I gave a practise version of the talk and found it was too long, so I had to change the title a little. I've kept the good bits but I'm no longer automatically generating tamper-evident data structures, although I am operating on them in a sense. So I apologise. With that let's start.

Purpose

- Show you a cute correspondence between Merkle membership proofs and differential calculus
- Get you interested in blockchains

The meat of this talk is about type theory and an interesting correspondence between it and the differential Calculus, but I also hope to convey some of the interesting concepts in the blockchain space. Yes, it's over-hyped and many business people think it's some kind of magical pixie dust that will solve all their problems, but at the heart of it is some really nice stuff. If you're late to the party I still highly recommend that you join.

NEXT: First, let's do a quick recap on hash functions.

Hashes

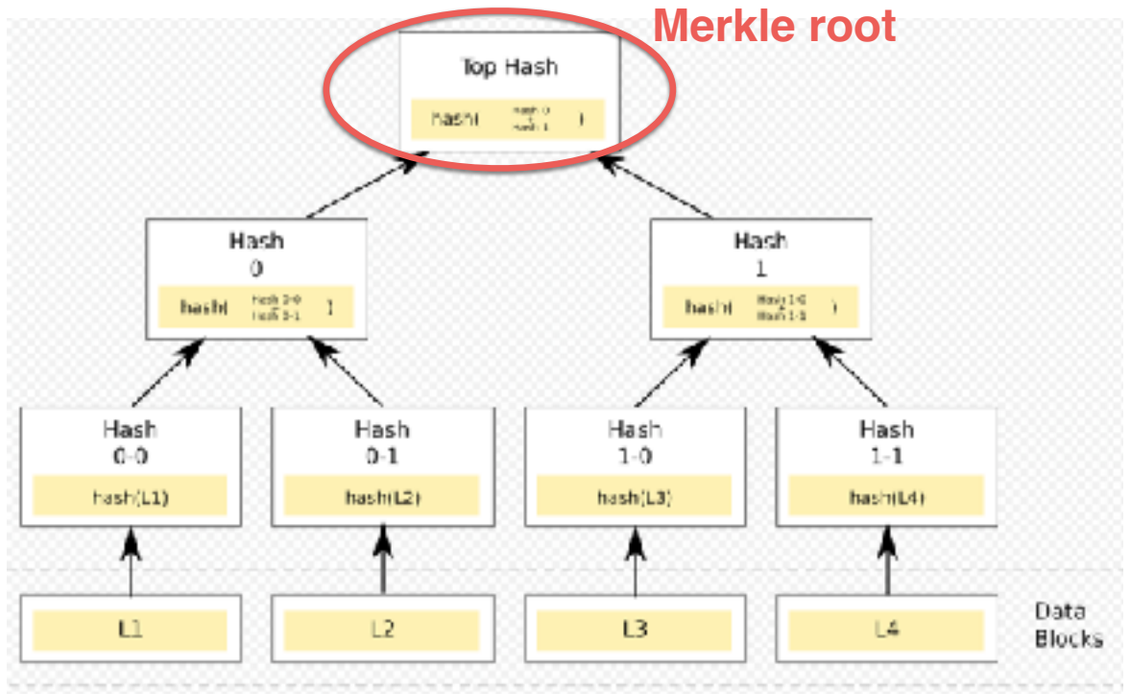
- Known as *one-way* or *trapdoor* function. Easy to go one way, hard the other
- *h(x) = y*. Easy to get y given x, very hard to get x given y. (x is pre-image, y is image)
- Property: **Second pre-image resistance**. Given *x1* with *h(x1) = y* it should be hard to find a different *x2* such that *h(x2) = y*

But first, let's quickly talk about hashes
Hash functions are an example of something known as either a one-way or trapdoor function. It's easy to go one way but hard to go the other.

If h is a hash function, and x is its input and y is the output, then it is hard given y to get back to x.
Hash functions have a couple of properties but the one we are interested in is second pre-image resistance. This property specifies that if you know x1 and h(x1) = y, then it is hard to find another x2 such h(x2) = y.

We will see that this has applications in so-called tamper-evident data structures.

Merkle Trees



Let me introduce you to a little data structure known as the Merkle Tree. Named after Ralph Merkle it is an example of a *tamper evident data structure*.

In it's original formulation, and in usage today, the values it contains exist only in the leaves. Also, it is meant to be an immutable, or at at the very least an append-only data structure.

It is constructed as follows. First you hash the values in the leaves, L1 and L2 to get hashes 0-0 and 0-1 and then hash those to get hash 0, and you do the same for the right subtree. Finally you get a hash for the whole tree called the **Merkle root**. The Merkle root is published widely so that it is publicly known and the public expects that this Merkle root *will never change*. In Bitcoin, each block contains a Merkle Tree of transactions which should never change.

Now why is this a tamper evident data structure? Well, if you change any of the leaves, then the hashes for those values will change which will cause a propagation of changed hashes all the way up to the Merkle root. This is where we rely heavily on the second pre-image resistance property of hashes. Since we believe modern hash functions to have this property we can be certain to a high degree of probability that it is basically impossible the change the values in such a way that you could get something with the same Merkle root.

NEXT: Now this data structure is pretty interesting in its own right, but it turns out that Merkle Trees also have another fascinating application.

TODO
[Draw your own where you don't have the initial hash blocks]

Collection membership

- Another application of Merkle data structures is collection membership proofs
- You want to *verify* that an element is in a large *possibly secret* Merkle thing.
- Verifier asks prover to provide a *Merkle membership proof*

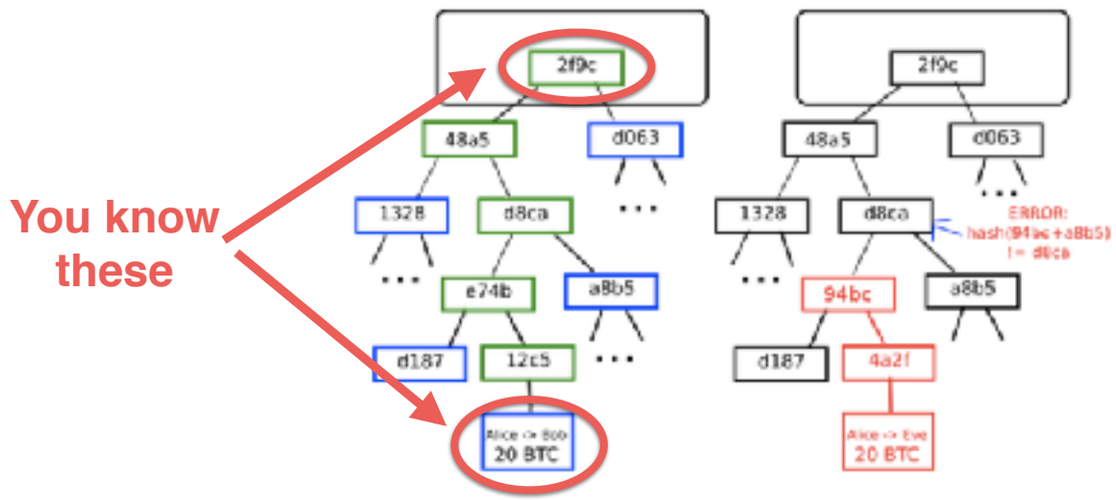
Another application of Merkle Trees is using them in a prover-verifier scheme where a verifier asks a prover whether an element is a member of a Merkle tree that they already know the *merkle root* for.

One reason you might want to do this is for performance or space reasons. It might just be more convenient for the data structure to be kept somewhere centrally, or perhaps it's just too big to fit on your mobile phone. (Mobile wallets for Bitcoin have this problem). The use case is that you want to add things to the Merkle Tree (e.g. a transaction) but you want to verify that it is in there without having to see the whole data structure.

Interestingly, you can also use this prover-verifier scheme in a situation where the values in Merkle Tree are kept secret. You don't know what else in the tree but you can always verify that *your* values are in the tree.

NEXT: Let's look an example of what I will call a Merkle membership proof.

Collection membership



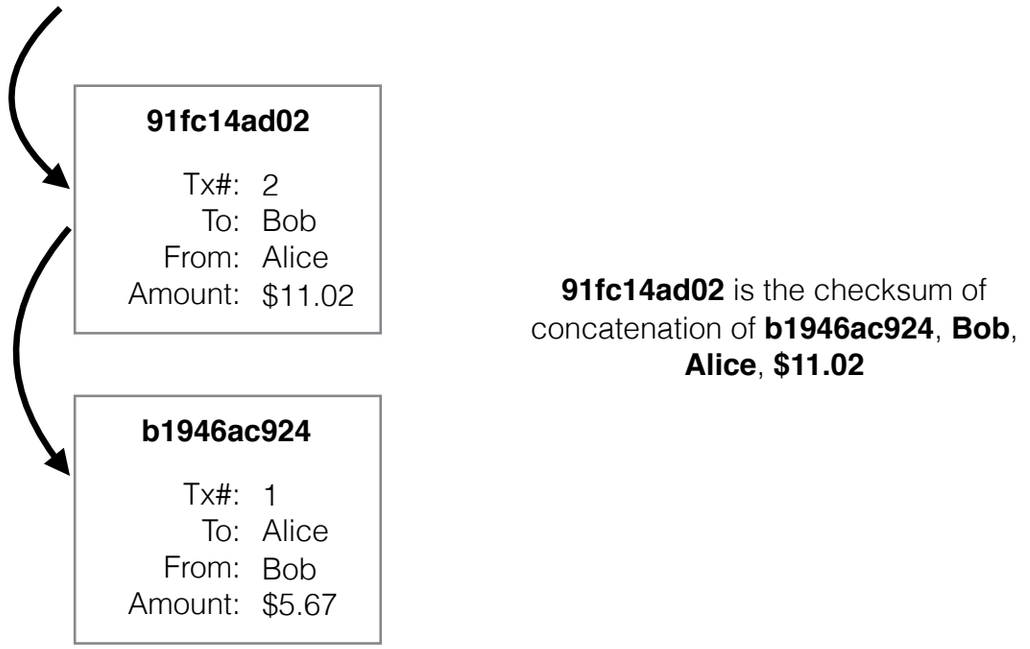
In this example **you** know the merkle root 2f9c. You want a *proof* that the transaction “Alice -> Bob 20 BTC” is member of the collection.

[Mention that the blue nodes are all you need. That is what constitutes the “proof”]

Now when I gave this talk a couple of weeks ago there was some misunderstanding about whether the prover could lie to the verifier and just create a bunch of hashes that matched up with the value they wanted a membership proof for, but no, this is computationally infeasible thanks to second pre-image resistance. It’s really hard to craft hashes such that they match the merkle root. The prover really needs to have the data structure at hand to provide the proof.

NEXT: Next I’ll show you another Merkle data structure which I’ll call a Merkle list

A tamper evident distributed ledger



91fc14ad02 is the checksum of concatenation of **b1946ac924**, **Bob**, **Alice**, **$11.02**

This is an example of a Merkle list of transactions. Hint: this is similar to what Bitcoin does, but it has multiple transactions in what are known as blocks. Hence block ***chain.*** A chain is just a linked-list by any other name.

In this example the hash is created just by concatenating all the values in the transaction *plus the hash of the previous transaction.* Here **91fc14ad02** is created by hashing **Bob, Alice, $11.02** and the previous hash **b1946ac924**

This is also obviously tamper evident. Change a value anywhere and the hashes will have to change all the way up the chain. Tampering would be immediately evident.

Membership proof for Merkle lists

- A bit more long winded since you need every intermediate value up to the one in the set.
- Say you wanted to known that C was a member of some (hidden) list [A,B,C,D,E]
- Then proof would be **([A,B], merkleRoot([D,E]))** where **merkleRoot([D,E]) = h(D•h(E))])**
- With this you can calculate:
  merkleRoot [A,…,E]= h(A•h(B•h (C•h(D•h(E))))

You can also do membership proofs for lists but they aren't nearly as secret as for Merkle Trees since you can't just give someone a list of hashes. You need to given them all the intermediate values up to the thing you want to prove membership of.

NEXT: Now I want to interject with an interlude, that although may not be immediately

Would you believe that you can *derive* a Merkle membership proof data structure from its type!?

We've already seen that the data structure for Merkle membership proofs is quite different between Merkle Trees and Merkle lists. Given a data structure, how could we automatically generate a data structure whose values would serve as a Merkle membership proof? Is this even possible? It turns out yes, if your type is a *regular data type*!

Algebra of types

Name	Algebraic Rep	Haskell equivalent
Void	0	`data Void`
Unit	1	`()`
Value	a	type variable
Sum	$S + T$	`data Ty … = C1 <S…> | C2 <T…>`
Product	ST	`(<S…>,<T…>)` `data Ty … = C <S…> <T…>`
Composition	$S \circ T$	`data Ty … = C <S (T…)>`

Ever wondered why they were called *algebraic* data types?
You can build up what is called a *regular data type* using the building blocks unit, value, sum, product and a fix-point operator.

Examples

```
data Bool = True | False
```

$$B = 1 + 1$$

```
data Maybe a = Nothing | Just a
```

$$M(a) = 1 + a$$

```
data Pair a b = Pair a b
```

$$P(a, b) = ab$$

[Use the triple line equals here]

NEXT: Now let’s look at some more complicated examples involving recursion

More examples

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

$$L(a) = 1 + aL(a)$$

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

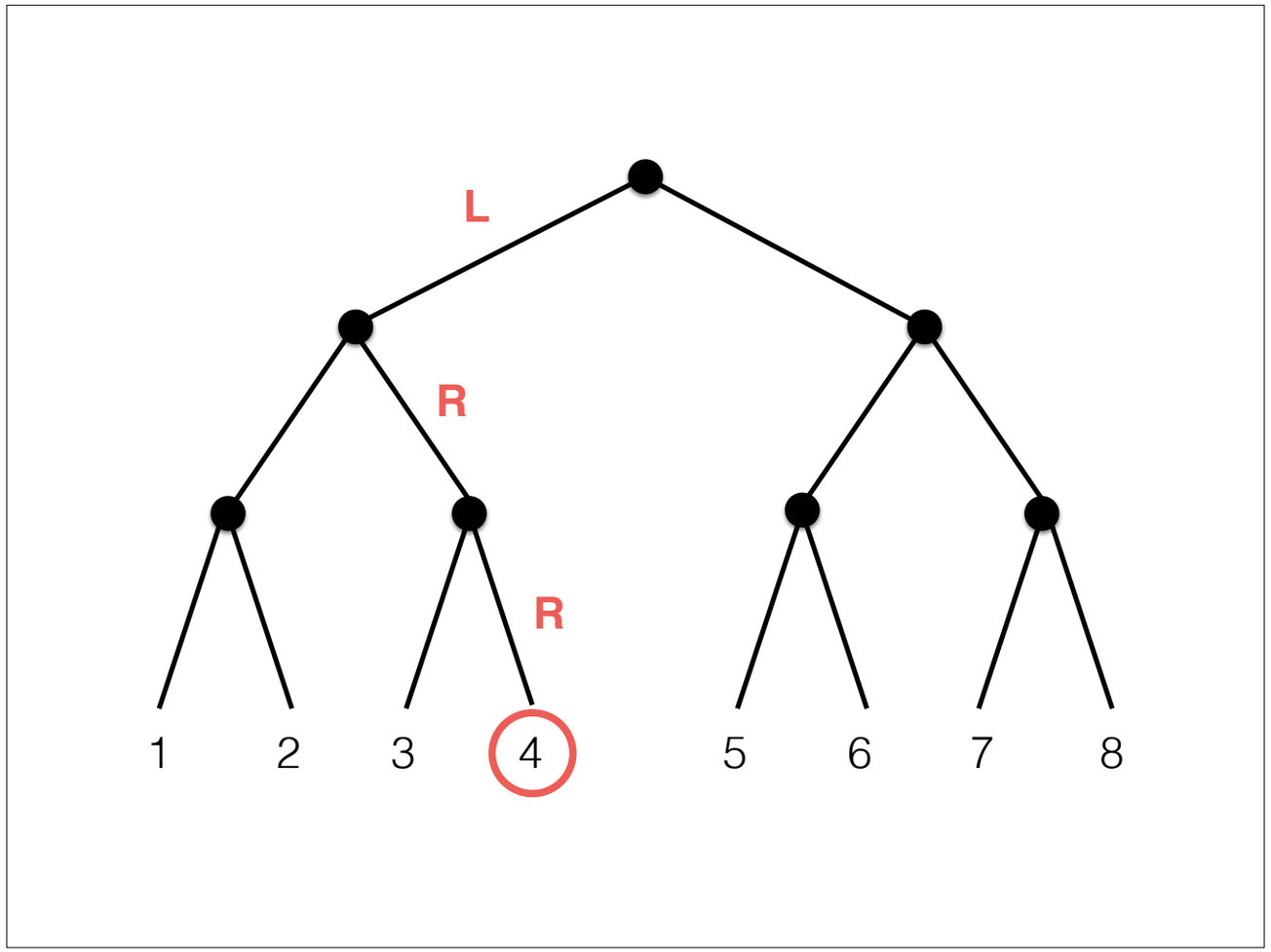
$$T(a) = a + T(a)^2$$

```
data Rose a = RLeaf a | RNode [Rose a]
```

$$R(a) = a + L(R(a))$$

Interlude: the Zipper

So, now that we've seen how to express regular data types in algebraic terms I'd like to interject with an interlude about the Zipper. It turns out there is some theory connecting the algebraic representation of a regular data types with something known as a zipper, and the zipper has a further connection with Merkle membership proofs, which is the tiny insight I had that inspired this talk.



The zipper data structure has probably been independently invented a bunch of times, but it was Gerard Huet who popularised them in his Functional Pearl “The Zipper”. Put simply, the zipper is a data structure for efficiently representing a “focus” **CLICK** in a data structure along with the surrounding “context”.

Using the zipper you can efficiently move the focus, change the value at the focus, and then efficiently reconstruct the original data structure.
[TURN INTO BULLET POINTS].

Here is an example. Our “focus” is the value 4. Now, if the focus had started at the root of the tree, what path would we have had to take to get to the leaf value?

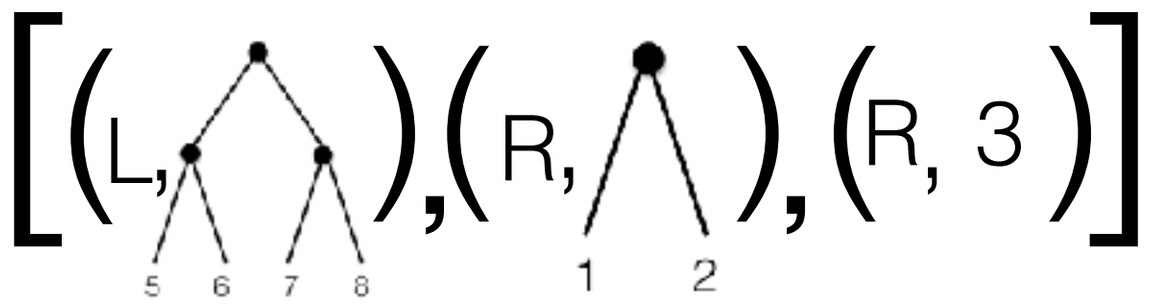
Zipper

```
data Dir = Left | Right

data ZipperCtx a = [(Dir, Tree a)]
Type Zipper a = (a, ZipperCtx a)
```

(4,C)

where C =



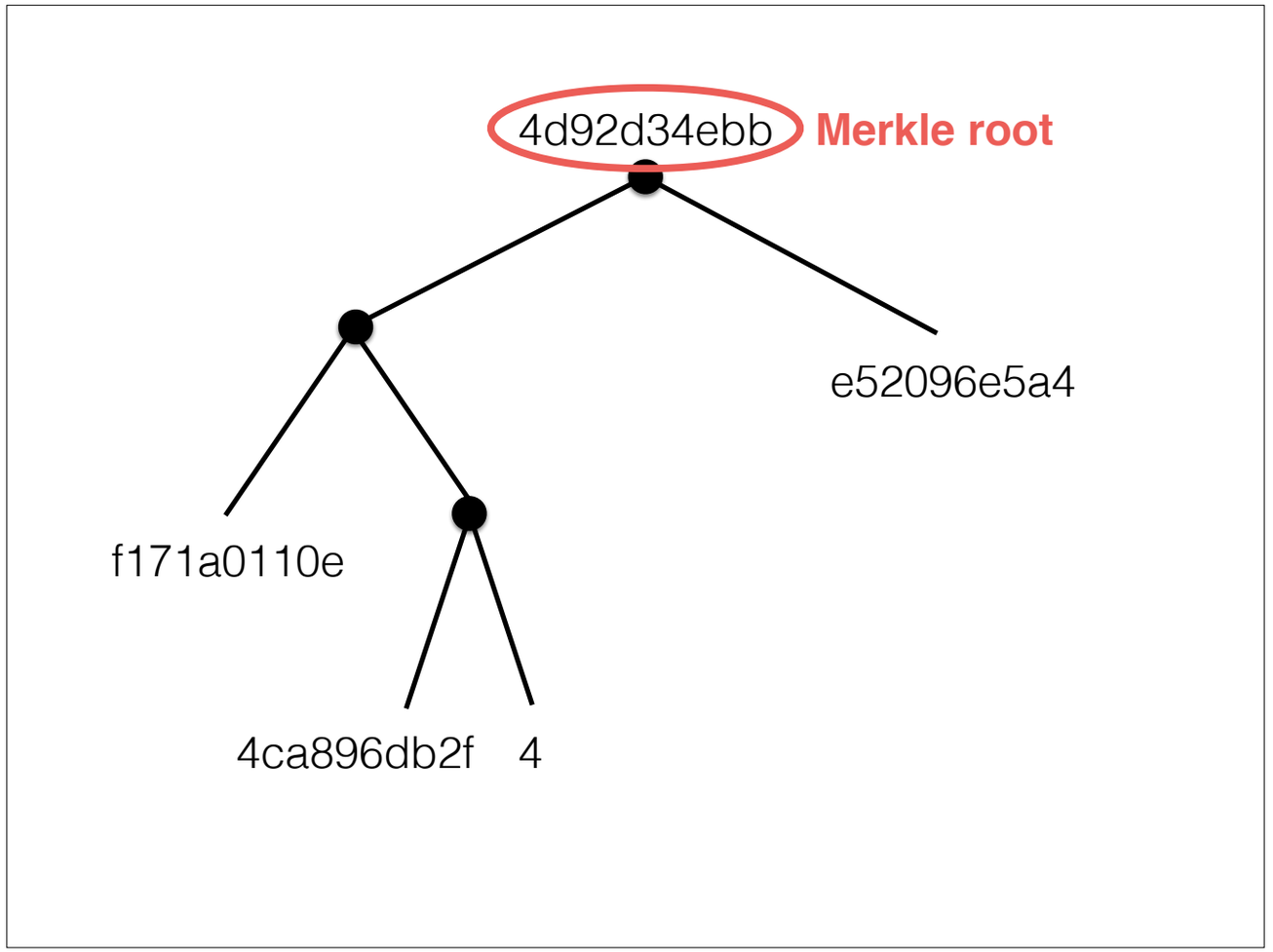
This is great because you could easily change the 4 to something else and reconstruct the tree in time proportional to the height of the tree.

NEXT: And now we come to the central insight that inspired this talk in the first place

Insight!

Replace the subtrees in the *zipper context* with their Merkle roots and you have a Merkle membership proof!

NEXT: To see this, let’s look at a slightly modified version of the previous zipper example


4d92d34ebb
Merkle root
e52096e5a4
f171a0110e
4ca896db2f
4

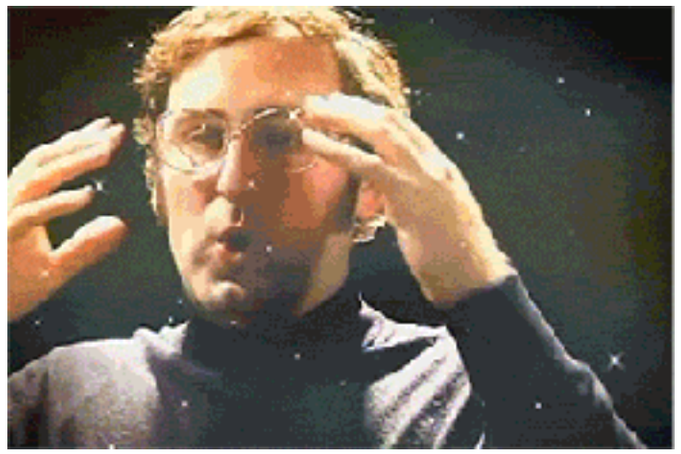
Why is this exciting?

Now you might be thinking, sure Sean, you've shown a correspondence between Huet's zipper and Merkle tree membership proofs. But why should I care?

Because, would you believe, there is a way to derive a zipper context for any regular data type given the algebraic representation of its type.

In short, there is theory of zippers and we can leverage that theory.

The zipper context is the
differential calculus derivative
of the algebraic representation
of the type!



As I wrote down the rules corresponding to the empty type, the unit type, sums and products, I was astonished to find that Leibniz had beaten me to them by several centuries: they were exactly the rules of differentiation I had learned as a child.

— Conor McBride in The Derivative of a Regular Type is its Type of One-Hole Contexts

[Make joke about it being a pity that I can't go back in time to give this talk with Conor available]

NEXT: Now before we go into the theory, let's go through a sample differentiation to see the magic in action.

Derivation

$$T(a) = a + T(a)^2$$

$$\begin{aligned}
\frac{dT}{da} &= \frac{d}{da}a + \frac{d}{da}T^2 \\
&= 1 + \frac{d}{dT}T^2\frac{dT}{da} \\
&= 1 + 2T\frac{dT}{da}
\end{aligned}$$

$$T'(a) = 1 + 2T(a)T'(a)$$
$$T'(a) = L(2T(a))$$

Now, one thing to note is that the type appears on the left hand side and on the right hand side, so we have to use implicit differentiation.

NEXT: Now let’s have a look at how that compares with the Zipper context I showed you before

Zipper comparison

$$T'(a) = L(2T(a))$$

VS

```
data Dir          = Left | Right
data ZipperCtx a = [(Dir, Tree a)]
```

Merkle membership proof comparison

$$T'(a) = L(2T(a))$$

VS

```
data MerkleTreeProof = [(Dir, Hash)]
```

Similarly

Rule to get a Merkle membership proof

In the derivative, replace occurrences of the original type with hashes!

What's the connection with calculus?

- There doesn't seem to be an intuitive connection between differential calculus notions such as tangents and limits, and zipper contexts
- But one can show *correspondence* between differential calculus derivatives and *zipper contexts/merkle proof data structures*

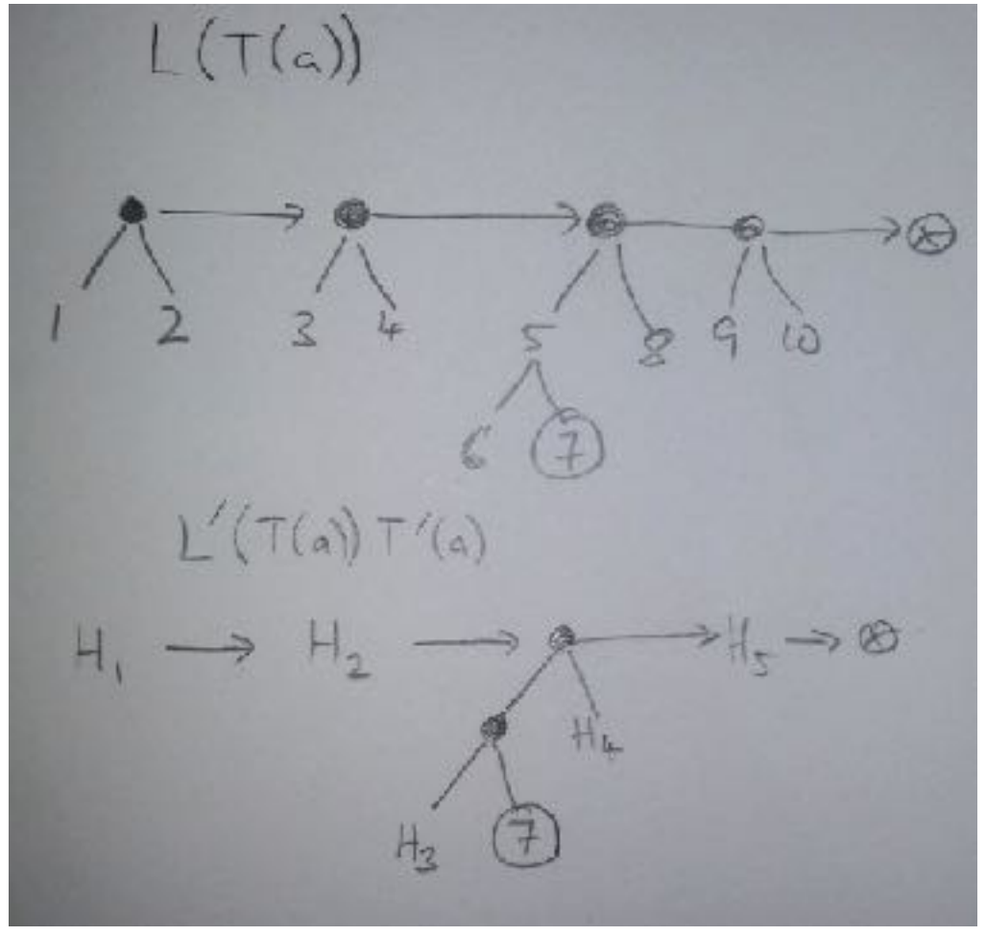
NEXT: I'm going to show you how to find the Merkle membership data structures for the building blocks of algebraic types and show how they correspond to the rules of integral calculus.

Name	Algebraic Rep	Derivative	
Void	0	0	
Unit	1	0	
Value	a	1	
Sum	$S(a) + T(a)$	$S'(a) + T'(a)$	
Product	$S(a)T(a)$	$S'(a)T(a) + S(a)T'(a)$	**Product rule!**
Composition	$S(T(a))$	$S'(T(a))T'(a)$	**Chain rule!**

- Now remember, when ever you see a non-primed data structure in the derivative, that's a HASH. A primed data structure is a Merkle proof data structure.
- **Also I want to make it clear I'm cheating here. I really need to be talking about partial differentiation because the types could depend on multiple type variables. But we'll just assume it's one in this case.**
- There is no Merkle proof for unit. No "a"s in there. Nothing to see here.
- If there's just one value in there then no Merkle proof required. You've already got the merkle root
and you can verify that root matches the value (which you also have)
- For a sum type you have choice. Either you need a Merkle proof that it's in the S or a Merkle Proof that it's in the T and you need to be able to distinguish between the two proofs.
- For a product type, this is a bit more interesting. Remember S(a)T(a) is a pair. So either the element is in the S part or the T part. If it's in the S part then you need a merkle proof for that and you need a hash for the T. If it's in the T part then vice versa, but there are two cases and you need to distinguish between them. OMG! This is the product rule from calculus! What the hell?
- For composition we have a collection of collections. For a Merkle proof you first need to travel along in the outer data structure and then into the inner data structure to find the element. So in the result you actually need a pair of Merkle data structures, one for the outer, one for the inner. This one's a bit tricky to visualise so I'm going to show you an example on the next slide.

Name	Algebraic Rep	Derivative	
Void	0	0	
Unit	1	0	
Value	a	1	
Sum	$S(a) + T(a)$	$\pi_S(a) + \pi_T(a)$	
Product	$S(a)T(a)$	$\pi_S(a)h_T + h_S\pi_T(a)$	**Product rule!**
Composition	$S(T(a))$	$\pi_S(h_T)\pi_T(a)$	**Chain rule!**

- Now remember, when ever you see a non-primed data structure in the derivative, that's a HASH. A primed data structure is a Merkle proof data structure.
- **Also I want to make it clear I'm cheating here. I really need to be talking about partial differentiation because the types could depend on multiple type variables. But we'll just assume it's one in this case.**
- There is no Merkle proof for unit. No "a"s in there. Nothing to see here.
- If there's just one value in there then no Merkle proof required. You've already got the merkle root
and you can verify that root matches the value (which you also have)
- For a sum type you have choice. Either you need a Merkle proof that it's in the S or a Merkle Proof that it's in the T and you need to be able to distinguish between the two proofs.
- For a product type, this is a bit more interesting. Remember S(a)T(a) is a pair. So either the element is in the S part or the T part. If it's in the S part then you need a merkle proof for that and you need a hash for the T. If it's in the T part then vice versa, but there are two cases and you need to distinguish between them. OMG! This is the product rule from calculus! What the hell?
- For composition we have a collection of collections. For a Merkle proof you first need to travel along in the outer data structure and then into the inner data structure to find the element. So in the result you actually need a pair of Merkle data structures, one for the outer, one for the inner. This one's a bit tricky to visualise so I'm going to show you an example on the next slide.


L(T(a))
1
2
3
4
5
6
7
8
9
10
L'(T(a)) T'(a)
H1
H2
H3
H4
H5
7

Tree with values in nodes example

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

$$
\begin{aligned}
T(a) &= a + aT(a)^2 \\
&\vdots \\
T'(a) &= L(2T(a)^3) \\
&= L(2T(a)T(a)^2)
\end{aligned}
$$

```
data ZipperCtxTree a = [(Dir, Tree a, (Tree a, Tree a))]
```

```
data MerkleProofTree a = [(Dir, Hash, (Hash, Hash))]
```

Merkle Trees generally only have values in the leaves, but there is really nothing stopping you from having them in the nodes as well. You just need to perform the hashing on the node values as well. Designing these hashing schemes for data structures can involve a little bit of thought. What's really nice is that the derivative gives us a way of coming up with an adequate hashing scheme in a completely mechanical way.

I've written T(a)T(a)^2 instead of T(a)^3 because it's more convenient.

Rose tree example

```
data Rose a = RLeaf a | RNode [Rose a]
```

$$R(a) = a + L(R(a))$$
$$\vdots$$
$$R'(a) = L(L'(R(a))$$
$$R'(a) = L(L(R(a)^2)$$

Using $L'(a) = L(a)^2$

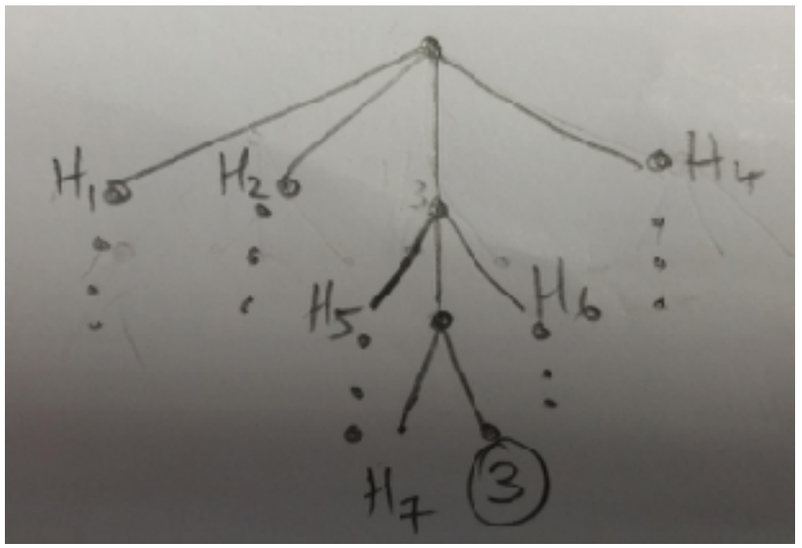
```
data ZipperCtxRose a = [([Rose a], [Rose a])]
```

```
data MerkleProofRose = [([Hash], [Hash])]
```

Here's one I prepared earlier.

The derivative of this one involves using the chain rule because we have composition of lists and roses here. [Point to L(R(a)) term]

Can you see how this serves as Merkle proof?
As you traverse down a path in the Rose tree you need Merkle roots for everything to the left of the (possibly empty)



Merkle Proof of 3's membership =

[([H_1, H_2], [H_4]),([H_5],[H_6]), ([H_7],[])]

A question from the Bitcoin StackExchange

https://bitcoin.stackexchange.com/questions/42281/what-is-the-canonical-way-of-creating-merkle-tree-branches

However, since the order of nodes matter for creating the merkle root, I would also need to list whether the node was on the left or right side.

I'm wondering - is there a canonical way of listing this information to create the merkle branches?

Look what I found!

It asks a question about canonicity. Now we have an even better answer, from theory, the derivative is the canonical way. All else is ad hoc.

END

References

http://strictlypositive.org/diff.pdf

https://pavpanchekha.com/blog/zippers/derivative.html

Contact

sean.seefried@gmail.com
sean.seefried@digitalasset.com

Consider adding Authenticated Data Structures

https://www.cs.umd.edu/~mwh/papers/gpads.pdf