Hashing Algebraic Datatypes

Ashutosh Upadhye

2018-04-12

Contents

Abstract

In this report I propose a way to cryptographically hash algebraica datatypes. The Hashing protocol essentially involves converting the algebraic datatype to a Sakura tree hash coding. Cryptographic hashing of algebraic data types is a nontrivial problem. In order to Hash algebraic data types in Haskell, one must ensurethat different values of same data types yield different hashes while ensuring thatthe same hash is not generated by any value of some other data type.

Acknowledgement

I would like to express my sincere gratitude to my supervisor Dr. Piyush Kurur for providing their invaluable guidance, comments and suggestions throughout the course of the project. I would like to especially thank Dr. Piyush Kurur for constantly motivating me to work harder. Also, I would like to thank the Haskell community which helped me when I got stuck with non trivial issues.

1. Introduction

Algebraic Datatypes is the ultimate gift of Functional Programming. There are many real world applications of Algebraic Datatypes which make programming intuitive, and efficient. There are many Real life applications of hashed data structures and hashing protocols. If we find a way to hash Algebraic Datatypes, we could essentially hash every data structure that can be represented as Algebraic Datatypes in Haskell.

Cryptographic Hash of algebraic data types has a multitude of applications ranging from networking to blockchains. Cryptographic hashing of algebraic data types is a non trivial problem. In order to Hash algebraic data types in Haskell, one must ensure that different values of same data types yield different hashes while ensuring that the same hash is not generated by any value of some other data type.

A way to hash any given data type is using the `Hash . Serialize` function, but this generates trivial collisions and the ultimate goal of cryptographic hashing is to avoid such collisions.

1.1. Organization of the report

Following section discusses related work and builds a background to Hashing and Algebraic datatypes. The next section shows an implementation of Sakura tree encoding followed by a wrapper to generate Merkle hash of a given string along with briefs on methods of validating the implementation.

2. Algebraic Data Types

2.1. Introduction

In computer programming, especially functional programming and type theory, an algebraic data type is a kind of composite type, i.e., a type formed by combining other types. Algebraic datatypes in Haskell have one or more constructors. Each data constructor can have zero or more arguments. The definitions can be recursive too.

One can pattern match over the constructors. Pattern matching is essentially matching values against patterns. Apart from allowing one to match patterns, algebraic datatypes also bind the variables to succesful matches.

2.2. Generic Representation of Algebraic Datatypes

A datatype can have *parameters*, *alternatives* and *fields*

```haskell
data D p = Alt1 | Alt2 Int p
```

2.2.1. Alternatives

Alternatives are often called as **sums**. A typical datatype concisting only of alternatives is shown below.

```haskell
    data AltEx = A1 Int | A2 Char
```

The Alternatives are very similar to another datatype `Either`. We use a similar datatype, `:+:`to represent alternatives generically.

```haskell
    data a :+: b = L a | R b
```

This can also be used to represent types with more than 2 alternatives. For example the following datatype `AltEx2`,

```haskell
data AltEx2 = B1 Int | B2 Char | B3 Float
```

could be easily repsresented using nesting as follows.

```haskell
type AltEx2 = Int :+: (Char :+: Float)
-- Note the smart constructors:
b1 :: Int -> AltEx2
b1 = L
b2 :: Char -> AltEx2
b2 = R. L
b3 :: Float -> AltEx2
b3 = R . R
```

2.2.2. Fileds

Fields are often called as **products**. A typical datatype consisting only of fields is shown below.

```haskell
data FldEx = FldEx Int Char
```

The Fields are very similar to another datatype, the pair, `(,)` function. We use a similar datatype, `:*:` to represent fields generically.

```haskell
data a :*: b = a :*: b
```

This can also be used to represent types with more than 2 alternatives. For example the following datatype `FldEx2`,

```haskell
data FldEx2 = FldEx2 Int Char Float
```

could be easily represented using nesting as follows.

```haskell
type FldEx2' = Int :*: (Char :*: Float)
-- note the smart constructor.
fldEx2' :: Int -> Char -> Float -> FldEx2'
fldEx2' x y z = x :*: (y :*: x)
```

2.2.3. Sum of Products

Algebraic Datatypes in Haskell could now be represented generically as sums of products as follows. The datatype `D` that takes a parameter `p` is defined as follows.

```haskell
type D p = Alt1 | Alt2 Int p
```

For this datatype `D`, we can define an identical datatype `RepD` as follows.

```haskell
type RepD p = U :+: Int :*: p
```

Here, we use *unit* type `data U = U`, *(identical to standard type,* `()`*)* to represent an alternatice without fields. Also, the precedence order of `:+:` is infix 5 and that of `:*:` is infix 6, hence we need not use the parantheses.

3. Hashing

A hash function is any function that can be used to map data of arbitrary size to data of a fixed size. The values returned by a hash function are called hash values, hash codes, digests, or simply hashes. Hash functions are often used in combination with a hash table, a common data structure used in computer software for rapid data lookup. Hash functions accelerate table or database lookup by detecting duplicated records in a large file. One such application is finding similar stretches in DNA sequences. They are also useful in cryptography. A cryptographic hash function allows one to easily verify that some input data maps to a given hash value, but if the input data is unknown, it is deliberately difficult to reconstruct it (or any equivalent alternatives) by knowing the stored hash value. This is used for assuring integrity of transmitted data, and is the building block for HMACs, which provide message authentication.

> Pour the initial value in a big cauldron and place it over a nice fire. Now slowly add salt if needed and stir well. Marinade your input string by appending some strengthened padding. Now chop the resulting bit string into nice small pieces of the same size and stretch each piece to at least four times its original length. Slowly add each single piece while continually stirring at the speed given by the rotation constants and spicing it up with some addition constants. When the hash stew is ready, extract a portion of at least 128 bits and present this hash value on a warm plate with some garnish.
>
> *- Attacks on hash functions and applications, by Marc Stevens, University Leiden, 2012*

3.1. Constructing a Hash Function

Figure 1 represents a method used to create Hashes of input strings. The method is called Merkle Dangard construction.

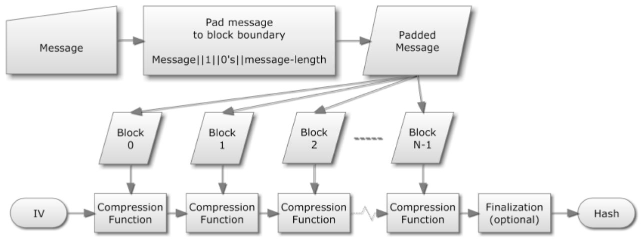

Figure 1: Constructing a Hash Function.

3.2. Tree Hashing

Merkle and others have proposed a method to authenticate any leaf with respect to the hash at the root with a logarithmic number of hash computations.

It enables parallel computation for validation and an incremental update to the root hash after a leaf changes.

3.3. Merkle Tree

Figure 2 shows a simple Merkle tree. The black nodes represent the update sequence if node corresponding to the blue box is changed.

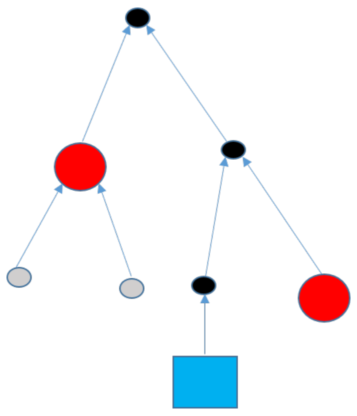

Figure 2: Merlke Tree

3.4. Sakura

Sakura is a tree hash mode which is more flexible than other tree hash modes. In Sakura you can have multiple modes of trees.

More mathematically, Sakura can be defined as following.

```
Sakura :: Mode -> Innerhash function -> Input -> Hash
```

Sakura takes Mode and innerhash function as parameters, along with the input string that needs to be hashed. A hashing mode can be seen as a recipe for computing digests over messages by means of a number of calls to an underlying function. The hashing mode splits the message into substrings that are assembled into inputs for the inner function

3.4. Collisions in Tree Hashing

Trees in *hashing mode* could be of any shape. Figure 3 represents one such illustration.

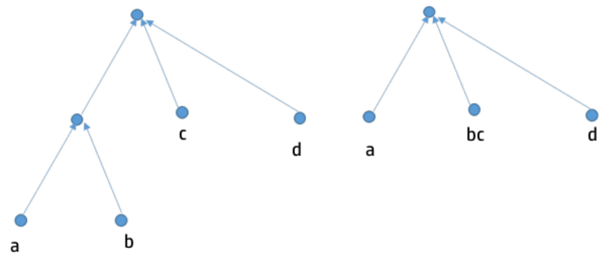

Figure 3: Shapes of trees representing hashing modes.

We need to ensure not to have trivial collisions when having multiple shapes. Trivial collisions are the ones that allows one generate same hashes for two different values. One such collision is illustrated in Figure 4.

3.5. How Sakura Hashing works

We represent trees in terms of hops that model how message and chaining values are distributed over nodes. There are two distinct types of hops: message hops that contain only message bits and chaining hops that contain only chaining values.

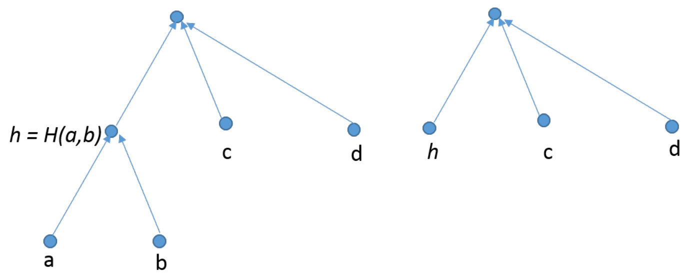

Figure 4: Trivial collisions.

The hops form a tree, with the root of the tree called the final hop. Such a hop tree determines the parallelism that can be exploited by processing multiple message hops or chaining hops in parallel.

An example hop tree from Sakura is shown in figure 5. The Encoding for the hop tree is represented in Figure 6.

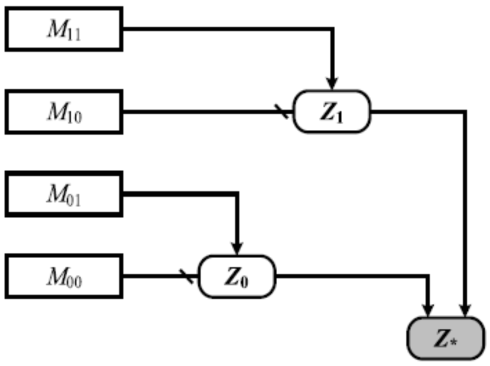

Figure 5: Hop Tree

In Figure 2 there are in total 7 hops: 4 message hops M_{00} , M_{01} , M_{10} , M_{11} , and three chaining hops Z_0 , Z_1 and Z_*. The final node contains only the final hop Z_*. The hops M_{00} and Z_0 are in a single node. Similarly, M_{10} and Z_1 are in a single node. The total number of nodes is 5.

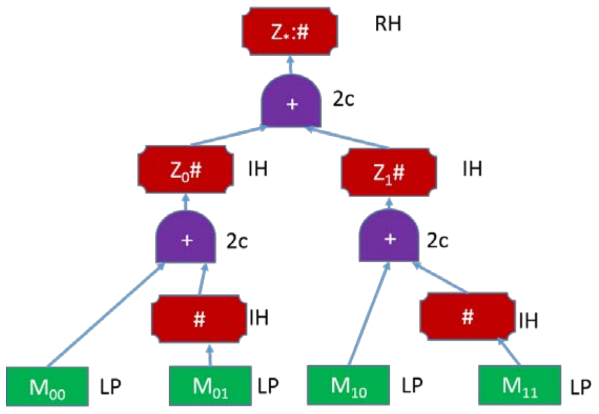

Figure 6: Encoded Tree

3.6. Sakura Implementation

3.6.1. Capturing the Shape

The shape of Sakura Tree can be captured as follows:

```
data HShape = InnerHash HShape
    | Concat [HShape]
    | Interleaving [HShape]
    | Slice Int Int
    | Pad BStr
```

3.6.2. Serial Hash Computation

Renaming types.

```
    type BStr = [Word8]
    type HashF = [Word8] -> [Word8]
```

Slices a BStr from from to to.

```
    my_slice :: Int -> Int -> BStr -> BStr
    my_slice from to = (drop from).(take to)
```

Computes the hash for a given string and a HShape.

```
    s :: HashF -> HShape -> BStr -> BStr
    -- Serial Hash Function
    s h (InnerHash aShape) bStr = h $ s h aShape bStr
    s h (Concat l) bStr = concat $ map (\x -> s h x bStr) l
```

```
s _ (Slice from to) bStr = my_slice from to bStr
s _ (Pad x) _ = x
```

3.6.3. Parallel Hash Computation

In above definition, the computation of hashes in Concat l can be parallelised as follows:

```
p :: HashF -> HShape -> BStr -> BStr
-- Parallel Hash Function
p h (InnerHash aShape) bStr = h $ p h aShape bStr
p h (Concat l) bStr =
concat $ parMap rpar (\x -> p h x bStr) l
p _ (Slice from to) bStr = my_slice from to bStr
p _ (Pad x) _ = x
```

3.6.4. Wrapper for Sakura

Chunker for Merkle Tree

We need to create a HShape for a given string and a given block size.

```
chunker :: Int -> Int -> BStr -> BStr -> HShape
chunker n size innerpad rootpad =
  let
    b = quot n size
    make_node i = InnerHash (Concat[(Slice (i*size) ((i+1)*size)), (Pad innerpad)])
    ranges = map make_node [0 .. b]
    all_ranges =
      if rem n size == 0 then
        ranges
      else
        ranges ++ [InnerHash (Concat [(Slice (b*size) n), (Pad innerpad)])]
  in
    InnerHash (Concat (all_ranges ++ [(Pad rootpad)]))
```

a_block_mode

```
a_block_mode x block_size = chunker (length x)
    block_size (c2w8 i_padding) (c2w8 r_padding)
```

Now to calculate hash of any string,

```
hash_val = (s hashf (a_block_mode string 1)) string
```

3.6.5. Validation of the implementation

Can we really validate if a hashing protocol is actually working? The code could be validated by using id function as the hash function and comparing the expected output with the obtained output. Following are the results of validation of the simple HShapes:

```
    b1 : [97,98,99,100]
    b1': [97,98,99,100]
```

To validate the Merkle shape chunker, I created the Merkle tee from Scratch and Padded bits as and where required, as per the protocol. The following are the results.

```
chunker:   [97,73,98,73,99,73,100,73,101,73,102,73,103,73,104,82]
from tree: [97,73,98,73,99,73,100,73,101,73,102,73,103,73,104,82]
```

4. Generic Programming in Haskell

4.1. Introduction

Generic programming is a style of computer programming in which algorithms are written in terms of types to-be-specified-later that are then instantiated when needed for specific types provided as parameters. This approach, pioneered by ML in 1973, permits writing common functions or types that differ only in the set of types on which they operate when used, thus reducing duplication. Such software entities are known as generics in Python, Ada, C#, Delphi, Eiffel, F#, Java, Rust, Swift, TypeScript and Visual Basic .NET.

Generic programming is a powerful way to define a function that works in an analogous way for a class of types. Most of the programming languages have some sort of generic programming. Some are listed below.

- Generics in Java / C#
- Templates in C++
- Generic packages in Ada

The goal of generic programming is often the same, that is to achieve a higher level of abstraction than “normally” available.

The technique is also often the same: some form of parametrization and instantiations.

Example of Generic programming in Java / C#.

```
public class Stack<T>
{
    public void push (T item) {..}
    public T pop () {..}
}
```

Example of Generic Programming in C++.

```
template <typename T, typename Compare>
T & min (T&a, T&b, Compare comp) {
    if (comp (b, a))
        return b;
    return a;
}
```

4.2. Generic Programming in Haskell

4.2.1. Parametric Polymorphism

In programming languages and type theory, parametric polymorphism is a way to make a language more expressive, while still maintaining full static type-safety. Using parametric polymorphism, a function or a data type can be written generically so that it can handle values identically without depending on their type.

In Haskell, parametric polymorphism refers to when the type of a value contains one or more (unconstrained) type variables, so that the value may adopt any type that results from substituting those variables with concrete types.

For example,

```
id  :: a -> a
map :: (a -> b) -> [a] -> [b]
```

`a` and `b` are unconstrained types and could be any kind of type, making the functions `id` and `map` more generic in some sense.

This is similar to generic programming in Java and C#. Parametric polymorphism is native to Haskell and so we don't call it generic programming.

4.2.2. Ad-hoc Polymorphism

Ad-hoc polymorphism refers to when a value is able to adopt any one of several types because it, or a value it uses, has been given a separate definition for each of those types. For example, the + operator essentially does something entirely different when applied to floating-point values as compared to when applied to integers – in Python it can even be applied to strings as well. Most languages support at least some ad-hoc polymorphism, but in languages like C it is restricted to only built-in functions and types. Other languages like C++ allow programmers to provide their own overloading, supplying multiple definitions of a single function, to be disambiguated by the types of the arguments. In Haskell, this is achieved via the system of type classes and class instances.

In Haskell, this is achieved via the system of type classes and class instances.

So, for example, if my type can be compared for equality (most types can, but some, particularly function types, cannot) then I can give an instance declaration of the `Eq` class. All I have to do is specify the behaviour of the `==` operator on my type, and I gain the ability to use all sorts of functions defined using that operator, e.g. checking if a value of my type is present in a list, or looking up a corresponding value in a list of pairs.

```
class Eq a where
    (==) :: a -> a -> Bool
```

This is similar to generic programming in C++ templates. Type classes make ad hoc polymorphism sort of native to the language, so we don't call it generic programming in Haskell.

4.2.3. Polytypism: Shape/Structure polymorphism.

Can there be a higher level of abstraction? Can we abstract over isomorphic types, the ones that have similar representations?

When we abstract over the shapes and structures of the datatypes, it is known as polytypism or shape/structural polymorphism.

4.3. Generic Representation of Algebraic Datatypes

A datatype can have *parameters*, *alternatives* and *fields*

```
data D p = Alt1 | Alt2 Int p
```

4.3.1. Alternatives

Alternatives are often called as **sums**. A typical datatype concisting only of alternatives is shown below.

```
data AltEx = A1 Int | A2 Char
```

The Alternatives are very similar to another datatype `Either`. We use a similar datatype, `:+:`to represent alternatives generically.

```
data a :+: b = L a | R b
```

This can also be used to represent types with more than 2 alternatives. For example the following datatype `AltEx2`,

```
data AltEx2 = B1 Int | B2 Char | B3 Float
```

could be easily repsresented using nesting as follows.

```
type AltEx2 = Int :+: (Char :+: Float)
-- Note the smart constructors:
b1 :: Int -> AltEx2
b1 = L
b2 :: Char -> AltEx2
b2 = R. L
b3 :: Float -> AltEx2
b3 = R . R
```

4.3.2. Fileds

Fields are often called as **products**. A typical datatype consisting only of fields is shown below.

```
data FldEx = FldEx Int Char
```

The Fields are very similar to another datatype, the pair, `(,)` function. We use a similar datatype, `:*:` to represent fields generically.

```
data a :*: b = a :*: b
```

This can also be used to represent types with more than 2 alternatives. For example the following datatype `FldEx2`,

```
data FldEx2 = FldEx2 Int Char Float
```

could be easily represented using nesting as follows.

```
type FldEx2' = Int :*: (Char :*: Float)
-- note the smart constructor.
fldEx2' :: Int -> Char -> Float -> FldEx2'
fldEx2' x y z = x :*: (y :*: x)
```

4.3.3. Sum of Products

Algebraic Datatypes in Haskell could now be represented generically as sums of products as follows. The datatype `D` that takes a parameter `p` is defined as follows.

```
type D p = Alt1 | Alt2 Int p
```

For this datatype `D`, we can define an identical datatype `RepD` as follows.

```
type RepD p = U :+: Int :*: p
```

Here, we use *unit* type `data U = U`, *(identical to standard type,* `()`*)* to represent an alternatice without fields. Also, the precedence order of `:+:` is infix 5 and that of `:*:` is infix 6, hence we need not use the parantheses.

4.3.4. Isomprphism

In order to prove that RepD correctly represents D we need to construct an isomorphism An isomorphism is a way of casting types into each other without loss of information. It is essentially an equivalence relation between the types.

So, for RepD and D we define the following isomorphism.

```
fromD :: D p -> RepD p
fromD Alt1 = LU
fromD (Alt2 i p) = R (i :*: p)
toD :: RepD p D p
toD (L U) = Alt1
toD (R (i :*: p)) = Alt2 i p
```

This isomorphism allows us to convert from RepD to D without any loss of information, and hence RepD is a valid representation of D.

4.3.5. Type representation: Metadata

The above mentioned way of representation is compleplete but we can add more information in the representation like Constructors and other metadata.

```
data C a = C String a
```

And, RepD can now be more informative.

```
type RepD p = C U :+: C (Int :*: p)
fromD Alt1 = L (C "Alt1" U)
fromD (Alt2 i p) = R (C "Alt2" (i :*: p))
```

4.4. Generic Functions

A function that is defined on each possible case of the structural representation of datatypes is a generic function. A generic function will work for every type that has an isomorphism to the generic representation of the datatypes.

Let's look at a generic function show.

4.4.1. show: a Generic Function

The type signature of show is as follows.

```
show :: a -> String
```

In order to completely define `show` we need to define it for all possible structure cases.

- Unit:

  ```
  showU :: U -> String
  showU U = ""
  ```

- Constructor name:

  ```
  showC :: (a -> String) -> C a -> String
  showC sA (C name a) = "(" ++ name ++ " " ++ sA a ++ ")"
  ```

- Binary Product:

  ```
  showP :: (a -> String)->(b -> String) -> a:*:b -> String
  showP sA sB (a:*:b) = sA a ++ " " ++ sB b
  ```

- Binary Sum:

  ```
  showS :: (a -> String)->(b -> String) -> a:+:b -> String
  showS sA _ (L a) = sA a
  showS _ sB (R b) = sB b
  ```

Now `show` for RepD can be defined as:

```
-- assumming showInt is known.
showRepD :: (p -> String) -> RepD p -> String
showRepD sP = showS (showC showU) (showC (showP sInt sP))
```

The representation has a fairly predictable pattern. The above functions are sort of recursive, but with differing arguments.