1 | Chapter 0: Algebraic Type Theory

1.1 | Semi-Groups

- Elements
- Operator
- Identity
- Associativity
- Commutativity

So its a field with one operator

1.2 | Semi-Rings

Groups with two operators.

1.3 | Types

Define two types:

- (): Unit
- Option<Unit>: Bool

1.4 | Tuples are Multiply

1.4.1 | The "Tuple" Operator

These are isomorphic:

```
fn SOME () => true | None => false;
fn true => SOME () | false => None |
```

(elements proven)

a * b = a * b (closed under operator. Operator shown.)

1.4.2 | Proving Properties

- | unit * a | = 1 * | a | = | a | (operator identity exists. Identity proven)

Operator is associative

```
fn ((x,y),z) => (x,(y,z))
```

(associativity proven)
The Reverse Function Exists.
(commutativity proven) Proof left for the reader, or Axler.

1.5 | Datatypes are Add

1.5.1 | The "Add" Operator

$a + b = a + b$ (you are having a value that has EITHER type a or b. Closed under operator.)

(Operator 2 shown.)

1. Relationship between Operators

$$a + a = 2 * a = bool * a = bool * a$$

We could show this:

```
datatype ('a, 'b) either = Left of 'a | Right of 'b;
fn Left x => (false, x) | Right x => (true x);
```

Therefore "either" is a union (OR) of two types.
(map between operators?)

1.5.2 | Proving Properties

Also, we show that this second operator has an identity

```
datatype void = .
```

$$a + void = a$$

because nothing could possibly have the type of `void`, so if we have a Either 'void 'a, it could only be Right a.

(identity proven.)

Commutivity and associativity is left for the reader. Or anyone else. I didin't catch it.

1.6 | Proving Distribution

```
fn (y, Left y) => Left (x,y) | (x, Right z) => Right (x,z)
```

Yeah.

1.7 | Functions as a Type

How many functions does the following thing?

'a -> 1?

One. The following thing:

```
fn _ => ()
```

How many functions does the following thing?

1 -> ’a

A.

```
fn _ => 1
fn _ => 2
```

etc. etc. etc.

So! We understand the following things:

- |’a -> 1| = | 1 |
- | 1 -> ’a | = | ’a |
- | 2 -> ’a | = | ’a $|^2$ (true, alpha. False, alpha. etc. etc.)
- | ’a -> 2 | = $2^{(|'a|)}$

Hence, ”function” is the exponent operator.

1.8 | Currying and Uncurring form an Isomorphism

So:

Blue box:

’a x ’b -> ’c (uncurried tuple input) is isomorphic to ’a -> ’b -> ’c

Proof:

(a * b) -> c = a -> (b -> c)

Rewriting in Exponent form, as established above

c ^ (a * b) = (b -> c) ^ a

Because of that property I dm’d zach

c ^ (a) * (b) = (b -> c) ^ a

Expanding the right function into exponent form

c ^ (a) * (b) = (c ^ b) ^ a

a^{bc} = a * bc

qed, ig?

1.9 | Lists

Linked lists!

L('a) = 1 + 'a * L('a)

We could do some things on it, by moving things around

- L('a) - 'a * L('a) = 1
- L('a) (1 - 'a) = 1
- L('a) = 1/(1-'a)

Wait wait wait

L('a) = 1/(1-'a)

that's an infinite series! Let's taylor expand it

L('a) = 1/(1-'a) = 1 + a + a^2 …

And, indeed. A list is a datatype of empty, OR linked to one element, OR linked by 2 elements (all possiblities of first element, times all the second, so 'a^2), OR linked by 3 etc.

1.10 | And now, Calculus.

So what exactly is: d/d'a L ('a)?

Expanding the definition and taking low 'd high, high'd low:

d/da L('a) = d/da 1/(1-'a) = ((1-'a)(0) - 1(d/d'a (1 - 'a)))/$(1\text{-'a})^2$

d/da L('a) = 1/$(1\text{-'a})^2$ = $(1/(1\text{-'a}))^2$

So we know that:

d/da L('a) = $(1/(1\text{-'a}))^2$ = $(L(\text{'a}))^2$

Therefore: the derivative of a list is two lists! Or a tuple of lists.

1.11 | Formalizing a Derivatives

Say we want to remove an 'a from a list:

- 'a * 'a = 2 ways of removing an element of 'a. The result of this is bool (which one punched), val
- 'a * 'a * 'a = 3 ways of removing an element of 'a
- 'a * 'b = 1 way of removing an element of 'a

This all makes sense. But its not motivated.

1.11.1 | Statements from the utterly deranged

- What if we have an 'b, and punch out alpha? We get Void.
- What if we have an (), and punch out alpha? We get Void.
- Why are these impossible?

1.11.2 | So this is a derivative

Think:

'a * 'a = 2 ways of removing an element of 'a. The result of this is bool (which one punched) + val = 'a * 2

$a^2 = 2a$

This is also known as "one-hole context."

1.12 | Revisiting Lists with One-Hole Context on Context

From above:

d/da L('a) = $(L('a))^2$

We could see: if we "punch" (create a hole) in a linked list, we create TWO linked lists. A tuple with everything before the punched element, and everything after the punched element.

Also, you will realize that one-hole contexts allow fast access of things near holes

1.13 | More with One-Hole Contexts

A tree is defined as…

T('a) = 1 + T('a) * a * T('a)

Let's find its one-hole context:

We are just believing Avery.

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

This tells us that, for any given hole, we have

- T('a): a left tree
- T ('a): a right tree
- L: a list of parent nodes
  - 'a: the current value
  - 2: a boolean of whether to go right or left
  - T ('a): a tree that diverged from that point

2 | Chapter 1: Church's Lambda Calculus

EVERYTHING IS A FUNCTION!

2.1 | Expressions

An expression should look like one of three things:

- `lambda x . exp` (a function)
- `exp(exp)` (a compound expression)

- `x` (a value)

Congratulations, you made a turing machine (the simplest one, in fact). Now let's just put some meaning on things:

2.2 | Basic Functions

- lambda x . x
- lambda x . lambda f . fx

2.3 | Booleans

- true = lambda x. lambda y. x (curred function returns first)
- false = lambda x. y lmbda y. y (curred function returns second)
- if a then b else c = a b c
  - a is a boolean, as defined above
  - true bc => (lambda x. lambda y. x) bc = b (the "then" case")
  - false bc => (lambda x. lambda y. y) bc = c (the "else" case")

2.4 | Numbers

- 0 = lambda f.lambda.x.x ("run the function on base case times")
- 1 = lambda f.lambda x.fx ("run the function on base case one times")
- 2 = lambda f.lambda y.f(fx) ("run the function on base case two times")

2.5 | Numerical Operators

- succ (succ = fn x => x+1) = lambda n. lambda f. lambda x. f(nfx) (remember that n, a number, is a function too per above)
- add = lambda a. lambda b. lambda f. lambda x. b f(afx)
- mult = lambda a. lambda b. lambda f. lambda x. a(bf)x (run it a times b times)

2.6 | Y-Combinator and Recursion

Y is a recursion helper with the following property:

Y f x = f (Y f) x

"Y" passes itself to the inner function as a parametre. It is defined as follows:

Y = lambda f.(lambda x.f(x x))(lambda x.f(x x))

With this, we could now have the factorial function

Y(lambda f. lambda n. if n=0 then 1 else mult n(f(pred x)))