B Tech Project

Ashutosh Bharat Upadhye

Supervisor: Dr. Piyush P Kurur

Functional Programming in Haskell

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

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

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

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

taken from 'Bloc: Library for handling large binary objects in Haskell' by Anshu Avinash

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.

Cryptographic Hash Functions *under the hood*

> 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.'**

'Attacks on Hash functions and Applications' by Marc Stevens, Universiteit Leiden, 2012.

Constructing a Hash Function

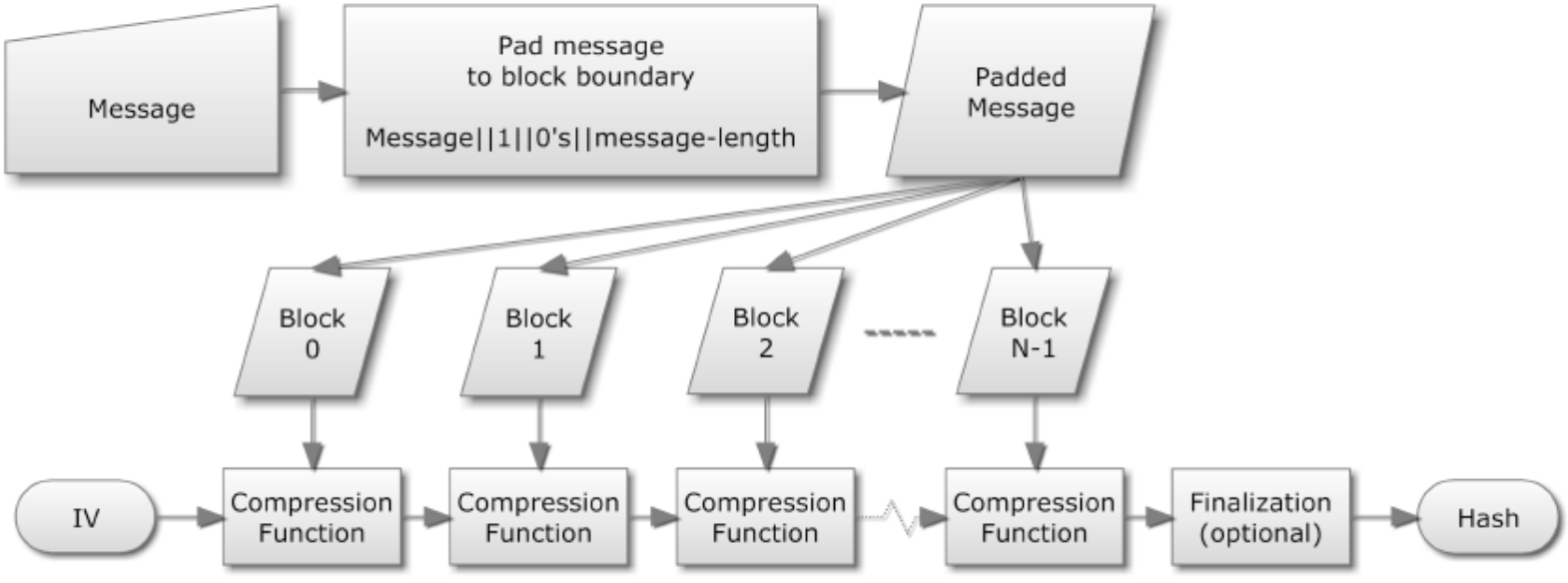
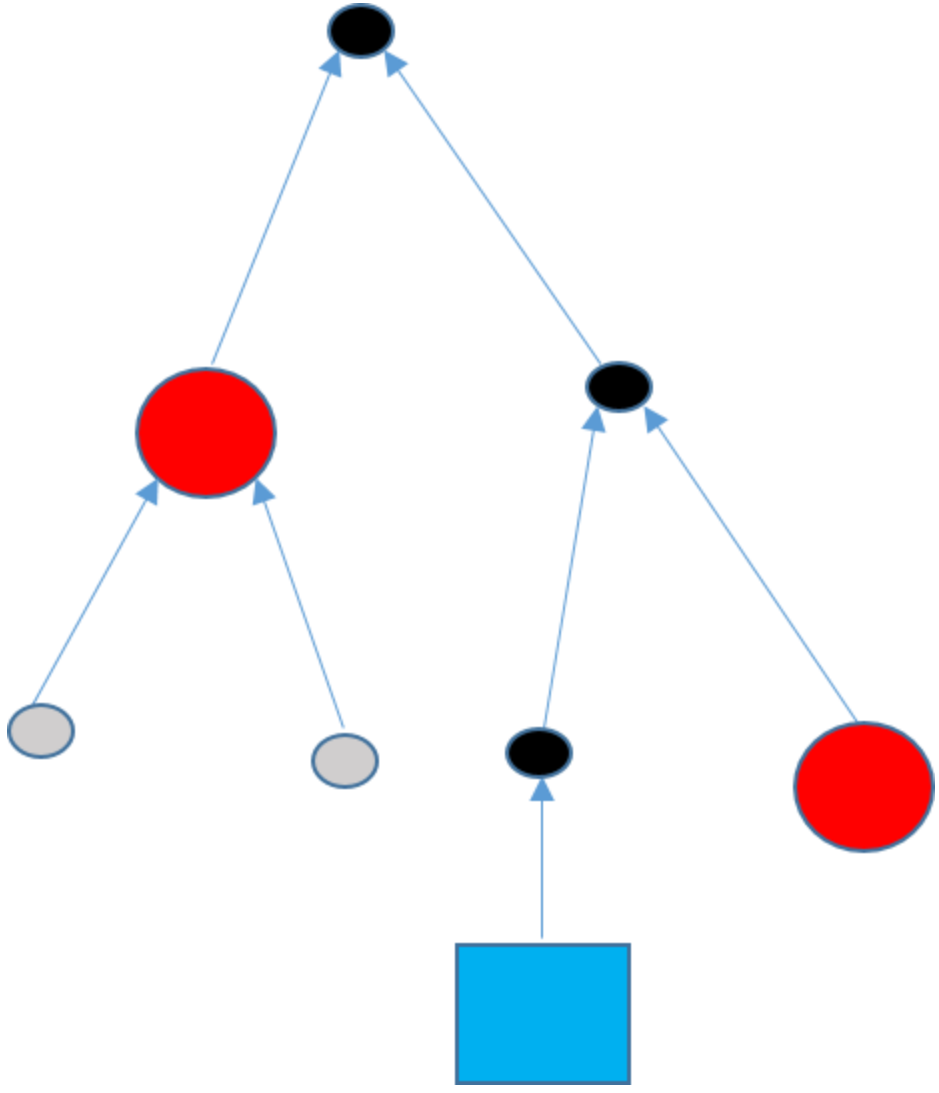


Figure 3: *Merkle-Damgård Construction*

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

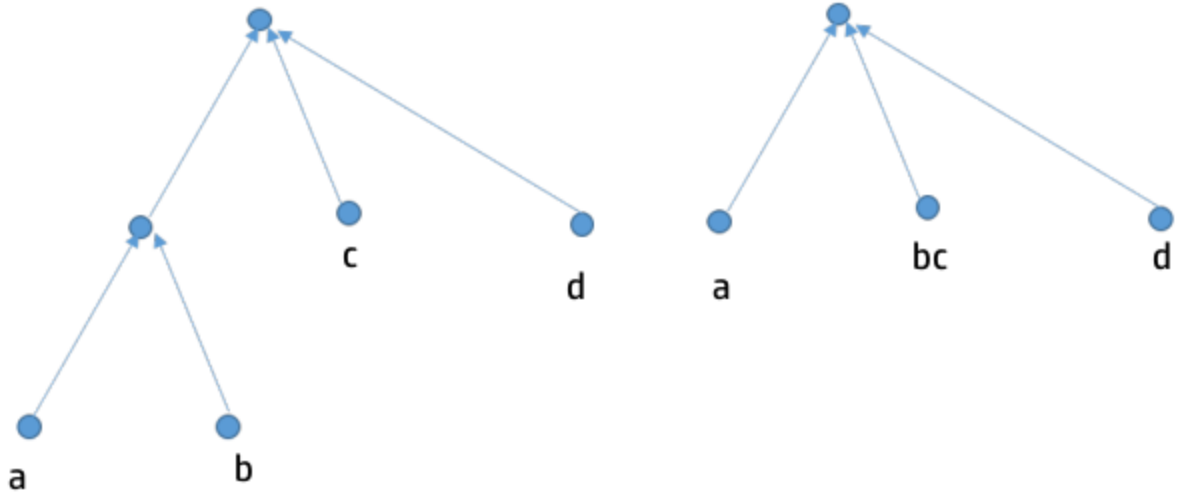


About Sakura

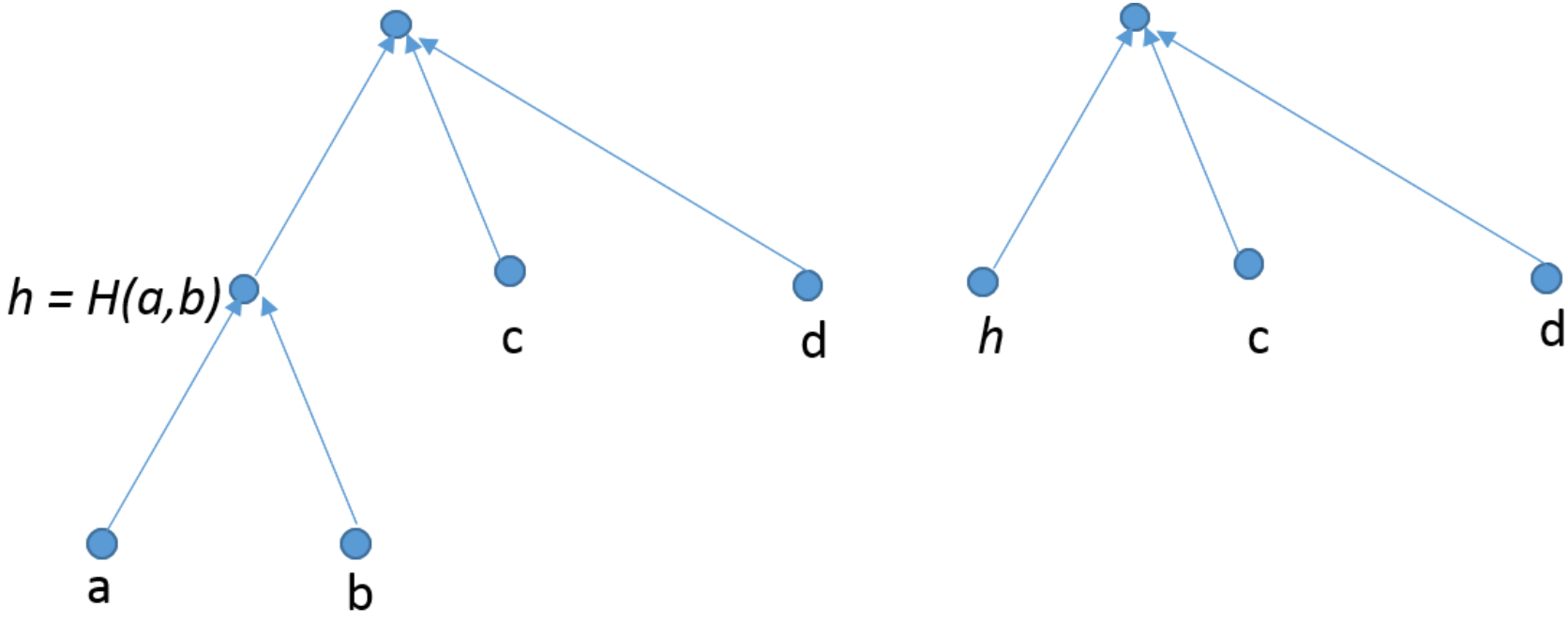
- Tree Hash Mode.
  - More flexible than other tree hash modes.
  - multiple shapes of trees possible.
- Takes an inner hash function as a paramater.

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

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*.

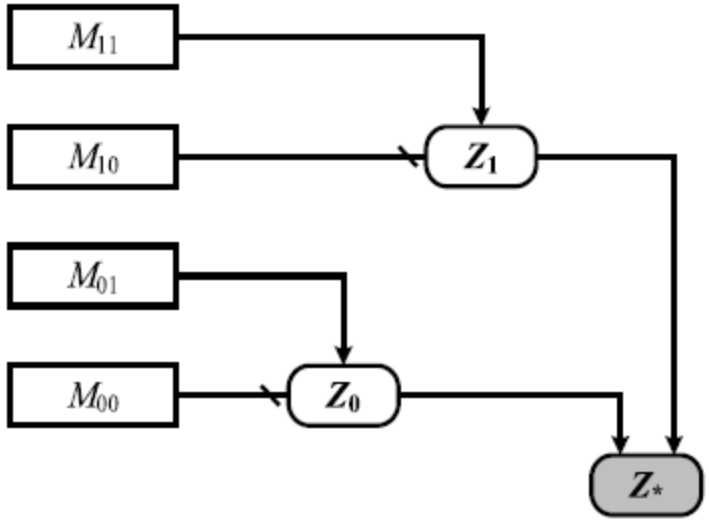
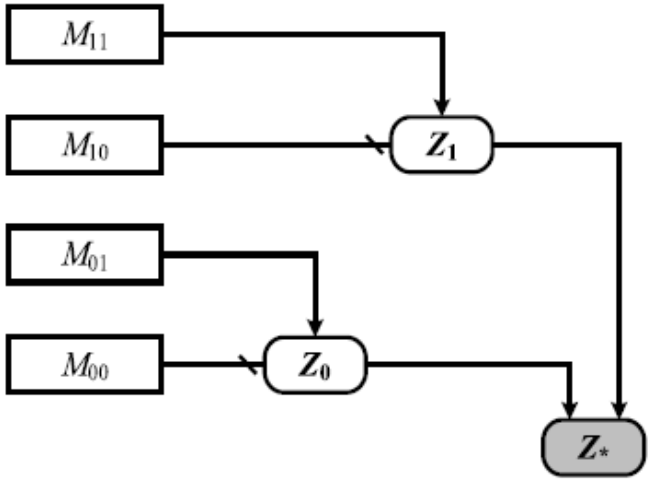
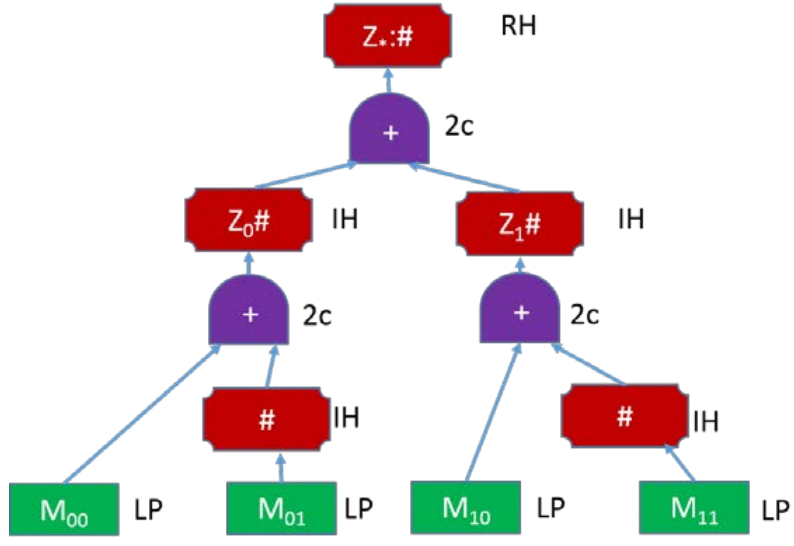


Image from Sakura Paper

Encoding the Example





Implementation of Sakura

Capturing the Shape

- InnerHash, Concat, Interleave, Slice, Pad

```
data HShape = InnerHash HShape
            | Concat [HShape]
            | Interleaving [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
```

Parallel Hash Computation

```
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
```

Sakura Wrapper for a Block Mode

chunker

```
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
```

Validating Hashing functions

Can we really validate if a hashing protocol is actually working?

Using identity function as the hash function!

```
id_hash x = x       -- Identity function as hash funt.

ipad = Pad (c2w8 i_padding)
rpad = Pad (c2w8 r_padding)

shape0 = (Slice 0 4)
-- just a slice of bit string
shape1 = (Concat [(InnerHash (Slice 0 4)),
  (InnerHash (Slice 4 8))])
-- Essentially concatanation of first 8 characters
shape2 = (Concat [Concat [(InnerHash (Slice 0 4)),ipad],
  Concat [(InnerHash (Slice 4 8)),ipad],rpad])
-- slightly more complex, but straightforward

b1 = my_slice 0 4 (c2w8 "abcdefgh")
b1' = s id_hash shape0 (c2w8 "abcdefgh")
-- b1 = b1'
hash1 = s id_hash shape1 (c2w8 "abcdefgh")
hash2 = s id_hash shape2 (c2w8 "abcdefgh")
```

What about the Chunker?

> Let's examine sakura.hs.

Algebraic DataTypes and Hashes

Aeson: The JSON parser

Records in Haskell

Records are unnatural in Haskell. Mere *Syntactic Sugar*

Sweet:

```
data Person = Person
            { name :: String
            , age :: Int
            }
```

Unsweet:

```
data Person = Person String Int
name :: Person -> String
name (Person name _) = name

age :: Person -> Int
age (Person _ age) = age
```

Records to JSON

```haskell
instance ToJSON Person where
    -- this generates a Value
    toJSON (Person name age) =
        object ["name" .= name, "age" .= age]

    -- this encodes directly to a bytestring Builder
    toEncoding (Person name age) =
        pairs ("name" .= name <> "age" .= age)
```

```haskell
instance FromJSON Person where
    parseJSON = withObject "Person" $ \v -> Person
        <$> v .: "name"
        <*> v .: "age"
```

Well, what is special?

You don't actually have to define those functions explicitly, the Aeson module provides a way to do it.

```
genericToEncoding ::(Generic a, ..)=>Options->a-> Encoding
toEncoding = genericToEncoding defaultOptions
```

and to decode,

```
decodeStrict :: FromJSON a => ByteString -> Maybe a
parseJSON = decodeStrict
```

instance Typable

```
class Typeable a where
    typeOf :: a -> TypeRep
```

```
cast :: (Typeable a, Typeable b) => a -> b
cast x
  | typeOf x == typeOf (undefined :: b) =
      Just (unsafeCast x)
  | otherwise = Nothing
```

Algebraic DataTypes and Hashes

Type Families *and Data Families*

```
class Add a where
    plus :: a -> a -> a
```

Should the return type be a? b?

```
class Add a b where
    plus :: a -> b -> ???

instance Add Integer Double where
    plus x y = fromIntegral x + y

instance Add Double Integer where
    plus x y = x + fromIntegral y
```

Solution: Type level functions

```
class Add a b where
    type SumTy a b
    plus :: a -> b -> SumTy a b

instance Add Integer Double where
    type SumTy Integer Double = Double
    plus x y = fromIntegral x + y

instance Add Double Integer where
    type SumTy Double Integer = Double
    plus x y = x + fromIntegral y

instance (Num a) => Add a a where
    type SumTy a a = a
    plus x y = x + y
```

Basic Idea

```haskell
-- Type family is used here
class Hashable a where
    data ID a
    type Node a
    hash :: Node a -> ID a
```

```haskell
instance Hashable a => Hashable [a] where
    ID [a] = ListID
    Node [a] = Nil
             | (ID a) : (ID [a])
```

A Novel Idea

```
class Hashable a where
    data ID a
    type Node a
    hash :: Node a -> ID a
    toHShape :: a -> HShape
```

Possible map:

```
HMap a = case (typeRep a) of
  "primitive" -> Concat [
      (InnerHash("typeOf_a")),
      (InnerHash ("slice_a"),
      (Pad premitive_padding)
      ]
  "alg-non-recursive" -> Concat [
      (InnerHash ("constructor-slice")),
      (HMap t1),
      (HMap t2)
      ]
  "alg-recursive" -> Concat []
  -- do we really need another one?
```

Future works:

- Deduce a standard method for mapping Algebraic Datatypes to Sakura tree.
- Check for feasibility. *Concerned because the small implementation I've shown wasn't very pretty*.
- Derive HShape from Typable instances of datatypes.