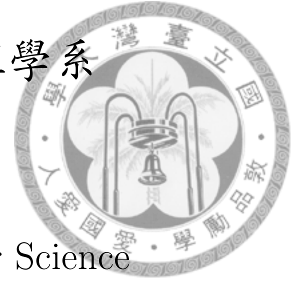
國立臺灣大學電機資訊學院電機工程學系
碩士論文
Department of Electrical Engineering
College of Electrical Engineering and Computer Science
National Taiwan University
Master Thesis



爲高效率密碼工程設計之特定領域語言

A Domain-Specific Language for Efficient Cryptographic Engineering

郝柏翔
Po-Hsiang Hao

指導教授：鄭振牟 博士
Advisor: Chen-Mou Cheng, Ph.D.

中華民國 103 年 6 月
June, 2014

誌謝

在一年多以前我對於 functional language 完全沒有概念，也從來沒有寫過 Haskell，根本不會想到竟然會以它爲我的研究的主軸。

首先要感謝我的指導教授鄭振牟教授，帶領我接觸 functional language 和 Haskell。他提供給學生們一個很好的研究環境，對於我們遇到的問題和困難，總是能提出很有效的建議。這段期間我學到了許多對問題的思考方式，及解決問題的方法。當然也要感謝另一位實驗室的大家長，中研院的楊柏因老師，在我研究所這段期間給予的協助。感謝口試委員們穆信成老師、王柏堯老師和陳郁方老師，對這篇論文給予了許多寶貴的意見。

接著要感謝實驗室的學長和同學們：陳明興、郭博鈞、楊上逸、吳崧銘、李鎬、陳韋翰、吳忠憲、李文鼎等等。他們陪伴我度過兩年的研究所生活，陳明興學長教我 Hydra 的指令集和模擬器的用法。郭博鈞學長負責幫實驗室買零食，而且常常規劃有趣的活動，例如：爬山、射箭、溜冰、攀岩等等，讓大家偶爾可以放鬆一下。楊上逸學長在我剛開始學 Haskell 時給我不少幫助，並且常常是除了老師之外少數在我報告時聽得懂的人。吳崧銘會和我討論並互相交換找研發替代役的資訊。忙碌之餘，我會看李鎬下網路圍棋，並和他一起嘲笑犯錯的對手。陳韋翰和李文鼎常常陪我一起討論演算法的問題，組隊參加線上的比賽。感謝吳忠憲和李鎬雖然自己兩天前先口試完了，但在我口試時也來加油。另外還要感謝百片學長每次 group meeting 時都幫大家訂麥當勞，並且容忍大家各式各樣的點餐要求。

最後一定要感謝我的家人，一路上無論我做任何決定他們都全力支持。感謝雅婷，當我在趕論文壓力很大時的對我的包容。還有許許多多在我求學過程中的老師、同學和朋友，因爲有你們的教誨、協助與支持，我今天才能完成這篇學位論文。謝謝你們。

摘要

實作密碼學系統時，常見許多多維代數結構間的運算。若要在較低階的組合語言上實作，必須轉換成基本元素的運算。運算數量龐大時，必須有自動化工具來輔助。此外，在低階語言上無法高階地描述系統或演算法，增加程式設計者的困難，以及出錯的可能性。

我們提出一個嵌於Haskell中的特定領域語言，讓程式設計者能以方便的語法和多維的代數結構，描述密碼演算法和系統。程式會被表示成樹狀的表示式，並且由編譯器自動展開代數結構的運算，轉成中間語言，再進行優化並產生目標語言。

編譯器結合了兩個優化器，並且實作了兩種目標語言，分別是Hydra處理器上的組合語言，以及C++，支援的代數結構有擴張體和矩陣。程式設計者也能加入自己所需的代數結構、優化或是目標語言。我們在此特定領域語言上實作了兩個應用：最佳配對和一個基於LWE的密鑰交換系統。

使用此特定領域語言實作密碼系統，可將數學演算法、優化和輸出語言各自獨立，節省重複的工作，並且程式設計者在實作時可把重點放在密碼系統高階的描述。

關鍵字: 密碼工程、特定領域語言、*Haskell*、*Hydra*處理器、編譯器優化

Abstract

Multidimensional algebraic structures are common in the description of cryptographic systems. They have to be translated to computations between basic elements by automation before being implemented on low-level assembly languages. Besides, the programmer cannot write programs in a high-level way, which makes them more error-prone.

In this thesis, we propose a domain-specific language embedded in Haskell, so that the programmer can implement cryptographic systems in convenient syntax. The computations of algebraic structures will be expanded, supporting extension fields and matrices.

Our compiler is combined with two optimizers, and supports two target languages: Hydra assembly and C++. The programmer can add his own algebraic structures, optimizations, and target language as needed. We also implement two applications in this DSL: optimal pairing and a key exchange with LWE.

The algorithm description, optimizations and code generations is separated and independent. The programmer can focus on the high-level descriptions of the cryptographic systems.

Keywords: *cryptographic engineering, domain-specific language, Haskell, Hydra coprocessor, compiler optimizations*

Contents

1 Introduction **1**

1.1 Embedded Domain-Specific Language 2

1.2 Hydra 3

1.3 Contribution 3

2 Overall Structure **5**

3 Language Embedding **6**

3.1 Expressions 6

3.1.1 Standard Mathematical Operators 7

3.1.2 Functions 7

3.2 Let-sharing 8

3.3 Control Flow 10

4 Algebraic Structure Expansion **11**

4.1 Extension Field 11

4.2 A Small Example 11

4.3 Haskell For Maths 12

4.3.1 Base Field 13

4.3.2 Extension Field 13

4.4 For General Algebraic Structures 15

5 Compiling Embedded Language **17**

5.1 Intermediate Representation 17

5.2 Expressions to IR 18

5.2.1 Type Class 18

5.2.2 Default Implementation 18

5.2.3 Instance Example 19

6 Optimizations and Code Generation **21**

6.1 Common Subexpression Elimination 21

6.2 Linear Register Allocation 21

6.3 Code Generation 22

6.3.1 Hydra 22

6.3.2 C++ 23

7 Applications **24**

7.1 Pairing 24

7.2 Key Exchange Protocol from LWE 25

8 Summary **26**

8.1 Related Work 26

8.2 Future Work 26

Bibliography **28**

List of Figures

2.1 The overall compilation process. The red blocks are the program in multiple representations, and the blue blocks are components of the compiler. 5

3.1 The in-memory representation of `square (x+1)`. 8

List of Tables

8.1 Comparison with related work 27

Chapter 1

Introduction

Cryptographic engineering is using cryptography to solve problems, such as ensuring *data confidentiality*, *authenticating* people or devices, or verifying *data integrity*. It is a complex, multidisciplinary field, including:

- Mathematics: finite groups, rings, fields, lattices, etc.
- Computer engineering: hardware design, ASIC, embedded system, FPGAs, etc.
- Computer science: algorithms, complexity theory, software design, etc.

In this thesis, we focus on the problems of implementing cryptographic systems in software.

Sometimes, the implementation has to be done in instructions for processors or coprocessors. The programmer will face the following challenges:

- Cryptographic protocols are usually expressed in terms of multidimensional algebraic structures, such as extension fields or matrices. For example, an $n \times n$ matrix over a field F represents n^2 elements in F. When the program is written in low-level languages without objects to represent matrices, the programmer will have to explicitly rewrite a single matrix multiplication into n^3 multiplications in F. This can be tedious without help from automation.

- Writing in low-level languages is inconvenient, and error-prone. Also, to generate code for another machine, the programmer will have to learn new languages.

- The program needs to be optimized for efficiency. Some optimizations are machine-independent, and should be abstracted to avoid duplicated work.

1.1 Embedded Domain-Specific Language

A domain-specific language (DSL) is a computer language specialized to a particular domain. A DSL can be embedded in a general purpose host language, while adding domain specific elements, such as data types, functions, etc. In this way the DSL can exploit the existing syntax, type system, and libraries of its host language, saving the designers from the details of language implementation.

There are two ways to embed a language: shallow embedding and deep embedding. Shallow embedding uses host language functions and values as its own functions and values, while deep embedding uses algebraic data types to represent the abstract syntax tree. Since we want the language to be compiled to an intermediate representation, deep embedding is our choice.

Embedding code-generating domain-specific languages in Haskell was originally advocated by Leijen and Meijer [LM99]. It has been popular choices for domains including parsing [Hut92], pretty-printing [Hug95], efficient image manipulation [EFDM03], robotics [PNH02] and hardware circuit design [BCSS98]. Its advantage includes:

- Functional. Good at expressing mathematical functions, no side effects.

- Strong, static typing. Bugs can be caught early.

- Higher ordered functions are convenient.

- Algebraic data types and pattern matching makes embedding language simpler.

1.2 Hydra

Hydra [CHH+] is a complete, proof-of-concept public-key cryptography (PKC) based system. It demonstrates that strong, hardware-assisted PKC can be feasible for M2M sensors [SHH+13].

Hydra contains a scalable and programmable cryptographic coprocessor. It has specialized instructions to perform modular arithmetic operations in finite fields with large characteristics.

1.3 Contribution

We propose a domain-specific language embedded in Haskell for efficient cryptographic engineering, with the following features:

- When the programmer is implementing an algorithm or a system, he/she can focus on the high-level description.

- The algorithm representation, target language implementation, and machine-independent optimizations can be implemented separately and reused. In particular, adding a new target language requires only an implementation from the intermediate representation to the new target language. The algorithms do not need to be rewritten.

- The language allows programmers to use multidimensional algebraic data types other than vectors or other basic data types. We demonstrate a method to translate those expressions into expressions in the base field. We provide two built-in algebraic structures: extension fields, and matrices. The program-

mer can easily add new algebraic structures as needed, based on the interface we provide.

- We provide two target language implementations: Hydra's assembly language and C++. The programmer can easily add new target languages implementations.

- We provide two applications: optimal pairing and key exchange with LWE.

- The compiler is combined with two optimizers. The programmer can also combine his/her own optimizers.

Chapter 2

Overall Structure

The programmer provides the Haskell program, which will be represented as an AST. Then, the computations of algebraic structures will be expanded, and compiled to an intermediate representation (IR). Finally, the IR goes through optimization and code generation, and the target code is produced.

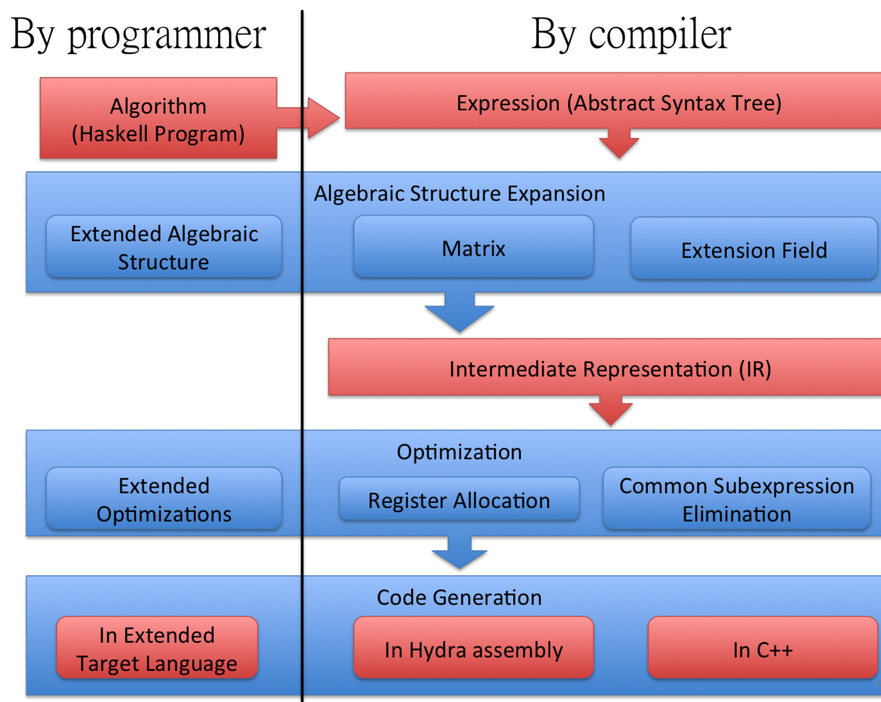


Figure 2.1: The overall compilation process. The red blocks are the program in multiple representations, and the blue blocks are components of the compiler.

Chapter 3

Language Embedding

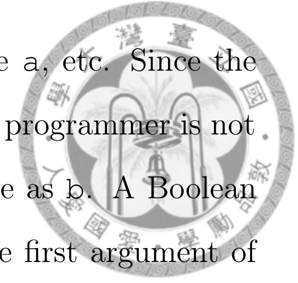
3.1 Expressions

Our goal is to allow the programmer to write programs almost as if he/she is writing in standard Haskell. Specifically, we do not want to force the programmer to write the program in *monadic* style.

We want to store the computation as abstract syntax trees in recursively defined expressions. In previous work [EFDM03, MM10], the underlying expressions are untyped, but wrapped by a *polymorphic* type with a *phantom type variable*. Untyped expressions are easier to handle, but they can only hold expressions of certain built-in types, such as `Int`, `Float` or `Arrays`, etc. We want to deal with many kinds of algebraic structures, so we use a typed expression with generalized algebraic data types (GADTs) like this:

```
data Exp a where
  Const :: a -> Exp a
  Input :: String -> Exp a
  Add :: Exp a -> Exp a -> Exp a
  Sub :: Exp a -> Exp a -> Exp a
  Mul :: Exp a -> Exp a -> Exp a
  Equal :: Exp a -> Exp a -> Exp Bool
  IfThenElse :: Exp Bool -> Exp a -> Exp a -> Exp a
```

An expression of type `a` holds either a constant of type `a`, a `String` to identify



an input variable, a binary operation of two expressions of type a, etc. Since the expressions are typed, Haskell will do the type check for us. The programmer is not allowed to add Exp a and Exp b together if a is not the same as b. A Boolean expression will be of type Exp Bool, which can be used as the first argument of IfThenElse.

3.1.1 Standard Mathematical Operators

The programmer should be able to use standard mathematical operators like $+$, $-$, and $*$. Thanks to Haskell's type classes, we can make Exp an instance of the Num type class to overload the operators.

```
instance Num a => Num (Exp a) where
  x + y = Add x y
  x - y = Sub x y
  x * y = Mul x y

  fromInteger = Const . fromInteger
```

The definition of fromInteger may seem recursive, but the signatures of them are different. The former one has signature Integer -> Exp a and the latter one has signature Integer -> a.

3.1.2 Functions

Our language directly uses Haskell's function. Consider a function square in Haskell:

```
square :: (Num a) => Exp a -> Exp a
square x = x * x
```

It simply multiplies the input value to itself. Since the mathematical operators are overloaded, the function body needs little modification. To get the representing computation in an expression, we evaluate square (Input "x"), and get the following representation:

```
Mul (Input "x") (Input "x")
```

3.2 Let-sharing

If the input is more complicated, say `x+1` for example, we will get this output expression:

```
Mul (Add (Input "x") (Const 1))
    (Add (Input "x") (Const 1))
```

When we traverse the tree, we end up processing the same expression repeatedly. Even though GHC will represent it in memory like in figure 3.1, we cannot directly observe the sharing introduced by Haskell bindings, even if we use the `let` expression in Haskell.

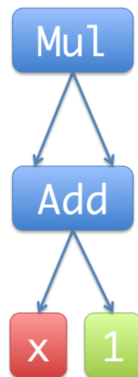
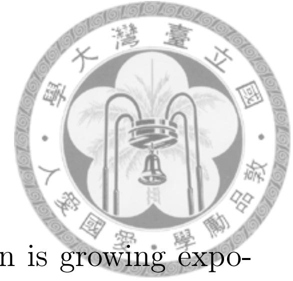


Figure 3.1: The in-memory representation of `square (x+1)`.

To compute large integer powers of a number fast, *square-and-multiply* algorithm is often used. Its complexity is linear to the number of bits of the integer power, because it repeatedly squaring the base. Now, consider `square` being iterated twice, which is `square (square (Input "x"))`. It evaluates to:

```
Mul (Mul (Input "x") (Input "x"))
    (Mul (Input "x") (Input "x"))
```

Then consider `square (square (square (Input "x")))`, which evaluates to:



```
Mul (Mul (Mul (Input "x") (Input "x"))
         (Mul (Input "x") (Input "x")))
    (Mul (Mul (Input "x") (Input "x"))
         (Mul (Input "x") (Input "x")))
```

We can see that the number of `Mul` operations in the expression is growing exponentially, because each time the function `square` is called, the input expression is evaluated twice.

To represent the sharing, we have to introduce some other types of expressions:

```
data Exp a where
  Let :: String -> Exp b -> Exp a -> Exp a
  Var :: String -> Exp a
```

The first argument of `Let` is the name of the variable assigned to the second argument `Exp a`, and the body of the whole `Let` expression is an `Exp b`. In the body, the programmer can use `Var "name"` to refer to the variable defined by `Let`, but the definition cannot be used outside the body of a `Let` expression.

Now, the programmer can rewrite the function `square` in the following way:

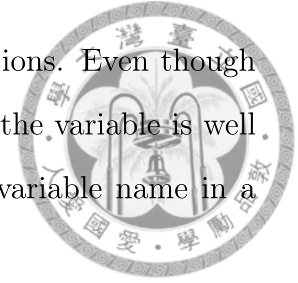
```
square :: (Num a) => Exp a -> Exp a
square x = Let "y" x ((Var "y") * (Var "y"))
```

In this way, our library can recognize the sharing.

There is no ambiguity to the use of `Var "y"`, because we always refer to the closest definition of the used variable. Even if there is an outer `Let` expression defining a variable also named `y`, in `square` the `Var "y"` will always refer to the input argument `x`.

Now, if the final version of `square` is iterated three times, the result would be:

```
Let (Var "y")
    (Let (Var "y")
         (Let (Var "y")
              (Input "x")
              (Mul (Var "y") (Var "y")))
         (Mul (Var "y") (Var "y")))
    (Mul (Var "y") (Var "y"))
```



The number of `Mul` operations is linear to the number of iterations. Even though there are three variables named `y` in this example, the scope of the variable is well defined. As long as the programmer does not reuse the same variable name in a single scope, there will be no ambiguity.

The explicit `Let` expressions require a little more modifications than we wanted. A technique was proposed by Gill [Gil09] to make the sharing implicit, using a *reification monad* to maintain a map from the stable name of an expression to its rewritten form, and thus allow the programmer to use Haskell's native let bindings. We leave the implementation of implicit let-sharing to future work.

3.3 Control Flow

There is no loop in our expressions. Loops in the embedded language are all unrolled.

Chapter 4

Algebraic Structure Expansion

4.1 Extension Field

Let K be a subfield of a field L. We also say that L is an *extension* of K. Quotient rings are often used to construct field extensions. Suppose K is some field and f is an irreducible polynomial in $K[x]$. Then the quotient ring $L = K[x]/(f)$ is a field whose *minimal polynomial* is f. The elements are the polynomials with coefficients in K. The addition and multiplication in L are under modulo f.

For example, consider the ring $\mathbb{R}[x]$ of polynomials in the variable x with real coefficients. The quotient ring $\mathbb{R}[x]/(x^2+1)$ is isomorphic to the field of complex numbers $\mathbb{C}$, with x playing the role of the imaginary unit i. The reason is that in the quotient ring, addition and multiplication are modulo x^2+1, so $x^2+1=0$, i.e. $x^2=-1$, which is the defining property of i.

4.2 A Small Example

Given $a, b \in K_2 = K[x]/(x^2+2)$ for some field K, we try to compute $c = ab$. We know a and b can be represented as $a_1x + a_0$ and $b_1x + b_0$ respectively. Hence,

$$c = (a_1x+a_0)(b_1x+b_0) = a_1b_1x^2+(a_1b_0+a_0b_1)x+a_0b_0 = (a_1b_0+a_0b_1)x+(a_0b_0-2a_1b_1).$$

Therefore, if c is represented as $c_1x + c_0$, we get the two equations:

$$c_1 = a_1b_0 + a_0b_1, \quad c_0 = a_0b_0 - 2a_1b_1.$$

That is, when we get an expression `Mul (Input "a") (Input "b")` of type `Exp K2`, it has to be expanded to two expressions of type `Exp K`

```
Add (Mul (Input "a1") (Input "b0"))
    (Mul (Input "a0") (Input "b1"))
```

and

```
Sub (Mul (Input "a0") (Input "b0"))
    (Mul 2 (Mul (Input "a0") (Input "b1")))
```

This may not seem difficult, but the extension fields in real applications are much more complicated. The implementation of the *optimal pairing* in Hydra uses a three-level extension field K_{12}:

$$\begin{aligned} K_{12} &= K_6[x]/(x^2 - y) \\ K_6 &= K_2[y]/(y^2 - z - 1) \\ K_2 &= K[z]/(z^2 + 2) \\ K &= F_p \end{aligned}$$

The expansion has to be done three times, and there will be 12 equations with 24 variables, which are kind of messy. That is why we need to expand them by the compiler.

4.3 Haskell For Maths

A benefit of embedding language in a rich host language is that the existing library can be exploited. We are going to use a library called *HaskellForMaths*[1], which

[1] http://hackage.haskell.org/package/HaskellForMaths

contains implementations to many algebraic structures.

4.3.1 Base Field

First, we use HaskellForMaths to define a base field with p elements, where p is a prime:

```
data Tp
instance IntegerAsType Tp
  where value _ = p
type K = Fp Tp
g1 = 1 :: K
```

The type class `IntegerAsType` is defined in HaskellForMaths as follows:

```
class IntegerAsType a where
    value :: a -> Integer
```

The data type `Tp` is a type holding an integer. The function `value` is defined such that the integer that `Tp` represents can be found by

```
value (undefined :: Tp)
```

Then `Fp Tp` is a field with p elements and modular arithmetic.

4.3.2 Extension Field

Then, we use HaskellForMaths to define an extension field.

```
data DefPolyK2
instance PolynomialAsType K DefPolyK2
  where pvalue _ = x^2 + 2 where x = UP [0, 1]
type K2 = ExtensionField K DefPolyK2
x = Ext (UP [0, 1]) :: K2
```

The type class `PolynomialAsType` means `DefPolyK2` holds a polynomial with coefficients in `K`, which we can retrieve from

```
pvalue (undefined :: (K, DefPolyK2))
```

since the definition of `PolynomialAsType` is:

```
class PolynomialAsType k poly where
    pvalue :: (k,poly) -> UPoly k
```

Then, `ExtensionField K DefPolyK2` defines an extension field, whose minimal polynomial is $x^2 + 2$. The constructor `UP` constructs a univariate polynomial from a list, and the constructor `Ext` constructs an element of an extension field from a univariate polynomial.

We can try to evaluate `(x+1)*(x+2)` and get `3x` because

$$(x+1)(x+2) = x^2 + 3x + 2 = 3x,$$

since the calculation are modulo $x^2 + 2$. The HaskellForMaths library will do the modular operation for us to make the degree of the results smaller than the modular polynomial.

Now when we get the expression `Mul (Input "a") (Input "b")`, where `Input "a"` and `Input "b"` are of type `Exp K2`. Since the degree of `K2` is 2, we expand `Input "a"` to `Ext (UP [Input "a0", Input "a1"])`, which is an extension field element with coefficients `Input "a1"` and `Input "a0"` in `Exp K`. The name `a0` and `a1` are created by appending numbers to the original name `a`.

Similar expansion is done to `Input "b"`, now we can multiply them together:

```
Ext (UP [Input "a0", Input "a1"])
* Ext (UP [Input "b0", Input "b1"])
```

Then, the two field elements are multiplied as if their coefficients are normal numbers, and we retrieve the two coefficients in `Exp K` respectively:

```
Sub (Mul (Input "a0") (Input "b0"))
    (Mul 2 (Mul (Input "a0") (Input "b1")))
```

and

```
Add (Mul (Input "a1") (Input "b0"))
```

```
(Mul (Input "a0") (Input "b1"))
```

4.4 For General Algebraic Structures

What we just did here was to expand the original expression of type:

```
Exp (ExtensionField K DefPolyK2)
```

to:

```
ExtensionField (Exp K) DefPolyK2
```

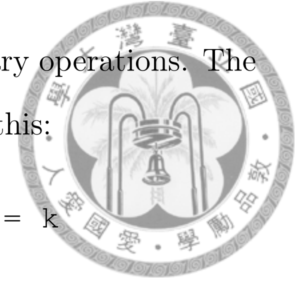
That is, turning an expression of an extension field to an element of an extension field whose coefficients are expressions. Then, the desired operations are done by Haskell libraries, and retrieve the resulting expressions from the coefficients.

To make this process more general, we define a type family `SubType`, and a type class `Expandable` as follows:

```
type family SubType a

class Expandable a where
  size :: a -> Int
  coefficients :: a -> [SubType a]
  expandSpec :: Exp a -> [Exp (SubType a)]
  expand :: Exp a -> [Exp (SubType a)]
  expand (Const ...) = ...
  expand (Input ...) = ...
  expand (Var ...) = ...
  expand (Let ...) = ...
  expand e = expandSpec e
```

To make type a expandable, first we have to define the `SubType` of a. Then we have to be able to infer the size of an element, because when `Input "a"` and `Input "b"` are expanded, we need to know the dimension of the extension. We also have to know how to get its coefficients, which is a list of `SubType a`. Once `size` and `coefficients` are implemented, a default implementation of `expand` will take care of the expressions in the form of `Const`, `Input`, `Var` and `Let`. The rest of



the cases will be implemented in `expandSpec`, such as the binary operations. The instance declaration of `ExtensionField k poly` looks like this:

```
type instance SubType (ExtensionField k poly) = k

instance Expandable (ExtensionField k poly)
  where
  size _ = deg (pvalue (undefined :: (k, poly))) - 1
  coefficients (Ext (UP xs)) = xs
  expandSpec (Add x y) = coefficients $ x′ + y′ where
    x′ = Ext (UP (expand x)) :: ExtensionField (Exp k) poly
    y′ = Ext (UP (expand y)) :: ExtensionField (Exp k) poly
  expandSpec (Sub x y) = ...
  expandSpec (Mul x y) = ...
```

If the programmer is dealing with algebraic structures other than the built-in ones, they can simply make an instance declaration, and then the expansion will be taken care of.

Chapter 5

Compiling Embedded Language

5.1 Intermediate Representation

After algebraic structure expansion, the expression representing users' program is translated into an intermediate representation that is more suitable for optimizations before generating target code. The intermediate language we use is *three-address code* (TAC). Besides, it is in *static single assignment form* (SSA). In three-address code, a complicated expression will be broken down into many separate instructions. They can be translated easily to different target languages, including assembly languages. It is also easier to detect common subexpressions.

Each TAC in Haskell looks like this:

```
type Address = Int

data IR = ConstI Address String
        | AddI Address Address Address
        | SubI Address Address Address
        | MulI Address Address Address
```

`Address` are symbolic addresses of each operand, and will later be translated to actual addresses. A TAC instruction $t_3 := t_1 + t_2$ will be `AddI 3 1 2`. The IR no longer has its own type variable like an expression, so the constants are stored in `String` form. Instruction $t_4 := 0$ is represented as `ConstI 4 "0"`

5.2 Expressions to IR

We replace the Input's String field with two addresses marking the begin and the end of the input's address:

```
data Exp a = ...
           | Input Address Address
```

Since the instructions are SSA, each instruction comes with a new symbolic address, and they are enumerated sequentially. We use the *State* monads to keep track of the address number and the variable mapping introduced by the Let expression.

5.2.1 Type Class

We defined the type class Compilable to make the function toIR polymorphic.

```
type Env = [(String, [Address])]

class Compilable a where
  toIRSpec :: Exp a -> State (Address, Env) ([IR], [Address])
  toIR :: Exp a -> State (Address, Env) ([IR], [Address])
  toIR (Input start end) = return ([], [start..end])
  toIR (Let (Var x) e1 e2) = ...
  toIR (Var x) = ...
  toIR e = toIRSpec e
```

The function toIR is a stateful computation. The state is the used address number so far, and the environment Env, which is an association list, storing the map from variable name to the address of the expression. The results of toIR are a list of three-address code, and a list of addresses, where the results of this expression are stored. There may be multiple addresses because a may be multidimensional.

5.2.2 Default Implementation

A default implementation is provided for several cases. Compiling Input is simply returning the addresses. Compiling Var is looking up the variable name in the environment. Compiling Let is a little more complicated:

```
toIR (Let (VarE x) e1 e2) = do
  (ir1, res1) <- toIR e1
  (i, env) <- get
  put (i, (x, res1):env)
  (ir2, res2) <- toIR e2
  (i', env') <- get
  put (i', delete (x, res1) env')
  return (ir1 ++ ir2, res2)
```

What the above code does is:

1. Compile e1.

2. Add the mapping from `x` to the resulting address of `e1` to the environment.

3. Compile e2.

4. Delete `x` from the environment, since its scope is over.

5. Return the IR and addresses, note that the result of the entire `Let` expression is the results of the body, `res2`.

If there are two or more variables with the same name, the newest one will be closer to the head of the association list, which will be returned by the `lookup` function. The older binding will be *shadowed* until the scope of the new one is over. This is consistent with normal programming languages.

5.2.3 Instance Example

The cases not implemented in `toIR` should be implemented in `toIRSpec` in the instance declaration. The instance declaration of extension field looks like this:

```
instance (PolynomialAsType k poly)
  => Compilable (ExtensionField k poly) where
    toIRSpec (Const ...) = ...
    toIRSpec (Add e1 e2) = ...
    toIRSpec (Sub e1 e2) = ...
    toIRSpec (Mul e1 e2) = ...
```

Compiling a constant of an extension field is to recursively compile each of its coefficients. The `Add`, `Sub` and `Mul` are expanded as described in the last chapter.

The recursion will boil down to the base field. The instance declaration of the base field is:

```
instance Compile (Fp a) where
  toIRSpec (Add e1 e2) = do
    (ir1, [res1]) <- toIR e1
    (ir2, [res2]) <- toIR e2
    i <- newAddress
    return (ir1++ir2++[Add i res1 res2], [i])
  toIRSpec (Sub e1 e2) = ...
  toIRSpec (Mul e1 e2) = ...
  toIRSpec (Const c) = getConstant (show c)
```

Compiling a binary operation of `e1` and `e2` is to compile them each, generate a new address, and append an instruction at the end. We always generate new addresses for new results, so if the target code is an assembly with limited number of registers, the IR should go through register allocation first.

Our compiler deals with constants in a special way. The function `getConstant` will append a `ConstI` instruction when the constant is asked for the first time, and stores the address in the environment. The next time the constant is asked for, the address will be returned.

```
getConstant s = do
  (i, env) <- get
  case lookup s env of
    Just res -> do
      return ([], res)
    Nothing -> do
      put (i+1, (s, [i]):env)
      return ([ConstI i s], [i])
```

Chapter 6

Optimizations and Code Generation

Now that we have an intermediate representation, we can perform optimizations on it. The programmer can also implement customized optimizations, as long as it takes the IR as input, and output an IR as well.

6.1 Common Subexpression Elimination

The multiplication of extension field elements is essentially polynomial multiplication. Karatsuba algorithm [KO63] is a way to reduce the number of operations in polynomial multiplications. Alternatively, Chen has developed a tool using Max-SAT to reduce the number of operations in the multiplications of polynomials in binary fields [Che14], and implemented an optimizer for our compiler. His tool takes our expanded expression as input, and output the IR, on which we can do further optimizations and code generations.

6.2 Linear Register Allocation

In the intermediate representation, we could use arbitrarily many variables, but in the assembly, we only have a small, finite set of registers to use. Memory accesses slow down the program, so register allocation is very important.

We use an implementation of *linear scan* register allocation by Yang [Yan13]. It is also implemented in Haskell, and the representation is very similar to our IR, so it is easy to be combined with our compiler.

In Hydra, in order for the operands of a binary operation to load in a single cycle, they should come from two different register banks. The register allocation algorithm by Yang was designed for ARM processors, so it did not have this constraint, so we end up wasting some time loading the operands. We leave this part to future work.

6.3 Code Generation

6.3.1 Hydra

Translating three-address code to assembly is pretty straightforward, but there are a few things to notice. The operations in Hydra are designed to modulo a large prime, set by the instruction `setrn`.

Montgomery Multiplication

The `mul` instruction performs *Montgomery multiplication* [Mon85], which given the operands a and b, calculates:

$$c = a \times b \times R^{-1} \pmod N,$$

where $R = 2^{256}$. The Montgomery algorithm makes it faster than a naive modular multiplication, $c = a \times b \pmod N$.

Each input and each constant should be transformed to the *residue*, defined by:

$$\bar{a} = aR \pmod N,$$
$$\bar{b} = bR \pmod N.$$

Addition and subtraction are the same. If $c = a + b$, then

$$\bar{c} = cR = (a + b)R = aR + bR = \bar{a} + \bar{b} \pmod{N}.$$

Now if $c = a \times b$, then

$$\bar{c} = cR = (a \times b)R = (aR \times bR)R^{-1} = (\bar{a} \times \bar{b})R^{-1}.$$

So we can perform all the operations in the residue form, and convert the results of the computations back by

$$c = \bar{c}R^{-1} \pmod{N}.$$

6.3.2 C++

We also implemented a code generation to produce C++ code.

Chapter 7

Applications

We have implemented two following applications.

7.1 Pairing

Let G_1, G_2 be additive groups, and G_T be multiplicative groups. A pairing is a map of the form $e : G_1 \times G_2 \to G_T$, where G_1, G_2 are additive groups and G_T is a multiplicative group. The following properties should hold:

1. Bilinear: For all $P \in G_1, Q \in G_2$ and for all $a, b \in \mathbb{Z}$, $e(aP, bQ) = e(P, Q)^{ab}$.

2. Non-degenerate: There exists $P \in G_1$ and $Q \in G_2$ such that $e(P, Q) \neq 1$.

3. Computable: Given $P \in G_1, Q \in G_2$, there is an efficient algorithm, to compute $e(P, Q)$.

Pairing can been used to construct identity-based encryption.

We implement the optimal pairing [Ver10], and compile it to Hydra assembly. The program in our DSL is about 300 lines, and the generated assembly contains about 2×10^6 instructions. The compilation takes about 4 minutes on a 4.4Hz AMD Phenom(tm) 9550 Quad-Core Processor, which is long but still tolerable, as we do not emphasize compile time. Speeding up the compilation process is left for future work.

7.2 Key Exchange Protocol from LWE

Learning with errors (LWE) is a problem that is as hard as several lattice problems. The problem is to distinguish polynomially many noisy inner-product samples $(a, b \approx \langle a, s \rangle)$ from uniformly random samples.

There is a provably secure key exchange protocol based on LWE [DL12]. Alice and Bob have secret keys $s_A, s_B \in \mathbb{Z}_q^n$ respectively. There are public parameters $M \in \mathbb{Z}_q^{n\times n}$. They send each other $p_A = Ms_A + e_A$ and $p_B = M^T s_B + e_B$. Upon receiving p_B, Alice computes $s_A^T p_B$, and Bob computes $s_B^T p_A$. The two values are very close to $s_A^T M^T s_B$, and a shared secret can be derived.

We implement the protocol, and generate codes in C++. Hydra cannot handle the protocol, because it requires random number generation. We need to call the library in C to generate random numbers. The target code length and compilation time are proportional to the square of vector size.

Chapter 8

Summary

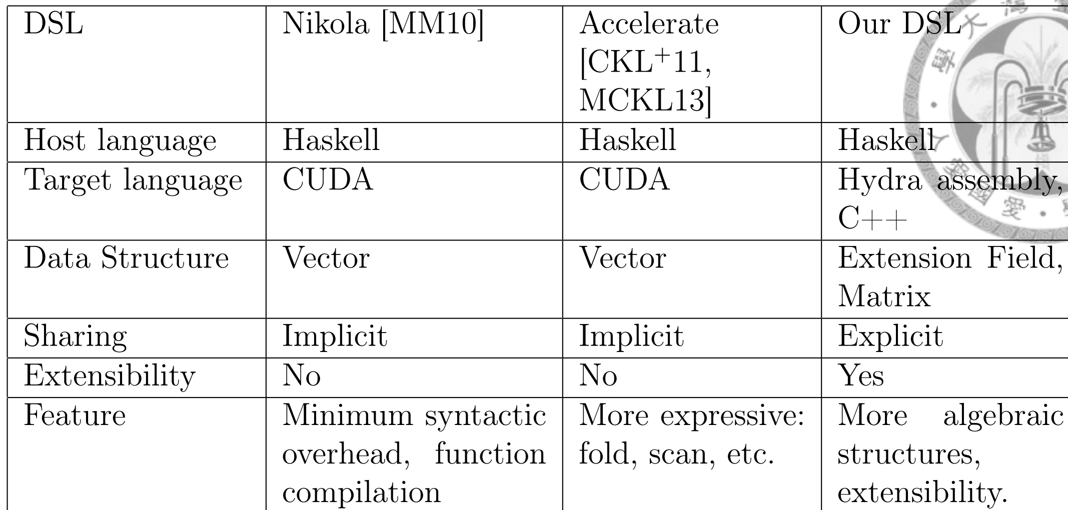
We present a domain-specific language embedded in Haskell and an embedded compiler that enables the programmer to write programs in high-level language, and using complex algebraic structures. We show that sharing is vital for tree-structured expressions, and use explicit let-sharing. Type classes are defined for the programmers to declare customized algebraic structure expansion, other than the built-in extension fields and matrices. We provide two applications with two target code generations, and our compiler is combined with two optimizers. The optimizations and code generation are separated from the program itself, so the programmer can focus on the high-level description of the cryptographic systems.

8.1 Related Work

See the comparison in table 8.1.

8.2 Future Work

- Implicit sharing. Allow programmers to use the native Haskell binding, and let the compiler figure out the expressions to be shared.
- Loops and functions. Making the expression able to represent loops and func-



DSL	Nikola [MM10]	Accelerate [CKL+11, MCKL13]	Our DSL
Host language	Haskell	Haskell	Haskell
Target language	CUDA	CUDA	Hydra assembly, C++
Data Structure	Vector	Vector	Extension Field, Matrix
Sharing	Implicit	Implicit	Explicit
Extensibility	No	No	Yes
Feature	Minimum syntactic overhead, function compilation	More expressive: fold, scan, etc.	More algebraic structures, extensibility.

Table 8.1: Comparison with related work

tion calls could reduce the target code size significantly.

- More optimizations. The possible ways are:
  - Implement more optimizations to our intermediate representation.
  - Use Hoopl, a Haskell tool for dataflow analysis and code transformation.
  - Try to connect our compiler with the middle-end and back-end of LLVM.
- Use the syntactic library [Axe12] to make our compiler extensible without modifying the existing code.
- Shorten the compilation time. When we build this compiler, the compilation time is not our first concern. Still it is better to reduce the compilation time from minutes to seconds. A possible way is to replace the lists with difference lists to save time for append and concatenation.

Bibliography

[Axe12] Emil Axelsson. A generic abstract syntax model for embedded languages. In *ACM SIGPLAN Notices*, volume 47, pages 323–334. ACM, 2012.

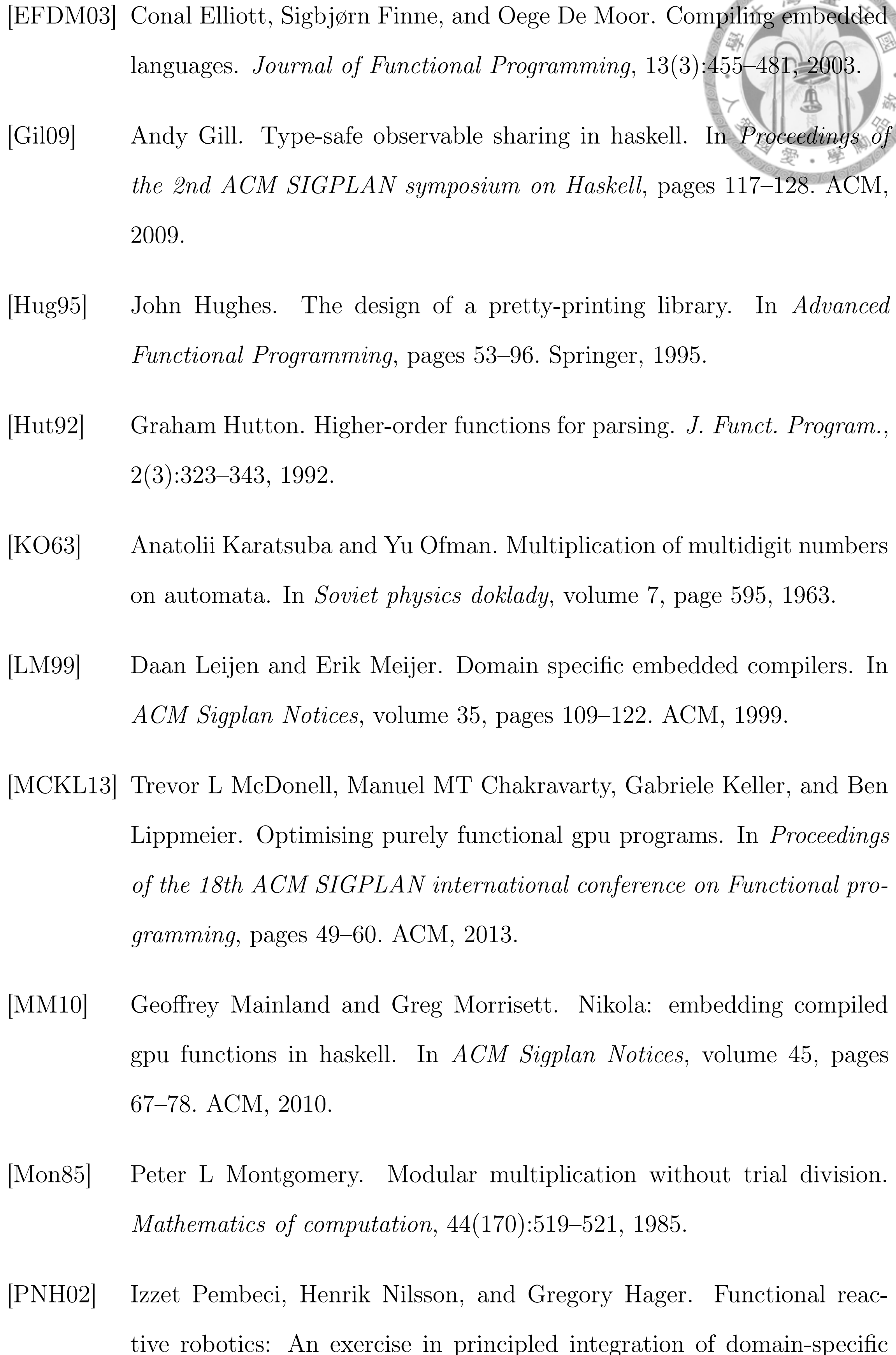
[BCSS98] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: hardware design in haskell. In *ACM SIGPLAN Notices*, volume 34, pages 174–184. ACM, 1998.

[Che14] W.-H. Chen. A simplification tool for expressions over binary fields using max-sat solver. Master's thesis, National Taiwan University, June 2014.

[CHH+] Y.-A. Chang, W.-C. Hong, M.-C. Hsiao, B.-Y. Yang, A.-Y. Wu, and C.-M. Cheng. Hydra: An energy-efficient programmable cryptographic coprocessor supporting elliptic-curve pairing over fields of large characteristics. To appear in the 9th International Workshop on Security (IWSEC 2014), Hirosaki, Japan, Aug. 2014.

[CKL+11] Manuel MT Chakravarty, Gabriele Keller, Sean Lee, Trevor L McDonell, and Vinod Grover. Accelerating haskell array codes with multicore gpus. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, pages 3–14. ACM, 2011.

[DL12] Jintai Ding and Xiaodong Lin. A simple provably secure key exchange scheme based on the learning with errors problem. *IACR Cryptology ePrint Archive*, 2012:688, 2012.



[EFDM03] Conal Elliott, Sigbjørn Finne, and Oege De Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(3):455–481, 2003.

[Gil09] Andy Gill. Type-safe observable sharing in haskell. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 117–128. ACM, 2009.

[Hug95] John Hughes. The design of a pretty-printing library. In *Advanced Functional Programming*, pages 53–96. Springer, 1995.

[Hut92] Graham Hutton. Higher-order functions for parsing. *J. Funct. Program.*, 2(3):323–343, 1992.

[KO63] Anatolii Karatsuba and Yu Ofman. Multiplication of multidigit numbers on automata. In *Soviet physics doklady*, volume 7, page 595, 1963.

[LM99] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *ACM Sigplan Notices*, volume 35, pages 109–122. ACM, 1999.

[MCKL13] Trevor L McDonell, Manuel MT Chakravarty, Gabriele Keller, and Ben Lippmeier. Optimising purely functional gpu programs. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, pages 49–60. ACM, 2013.

[MM10] Geoffrey Mainland and Greg Morrisett. Nikola: embedding compiled gpu functions in haskell. In *ACM Sigplan Notices*, volume 45, pages 67–78. ACM, 2010.

[Mon85] Peter L Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.

[PNH02] Izzet Pembeci, Henrik Nilsson, and Gregory Hager. Functional reactive robotics: An exercise in principled integration of domain-specific

languages. In *Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 168–179. ACM, 2002.

[SHH+13] Jie-Ren Shih, Yongbo Hu, Ming-Chun Hsiao, Ming-Shing Chen, Wen-Chung Shen, Bo-Yin Yang, An-Yeu Wu, and Chen-Mou Cheng. Securing m2m with post-quantum public-key cryptography. *IEEE J. Emerg. Sel. Topics Circuits Syst.*, 3(1):106–116, 2013.

[Ver10] Frederik Vercauteren. Optimal pairings. *Information Theory, IEEE Transactions on*, 56(1):455–461, 2010.

[Yan13] S.-Y. Yang. Code generation for fast pseudo-mersenne prime field arithmetic on arm processors. Master's thesis, National Taiwan University, July 2013.