Hashing Algebraic Datatypes

B Tech Project

Ashutosh Bharat Upadhye

Supervisor: Dr. Piyush P Kurur

DThash

A Haskell Package to Cryptographically Hash Algebraic Datatypes

Flow

Algebraic datatypes -> Generic Representation -> Hash of the Generic Representation

Algebraic DataTypes

- Has one or more data constructors.
- Each data constructor can have zero or more arguments.
- Can be recursive too.
- Pattern matching:
  - Match values against patterns.
  - Bind variables to successful matches.

Example

```haskell
data Shape = Rectangle Int Int
           | Square Int
```

```haskell
area :: Shape -> Int
area (Rectangle len breadth) = len * breadth
area (Square side) = side * side
```

```haskell
rec = Rectangle 3 4
main = print $ area rec
```

Some Data Structures in Functional Programming

Binary Tree

```
BTree a = NullBTree
        | BNode a (BTree a) (BTree a)
```

Rose Tree

```
RTree a = NullRTree
        | RNode a [RTree a]
```

List

```
List a  = Nil
        | Cons a (List a)
```

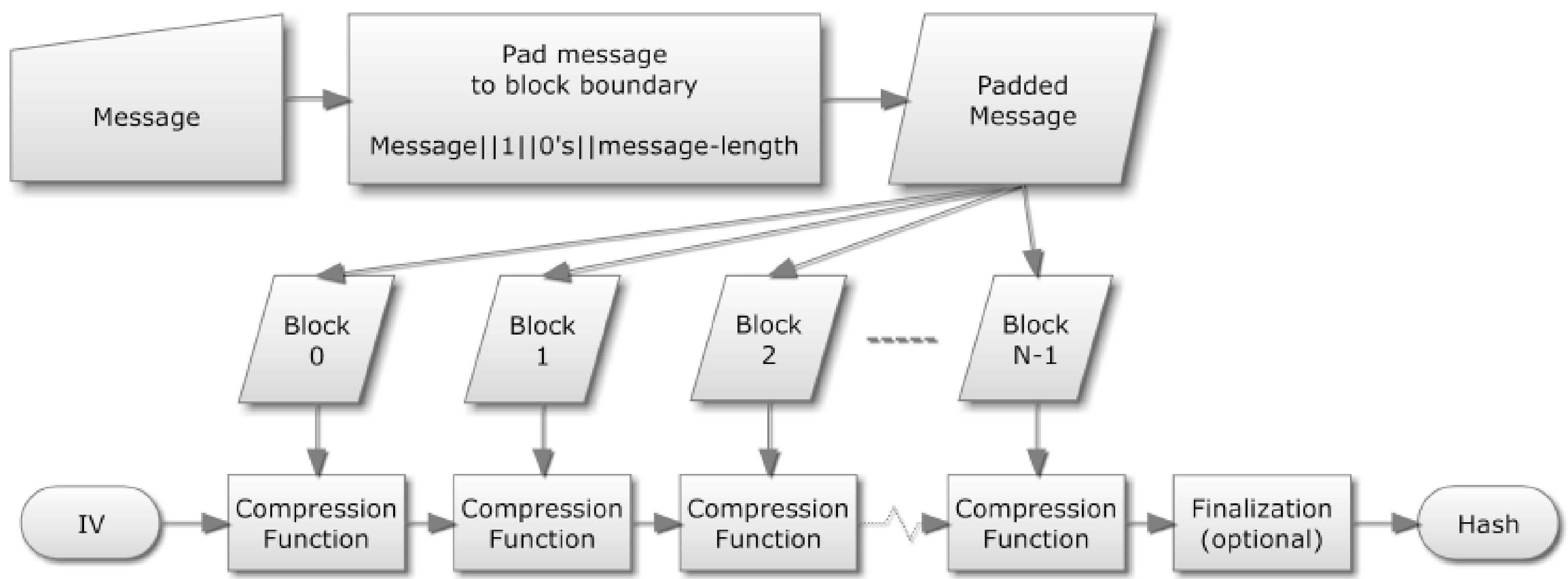
Hashing *a quick lookback*

A hash function h with some interesting properties.

$$h : \{0,1\}^* \to \{0,1\}^n$$

- It is extremely easy to calculate $h(x)$.
- It is extremely computationally difficult to calculate $h^{-1}(y)$.
- It is extremely unlikely that two slightly different messages have the same hash.

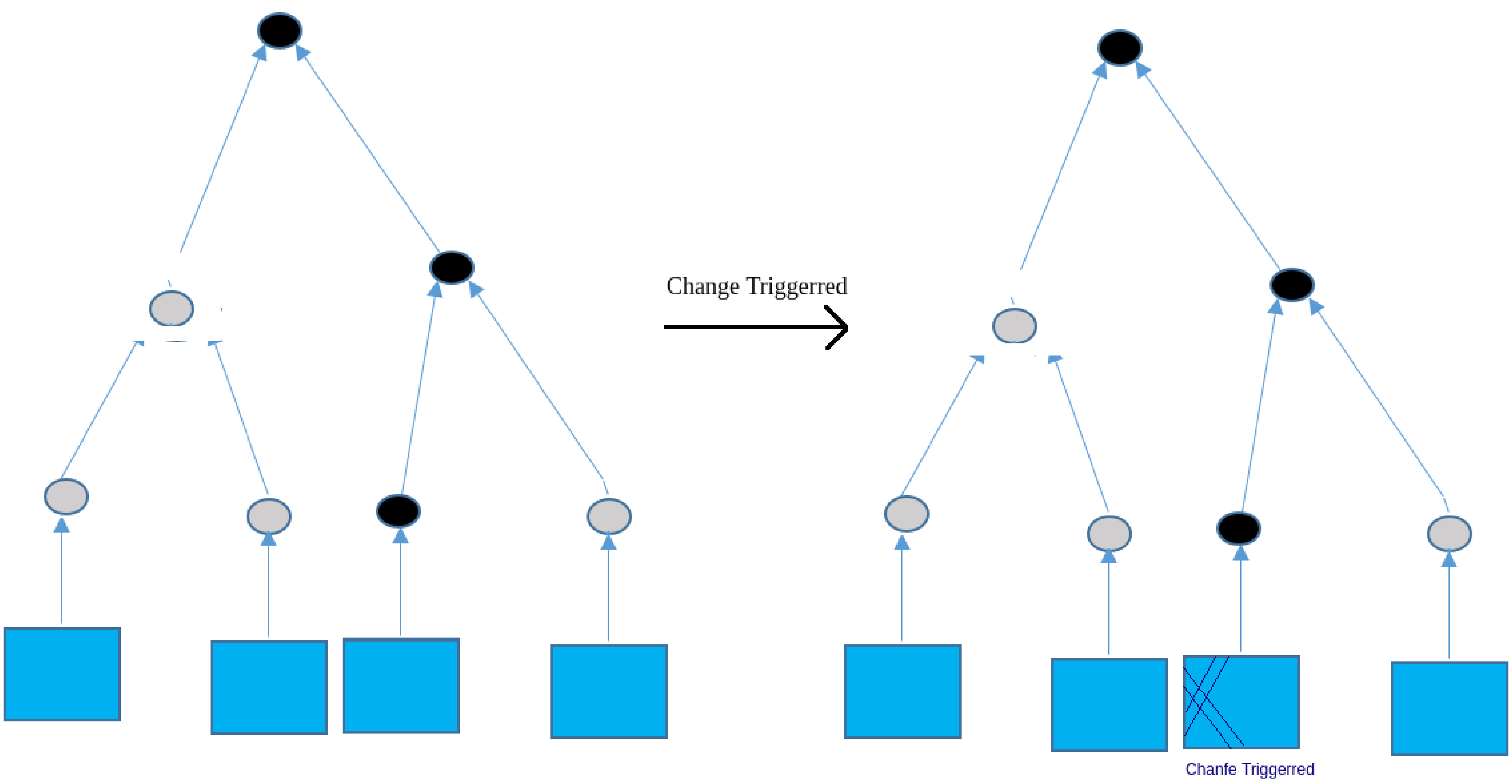
Constructing a Hash Function



Tree Hashing

- Merkle (1980): authenticate any leaf w.r.t. the hash at the root with a logarithmic number of hash computations.
- Enables:
    - Parallel Computation of nodes.
    - Incremental update to the root-hash after a leaf changes. *(old hash values are stored on the nodes.)*

Merkle Tree

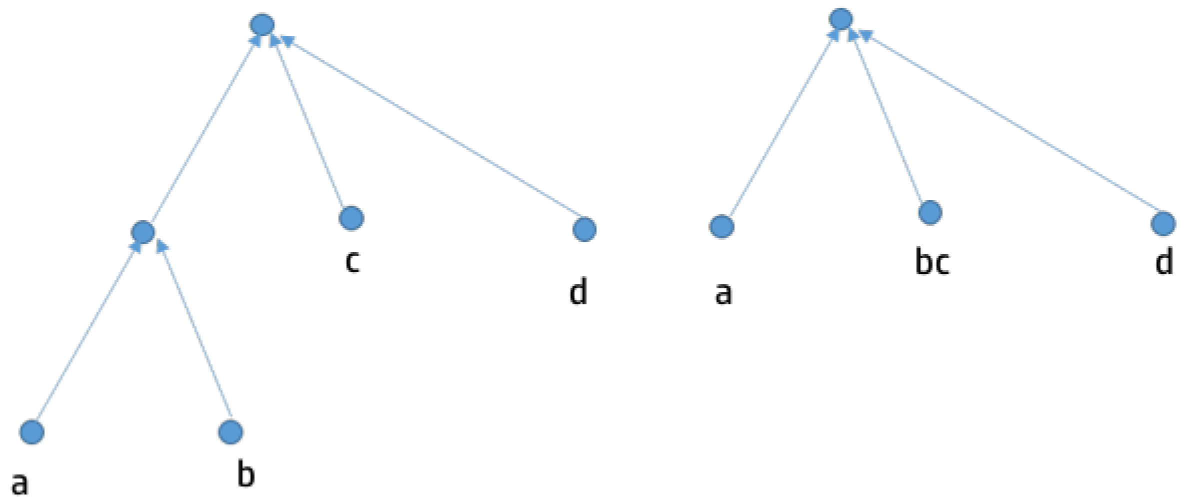


About Sakura

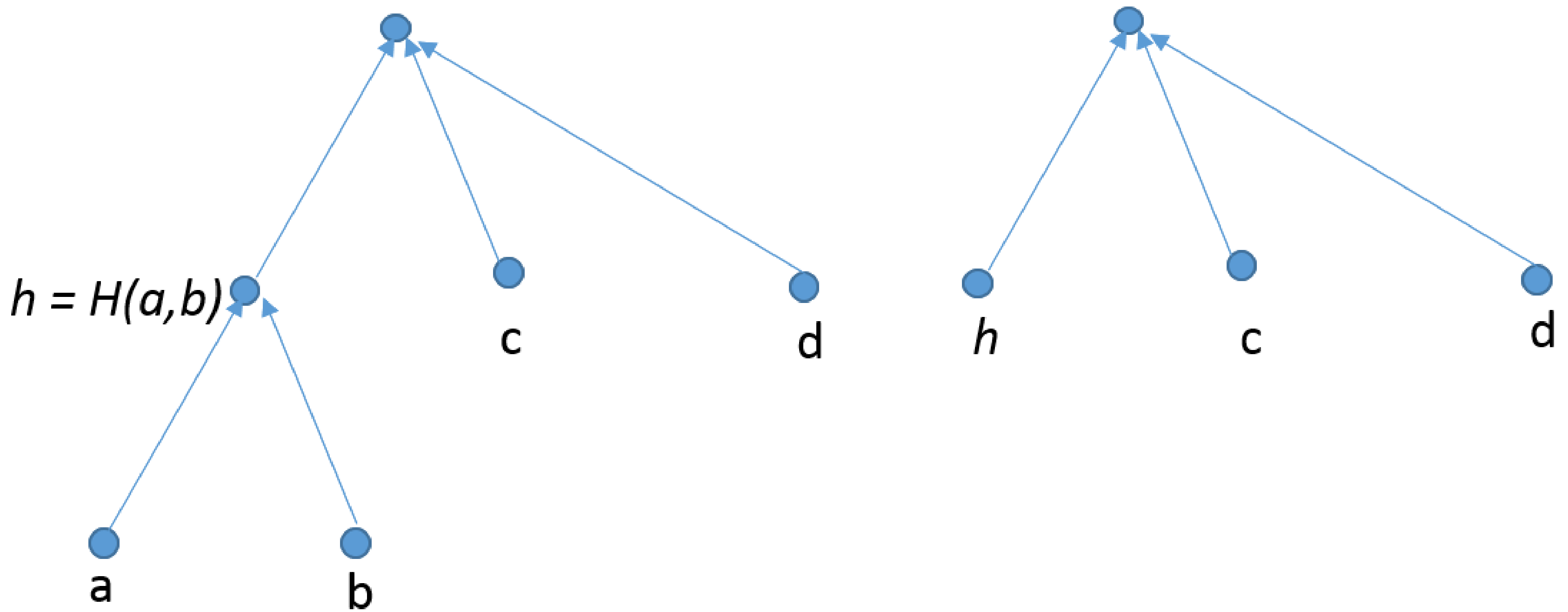
- Tree Hash Mode.
  - More flexible than other tree hash modes.
  - multiple shapes of trees possible.
- Takes an inner hash function as a paramater.

$$Sakura :: Mode \rightarrow Innerhashfunction \rightarrow Input \rightarrow hash$$

Tree Shapes



Stupid Collision



`abcd` and `hcd` should not have the same tree hash, otherwise, you have a collision.

Detailed Example *Hop Tree*

An example hop tree from *Sakura*.

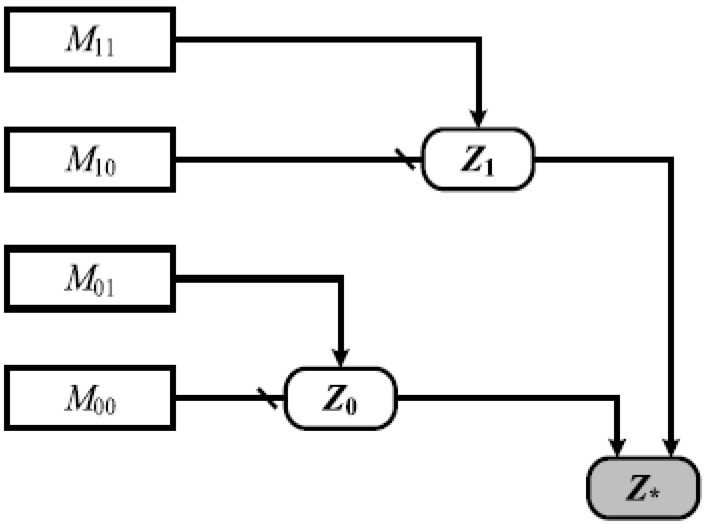
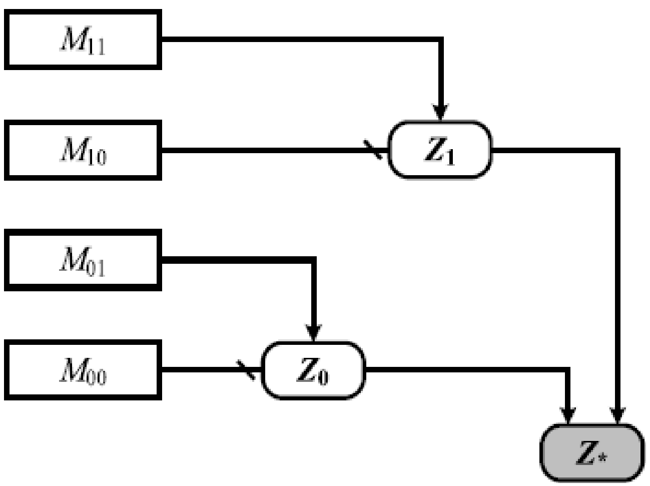
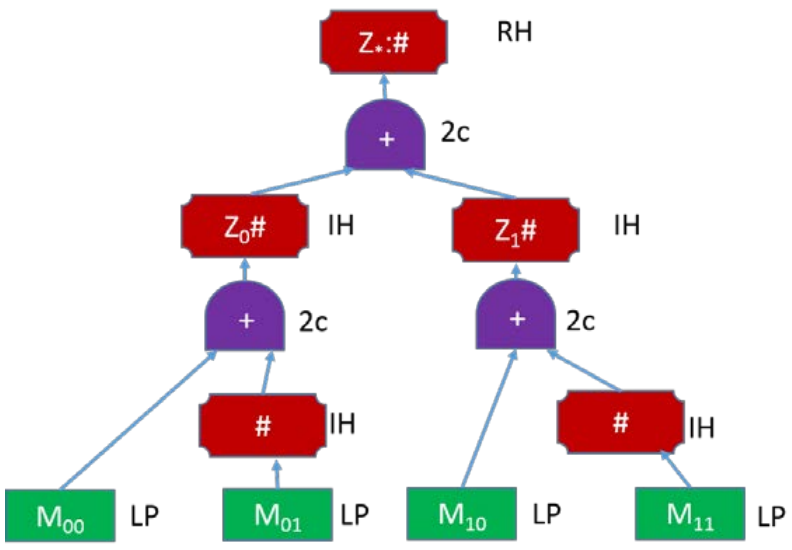


Image from Sakura Paper

Encoding the Example





Implementation of Sakura

Capturing the Shape

- InnerHash, Concat, Interleave, Slice, Pad

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

Serial Hash Computation

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

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

```
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
```

Generic Programming

What is Generic Programming?

the adjective "generic" is heavily overloaded!

- Java / C# generics
- C++ templates
- Ada generic packages

Generic Programming

- Java-stype generics ~ parametric polymorphism
- C++ templates ~ ad-hoc polymorphism

In Haskell:

- Both forms of polymorphism already exist.
- We don't call them generics because they are sort of native to the language.

Generic Programming: Haskell.

Datatype-generic Programming:

- Abstract over the structure of the datatype.
- Also known as "polytypism" and "shape / structure polymorphism".

Algebraic DataTypes and Generics

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

A datatype can have:

- Parameters: type variables (≥ 0)
- Alternatives: unique constructors (≥ 0)
- Fields: types for each constructor (≥ 0)

Structure of Datatypes: Sums and Products

Alternatives are often called as **sums**. We use another *identical* sum type to represent it, instead of Either.

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

The pair type is the basic binary product type. We use the following identical type instead.

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

Structure of Datatypes: Sums of Products

To "sum" it all up, recall the first example.

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

We can define an identical type using the sum and product types.

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

Notes:

- We use *unit* type data U = U, *(identical to standard type ())* to represent an alternatice without fields.
- `:+:` is `infix 5` and `:*:` is `infix 6`, so no parantheses.

Structure of Datatypes: Metadata

The representation lacked any information about the constructors (e.g. the names).

That's easily repaired with another datatype:

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

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

Encoding Isomorphisms

The type class:

```
class Generic a where
  type Rep a
  from :: a -> Rep a
  to :: Rep a -> a
```

- `Rep` is a type family, an associated type synonym.
- `Rep` can be thought as a function on types. Given a unique type (index) `T` you get a type synonym, `Rep T`.
- Also, two datatypes may have the same representation.

Polymorphic Recursion

We can encode polymorphic recursion in several ways. Most obvious one is the type classes.

- Standard classes already use polymorphic recursion for deriving instances.
- Class declaration specifies type signature.
- Each recursive case can be specified by an instance of the class.

GHashable class:

```
class GHashable hashf f where
    gcomputeHash :: hashf -> f a -> BStr
```

Polymorphic Recursion: ComputeHash

- Unit:

```
instance GHashable hashf U1 where
    gcomputeHash hash U1 = hashf (toWord8 "U1")
```

- Binary Product:

```
instance (GHashable hashf a, GHashable hashf b) =>
GHashable hashf (a :*: b) where
    gcomputeHash hashf (a :*: b) =
        concat [gcomputeHash hashf a,
                gcomputeHash hashf b,
                toWord8 "Pdt1"]
```

Polymorphic Recursion: ComputeHash

- Binary sum:

```
instance (GHashable hashf a, GHashable hashf b) =>
GHashable hashf (a :+: b) where
    gcomputeHash hashf (L1 x) =
        concat [gcomputeHash hashf x,
                toWord8 "Sum1L1"]
    gcomputeHash hashf (R1 x) =
        concat [gcomputeHash hashf x,
                toWord8 "Sum1R1"]
```

- Metadata:

```
instance (GHashable a) => GHashable (M1 i c a) where
    gcomputeHash hashf (M1 i c a) =
        concat [gcomputeHash hashf (i a),
                toWord8 c]
```

Polymorphic Recursion

`showRepD` without polymorphic recursion:

```
hashRepD :: (p -> BStr) -> RepD p -> BStr
hashRepD sP = hashS (hashC hashU)
                    (hashC (hashP hashInt hashP))
```

Compare to new version that's possible.

```
hash'RepD :: hashFun ->RepD p -> String
hash'RepD = gcomputeHash
```

Generic Show Function

Finally,

```
gshow :: (Show (Rep a), Generic a) => a -> String
gshow = show . from
```

DThash

Let's look at the code now.

Future Works

- It is serial computation of Hash. It doesn't essentially generates a HShape and then hashes the bitstream representation of the datatype. It can be parallelised.
- Datatype -> Representation -> HShape -> ComputeHash