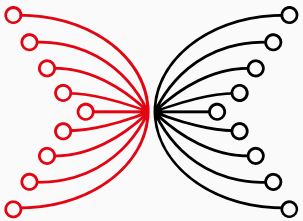
More on data structures

Haskell and Cryptocurrencies

Dr. Lars Brünjes, IOG
Robertino Martinez, IOG
Karina Lopez, IOG

August, 2023



INPUT | OUTPUT

Goals

- Vectors
- Unboxed types
- String-like types
- Binary Serialization
- Merkle Trees

Arrays

Arrays

Sometimes, we want fast (constant time) access to data and compact storage.

Note that sequences get close (operations at a low logarithmic cost), but when speed and space are really an issue, arrays may be better:

- there are simple arrays provided as part of the `array` package;
- the still relatively recent `vector` package is quickly becoming a new favourite, and that's what we'll discuss here.

Persistent arrays and updates

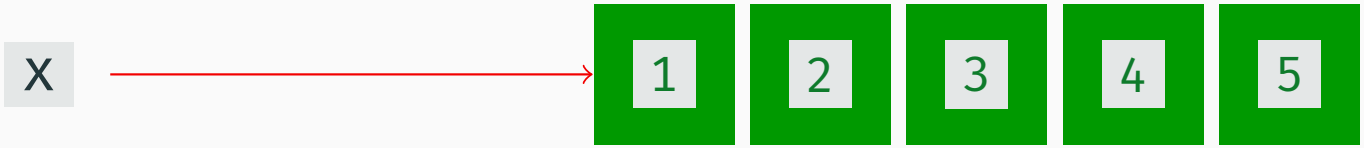
The expression

```
let x = fromList [1, 2, 3, 4, 5] in x // [(2, 13)]
```

evaluates to the same array as
`fromList [1, 2, 13, 4, 5]`.

Question: How expensive is the update operation?

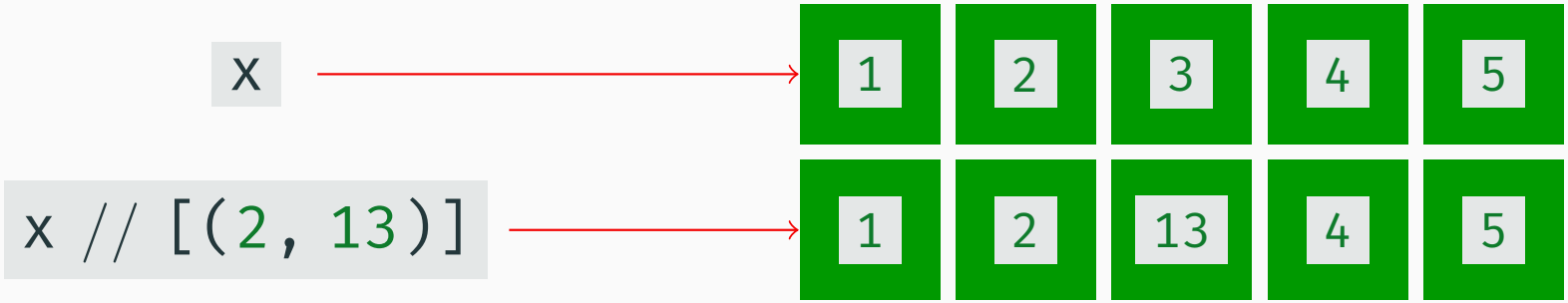
Array updates




```
x // [(2, 13)]
```

- Arrays are stored in a contiguous block of memory.
- This allows O(1) access to each element.
- In an imperative setting, a destructive update is also possible in O(1).

Array updates



- Arrays are stored in a contiguous block of memory.
- This allows O(1) access to each element.
- In an imperative setting, a destructive update is also possible in O(1).
- But if a persistent update is desired, the whole array must be copied, which takes O(n), i.e., linear time.

Advice on persistent arrays

Be careful when using them:

- stay away if you require a large number of small incremental updates – finite maps or sequences are usually much better then;
- arrays can be useful if you have an essentially constant table that you need to access frequently;
- arrays can also be useful if you perform global updates on them anyway.

`Vector` interface

This is an excerpt of `Data.Vector`:

```
data Vector a  -- abstract

empty    :: Vector a                              -- O(1)
generate :: Int -> (Int -> a) -> Vector a         -- O(n)
fromList :: [a] -> Vector a                       -- O(n)

(++)     :: Vector a -> Vector a -> Vector a      -- O(m + n)

(!)      :: Vector a -> Int -> a                  -- O(1)
(!?)     :: Vector a -> Int -> Maybe a            -- O(1)
slice    :: Int -> Int -> Vector a -> Vector a    -- O(1)

(//)     :: Vector a -> [(Int, a)] -> Vector a    -- O(m + n)

map      :: (a -> b) -> Vector a -> Vector b      -- O(n)
filter   :: (a -> Bool) -> Vector a -> Vector a   -- O(n)
foldr    :: (a -> b -> b) -> b -> Vector a -> b   -- O(n)
```

Note the efficient slicing.

Implementation of vectors

- The implementation of the vector library falls back on primitive built-in arrays implemented directly in GHC.
- It additionally stores a lower and upper bound. These are used to implement the slicing operations.

Unboxed types, unboxed vectors

The internals of basic types

```
GHCi> :i Int
data Int = GHC.Types.I# GHC.Prim.Int#
```

The internals of basic types

```
GHCi> :i Int
data Int = GHC.Types.I# GHC.Prim.Int#
```

Aha, so GHC thinks `Int` is yet another datatype?

- The `GHC.Types` and `GHC.Prim` are just module names.
- So there's one constructor, called `I#`.
- And one argument, of type `Int#`.

The internals of basic types

```
GHCi> :i Int
data Int = GHC.Types.I# GHC.Prim.Int#
```

Aha, so GHC thinks `Int` is yet another datatype?

- The `GHC.Types` and `GHC.Prim` are just module names.
- So there's one constructor, called `I#`.
- And one argument, of type `Int#`.

What is an `Int#`?

The internals of basic types (contd.)

To get names like `Int#` even through the parser, we have to enable the `MagicHash` language extension ...

```
GHCi> :i GHC.Prim.Int#
data GHC.Prim.Int#  -- Defined in ‘GHC.Prim’
```

So this one seems to be really primitive.

Boxed vs. unboxed types

The type `Int#` is the type of *unboxed* integers:

- unboxed integers are essentially machine integers,
- their memory representation is just bits encoding an integer.

Boxed vs. unboxed types

The type `Int#` is the type of *unboxed* integers:

- unboxed integers are essentially machine integers,
- their memory representation is just bits encoding an integer.

An `Int` is a *boxed* integer:

- it wraps the unboxed integer in an additional pointer,
- thereby introducing an indirection.

Boxed vs. unboxed types (contd.)

Pro unboxed:

- no indirection,
- faster,
- less space.

Boxed vs. unboxed types (contd.)

Pro unboxed:

- no indirection,
- faster,
- less space.

Pro boxed:

- only boxed types admit laziness,
- normal polymorphism scopes only over boxed types.

Boxing makes all types look alike, making it compatible with suspended computations and polymorphism.

Operations on unboxed types

```
3#    :: Int#
3##   :: Word#
3.0#  :: Float#
3.0## :: Double#
'c'#  :: Char#

(+#)       :: Int#    -> Int#    -> Int#
plusWord#  :: Word#   -> Word#   -> Word#
plusFloat# :: Float#  -> Float#  -> Float#
(+##)      :: Double# -> Double# -> Double#
```

Type representations

The kind of unboxed types

```
GHCi> :k Int#
Int# :: TYPE 'IntRep
GHCi> :k Word#
Word# :: TYPE 'WordRep
GHCi> :k Double#
Double# :: TYPE 'DoubleRep
```

The kind of unboxed types

```
GHCi> :k Int#
Int# :: TYPE 'IntRep
GHCi> :k Word#
Word# :: TYPE 'WordRep
GHCi> :k Double#
Double# :: TYPE 'DoubleRep
```

Actually, `*` is a synonym for `TYPE 'PtrRepLifted`.

The kind of unboxed types

```
GHCi> :k Int#
Int# :: TYPE 'IntRep
GHCi> :k Word#
Word# :: TYPE 'WordRep
GHCi> :k Double#
Double# :: TYPE 'DoubleRep
```

Actually, `*` is a synonym for `TYPE 'PtrRepLifted`.

You can think of `IntRep`, `WordRep`, and `DoubleRep` and `PtrRepLifted` as just kind parameters.

Polymorphism and kinds

Normal polymorphism restricts the kind to lifted types:

```
id :: a -> a
id x = x
```

is actually an abbreviated form of

```
id :: forall (a :: *) . a -> a
id x = x
```

(which requires `ExplicitForAll` and `KindSignatures`).

Kind errors

All these expressions produce kind errors:

```
let x = undefined :: []
3# +# 2
id 3#
[3#]
```

The kind system prevents polymorphic use of unboxed types.

GHCi

```
GHCi> id 3#
Couldn’t match lifted type with an unlifted type
When matching the kind of ‘GHC.Prim.Int#’
```

GHCi

```
GHCi> id 3#
Couldn't match lifted type with an unlifted type
When matching the kind of ‘GHC.Prim.Int#’
```

Nearly all polymorphic functions are only applicable to normal, lifted types.

Levity polymorphism

A select few functions have “levity-polymorphic” types:

```
GHCi> :i ($)
($) ::
  forall (r :: RuntimeRep) a (b :: TYPE r) .
  (a -> b) -> a -> b
```

Function application is compatible with unlifted types if the function returns an unlifted type.

Example

```
unpackInt :: Int -> Int#
unpackInt (I# x) = x
```

This works due to levity-polymorphic `($)`:

```
GHCi> I# (unpackInt $ 3)
3
```

Advice on unboxed types

You very rarely have to use unboxed types directly:

- GHC's optimizer is quite good at removing some unnecessary boxing and unboxing;
- there are libraries that offer good abstractions of internally unboxed values;
- one can instruct GHC to specifically unbox certain values via `UNPACK` pragmas.

Unboxed vectors

The vector package provides unboxed vectors next to the regular, boxed ones:

- provided by the `Data.Vector.Unboxed` module;
- more compact and more local storage;
- restricted w.r.t. the element type.

The `Unbox` class

General user-defined datatypes cannot be unboxed, so somewhat necessarily unboxed vectors are only available for a limited class of element types:

```
class Unbox a where
  ...
```

The `Unbox` class

General user-defined datatypes cannot be unboxed, so somewhat necessarily unboxed vectors are only available for a limited class of element types:

```
class Unbox a where
  ...
```

```
instance Unbox Int
instance Unbox Float
instance Unbox Double
instance Unbox Char
instance Unbox Bool
instance (Unbox a, Unbox b) => Unbox (a, b)
```

Interface of unboxed vectors

`Data.Vector.Unboxed` is similar to `Data.Vector`:

```
data Vector a  -- abstract

empty    :: Unbox a => Vector a
generate :: Unbox a => Int -> (Int -> a) -> Vector a
fromList :: Unbox a => [a] -> Vector a

(++)   :: Unbox a => Vector a -> Vector a -> Vector a

(!)    :: Unbox a => Vector a -> Int -> a
(!?)   :: Unbox a => Vector a -> Int -> Maybe a
slice  :: Unbox a => Int -> Int -> Vector a -> Vector a

(//)   :: Unbox a => Vector a -> [(Int, a)] -> Vector a

map    :: (Unbox a, Unbox b) => (a -> b) -> Vector a -> Vector b
filter :: Unbox a => (a -> Bool) -> Vector a -> Vector a
foldr  :: Unbox a => (a -> b -> b) -> b -> Vector a -> b
```

Complexity as before.

Implementation of unboxed vectors

- While the internals of unboxed vectors are partially built into GHC as well, the outer interface and the `Unbox` class are actually implemented as a library.
- The library selects an appropriate implementation automatically depending on the type of array element, by means of a *datatype family*. More on (data)type families later.

Mutable vectors

Ephemeral data structures in Haskell

Some (surprisingly few, but some) algorithms can be implemented more efficiently in the presence of destructive updates:

- Haskell has mutable data structures as well as immutable ones;
- most operations on mutable data structures have `IO` type.

We have already seen simple mutable references (`IORef`s, `TVar`s, `MVar`s).

Mutable vectors

Both `Data.Vector.Mutable` and `Data.Vector.Unboxed.Mutable` export a datatype

```haskell
data IOVector (a :: *)  -- abstract
```

of mutable vectors.

Mutable vector interface

```
new       :: Int -> IO (IOVector a)
replicate :: Int -> a -> IO (IOVector a)

read  ::  IOVector a -> Int -> IO a
write ::  IOVector a -> Int -> a -> IO ()

...
```

Strings

Haskell strings

By default, Haskell strings are lists of characters:

```haskell
type String = [Char]
```

Haskell strings

By default, Haskell strings are lists of characters:

```
type String = [Char]
```

This definition is quite convenient for implementing basic text processing functions, as one can reuse the rich libraries for lists, but the list representation is quite inefficient for dealing with large amounts of text.

Question

How much memory is needed to store a `String` that is three characters long?

Size of a linked list

- Each cons-cell is three words long (info table, two pointers for the payload).
- Each character is boxed, and the box is three words long (info table, two words for the character).
- The empty list is one word (info table, no payload).

Size of a linked list

- Each cons-cell is three words long (info table, two pointers for the payload).
- Each character is boxed, and the box is three words long (info table, two words for the character).
- The empty list is one word (info table, no payload).

So all in all 16 words (or 128 bytes on a 64-bit machine).

Size of a linked list

- Each cons-cell is three words long (info table, two pointers for the payload).
- Each character is boxed, and the box is three words long (info table, two words for the character).
- The empty list is one word (info table, no payload).

So all in all 16 words (or 128 bytes on a 64-bit machine).

In fairness, the empty list is shared, and some frequently used characters may be shared as well, but still …

`Text`

The `text` package offers

```
data Text  -- abstract
```

a packed representation of (Unicode) text.

- Much less memory overhead than a `String`.
- Still uses UTF-16 encoding per character, so 2 bytes per character.
- Performance characteristics more like functional arrays.

Converting between `String` and `Text`

From `Data.Text`:

```
pack   :: String -> Text  -- O(n)
unpack :: Text -> String  -- O(n)
```

Some common operations

```
cons    :: Char -> Text -> Text                  -- O(n)
(<>)    :: Text -> Text -> Text                  -- O(n)
length  :: Text -> Int                           -- O(n)
map     :: (Char -> Char) -> Text -> Text        -- O(n)
filter  :: (Char -> Bool) -> Text -> Text        -- O(n)
foldr   :: (Char -> a -> a) -> a -> Text -> a    -- O(n)
toUpper :: Text -> Text                          -- O(n)
strip   :: Text -> Text                          -- O(n)
lines   :: Text -> [Text]                        -- O(n)
head    :: Text -> Char                          -- O(1)
last    :: Text -> Char                          -- O(1)
index   :: Text -> Int -> Char                   -- O(n)
```

Stream fusion

Text makes use of stream fusion internally:

- some subsequent traversals, such as multiple `map`s followed by a `fold`, will be fused together, so that only a single traversal is required.

Overloaded string literals

With the `OverloadedStrings` language extension, string literals become overloaded.

Without:

```
GHCi> :t "foo"
"foo" :: [Char]
```

With:

```
GHCi> :t "foo"
"foo" :: IsString t => t
```

Lazy text

The same package also offers a module `Data.Text.Lazy` with again

```
data Text  -- abstract
```

The internal representation is a linked list of *chunks*, which are strict text values.

For streaming purposes, not the entire text has to be present in memory at once.

Conversion from/to lazy text

Assuming `Data.Text.Lazy` is available qualified as `Lazy`:

```
toStrict   :: Lazy.Text -> Text
fromStrict :: Text -> Lazy.Text
```

Both `Text` types are an instance of the `IsString` class.

Building text

- Appending text values is not very efficient.
- However, often the building and inspecting phases are separate.
- Then it's useful to build text using `Builder`, and convert it to text after building is complete.
- Intuitively a builder simply allocates a buffer and fills it with incoming data.

Builder interface

Defined in `Data.Text.Lazy.Builder`:

```
data Builder  -- abstract

toLazyText   :: Builder -> Lazy.Text              -- O(n)
fromText     :: Text -> Builder                   -- O(1)
fromLazyText :: Lazy.Text -> Builder              -- O(1)
(<>)         :: Builder -> Builder -> Builder     -- O(1)
```

`ByteString`

Available in the **`bytestring`** package.

```
data ByteString  -- abstract
```

A bytestring is a packed representation of a sequence of bytes.

- Again, much less memory overhead than a `String`.
- Less overhead even than `Text`.
- No interpretation of the characters (no encoding).
- Only useful for ASCII formats or (better) binary data.

Variants of `ByteString`

- Like `Text`, `ByteString` comes with a strict and a lazy variant.
- Like `Text`, both `ByteString` types are an instance of `IsString`.
- Like `Text`, `ByteString` has a `Builder` type.

Conversion between `Text` and `ByteString`

As `ByteString` consists of pure bytes, but `Text` is interpreted, we need an encoding in order to translate between the two.

From `Data.Text.Encoding`:

```
encodeUtf8 :: Text -> ByteString
decodeUtf8 :: ByteString -> Text  -- partial!
```

Decoding can fail

Not all byte sequences are valid UTF-8 encodings.

```
GHCi> decodeUtf8 "x\255"
"*** Exception: Cannot decode byte ...
```

Decoding can fail

Not all byte sequences are valid UTF-8 encodings.

```
GHCi> decodeUtf8 "x\255"
"*** Exception: Cannot decode byte ...
```

If you want to be robust against invalid inputs, you either have to catch exceptions or use `decodeUtf8'`:

```
GHCi> :t decodeUtf8'
decodeUtf8' ::
  ByteString -> Either UnicodeException Text
GHCi> isRight $ decodeUtf8' "x\255"
False
```

Conversion between `String` and `ByteString`

Defined in `Data.Text.Char8`:

```
pack   :: String -> ByteString  -- unsafe
unpack :: ByteString -> String  -- unsafe
```

Note that these will only work correctly on the ASCII subset of characters, so these should be used with extreme care.

More data structures on Hackage

On Hackage, there are several additional libraries for data structures.

Some examples: heaps, priority search queues, hash maps, heterogeneous lists, zippers, tries, graphs, quadtrees, ...

Binary Serialization

Serialization & deserialization

- Serialization denotes the process of converting data into a format that can be written to disk or sent over a network connection.
- Deserialization is the opposite process: (Re-)Constructing data from a file on disk or from something sent over the network.

Serialization & deserialization

- Serialization denotes the process of converting data into a format that can be written to disk or sent over a network connection.
- Deserialization is the opposite process: (Re-)Constructing data from a file on disk or from something sent over the network.
- Haskell's `Show` and `Read` classes can actually be used for serialization and and deserialization respectively, but they do not use very effficient representations (focus on human readability instead).

Serialization & deserialization

- Serialization denotes the process of converting data into a format that can be written to disk or sent over a network connection.
- Deserialization is the opposite process: (Re-)Constructing data from a file on disk or from something sent over the network.
- Haskell's `Show` and `Read` classes can actually be used for serialization and and deserialization respectively, but they do not use very efffficient representations (focus on human readability instead).
- *Parsing* is a form of deserialization.

Textual serialization formats

- There are several widely-used *textual* serialization formats.

Textual serialization formats

- There are several widely-used *textual* serialization formats.
- XML and JSON are important in web technologies.
- Another quite popular format, used for example for configuration files, is YAML.
- CSV is often used for record-like data, for example to serialize spreadsheets or database tables.

Textual serialization formats

- There are several widely-used *textual* serialization formats.
- XML and JSON are important in web technologies.
- Another quite popular format, used for example for configuration files, is YAML.
- CSV is often used for record-like data, for example to serialize spreadsheets or database tables.

Haskell support

For all of these formats, there are popular Haskell-libraries on Hackage that support them. We will probably look at some of them later.

Binary serialization & deserialization

- Another important class of serialization formats are binary formats, represented by (lazy) Haskell `ByteString`s.
- Such binary representations can be much more compact and efficient than textual, human-readable ones.
- There are several Haskell libraries that support binary serialization: `binary`, `store`, `cereal`...,
- We will have a closer look at `binary` in this lecture.

The `Binary` class

```haskell
class Binary a where
  put : a -> Put
  get : Get a
```

```haskell
encode :: Binary a => a -> ByteString
```

```haskell
decode :: Binary a => ByteString -> a
```

```haskell
decodeOrFail :: Binary a => ByteString
   -> Either (ByteString, ByteOffset, String)
             (ByteString, ByteOffset, a)
```

The `Binary` class

```
class Binary a where
  put : a -> Put
  get : Get a
```

```
encode :: Binary a => a -> ByteString
```

```
decode :: Binary a => ByteString -> a
```

```
decodeOrFail :: Binary a => ByteString
   -> Either (ByteString, ByteOffset, String)
             (ByteString, ByteOffset, a)
```

unconsumed input

The `Binary` class

```haskell
class Binary a where
  put : a -> Put
  get : Get a
```

```haskell
encode :: Binary a => a -> ByteString
```

```haskell
decode :: Binary a => ByteString -> a
```

```haskell
decodeOrFail :: Binary a => ByteString
   -> Either (ByteString, ByteOffset, String)
             (ByteString, ByteOffset, a)
```

number of consumed bytes

The `Binary` class

```haskell
class Binary a where
  put : a -> Put
  get : Get a
```

```haskell
encode :: Binary a => a -> ByteString
```

```haskell
decode :: Binary a => ByteString -> a
```

```haskell
decodeOrFail :: Binary a => ByteString
   -> Either (ByteString, ByteOffset, String)
             (ByteString, ByteOffset, a)
```

human-readable error message

What are `Get` and `Put`?

```
data Get a   -- abstract
```

instances:

- `Functor`
- `Applicative`
- `Monad`
- `Alternative`
- `MonadPlus`

```
type Put = PutM ()
data PutM a   -- abstract
```

instances:

- `Functor`
- `Applicative`
- `Monad`

`Data.Binary` – primitives and helpers

```haskell
putWord8 :: Word8 -> Put
```

```haskell
getWord8 :: Get Word8
```

```haskell
fail :: String -> Get a
```

```haskell
encodeFile :: Binary a => FilePath -> a -> IO ()
```

```haskell
decodeFile :: Binary a => FilePath -> IO a
```

```haskell
decodeFileOrFail :: Binary a => FilePath
   -> IO (Either (ByteOffset, String) a)
```

Example

```haskell
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

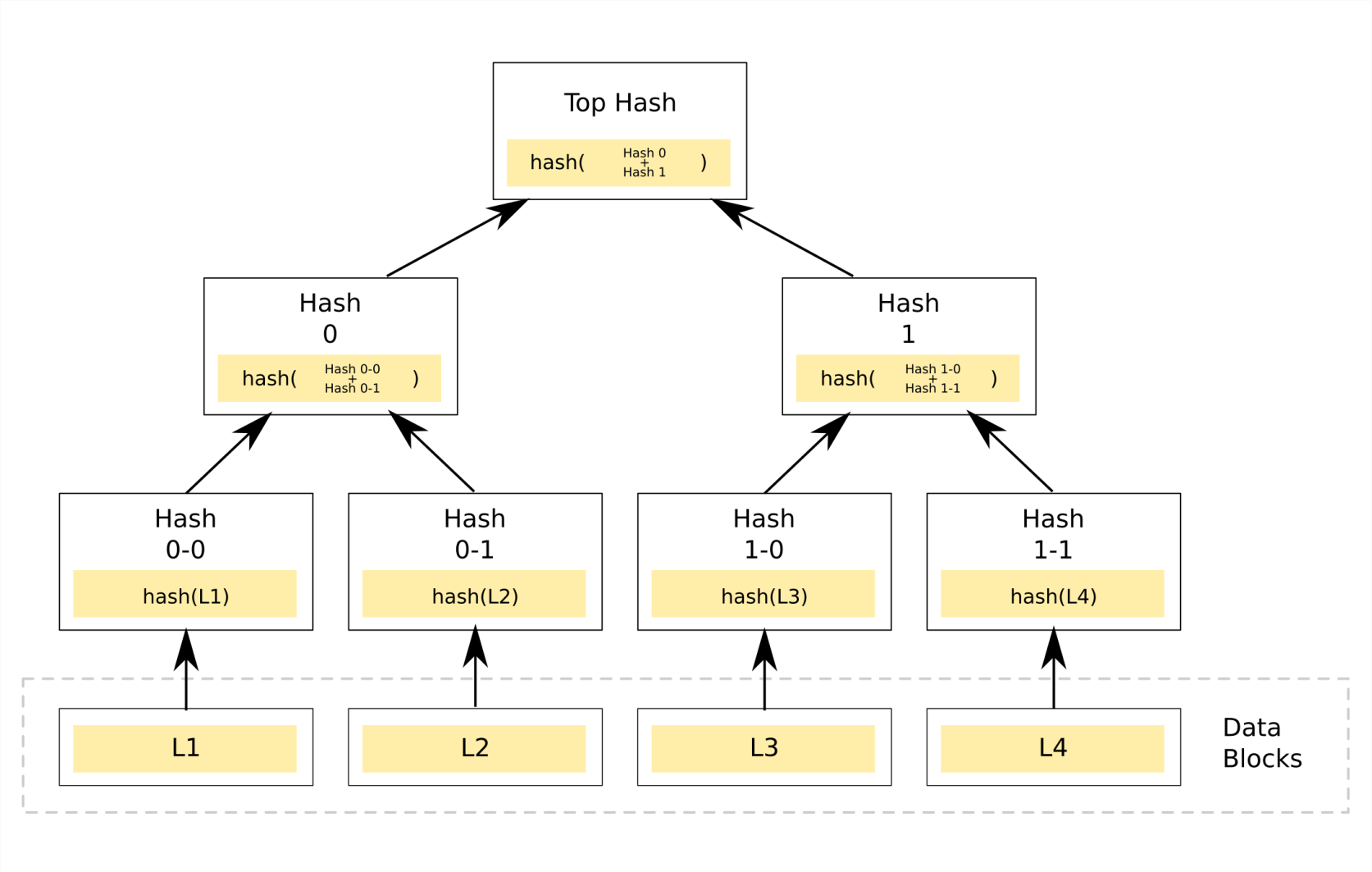
```haskell
instance Binary a => Binary (Tree a) where
  put (Leaf a)   = putWord8 0 >> put a
  put (Node l r) = putWord8 1 >> put l >> put r

  get = do
    tag <- getWord8
    case tag of
      0 -> Leaf <$> get
      1 -> Node <$> get <*> get
      _ -> fail "not a tree"
```

Merkle Trees

Merkle Tree

- A Merkle tree (or hash tree) is a (normally perfect binary) tree in which all nodes are labelled with a *hash*.
- There is a data block attached to each leaf, and the hash label is the hash of the data.
- For other nodes, the hash label is the hash of the concatenation of its child hashes.
- Merkle trees allow efficient and secure verification of the contents of large data structures

Merkle tree illustration

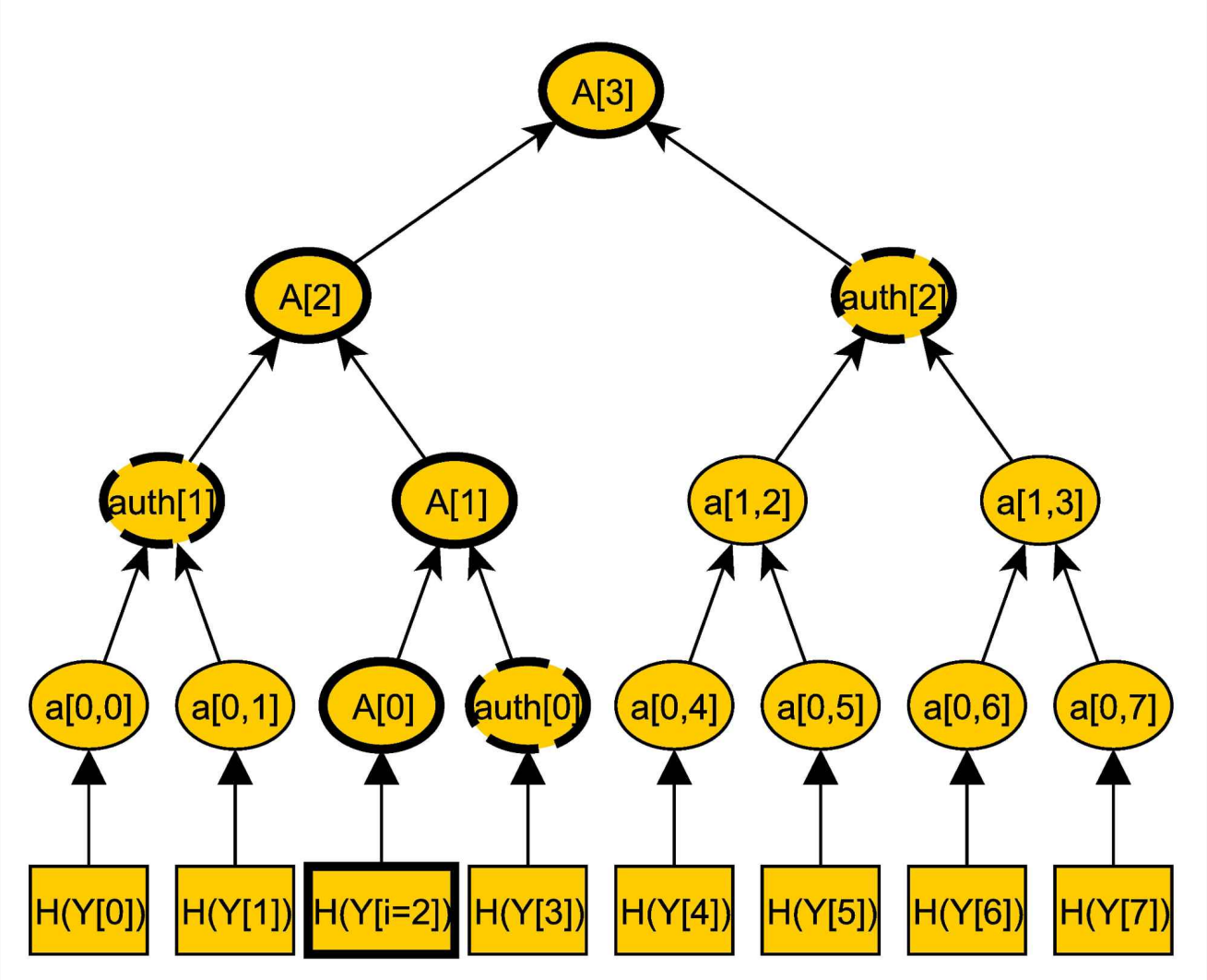


By Azaghal - Own work, CC0

Use in Bitcoin

- In Bitcoin, each block contains a Merkle tree of all included transactions.
- If there is an *odd* number of transactions or of nodes on a higher level, the last hash is duplicated.
- A *Simplified Payment Node (SPN)* does not save all transaction, but only the Merkle root, the hash labelling the Merkle tree root.
- A *full node* can prove to an SPN that a given transaction is in the block by providing a Merkle path.

Merkle path illustration



By Georg987 (Own work) [CC BY-SA 3.0 or GFDL], via Wikimedia Commons

Merkle tree code demo

Let’s look at some code!

Summary

- Arrays – in particular array updates – should be used with care.
- When dealing with textual data, `Text` and `ByteString` can be much more efficient than `String`.
- Consider using the **`binary`** package for binary formats.