N = width xs;
                products k =imap (\n->\x->lib.mul(x,lib.expi (lib.fromInt (2*pi*k*n/N))),xs);
                        //usage of fromInt is a simplification because pi is not float in
                        //practice and division by N does not always yield an integer.
                summation xs = sum(lib,xs);
        };
```

<u>Description</u>

- `size` is defined as a type constant 64. i.e. it is a 64-point DFT.
- DFT may involve any type such as integer, floats or Complex therefore we give the algorithm a generalized type (parameterized with variable `b`) named `dft_type`. `dft_type` is a function from a pair of library of type `b` and sequence of type `b` to a sequence of type `b;` while, `b` stands for any arithmetic type.
- `add,mul` are addition and multiplication for type `b`.
- `expi` is a function in the library that takes an argument of type `a` and returns a value of type `a`; it has been assumed here for simplification otherwise, it produces a complex number for which we would have to define other operations.
- `summation` sums a sequence of type `b` and produces a result of type `b`. It uses `sum` defined in section 3.5.2.
- `products` multiplies corresponding elements of sequence of samples with a sequence generated by enumerating values of $e^{(-2*\pi*j*k*n)/N}$ when `n` runs from `0` to `N-1`.
- If we implement Arithmetic library for floating point numbers and name it `FloatArith`, then we can use following function to get a DFT that works with floating point numbers.

```
                    float_dft xs = dft(FloatArith,xs);
```

Limitations

- This implementation cannot be run because of unavailability of floating points and trigonometric functions in Cryptol and,
- It is not possible to verify this implementation because of the same reason.

5.2.2 Fast Fourier Transform in Cryptol

FFT is a fast implementation of DFT. There are many such implementations but Cooley-Tukey is the most common FFT algorithm. Cooley-Tukey breaks the problem of computing a DFT of size N=k*m to k DFTs of size m which can be computed efficiently. This is based on the Decimation-in-Time (DIT) algorithm, which arranges the DFT equation into two parts. One is sum over even indices of discrete time and the other over the odd indices of discrete time. It can be proved that transform X[k] of samples can be computed from two N/2-DFTs of even and odd indices; represented by $X_e[k]$ and $X_o[k]$ respectively. X[k] is a sum of $X_e[k]$ and $X_o[k]$ times a number called twiddle factor; represented as $W_N^k = e^{-(j2nk/N)}$. $X_o[k]$ and $X_e[k]$ are periodic with a period of N/2 therefore, they can be used to compute X[k+N/2] but with a twiddle factor of -1. Same decimation in time can be applied to 2 N/2-DFTs(X_o and X_e) to produce 4 N/4-DFTs and this is continued till size 2-DFTs are obtained. Size 2-DFT is simply a butterfly operation represented in Fig. 5.1; taken from [7]. Therefore, the problem breaks from one large DFT to several smaller but faster DFTs. This divide-and-conquer approach accelerates the DFT algorithm from a complexity of $O(N^2)$ to O(N*log N).

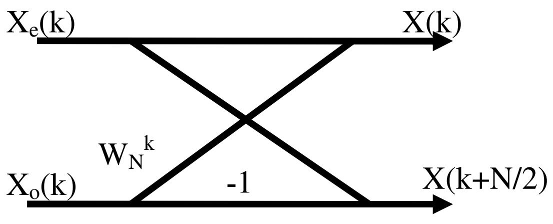

Fig. 5.1: Butterfly Operation of Size 2

Thus, a DFT of size $N=2^r$ is calculated using r stages and each stage consists of N/2 butterfly operations. Output samples have to be permuted by storing each sample at the index corresponding to the bit-reversed index of its previous position. This is required because of the reordering at each stage. Fig. 5.2 shows a complete graph of 3 stages of FFT of size N=8.

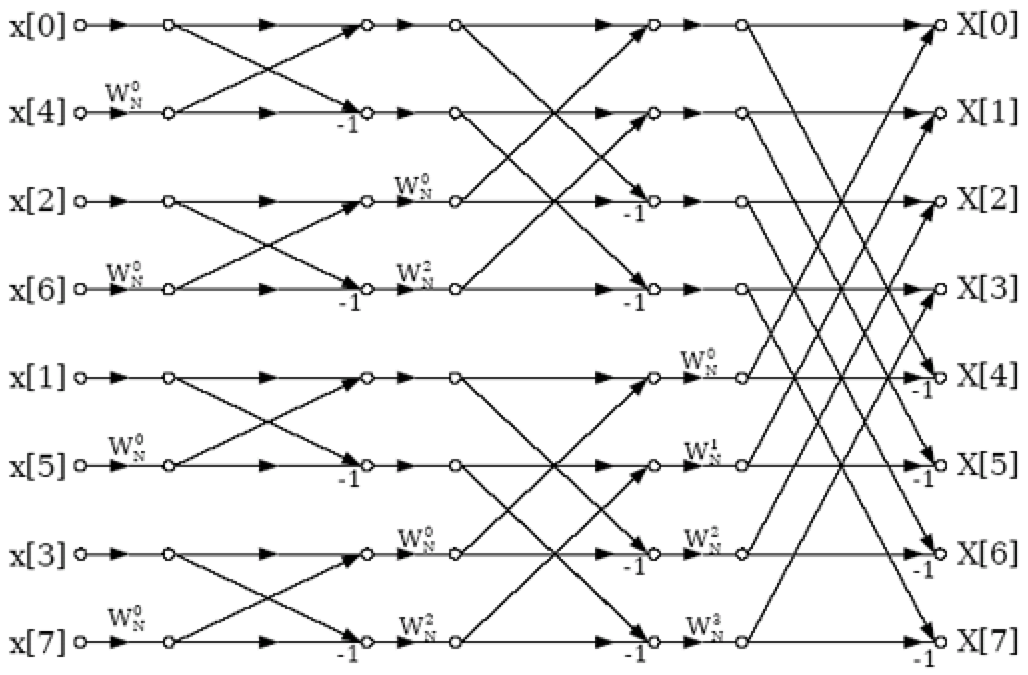

Fig 5.2: Stages of an FFT of Size 8

Here is a Cryptol definition for FFT which is translated from a C implementation given in [7].

```
type size = 128;
type fft_type b = (Arithmetic(b),[size]b)->[size]b;

fft:{b}fft_type b;
fft (lib,xs) = itransform(\idx->\ys->outerloop(idx,ys),brevxs,(lg2 N))
where
{
        N = width xs;
        brevxs = bitreverse xs;
        outerloop (i,xs) = itransform(\idx->\ys->middleloop(idx,ys),xs,g)
         where
         {
                g = N/2**i;
                s = 2**i;
                b = (s)/2;
                r = lg2 N;
                middleloop (k,xs) = itransform(\idx->\ys->innerloop(idx,ys),xs,b)
                where
                {
                        innerloop (m,xs) = main xs
                        where
                        {
                                main xs = subs2(subs1 xs);
                                theta = 2*pi*m*(g)/N;
                                w = lib.expi(lib.fromInt theta);
                                n = k*s+m;
                                y = lib.mul(w,(xs@(n+b)));
                                subs1 list = imap (\j->\x->(if j==n then (lib.add((list@n),y))
                                             else list@j),list);
                                subs2 list = imap (\j->\x->(if j==(n+b) then (lib.sub((list@n),y))
                                             else list@j),list);
                        };
                };
         };

        bitreverse input = imap (\n->\x->input@(reverse n),input)
                where
                {
                        W = lg2(width input);
                };

        };
```

Description of implementation

- `pi` is a constant integer, with the same assumption as in the DFT.
- We first defined a type `fft_type` (parameterized by a variable b) which is a function from a pair of an arithmetic library of type `b` and a sequence of type `b` to a sequence of type `b`. We assume that, all our FFT implementations will have this type.

- `add, mul and sub` perform the arithmetic operations on the type b.
- It consists of three nested loops; a nested loop is defined by a local function, which calls itself a certain number of times (specified by an argument) and passes the intermediate results in another argument.
- `outerloop` runs log2 N times which is the number of stages in an FFT computation.
- `middleloop` is used to generate how many butterfly operations are needed for a certain stage.
- `innerloop` performs the butterfly operations for respective indices and sizes.
- `bitreverse` permutes the input sequence by placing each element on the index corresponding to bit-reversed position of the original sequence.
- Each of these loops is implemented using `itransform` pattern defined in section 3.5.2. The use of `itransform` is necessary because operation perform in each iteration is a function of current loop index.
- Nested loops are simulated by defining a loop as a local function. e.g. `middleloop` is a local function in `outerloop`. Local definition would not be necessary if inner loops were not dependent on indices of enclosing loops.

<u>Limitations</u>

- It cannot be run in the Cryptol interpreter since there is no floating point numbers available.
- Twiddle factors cannot be calculated since $e^{-(j2nk/N)}$ cannot be computed in Cryptol. However we assume the existence of twiddle factors and code the structure of algorithm.

Chapter 6

Channel Coding and Modulation

This chapter covers two aspects of digital signal processing: channel coding and digital modulation.

6.1 Channel Coding

"Channel coding is mapping an incoming data sequence into an output data sequence in such a way that the overall effect of channel noise on the system in minimized [25]."

There are many channel coding algorithms in practice including applications such as: communication, storage and multimedia. We shall discuss the following channel coding algorithms:

- Cyclic Redundancy Check
- Block-Interleaving
- Convolution codes

6.1.1 CRC Algorithm

Cyclic Redundancy Check is commonly used technique for error detection in a received frame; it can easily be modeled and implemented in Cryptol.

6.1.1.1 Modeling CRC in Cryptol

Cryptol has good support for modeling the CRC algorithm; it can perform various useful manipulations on block of bits which simplify the implementations significantly. A CRC algorithm takes a frame (block of bits) input and outputs a Frame Check Sequence (FCS).

6.1.1.2 CRC Algorithm in Cryptol

CRC uses a polynomial of binary coefficients which is generally called `P`; this is decided once for a CRC scheme. If polynomial is `n+1` bits wide frame check sequence is `n` bits wide.
It can be realized in several ways, the most simplified and slow implementation is called straightforward implementation [22].

a) Straightforward CRC Implementation

1. Load a register of length n with zeros.
2. Augment the frame by appending `n` (one less than size of poly) zero bits to the end of it.
3. While (more bits in the augmented frame)
4. Begin
    a. Shift the register left by one bit, reading the next bit of the augmented frame into register bit position 0.
    b. If (a 1 bit popped out of the register during step a)
        i. register = register XOR Poly.
5. End
6. The register now contains the FCS.

Here is a Cryptol translation of this algorithm; `n` and `P` have been defined as constants:

```
n = 20;
P = <|x^20+x^15+x^12+x^9+x^8+x^4+x^2+1|>;

crc:{a}(fin a)=>[a]->[n];
crc input = shiftandxor(0,zero)
  where
  {
     N = width input;
     augmented:[N+n];
     augmented = input#zero;
     shiftandxor (i,reg) =
               if i<(N+n) then
                  (shiftandxor(i+1,((tail reg)#[(augmented@i)])^(zeroorp (reg@0))))
               else
                  reg
               where
               {
                      zeroorp cond = if cond then (tail P) else zero;
               }
  };
```

Description of algorithm

- `crc` is accepts frame of any size and returns a CRC of size n.
- `augmented` frame is obtained by appending `n` zeros to the original frame.
- `shiftandxor` is an in-lined function that is implemented recursively, it shifts 1 bit of the augmented frame into the register and based on output bit from the register either XORs the polynomial with the register or not. `shiftandxor` runs as long as the augmented frame is not completely consumed and then it returns the remaining contents of the register, which is the desired FCS. The second argument of this function serves as register which is initialized in the beginning with zeros.

- `zeroorp` returns last n bits of the polynomial or zeros which are used in the implementation of `shiftandxor`.

b) Table-Driven CRC Implementation

The straightforward implementation is quite slow because it works on 1 bit at a time: in case of large frame this would become very inefficient. Thus, a faster implementation exists that handles 1 byte instead of 1 bit of the register at a time; called Table-Driven implementation [22]. The algorithm is based on the observation that the end result is independent of whether we XOR the polynomial for each shifted instance of the register or we XOR all the shifted instances of the polynomials and then XOR the result with the original register. If we consider 8 shifts together we can build a table that maps top byte (8 bit) of the register to the XORs of the shifted polynomials. In this table each entry is addressable via an 8-bit number and, contains XORs of shifted polynomials. For XORs only those polynomials are considered where there is a 1 in its address. Following are the steps of this algorithm:

```
1. Load a register of length n with zeros.
2. Augment the frame by appending n (one less than size of poly) zero bits to the end of it.
3. while (augmented frame is not consumed)
4. Begin
    a. top = top_byte(register);
    b. register = (Register << 24) | next_augmessage_byte;
    c. register = register XOR precomputed_table[Top];
5. End
6. The register now contains the FCS.
```

Following is a Cryptol implementation of this algorithm where `table` is pre-computed for a given polynomial. Polynomial and contents of the register has been omitted to save space.

```
n = 32;
crc:{a}(fin a)=>[a]->[n];
crc input = shiftandxor(0,zero)
    where
    {
        N = width input;
        augmented:[N+n];
        augmented = input#zero;
        shiftandxor(i,reg)=if i<(N+n) then (shiftandxor(i+8,reg'^sum))
                                   else reg
            where
            {
                t = reg@@[0..7];
                sum = table@t;
                reg' = (reg@@[8..31])#(augmented@@[(8*i)..(8*i+7)]);
            };

    };
```

Description of algorithm

- Interface for this CRC implementation is the same as the previous one, only `shiftandxor` has been modified to shift 8 bits at a time and lookup the pre-computed sum of polynomials in a table called `table`. A definition for `table` is omitted.
- `zeroorp` has been removed because it was just used to decide polynomials should be XORed or not, which should have been done while computing the table.

6.1.2 Block Interleaving

Block interleaving is used in digital communication and storage to mitigate the effect of burst errors. In block interleaving bits are reordered in a pre-defined way and upon reception the process is reversed; in this way, consecutive errors are spread on a certain length of block and can be easily corrected with some less sophisticated technique.

6.1.2.1 Modeling Block Interleaving in Cryptol

A block interleaver accepts a sequence of bits or symbols as inputs and outputs a same number of bits or symbols but in different order. Therefore block interleaver can be modeled with a function that takes a sequence of bits as input and gives a sequence of bits as output.

6.1.2.2 Block Interleaving in Cryptol

The block interleaver takes a sequence of bits and stores them into a memory area which can be assumed to be a matrix of order n x m. It writes the data bits row wise i.e. fills first row first and then second and then third after that it reads the data out column wise i.e. reads all data from first column then from second column and so on. Following Cryptol definition achieve the similar effect:

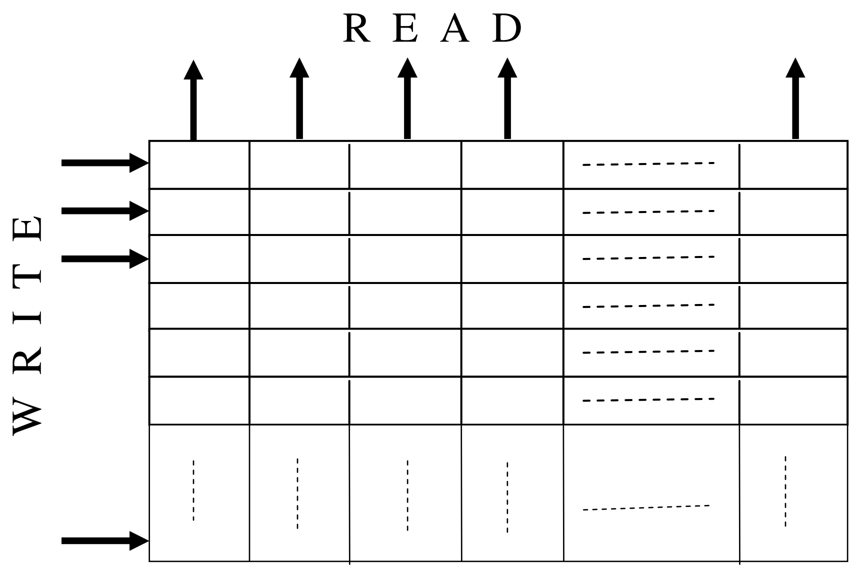

Fig. 6.1: A Block Interleaver

```
rows = 4;
cols = 6;
size = rows*cols;

interleave:{a b}(a>=size)=>[a][b]->[size][b];
interleave xs = read
        where
        {

                block = take (size,xs);
                write = (groupBy (rows,block));
                read = join (transpose write);
        };

deinterleave:{a b}(a>=size)=>[a][b]->[size][b];
deinterleave xs =  read
        where
        {
                block = take (size,xs);
                write = groupBy (cols,block);
                read  = join (transpose write);

        };
```

Description of Algorithm

- `interleave` takes a block (of length `size`) of original data. It uses `groupBy` to get a matrix of dimension `rows` x `cols`. It then applies transpose to this matrix to get a matrix of dimension `cols` x `rows` which is then flattened (using `join`) to produce the interleaved data.

- `deinterleave` also takes a block (of length `size`) of interleaved data. It uses `groupBy` to get a matrix of dimension `cols` x `rows`. It then transposes the matrix into a matrix of dimension `rows` x `cols` which is then flattened (using `join`) to produce the original data.

6.1.3 Convolution Coding

Convolution codes are most widely used error correction codes in digital communication; they are often part of digital radio, mobile phones and satellite links.

6.1.3.1 Modeling Convolution Codes in Cryptol

Convolution coder has similar inputs and outputs as that of an interleaver. It also takes a sequence of bits as input and gives a sequence of bits as output. Output sequence is generally larger than the input sequence. However, output depends not only on the currently input bit but also on previously received bits; therefore, a model similar to that of digital filter is suitable. Thus we model it as a step function that takes a input bit a set of state (register) and outputs a number of bits and new state.

(input_bit, state_register) → (outputs, newstate_register)

6.1.3.2 Convolution Coding in Cryptol

A convolution coder has a register of certain length that stores a sequence of bits received so far; upon receipt of a new input bit the contents of the register are shifted right and the left most bit position is occupied by a new bit. There can be more than one output from a convolutional encoder, which are multiplexed[15] to form a single output sequence. Each output is formed by XORing certain bit positions[16] of the register. The number of outputs of convolution encoder divided by the number of input denotes the rate of convolution encoder.

Following is a Cryptol representation for Convolution coder:

```
n = 3; // number of outputs
k = 1; // number of inputs
m = 3; // size of register
poly:[n][m][1];
poly = [[1 0 1] [1 1 1] [0 1 1]];
convcode:([1],[m][1])->([n][1],[m][1]);
convcode (in,s) = ([|(select (0,poly@i))||i<-[0..(n-1)]|],xs')
        where
        {
                xs' = [in]#tail(s>>1);
                select (i,g) = [(i<m)]&((g@i)&(xs'@i))^(select ((i+1),g));

        };
```

[15] Multiplexing is to take data from many streams and output them on a single stream in a particular order.
[16] Bit positions are determined by position of 1's in the polynomial used for corresponding output.

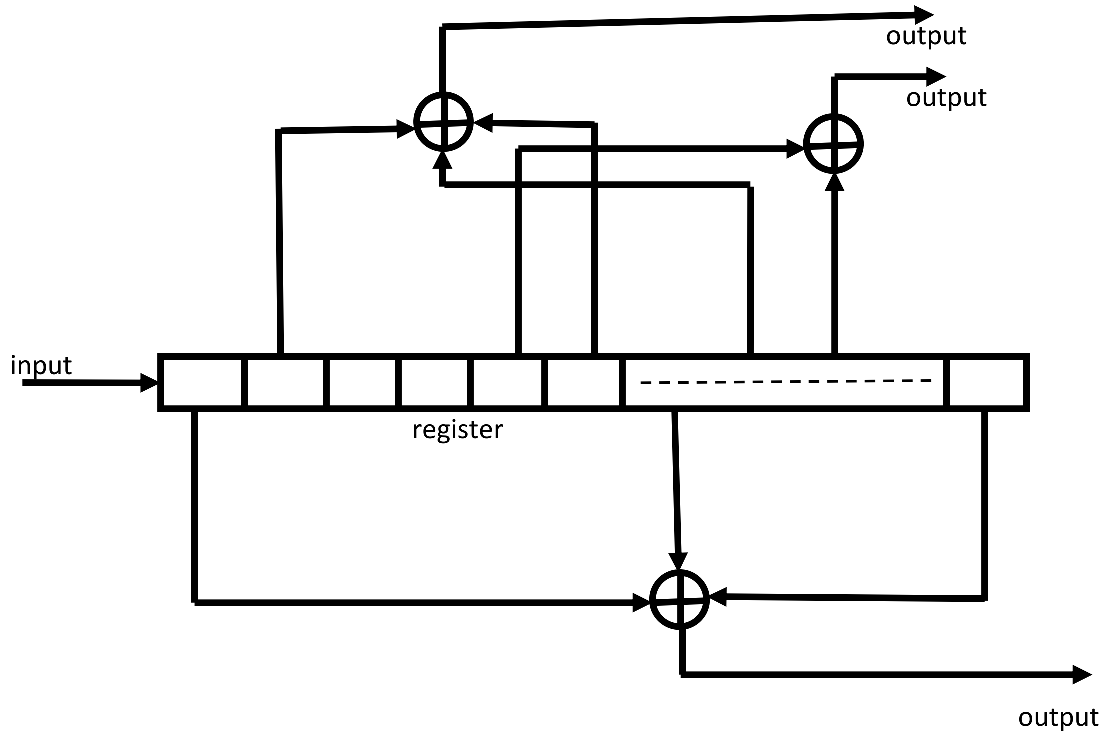

Figure 6.2: Convolutional Encoder

<u>Description of algorithm</u>

- Encode takes two arguments one is new input bit and the other is state representing the register used in convolution encoder.
- It takes one input bit, shifts the state register right and replaces the left most bit position with the new one.
- `select` examines each register bit for a given polynomials and computes the XOR of all the bit positions specified by the polynomial.
- Top level function call just calls `select` as many times as the number of output bits are and keeps the result in a sequence. This causes all the outputs to be multiplexed to form the output sequence.
- New state `xs'` is the register after the shift and replacing the left most bit with a new input bit.

6.2 Digital Modulation

In this part of this chapter an algorithm of digital modulation will be modeled and discussed.

"Modulation is the process of varying one waveform in relation to another waveform. In digital modulation, an analog carrier signal is modulated by a digital bit stream [24].*"*

Today, a number of digital modulation techniques are in use, but for our case study we shall take into account one famous technique called: QPSK (Quadrature Phase Shift Keying).

6.2.1 Quadrature Phase Shift Keying

QPSK is one type of phase shift key: a modulation technique, in which phase of the carrier[17] is varied based bits in the bit-stream to be modulated [23]. It is called *Quadrature* because it uses 2 bits of the bit-stream to be modulated, hence 4 different phases changes of the carrier. It is an implementation choice to select the phase change; for example 0, 90,180,270 or 45,135,225 etc. A mathematical equation describing QPSK might be a bit out of the scope of this thesis; therefore, I present a block diagram (taken from [23]) showing how input is transformed into output in QPSK modulation.

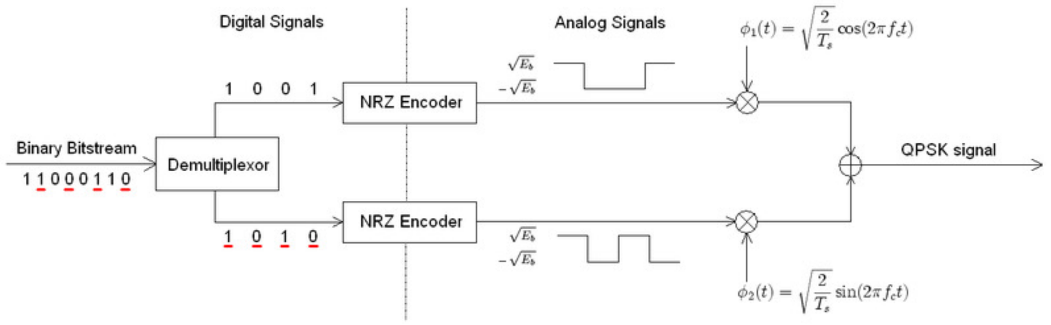

Fig. 6.3 Block diagram showing stages of QPSK [23].

In the diagram above some of the terms are familiar, while some deserve explanation: NRZ[18] Encoder is a block that takes bits and produces a positive analog value corresponding to 1 and negative for a 0, $\varphi_1(t)$ and $\varphi_2(t)$ are carrier waves, circles with a 'x' in the middle stand for multiplication and those with a '+' in the middle represent addition.

[17] Carrier is a high frequency analog (generally) signal used to modulate the information signal.
[18] Non Return to Zero

6.2.1.1 Modeling QPSK in Cryptol

In the algorithm presented in the block diagram above, modeling the carrier wave requires some explanation. Otherwise, demultiplexing[19], NRZ encoding, multiplication and addition of sequences is fairly easy to perform in Cryptol. In order to model carrier wave, I defined a sequence of samples of one cycle of sine/cosine wave at fixed intervals (e.g. at 15^o), and to generate a complete sine wave samples are replicated infinitely many times. The rest of the algorithm is quite natural to constructs available in Cryptol.

6.2.1.2 QPSK in Cryptol

For our implementation we shall be using Fig. 6.3 as a reference. It shows that input bit-stream is first demultiplexed into two bit streams: one from even bit positions and one from odd. NRZ encoding is only a transformation on the bit stream which produces √Es and -√Es; where Es is a constant value. It is not possible to calculate square root in Cryptol therefore, we define √Es as a constant and use it in our definition.
Here is a Cryptol definition for QPSK algorithm:

```
sinsamplepos,cossampleneg,cossamplepos,sinsampleneg:[12][32];

sinsamplepos = [0 258 500 707 866 965 1000 965 866 707 500 258];
sinsampleneg = [0 (-258) (-500) (-707) (-866) (-965) (-1000) (-965) (-866) (-707) (-500) (-258)];
cossamplepos = [1000 965 866 707 500 258 0 (-258) (-500) (-707) (-866) (-965)];
cossampleneg = [(-1000) (-965) (-866) (-707) (-500) (-258) 0 (258) (500) (707) (866) (965)];

sinewave = join [|if k%2==0 then sinsampleneg else sinsamplepos||k<-[1..]|];
cosinewave = join [|if k%2==0 then cossampleneg else cossamplepos||k<-[1..]|];

sqrtEs = 5; // an example value.

nrz:{a}[a][1]->[a][32];
nrz xs = [|if x==1 then sqrtEs else (-sqrtEs)||x<-xs|];

qpsk xs = zipwith(\i->\q->i+q,IRes,QRes)
        where
        {
                N = width xs;
                I = [|xs@i||i<-[0 2..N]|];
                Q = [|xs@i||i<-[1 3..N-1]|];
                IRes = join (map(\i->map(\x->x*i,take(100,sinewave)),nrz I));
                QRes = join (map(\q->map(\x->x*q,take(100,cosinewave)),nrz Q));
        };
```

<u>Description of Algorithm</u>

- `qpsk` is the top level function, which performs the final sum(as shown in the block diagram)

[19] Take sequence of bits from one steam and split them on two streams based on time etc.

- `nrz` performs the NRZ encoding; it takes a sequence of bits and produces an NRZ encoded sequence, using sequence comprehension.
- `I` and `Q` denote even and odd components of input bit-stream; their names are derived from being In-phase and Quadrature.
- `sinewave` and `cosinewave` are sine and cosine samples(for one cycles) replicated infinite times. We have assumed an interval of 15^o, which gives 12 samples for a complete cycle and each sample is 32-bit.
- `QRes` and `IRes` are result of multiplying NRZ encoded bits with the carrier waves; for our case, we have assumed that during 1-bit period there are 100 samples of carrier wave. `QRes` and `IRes` are formed by taking 100 samples from the carrier and multiplying each sample by one element of NRZ coded sequence. This is done by two maps, one that multiplies an NRZ coded number with each sample of a sequence of 100 of samples (of carrier wave) and the other that applies this operation on each element of NRZ coded sequence. Joining the elements of the sequence produced from the two maps gives the multiplication of whole NRZ sequence by the carrier wave.
- Corresponding elements of sequences `QRes` and `IRes` are summed using `zipwith`.

Chapter 7

Vector and Matrix Arithmetic

In this chapter we look at how vector and matrices manipulation can be specified using Cryptol.

7.1 Vector Arithmetic

It is very natural in Cryptol to perform operations on two vectors; it is because of the useful sequence operations in Cryptol.

7.2 Modeling Vectors in Cryptol

In Cryptol sequence can be used to model a vector; more precisely, a vector can be represented with a type [a]b, where a is the length of sequence and b is shape of each element of sequence. We shall be using many abstractions discussed in section 3.5.2 to implement vector operations.

7.3 Vector Operations in Cryptol

Here are some vector operations in Cryptol.

7.3.1 Element-wise Multiplication of All Elements of Two Vectors

Multiplication of corresponding elements of two sequences can be done using `zipwith`.

```
vmult (v1,v2) = zipwith(\x->\y->x*y,v1,v2);
```

7.3.2 Scaling

Scaling is a multiplication of each element of a vector by a constant; clearly, this can be achieved using `map`.

```
vscale (k,v) = map (\x->k*x,v);
```

7.3.3 Element-wise Square

Element wise square is also a `map` of function; which squares its argument, over the list.

```
vsquare (v) = map (\x->x*x,v);
```

7.3.4 Element-wise Sum of Two Vectors

Element wise sum is also a `zipwith`, where the function provided to `zipwith` sums two of its arguments.

```
vsum (v1,v2) = zipwith(\x->\y->x+y,v1,v2);
```

7.3.5 Addition of a Constant to All Elements of a Vector

Element wise addition of a constant can also be realized using `map`.

```
vaddscalar (k,v) = map (\x->k+x,v);
```

7.3.6 Sum of All Elements of a Vector

Sum of elements of a vector can be done by fold available in list library introduced in 3.5.2.

```
sumelem (v) = fold (\x->\y->x+y,0,v);
```

Using 0 as initial element here necessitates that all the elements of the vector are integers however, it can be generalized to use addition of any arithmetic type defined in arithmetic library

7.3.7 Sum of Square of All Values of a Vector

It can be done using `sumelem` defined above and map as follows:

```
sumsquare (v) = sumelem (map(\x->x*x, v));
```

7.3.8 Real and Imaginary Values of All Elements of a Vector

A Cryptol function to compute real and imaginary parts of a complex vector can be written using `map`. In the following, `real` and `imag` return real and imaginary part of a complex number respectively, and `Complex a` is the type defined in arithmetic library.

```
imag,real:{a}Complex a->a;
real e = e.real;
imag e = e.imag;

vreal (v) = map (real,v);
vimag (v) = map (imag,v);
```

7.3.9 Max and Min Value of a Real Valued Vector

Maximum and minimum values of a vector can be calculated using `fold` of `max` and `min` (available in Cryptol) over the vector. A lambda expression converts the uncurried version of these to curried version; since our implementation of `fold` requires the function to be in curried form.

```
vmax (v) = fold (\x->\y->max(x,y),v@0,v);

vmin (v) = fold (\x->\y->min(x,y),v@0,v);
```

7.4 Matrix Operations

Despite it is easy enough in Cryptol to represent matrices and manipulate them, a subset of matrix algorithms cannot be implemented because of the restrictive type system of Cryptol.

7.4.1 Modeling Matrices in Cryptol

Matrices are modeled via three dimensional sequences in Cryptol; since in matrices there are rows and columns, while each element could be a sequence of bits or some other type.

7.4.2 Matrix Operations in Cryptol

7.4.2.1 Matrix Inverse

Many algorithms exist for finding an inverse of a matrix, but the Gauss-Jordan method is easier to implement as a program:

Following is an explanation of Gauss-Jordan method:

- Take the original square matrix of some order n x n.
- Augment this matrix with an n x n identity matrix to produce a new matrix of order n x 2n. The augmented matrix is formed by coinciding rows of identity matrix with original matrix and appending the columns side wise.
- Perform the elementary row operations on the augmented matrix to convert it into reduced echelon form: a reduced echelon form is an n x 2n matrix in which the first n columns and n rows constitute an identity matrix of order n x n.
- In the reduced echelon form the sub-matrix constituted by columns n+1 to 2n and n rows is the inverse of the original matrix.

Here is a Cryptol specification of this algorithm translated from algorithm given in [8]:

```
augment:{a}(fin a,a>=1)=>[a][a][32]->[a][2*a][32];
augment matrix = augmented
          where
          {
                augmented = imap(\k->\row->(row)#(shiftnsplit k),matrix);
                shiftnsplit i = (ident>>i);
                N = width matrix;
                ident:[N][32];
                ident = [1]#zero;

          };

dostep (matrix,i) = step2 (step1 matrix)
          where
          {
                elemi m= m@i@i;
                rowi m = m@i;
                step1 m= imap(\k->\row->(if k==i then (divby (elemi m,row)) else row),m);
                step2 m= imap(\k->\row->(if k==i then row else (row-(mulby (row@i,rowi m)))),m);
                mulby (n,row) = map(\x->x*n,row);
                divby (n,row) = map(\x->x/n,row);

          };

inverse:{a b}(fin a,a>=1)=>[a][a][32]->[a][a][32];
inverse matrix = [|(r@@[(L/2)..(L-1)])||r<-result|]
          where
          {
                result = itransform (\k->\x->dostep(x,k),augmented,W);
                W = width augmented;
                L = width (augmented@0);
                augmented = augment matrix;
          };
```

Description of algorithm

- It consists of three stages; `augment` creates the augmented matrix by appending the unity matrix of the same size to the original matrix. It makes use of `imap` defined in list library in section 3.5.2. The type signature for `augment` shows that there are twice as many columns as in the input matrix.
- `dostep` first divides a row (identified by the argument `i`) by its first element and then it subtracts it (resulting row) from each of the other rows after multiplying it with the `i`th element of corresponding row.
- Inverse applies `itransform` using the function `dostep` on the augmented matrix W (# of rows) times.
- Toplevel sequence compreshension selects the right half of the matrix to form the inverse matrix.

Now we see some matrix algorithms that could not be coded in Cryptol because of the conservative type system of Cryptol. Such algorithms, fall under classical divide-and-conquer problems and they suffer from limitations defined in section 2.2.5. As stated before we cannot say that it is impossible but, it will be useless to invent some complicated way to implement such algorithms. Following paragraph illustrates the problem:

Whenever a function is defined in terms of itself with increasing or decreasing size of one of its argument then it becomes non-trivial to code it in Cryptol. Cryptol either knows or infers types of all definition at compile time. However, if the function accepts a variable sized argument then the size of the argument is inferred when the function is used. In Cryptol it becomes a problem, if an algorithm requires recursion on increasing or decreasing size of its one of its argument. An example of recursion with increasing size of argument could be a function that accumulates an element to a sequence and passes the sequence as an argument in the recursive call. Here is an example of a function that tries to sum a sequence. It is possible to sum a sequence otherwise; but, this function illustrates the problem:

```
sum xs = sum'(0,xs)
        where
        {
                N = width xs;
                sum' (i,ys) = if i<N then (xs@0)+sum'(i+1,tail ys) else 0;
        };
```

In the above code snippet, `sum'` calls itself by incrementing its first argument and skipping head of the second argument `ys`; in this way, second argument is used in the recursive call with length 1 less than its previous length. If this definition is compiled using Cryptol it says "`inf is not finite`" at the compile time, it happens because Cryptol tries to infer an argument finite and infinite at the same time. Use of `tail ys` in the recursive call is the root of problem; for example, when `sum'` is called with a sequence of N 32-bits element then Cryptol thinks that its input type is [N][32]. But, in the recursive call the same function is called with a sequence of size N-1 that suggest that its input type is [N-1][32]; this failure to infer the correct type is the reason for this compile-time error.

If same function is written this way

```
sum2 xs = if N>0 then (xs@0)+ (sum2 (xs@@[1..(N-1)])) else 0
        Where
        {
                N = width xs;
        }
```

Then it type-checks but it gives runtime error. For example, if `sum2` is called with [2 3 4] as argument it says that "`size mismatch between 2 and 3`".

When `sum2` is called from the interpreter with a sequence of size 3 Cryptol infers that it takes sequence of size 3. But `sum2` calls itself with the tail of sequence it has received before which is of size 2 and therefore it does not match with signature of `sum2` which has been inferred to accept a sequence of size 3. The next 3 algorithms fall into same category

7.4.2.2 Determinant of a Matrix

There are a few algorithms to compute determinant of a matrix and the most notable of those is Laplace Formula [8]. It has the following definition:

$$\det(A) = \sum_{j=1}^{N} A_{i,j}(-1)^{i+j}M_{i,j}$$

In the above equation, A is the matrix whose determinant is to be computed, $A_{i,j}$ is the element at ith row and jth column; while $M_{i,j}$ is called minor which is determinant of a matrix obtained by removing ith row and jth column of A. That is, to calculate determinant of an n x n matrix we need to compute determinant of an (n-1) x (n-1) matrix. This algorithm is recursive in size as it involves recursion based on size of matrix.
Here is an attempt to implement this algorithm in Cryptol; it is not very neat but included just to explain the problem.

```
det:{a b}(fin a,a>=1)=>[a][a][b]->[b];
det input = evaluate (input,0)
where
{
        N = width input;
        eliminate (list,target) = take (N,reordered)
        where
        {
                reordered = [|shift (i,target)||i<-[0..N]|];
                N = width list -1;
                shift (i,target) =
                                if i<target then
                                        list@i
                                else
                                        if i<N then
                                                list@(i+1)
                                        else
                                                list@(target)

         };

        omit (mat,r,c) = res
        where
        {
                rowomitted = eliminate (mat,r);
                res = [| eliminate (row,c) ||row<-rowomitted|];
        };

        evaluate (mat,i) = if i<M then
```

```
                       ((mat@0)@i)*(det (omit (input,0,i))) + evaluate (mat,i+1)
                     else 0
               where
               {
                     M = width (mat@0);
               };
       };
```

Description of Algorithm

- Implementation consists of a number of in-lined definitions.
- `eliminate` removes one element(specified by argument target) and returns the resulting sequence.
- `omit` takes a matrix and removes its one row and one column specified by arguments `r` and `c`, and returns the resulting matrix.
- `evaluate` actually performs the determinant calculation; it expands the matrix with the first row. It calls `det` recursively with a smaller matrix having first row and ith column omitted.

Problem

The above implementation of determinant does not type check. Following is the error reported by Cryptol interpreter:

```
While checking type of det, with the following declared signature:
       {a b}(fin a,a>=1)=>[a][a][b]->[b];
Inferred the following constraints not implied by the signature:
       (a-1>=1)
```

which says that the type signature of function `det` does not have constraint: `a-1>=1`. In the declared signature a constraint `a>=1` has been supplied to make sure that the input matrix has at least 1 row and 1 column. But Cryptol requires a constraint: `a-1>=1`; if we provide this constraint in the type signature it will require a constraint: `a-2>=1`. The reason for this is `det` is called with a smaller matrix which will call itself with even smaller size of matrix. Thus, Cryptol fails to infer the right type of the function.

There is no straight forward way to code this structure in the Cryptol because Cryptol cannot generate the correct constraints. The function which computes the determinant, calls itself with a smaller matrix; since in Cryptol size determines the type thus, the type inference fails to determine the correct type of the function. However there might be some implementation that probably will become unnecessarily complex for such a simple algorithm. Algorithms discussed before have their limitations as well but they were specifiable in Cryptol, but this algorithm cannot be coded without avoiding complexity.

7.4.2.3 Cholesky Factorization

Cholesky factorization is often called a square root of a matrix. It factors a matrix into two matrices which are triangular matrices[20] and transpose of one another. Transpose of a matrix is obtained by exchanging rows and columns of a matrix.

$$A = L \times L^T$$

L^T denotes the transpose of factor L.

Following steps are involved in Cholesky factorization of a matrix as given in [9]:

If A is an n x n matrix to be factorized then write it in following form

$$A = \begin{bmatrix} a_{11} & A_{21}^T \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} l_{11} & L_{21}^T \\ 0 & L_{22}^T \end{bmatrix}$$
$$= \begin{bmatrix} l_{11}^2 & l_{11}L_{21}^T \\ l_{11}L_{21} & L_{21}L_{21}^T + L_{22}L_{22}^T \end{bmatrix}$$

Where l_{11} and a_{11} are scalars L_{21} and A_{21} are an (n-1) x 1 matrices L_{22} and A_{22} are (n-1) x (n-1) matrices.

Evaluate first column of L:

$$l_{11} = \sqrt{a_{11}} \text{ and } L_{21} = (1/\sqrt{a_{11}})\, A_{21}$$

While $L_{22}L_{22}^T$ is itself a Cholesky factorization of $A_{22}-L_{21}L_{21}^T$.

It means that Cholesky factorization is defined in terms of Cholesky factorization of smaller matrix $A_{21}-L_{21}L_{21}^T$ and it is the same problem we faced in case of determinant.

7.4.2.4 QR Decomposition

QR-factorization finds out two factors Q and R, a matrix A of order m x n can be written as a product of two matrices Q and R.

$$A = Q \times R$$

[20] Triangular matrix is one whose elements above/below diagonal are zeros. If elements above diagonal are zeros we call it lower triangular matrix and vice versa.

Where R is an upper-triangular matrix of order n x n with all positive diagonals and Q is a matrix of order m x n satisfying Q x Q^T = I or Q is orthogonal. Following is an algorithm for QR-decomposition as given in [9].

Split the matrix A into a1 and A2

$$A = [\, a_1 \quad A_2]$$

Where a1 is m x 1 matrix and A2 is m x (n-1) matrix, then write [a1 A2] as follows:

$$\begin{bmatrix} a_1 & A_2 \end{bmatrix} = \begin{bmatrix} q_1 & Q_2 \end{bmatrix} \begin{bmatrix} r_{11} & R_{12} \\ 0 & R_{22} \end{bmatrix} = \begin{bmatrix} q_1 r_{11} & q_1 R_{12} + Q_2 R_{22} \end{bmatrix}$$

Calculate:

1. R_{11} = || a_1 || and $q_1 = (1/r_{11})\, a_1$
2. $R_{12} = q_1 T A_2$
3. $Q_2 R_2 = A_2 - q_1 R_{12}$

Q_2R_2 is can be found by QR-factorizing $A_2 - q_1R_{12}$.

This algorithm also calls itself with smaller matrix $A_2 - q_1R_{12}$; thus, this algorithm is also a divide and conquer problem and not possible to specify in Cryptol.

Chapter 8

Experiences and Extensions

In this chapter I discuss my findings during implementation of DSP algorithms in Cryptol and also the necessary extensions that could make Cryptol suitable for specification of DSP algorithms.

8.1 Summary of Experiences

Following is a description of how feasible is Cryptol for different types of DSP algorithms:

8.1.1 Filtering Algorithms

- It is possible to easily implement these algorithms with the available constructs in Cryptol.
- Filtering algorithms capture one sample input and they produce output sample based on previous inputs and previous outputs. Output is usually a sum of product of previous inputs/outputs with some constants which is fairly easy to do in Cryptol.
- However, IIR filters that require a feedback from the previous outputs become a bit tricky to implement.
- I currently modeled a digital filer with a step function that takes one new sample, state and produces the output and the new state.
- Floating point operations are not supported. Therefore, results are correct only for integer samples and coefficients.

8.1.2 Spectral Analysis

- I analyzed two algorithms of this domain: DFT and FFT. Both produce the same result but they do it in different ways.
- DFT was easier to implement in Cryptol because it simply accumulates the sum of products of sequences of varying sizes to form a new sequence.
- FFT was a bit different because it involves transforming a sequence of samples in several stages. I based my implementation on a C implementation I found in [7], which consist on three nested loops. It is not very difficult to simulate loops in Cryptol, but in case of nested loops if the inner most nesting uses arguments passed to top level loop then it becomes a problem. I used an in-lined function to model the nested loop because then all the arguments passed to enclosing function are in scope. Although, this way of simulating the nested loops is a bit clumsy and rigid.

- It is not possible to run these algorithms because of unavailability of floating point operations.
- Such algorithms involve complex numbers and exponentiation of a complex number which is not possible in Cryptol.
- Some of these also involve trigonometric functions which is also not possible in Cryptol.

8.1.3 Vector Arithmetic

- It is quite easy to implement and test algorithms involving vectors of integer.
- Scaling, squaring and shifting all elements of a vector can be done using the primitives defined in the list library.
- Only limitation is unavailability of floating point numbers.

8.1.4 Matrix Manipulation

- Matrix multiplication can be done in a compact way in Cryptol.
- Matrix inverse was implemented using gauss-jordan method; it was a tricky but does not become very untidy by use of list library.
- It was not possible to write an algorithm to find determinant of a matrix using Laplacian expansion because of the problem described in 7.4.2.2. Another method of finding determinant of a matrix is to factorize the matrix into a lower and an upper triangular matrix using LU[21], QR or Cholesky factorization, but factorization itself becomes a divide-and-conquer problem.
- QR-decomposition and Cholesky factorization also have the same structure as that of determinant and therefore, could not be coded in Cryptol.
- Floating point is not supported.

8.1.5 Channel Coding

- CRC, Interleaving/De-interleaving, Convolutional encoding were implemented in Cryptol quite successfully. The reason for this is that they manipulate sequence of bits in various ways; which is something Cryptol is very good at.
- There are other similar algorithms; such as, scrambling which was also defined in Cryptol but not included in this report.

[21] Another method of matrix factorization, which decomposes a matrix into a lower and upper triangular matrices. In this method, resulting lower triangular matrix has only 1's in the diagonal.

8.1.6 Digital Modulation

- Digital modulation algorithms can also be implemented easily with the assumption of availability of floating point operations. It was found fairly easy to generate carrier waves and implement the structure of QPSK algorithm.

8.2 Analysis of Results and Extensions

It has been observed that not all DSP algorithms can be specified in Cryptol; however, there are some language constructs that facilitate the specification of DSP algorithms in Cryptol while some do not. Here is an enumeration of what is good for DSP algorithms and what is not.

8.2.1 Why is Cryptol Good for DSP?

- Sequences and sequence comprehension
- Support of recursive function
- Higher order functions
- Split, join, group etc
- @ and @@ operators
- Sequence operations tail, drop and take.
- Arithmetic sequence generation
- Shift and rotate operators
- Append # operator
- zero sequence
- Polynomials and polynomial operators
- Type synonyms
- width function
- record type
- Element wise arithmetic operations on sequences

8.2.2 Why is Cryptol Bad for DSP?

- Cryptol's type system is not suitable for many DSP algorithms. It seems to be the root of many problems that are faced while implementing DSP algorithms in Cryptol.
- Cryptol only supports unsigned numbers, while DSP algorithms require both positive and negative numbers; this problem makes Cryptol less useful for DSP algorithms.
- Recursion with increasing/decreasing size of one of argument involved in the call is not possible. The reason for this is that the Cryptol associates types with sizes.
- Infinite lists have to be typed differently than the finite list and this prevents definition of generic algorithms for finite and infinite lists. It is desirable for a list to have arbitrary size and it should be possible to define algorithms independent of the sizes of lists they

operate on. Type system seems to be cause of this problem as well because it makes types dependent on the sizes of values.

- In Cryptol sizes of sequences we want to construct should be known at compile time or should be derivable from size of an input sequence, this resulted in weird implementation of many DSP algorithms. This is also the reason why tail and drop cannot have length of the sequence as variable.
- It is always required to explicitly apply arguments to a function using parentheses. This makes the code quite dirty, though it is not something specific to DSP algorithms.
- Limited conditional constructs which is however not very bad but at least it's nice to have an optional "else if" branch in the if-else statement.
- Formal verification available in Cryptol cannot be used to verify the correctness of DSP algorithms. Because, for cryptographic algorithms verification is done either by finding satisfying assignments, equivalence or by generating a formal model. For example for a cryptographic algorithm it is possible to state a property stating that no two block of plain text should have the same cipher text and, this can be verified by satisfiability solver. Contrary to this, it is hard to state such a property for DSP algorithm. Also, DSP algorithms work with floating point numbers and search space in that case becomes infinitely large.

Now we make some proposals about how Cryptol can be improved to facilitate specification of DSP algorithms. The proposals are based on experience gained from case studies and study of MATLAB which is a domain specific language for scientific computations.

8.2.3 Proposal for Extensions to Cryptol

Now, we list and elaborate necessary extensions to Cryptol to make it suitable for DSP domain.

8.2.3.1 Numeric Data Types

Cryptol works with bits; a literal, if not given suitable type annotation is inferred to have the minimum size required to store it. This property does not suite DSP algorithms very well; because, DSP algorithms work with numbers and it is not important that how many bits are required to store them. Thus, it is necessary to have a few numeric types, each of which should be capable to store a specified range of values. Integers could be one important such types, because it is common to write fixed point DSP algorithms. Making integers available for programs should not disable access to their individual bits, since in some of the algorithms that is required as well; it would be nice if @ could also be used to access individual bits of an integer as well. A Float data type is quite crucial for a DSL for DSP, in Cryptol this is absent which is the cause of many problems described earlier. Therefore, one very important extension to Cryptol is to provide support for float data type of a reasonable precision. It is of less important that which standard for integers or floats is adopted; any suitable reference for implementation will be sufficient.

8.2.3.2 Scientific Functions

DSP programs work extensively with scientific function such as, sin, cos exp etc; but, in Cryptol they are not available. Hence, one necessary extension to Cryptol is to provide a library of scientific functions. Clearly, this extension will only work once we have numeric data types available.

8.2.3.3 Type System

In Cryptol the size of literal or a sequence determines its type. This may be suitable for cryptography because cryptographic algorithms are often restrictive about the sizes of inputs and outputs. But, there are a number of problems that are faced while implementing DSP algorithms because of this typing scheme. One desirable operation which is often required is to build a sequence at runtime but, Cryptol type system does not allow this. The reason for this is that the Cryptol needs to know all the types at compile time while, if a sequence is built at runtime its type will not be determined until runtime. As stated earlier, recursion on varying sizes of sequences is also not possible because of the type system; when a function calls itself recursively with changing size of its argument then Cryptol becomes confused about the right type of function (because of different sizes in recursive calls).
Because of the above problems we propose modifications to Cryptol type system. We have removed the size-dependent types because they not suitable for DSP algorithms. The following notation captures changes to the current type system of Cryptol:

```
Num            ::= Float | Integer
Scalar         ::= Bit | Num | Record | Type-Synonym | Tuple | Generic -> Generic
Generic        ::= Scalar | Sequence
Sequence       ::= [ Generic ]
```

It suggests that we should have a number type Num; old Cryptol types Bit, Record, Type synonym Tuples and Function types should be preserved. Sequences are now different from old Cryptol sequences: they are denoted by a type in a square bracket which could also be a sequence. Extension to the type system also requires new types for the existing Cryptol's primitives. Next, types of some of Cryptol's primitives are stated according to the new type system:

```
+,-,*,/,**: (Num,Num)->Num
&,|,~,^    : (Bit, Bit) -> Bit
&&,||,^^  : (Integer, Integer) -> Integer
#,<<,<<<,>>,>>> :[a]->[a]
@,! : [a]->a
tail, @@,!! : [a]->[a]
```

```
width : [a]->Integer;
drop : (Integer,[a])->[a]
transpose : [[a]]->[[a]]
splitBy,groupBy : (Integer,[a]) ->[[a]]
join : [[a]]->[a]
>,>=,<,<=,!=,== : (a,a)-> Bit
```

However, I have not explained the new type system in detail. But, the most significant change is the elimination of size dependent types. Also, it is desirable to have curry notation, but that is not depicted in the types above. With these modifications to the type system, it will not be a problem to build a sequence at runtime also, recursion on varying sizes of list will be possible to implement.

8.2.3.4 List operations

It was observed from the case studies that list operations such as `fold`, `map`, `zipwith` etc. are very useful in implementing DSP algorithms. For our implementations we defined them in a library; discussed in section 3.5.2. These list primitives will make implementations compact and easier thus, they should be part of Cryptol as a library.

8.2.3.5 Iterations

It was found during the study that it is often required to perform a computation iteratively while using the result from the previous iterations. In imperative languages it is achieved by loops; therefore in Cryptol there should be some abstraction to simulate loops.

8.2.3.6 Complex Numbers

It is often the case that a DSP algorithm involves complex numbers; such as, DFT and FFT. In Cryptol there is no simple way to represent and use complex numbers. Therefore, an important extension to Cryptol should be to provide complex numbers and operations on them.

8.2.3.7 Infix notation

Cryptol only supports prefix notation and using prefix notation for function destroys the readability of DSP algorithms. In order increase the DSP algorithms Cryptol should allow users to use their functions in infix notations.

8.2.3.8 Custom Operators

This extension is not quite important; however, it would be nice if users can define their custom operators. The merit of this extension is that specifications in this DSL will become more readable.

Chapter 9

Conclusion

This chapter begins with some references to the current research in the area of domain specific languages then it describes conclusive results from this study, while future possible directions in this project comprise the last part of this chapter.

9.1 Related Work

DSLs are in use in many forms but their application to DSP domain has not been explored until recently. One reason seems to be that the DSP people wanted efficiency more than the abstraction. Nevertheless, there are some projects that happen to be quite related. SPIRAL[16] is the most relevant research that targets specification of numerical algorithms in mathematical form and their automatic optimization, it applies an intelligent approach to look for the best implementation of a DSP transform on a specific architecture. OL[21] is another related work which extends SPIRAL to non-transform DSP algorithms, it uses rewriting rules, structural architecture system and empirical search to generate very fast C implementation of non-transform DSP algorithms.

Currently, Chalmers' Functional Programming group is working with Ericsson for the design of a domain specific language for digital signal processing algorithms; this research has so far produced an embedded language in Haskell called Feldspar. Feldspar is now in evaluation phase however it is expected that some publications about its design and suitability for DSP will be released in the future.

9.2 Final Remarks

This study revealed many aspects of Cryptol, the nature and structure of DSP algorithms and the applicability of Cryptol to DSP algorithms. Case studies were done by modelling DSP algorithms in Cryptol and then implementing them (under some assumptions). Results of the evaluation show that Cryptol can specify only a specific class of DSP algorithms. This class includes CRC, convolution codes and spreading/dispreading algorithms. Cryptol however can specify some DSP algorithms with the assumption that floating points and certain trigonometric functions are available. This includes filtering and spectrum analysis algorithms. But DSP algorithms sometimes involve size varying recursive structure; these types of algorithms could not be specified at all in Cryptol. This renders Cryptol quite useless for the DSP domain. To make it usable for the DSP domain a number of extensions were proposed in the previous chapters. Some of the extensions would be trivial to incorporate in the language while, some of the necessary extensions involve changes to type system of Cryptol which would change the original shape of the language very much. Keeping this in view it seems that it will be far easier to develop a new domain specific language for DSP instead of extending Cryptol to make it suitable for the DSP domain.

Despite of being very peculiar to Cryptographic algorithms, some DSP algorithms can be modelled and implemented in Cryptol far more neatly than in any imperative language. The general impression of Cryptol is that it is easy to code the structure of many DSP algorithms but the lack of floating point numbers and scientific functions prevents us from simulating the specifications on test inputs. The most notable characteristic of Cryptol is its type system that serves as a root of all issues faced while implementing a DSP algorithm in Cryptol. Giving a new type system to a language is more than extending it; it's more like inventing a new language which was not our actual goal. It was part of the proposal to model the extensions in Haskell as well but it was found that required changes are major and cannot be modelled in Haskell, therefore this discussion is omitted from the report.

9.3 Future Work

There are a number of aspects in which this thesis can be expanded. One possible enhancement to this thesis will be to look for more extensions. It is also possible get inspiration from SPIRAL or Feldspar to introduce new functions to Cryptol. Additionally, code generation for target DSPs will be a slightly different but exciting project to work on. Finally modification of Cryptol to accommodate the suggested changes can also serve as a good thesis work.

Bibliography

[1] *Cryptol: The Language of Cryptography, Programming Guide,* Cryptol Documentation, Galois Inc, Version 1.8, October 2008.

[2] *From Cryptol to FPGA: A Tutorial,* Cryptol Documentation, Galois Inc, Version 1.8, October 2008.

[3] Levent Erkök, *Compiling Cryptol to Haskell, C++ and C*, Cryptol Documentation, Galois Inc, October 2008.

[4] Tom Dinkelaker, Mira Mezini, *Dynamically Linked Domain-Specific Extensions for Advice Languages,* AOSD workshop on Domain-specific aspect languages, Brussels, Belgium, April 2008.

[5] *Digital Signal Processing tutorial*, Retrieved July 01, 2009, from:
< http://www.dsptutor.freeuk.com/>.

[6] B. Preetham Kumar, *Digital Signal Processing Laboratory*, CRC Press, 2005.

[7] Robert J. Schilling, Sandra L. Harris, *Fundamentals of digital signal processing using MATLAB*, 2005.

[8] Gene H. Golub, Charles F. Van Loan**,** *Matrix computations*, Johns Hopkins Univ. Press, 1996.

[9] Dr.Brien Alkire, *Lecture Notes in Applied Numerical Computing Spring 05/06,* UCLA Electrical Engineering Department. Retrieved July 25, 2009, from :
<http://www.alkires.com/ee103.html>.

[10] *Case Study-Domain Specific Languages*, Galois Inc, 2008. Retrieved June 20, 2009, from:
< http://www.galois.com/files/Cryptol/Cryptol_casestudy.pdf>

[11] Robert Jan Ridder, *Programming Digital Signal Processor with High-Level Languages,* DSP Engineering, TASKING Inc, summer 2000.

[12] Jon Bently, *Programming Pearls: little languages,* Communications of ACM, Volume 29, Number 8, August 1986.

[13] Wave Report, *DSP Tutorial*. Retrieved July 20, 2009, from :< http://www.wave-report.com/tutorials/DSP.htm>.

[14] Robert Oshana, *Overview of Digital Signal Processing Algorithms, Part II,* IEEE Instrumentation & Measurement Magazine, Vol. 10, Issue 2, p 53-58, April 2007.

[15] Numerix Ltd, *Introduction to Digtial Signal Processing,* December 2006. Retrieved August 10, 2009, from :< http://www.numerix.co.uk/tutorials/DSP/DSPIntroduction.pdf>.

[16] M. Puschel, J.M.F. Moura, J.R. Johnson, D. Padua, M.M. Veloso, B.W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R.W. Johnson, N. Rizzolo, *SPIRAL: Code Generation Technique for DSP Transforms,* Proceedings of the IEEE, Vol. 93, Issue 2, p 232-275, February 2009.

[17] Jennifer Eyre and Jeff Bier, *The Evolution of DSP Processors*, White Paper, IEEE Signal Processing Magazine, Berkeley Design Technology Inc., 2000.

[18] Laurence Tratt, *Domain Specific Language Implementation via Compile-Time Meta-Programming*, ACM Transactions on Programming Languages and Systems, Vol. 30, No.6, Article 31, October 2008.

[19] Robert Oshana, *DSP Software Development Techniques for Embedded and Real-Time Systems,* Embedded Technology Series, Elsevier Inc. 2006.

[20] Hudak, Paul, *Modular domain specific languages and tools*, Fifth International Conference on Software Reuse, p 134-142, Jun 1998.

[21] Franz Franchetti, F. de Mesmay, Daniel McFarlin, and Markus P., *Operator Language: A Program Generation Framework for Fast Kernels,* IFIP Working Conference on Domain Specific Languages, LNCS, Springer, Vol. 5658, p 385-410, 2009.

[22] Ross N. Williams, *A Painless Guide to CRC Error Detection Algorithms.* Version 3, 19 August 1993. Retrieved September 25, 2009, from: <http://www.ross.net/crc/download/crc_v3.txt>.

[23] Wikipedia, the free encyclopedia, *Phase-Shift Keying*, 2009. Retrieved October 13, 2009, from: <http://en.wikipedia.org/wiki/Phase-shift_keying>.

[24] Wikipedia, the free encyclopedia, *Modulation*, 2009. Retrieved October 13, 2009, from : <http://en.wikipedia.org/wiki/Modulation>.

[25] Simon Haykin, Michael Moher, *Modern Wireless Communication*, Prentice Hall, 2005.

[26] Simon Peyton Jones, John Hughes and et al. *The Haskell 98 Report*, February 1999.