Functional Differentiable Programming

UCSD CSE 291 Differentiable Programming

Tzu-Mao Li

Functional programming

- everything is function

(this is equivalent to something called “lambda calculus”)

```
def f(x, y, z):
  # x is a function,
  # y is also a function,
  # z is also a function.
  return x(y, z)

# w is also a function.
let y = w(w) # functional representation of “2”
let z = w(w(w)) # functional representation of “3”

# add is also a function.
f(add, y, z)
# the expression above evaluates to
# w(w(w(w(w)))) ->
# it’s a functional representation of “5”
```

Why functional programming?

- it's mathematically cool
- function types help us verify correctness
- it allows us to write crazy polymorphic programs
- "pure" functional programming has no side-effects (mutation of variables)
  - easy to parallelize (c.f. mapreduce)
  - "=" is actually equal mathematically
    — allows us to prove/apply theorems of our programs
    — allows crazy compiler optimization
- anything else?

```
# assume "int" is all sets
# of compositions of w
def f(x : (int -> int) -> int,
      y : int,
      z : int):
  # x is a function,
  # y is also a function,
  # z is also a function.
  return x(y, z)

# assume w is "1"
f(add, w(w), w)
f(mul, w, w(w))
f(exp, w, w) # error
# partial evaluation
# aka "currying"
plus1 = f(add, w)
plus1(w(w)) # returns "3"
```

Usually, practical functional programming languages are less hardcore

- not literally everything is function (you have numbers at least)
- don't always have to supply types (often can be inference)

```
add y z = y + z

f :: (int -> int -> int) -> int -> int -> int
f x y z = x y z

main :: IO ()
main = print (f add 3 5)
```

Functional programming theory studies abstraction and patterns

i.e., what is the “essence” of my computation?

I have code that does both the following operations:

x : int, y : int
add(x, y)
mul(x, y)

X : matrix, Y : matrix
add(X, Y)
mul(X, Y)

Functional programming theory studies abstraction and patterns

i.e., what is the “essence” of my computation?

I have code that does both the following operations:

x : int, y : int
add(x, y)
mul(x, y)

X : matrix, Y : matrix
add(X, Y)
mul(X, Y)

```
class Ring a where
  add :: a -> a -> a
  mul :: a -> a -> a
  …
```

oh no, it’s a **ring**!

(Haskell’s typeclass doesn’t check
things like commutativity and associativity,
but some languages like Coq/Agda does)

Functional programming theory studies abstraction and patterns

i.e., what is the "essence" of my computation?

I have code that does both the following operations:

x : int, y : int
add(x, y)
mul(x, y)

refactor

```
class Ring a where
  add :: a -> a -> a
  mul :: a -> a -> a
  …
```

oh no, it's a **ring**!

Finally, a Polymorphic Linear Algebra Language

Amir Shaikhha
Department of Computer Science, University of Oxford, UK
amir.shaikhha@cs.ox.ac.uk

Lionel Parreaux
DATA Lab, EPFL, Lausanne, Switzerland
lionel.parreaux@epfl.ch

this paper pretty much did this

(Haskell's typeclass doesn't check
things like commutativity and associativity,
but some languages like Coq/Agda does)

Today: 2 ideas from functional programming for AD

Backpropagation with Continuation Callbacks: Foundations for Efficient and Expressive Differentiable Programming

Fei Wang
Purdue University
West Lafayette, IN 47906
wang603@purdue.edu

James Decker
Purdue University
West Lafayette, IN 47906
decker31@purdue.edu

Xilun Wu
Purdue University
West Lafayette, IN 47906
wu636@purdue.edu

Grégory Essertel
Purdue University
West Lafayette, IN, 47906
gesserte@purdue.edu

Tiark Rompf
Purdue University
West Lafayette, IN, 47906
tiark@purdue.edu

“Continuations” for AD

The Simple Essence of Automatic Differentiation

CONAL ELLIOTT, Target, USA

Automatic differentiation (AD) in reverse mode (RAD) is a central component of deep learning and other uses of large-scale optimization. Commonly used RAD algorithms such as backpropagation, however, are complex and stateful, hindering deep understanding, improvement, and parallel execution. This paper develops a simple, generalized AD algorithm calculated from a simple, natural specification. The general algorithm is then specialized by varying the representation of derivatives. In particular, applying well-known constructions to a naive representation yields two RAD algorithms that are far simpler than previously known. In contrast to commonly used RAD implementations, the algorithms defined here involve no graphs, tapes, variables, partial derivatives, or mutation. They are inherently parallel-friendly, correct by construction, and usable directly from an existing programming language with no need for new data types or programming style, thanks to use of an AD-agnostic compiler plugin.

The “essence” of AD

Today: 2 ideas from functional programming for AD

Backpropagation with Continuation Callbacks: Foundations for Efficient and Expressive Differentiable Programming

Fei Wang
Purdue University
West Lafayette, IN 47906
wang603@purdue.edu

James Decker
Purdue University
West Lafayette, IN 47906
decker31@purdue.edu

Xilun Wu
Purdue University
West Lafayette, IN 47906
wu636@purdue.edu

Grégory Essertel
Purdue University
West Lafayette, IN, 47906
gesserte@purdue.edu

Tiark Rompf
Purdue University
West Lafayette, IN, 47906
tiark@purdue.edu

"Continuations" for AD

The Simple Essence of Automatic Differentiation

CONAL ELLIOTT, Target, USA

Automatic differentiation (AD) in reverse mode (RAD) is a central component of deep learning and other uses of large-scale optimization. Commonly used RAD algorithms such as backpropagation, however, are complex and stateful, hindering deep understanding, improvement, and parallel execution. This paper develops a simple, generalized AD algorithm calculated from a simple, natural specification. The general algorithm is then specialized by varying the representation of derivatives. In particular, applying well-known constructions to a naive representation yields two RAD algorithms that are far simpler than previously known. In contrast to commonly used RAD implementations, the algorithms defined here involve no graphs, tapes, variables, partial derivatives, or mutation. They are inherently parallel-friendly, correct by construction, and usable directly from an existing programming language with no need for new data types or programming style, thanks to use of an AD-agnostic compiler plugin.

The "essence" of AD

Reverse mode implementation is messy

- especially when we have long chained function calls

```
f(g(h(i(j(k(…))))))
```

to make it efficient, we likely need to use a tape, pass it around functions to record intermediate values

can we have a simpler implementation?

turns out it is possible, with a “callback function”

A callback function implementation of reverse mode

```
def square(x : float) -> float:
  return x * x

def cube(x : float) -> float:
  return x * x * x
```

A callback function implementation of reverse mode

```
def square(x : float) -> float:
  return x * x

def cube(x : float) -> float:
  return x * x * x
```

```
def DTsquare(x : InOut[dfloat],
             k):
  ret : dfloat
  ret.val = x.val * x.val
  k(ret) # compute ret.dval
  x.dval += 2 * ret.dval * x.val
```

```
def DTcube(x : InOut[dfloat],
           k):
  ret : dfloat
  ret.val = x.val * x.val * x.val
  k(ret) # compute ret.dval
  x.dval += 3 * ret.dval * x.val * x.val
```

A callback function implementation of reverse mode

```
def DTsquare(x : InOut[dfloat],
             k):
  ret : dfloat
  ret.val = x.val * x.val
  k(ret) # compute ret.dval
  x.dval += 2 * ret.dval * x.val
```

```
def DTcube(x : InOut[dfloat],
           k):
  ret : dfloat
  ret.val = x.val * x.val * x.val
  k(ret) # compute ret.dval
  x.dval += 3 * ret.dval * x.val * x.val
```

```
let y = cube(x)
in z = square(x)
in square(z)
```

```
DTcube(x,
  lambda y : DTsquare(y,
    lambda z : DTsquare(z,
      lambda ret : ret.d = 1.0)
  )
 )
```

note that the order is not reversed! and all the intermediate values are safely stored in function local stacks

A callback function implementation of reverse mode

```
def DTsquare(x : InOut[dfloat],
             k):
  ret : dfloat
  ret.val = x.val * x.val
  k(ret) # compute ret.dval
  x.dval += 2 * ret.dval * x.val
```

```
def DTcube(x : InOut[dfloat],
           k):
  ret : dfloat
  ret.val = x.val * x.val * x.val
  k(ret) # compute ret.dval
  x.dval += 3 * ret.dval * x.val * x.val
```

k is often called “continuation” in functional programming

```
let y = cube(x)
in z = square(x)
in square(z)
```

```
DTcube(x,
  lambda y : DTsquare(y,
    lambda z : DTsquare(z,
      lambda ret : ret.d = 1.0)
  )
)
```

note that the order is not reversed! and all the intermediate values are safely stored in function local stacks

A callback function implementation of reverse mode

```
def DTsquare(x : InOut[dfloat],
             k):
  ret : dfloat
  ret.val = x.val * x.val
  k(ret) # compute ret.dval
  x.dval += 2 * ret.dval * x.val
```

```
def DTcube(x : InOut[dfloat],
           k):
  ret : dfloat
  ret.val = x.val * x.val * x.val
  k(ret) # compute ret.dval
  x.dval += 3 * ret.dval * x.val * x.val
```

k is often called “continuation” in functional programming

converting returning function calls into callback function forms is usually called the “continuation passing style” (CPS) conversion

```
let y = cube(x)
in z = square(x)
in square(z)
```

```
DTcube(x,
  lambda y : DTsquare(y,
    lambda z : DTsquare(z,
      lambda ret : ret.d = 1.0)
  )
)
```

note that the order is not reversed! and all the intermediate values are safely stored in function local stacks

A callback function implementation of reverse mode

```
def DTsquare(x : InOut[dfloat],
             k):
  ret : dfloat
  ret.val = x.val * x.val
  k(ret) # compute ret.dval
  x.dval += 2 * ret.dval * x.val
```

```
def DTcube(x : InOut[dfloat],
           k):
  ret : dfloat
  ret.val = x.val * x.val * x.val
  k(ret) # compute ret.dval
  x.dval += 3 * ret.dval * x.val * x.val
```

discussion: is this a good idea to implement in practice?

```
DTcube(x,
  lambda y : DTsquare(y,
    lambda z : DTsquare(z,
      lambda ret : ret.d = 1.0)
  )
 )
```

Getting rid of the side effects?

```
def DTsquare(x : InOut[dfloat],
             k):
  ret : dfloat
  ret.val = x.val * x.val
  k(ret) # compute ret.dval
  x.dval += 2 * ret.dval * x.val
```

```
def DTcube(x : InOut[dfloat],
           k):
  ret : dfloat
  ret.val = x.val * x.val * x.val
  k(ret) # compute ret.dval
  x.dval += 3 * ret.dval * x.val * x.val
```

Getting rid of the side effects?

```
def DTsquare(x : InOut[dfloat],
             k):
  ret : dfloat
  ret.val = x.val * x.val
  k(ret) # compute ret.dval
x.dval += 2 * ret.dval * x.val
```

```
def DTcube(x : InOut[dfloat],
           k):
  ret : dfloat
  ret.val = x.val * x.val * x.val
  k(ret) # compute ret.dval
x.dval += 3 * ret.dval * x.val * x.val
```

Denotationally Correct, Purely Functional, Efficient Reverse-mode Automatic Differentiation

MATHIEU HUOT, University of Oxford, UK
AMIR SHAIKHHA, University of Edinburgh, UK

Reverse-mode differentiation is used for optimization, but it introduces references, which break the purity of the underlying programs, making them notoriously harder to optimize. We present a reverse-mode differentiation on a purely functional language with array operations. It is the first one to deliver a provably efficient, purely functional, and denotationally correct reverse-mode differentiation. We show that our transformation is semantically correct and verifies the cheap gradient principle. Inspired by PROPs and compilation to categories, we introduce a novel intermediate representation that we call 'unary form'. Our reverse-mode transformation is factored as a compilation scheme through this intermediate representation. We obtain provably efficient gradients by performing general partial evaluation optimizations after our reverse-mode transformation, as opposed to manually derived ones. For simple first-order programs, the obtained output programs resemble static-single-assignment (SSA) code. We emphasize the modularity of our approach and show how our language can easily be enriched with more optimized primitives, as required for some speed-ups in practice.

Today: 2 ideas from functional programming for AD

Backpropagation with Continuation Callbacks: Foundations for Efficient and Expressive Differentiable Programming

Fei Wang
Purdue University
West Lafayette, IN 47906
wang603@purdue.edu

James Decker
Purdue University
West Lafayette, IN 47906
decker31@purdue.edu

Xilun Wu
Purdue University
West Lafayette, IN 47906
wu636@purdue.edu

Grégory Essertel
Purdue University
West Lafayette, IN, 47906
gesserte@purdue.edu

Tiark Rompf
Purdue University
West Lafayette, IN, 47906
tiark@purdue.edu

"Continuations" for AD

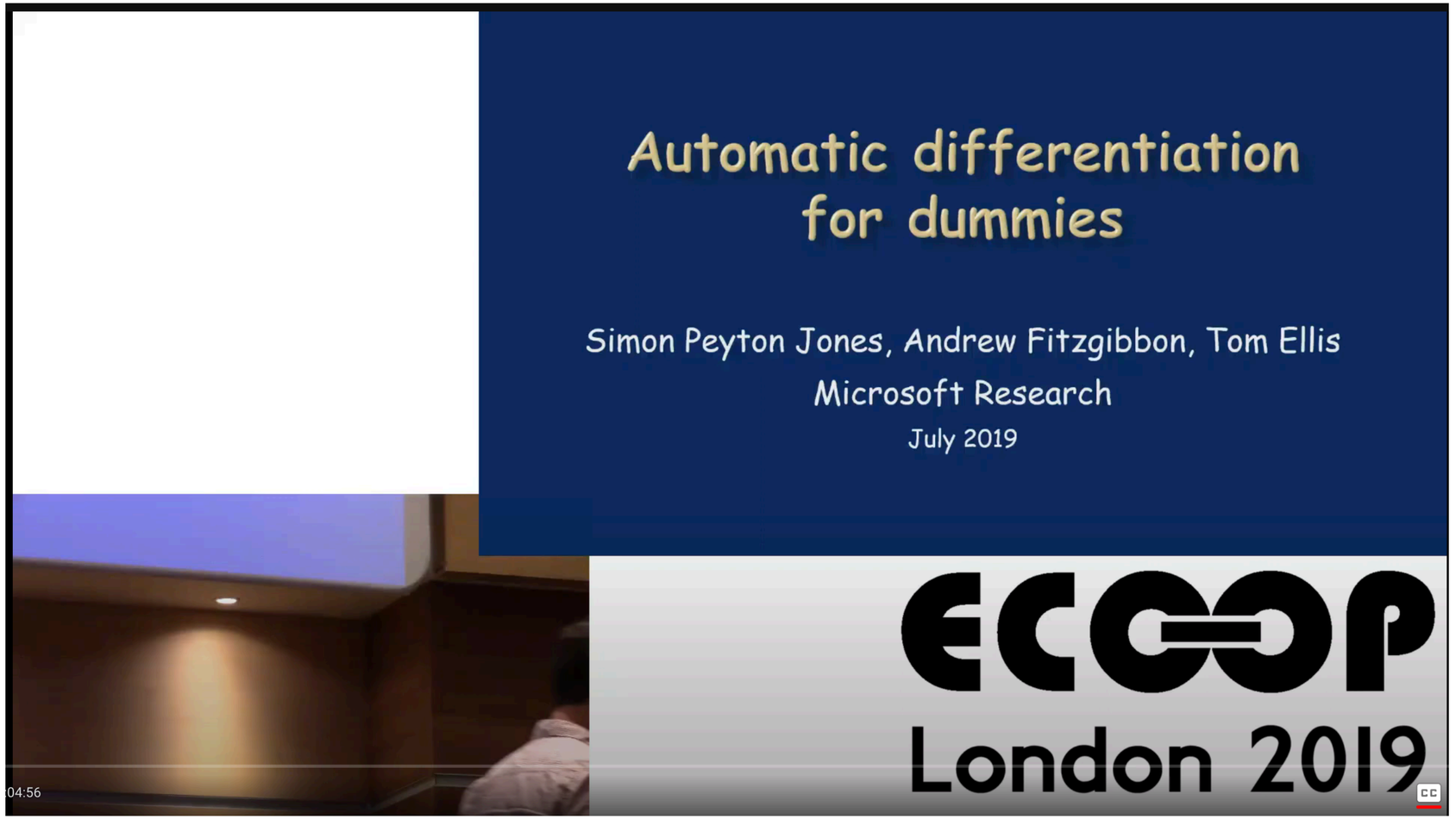
The Simple Essence of Automatic Differentiation

CONAL ELLIOTT, Target, USA

Automatic differentiation (AD) in reverse mode (RAD) is a central component of deep learning and other uses of large-scale optimization. Commonly used RAD algorithms such as backpropagation, however, are complex and stateful, hindering deep understanding, improvement, and parallel execution. This paper develops a simple, generalized AD algorithm calculated from a simple, natural specification. The general algorithm is then specialized by varying the representation of derivatives. In particular, applying well-known constructions to a naive representation yields two RAD algorithms that are far simpler than previously known. In contrast to commonly used RAD implementations, the algorithms defined here involve no graphs, tapes, variables, partial derivatives, or mutation. They are inherently parallel-friendly, correct by construction, and usable directly from an existing programming language with no need for new data types or programming style, thanks to use of an AD-agnostic compiler plugin.

The "essence" of AD

Based on a fantastic talk by SPJ



https://www.youtube.com/watch?v=FtnkqIsfNQc

How can we maximally generalize the idea of differentiation?

Given $f : S \to T$, what is the derivative of f?

How can we maximally generalize the idea of differentiation?

Given $f : S \to T$, what is the derivative of f?

Derivative is the closest linear approximation around a point:

$$f(p + \Delta_p) \approx f(p) + Df(p)(\Delta_p)$$

How can we maximally generalize the idea of differentiation?

Given $f : S \to T$, what is the derivative of f?

what is linear??

Derivative is the closest linear approximation around a point:

$$f(p + \Delta_p) \approx f(p) + Df(p)(\Delta_p)$$

Linear maps and application of linear maps

$f : S \to T$

$Df : S \to (S \multimap T)$

$S \multimap T$: a linear map from S to T

$f(p + \Delta_p) \approx f(p) + Df(p) \odot \Delta_p$

application of linear map

$\odot : (S \multimap T) \to S \to T$

Application of linear map defines its “semantics”

a linear map, $m : S \multimap T$ satisfies:

$$m \odot \left(x + y\right) = m \odot x + m \odot y$$

$$k * (m \odot x) = m \odot \left(k * x\right)$$

application of linear map
$\odot : (S \multimap T) \to S \to T$

Application of linear map defines its "semantics"

a linear map, $m : S \multimap T$ satisfies:

what is +?

$$m \odot (x + y) = m \odot x + m \odot y$$

$$k * (m \odot x) = m \odot (k * x)$$

what is *?

application of linear map

$\odot : (S \multimap T) \to S \to T$

Linear maps apply to vector spaces

a vector space V over a field F supports the following operations

$+ : V \to V \to V$
$* : F \to V \to V$ (scalar multiply)
$0 : V$
$\cdot : V \to V \to F$ (dot product)

plus a set of laws: associativity, distributivity, v+0=v, v+(-v)=0, …

Chain rule = composition of linear maps

$$f(x) = g(h(x))$$

$$f : R \to T \qquad Df : R \to R \multimap T$$

$$h : R \to S \qquad Dh : R \to R \multimap S$$

$$g : S \to T \qquad Dg : S \to S \multimap T$$

$$Df(x) = Dg(h(x)) \circ Dh(x)$$

what is the composition of linear maps??

$$\circ : (S \multimap T) \to (R \multimap S) \to (R \multimap T)$$

Semantic of chain rule

$$Df(x) = Dg(h(x)) \circ Dh(x)$$

$$\circ : (S \multimap T) \to (R \multimap S) \to (R \multimap T)$$

the semantic is defined by how it interacts with the application of linear map $\odot$

$$(m_1 \circ m_2) \odot \Delta_p = m_1 \odot \left(m_2 \odot \Delta_p\right)$$

“composition of m1 & m2” = “first apply m2, then apply m1”

Reverse mode: transposition of linear maps

application of linear map

$$\odot : (S \multimap T) \to S \to T$$

want to define the transpose

$$\odot_R : (S \multimap T) \to T \to S$$

Reverse mode: transposition of linear maps

application of linear map

$$\odot : (S \multimap T) \to S \to T$$

want to define the transpose

$$\odot_R : (S \multimap T) \to T \to S$$

$$(m \odot s) \cdot t = (m \odot_R t) \cdot s$$

this property uniquely defines $\odot_R$

$$< x | Ay > = < A^T x | y >$$

similar to how the above equation uniquely defines A^T

The Essence of AD

$$f : S \to T$$

$$Df : S \to (S \multimap T)$$

$$f(p + \Delta_p) \approx f(p) + Df(p) \odot \Delta_p$$

The Essence of AD

$$f : S \to T$$

$$Df : S \to (S \multimap T)$$

$$f(p + \Delta_p) \approx f(p) + Df(p) \odot \Delta_p$$

$$Df(x) = Dg(h(x)) \circ Dh(x)$$

$$m \odot (x + y) = m \odot x + m \odot y$$

$$k * (m \odot x) = m \odot (k * x)$$

The Essence of AD

$$f : S \to T$$

$$Df : S \to (S \multimap T)$$

$$f(p + \Delta_p) \approx f(p) + Df(p) \odot \Delta_p$$

$$Df(x) = Dg(h(x)) \circ Dh(x)$$

$$(m_1 \circ m_2) \odot \Delta_p = m_1 \odot (m_2 \odot \Delta_p)$$

$$m \odot (x + y) = m \odot x + m \odot y$$

$$k * (m \odot x) = m \odot (k * x)$$

The Essence of AD

$$f : S \to T$$

$$Df : S \to (S \multimap T)$$

$$f(p + \Delta_p) \approx f(p) + Df(p) \odot \Delta_p$$

$$Df(x) = Dg(h(x)) \circ Dh(x)$$

$$(m_1 \circ m_2) \odot \Delta_p = m_1 \odot (m_2 \odot \Delta_p)$$

$$m \odot (x + y) = m \odot x + m \odot y$$

$$k * (m \odot x) = m \odot (k * x)$$

$$(m \odot s) \cdot t = (m \odot_R t) \cdot s$$

Zipper in Haskell

this is called a “sum type”

```
union Tree2 {
  Leaf l;
  Node2 n;
}
```

this is called a “product type”

```
struct Node2 {
  Tree2 a, b;
}
```

https://en.wikibooks.org/wiki/Haskell/Zippers
https://en.wikipedia.org/wiki/Zipper_(data_structure)

Zipper in Haskell

this is called a “sum type”

```
union Tree2 {
  Leaf l;
  Node2 n;
}
```

this is called a “product type”

```
struct Node2 {
  Tree2 a, b;
}
```

derivative of data type w.r.t. x ~= “the relative location of x”

https://en.wikibooks.org/wiki/Haskell/Zippers
https://en.wikipedia.org/wiki/Zipper_(data_structure)

Zipper in Haskell

this is called a “sum type”

```
union Tree2 {
  Leaf l;
  Node2 n;
}
```

this is called a “product type”

```
struct Node2 {
  Tree2 a, b;
}
```

derivative of data type w.r.t. x ~= “the relative location of x”

for example, a root of a binary tree has the following type:
Tree2 -> Leaf + Tree2 x Tree2
If differentiate with Tree2, the derivative is 2 * Tree2, which is
1) a boolean value indicating whether the node is on left or right branches, and
2) the sibling branch of the tree

https://en.wikibooks.org/wiki/Haskell/Zippers
https://en.wikipedia.org/wiki/Zipper_(data_structure)

Zipper in Haskell

this is called a “sum type”

```
union Tree3 {
  Leaf l;
  Node3 n;
}
```

this is called a “product type”

```
struct Node3 {
  Tree3 a, b;
}
```

derivative of data type w.r.t. x ~= “the relative location of x”

for example, a root of a binary tree has the following type:
Tree3 -> Leaf + Tree3 x Tree3
If differentiate with Tree3, the derivative is 3 * Tree3 * Tree3, which is
1) a ternary value indicating which branch the node is on, and
2) the two sibling branches of the tree

https://en.wikibooks.org/wiki/Haskell/Zippers
https://en.wikipedia.org/wiki/Zipper_(data_structure)

Zipper in Haskell

this is called a “sum type”

```
union Tree3 {
  Leaf l;
  Node3 n;
}
```

this is called a “product type”

```
struct Node3 {
  Tree3 a, b;
}
```

Zipper is useful in functional languages to
1) traverse data structure
2) add/remove items in data structure

derivative of data type w.r.t. x ~= “the relative location of x”

for example, a root of a binary tree has the following type:
Tree3 -> Leaf + Tree3 x Tree3
If differentiate with Tree3, the derivative is 3 * Tree3 * Tree3, which is
1) a ternary value indicating which branch the node is on, and
2) the two sibling branches of the tree

https://en.wikibooks.org/wiki/Haskell/Zippers
https://en.wikipedia.org/wiki/Zipper_(data_structure)