Automatic Differentiation via Effects and Handlers: An Implementation in Frank

Jesse Sigal

School of Informatics, University of Edinburgh

PEPM'21
January 19, 2021

Summary

Summary

Automatic differentiation (AD) is an important family of algorithms which enables derivative based optimization. We show that AD can be simply implemented with effects and handlers by doing so in the Frank language.

Aside on AD

Aside on AD

- We will show the structure of AD algorithms, but not mathematically justify the algorithms.

Aside on AD

- We will show the structure of AD algorithms, but not mathematically justify the algorithms.
- Let $f, g \colon \mathbb{R} \to \mathbb{R}$ be smooth functions (i.e. infinitely differentiable at all points). The chain rule states that

$$(f \circ g)'(x) = f'(g(x)) \cdot g'(x).$$

  AD algorithms use this compositional property to incrementally calculate the derivative of an entire program one basic operation at a time during evaluation.

Aside on AD

- We will show the structure of AD algorithms, but not mathematically justify the algorithms.
- Let $f, g \colon \mathbb{R} \to \mathbb{R}$ be smooth functions (i.e. infinitely differentiable at all points). The chain rule states that

$$(f \circ g)'(x) = f'(g(x)) \cdot g'(x).$$

  AD algorithms use this compositional property to incrementally calculate the derivative of an entire program one basic operation at a time during evaluation.
- The two main categories of AD are *forward mode* and *reverse mode*.

Aside on AD

- We will show the structure of AD algorithms, but not mathematically justify the algorithms.
- Let $f, g \colon \mathbb{R} \to \mathbb{R}$ be smooth functions (i.e. infinitely differentiable at all points). The chain rule states that

$$(f \circ g)'(x) = f'(g(x)) \cdot g'(x).$$

  AD algorithms use this compositional property to incrementally calculate the derivative of an entire program one basic operation at a time during evaluation.
- The two main categories of AD are *forward mode* and *reverse mode*.
- For a program with n inputs and m outputs, forward mode is $O(n)$ and reverse mode is $O(m)$.

Frank introduction

Frank introduction

- Frank is a strict functional programming language with effects and handlers.

Frank introduction

- Frank is a strict functional programming language with effects and handlers.
- Frank's evaluation order is left-to-right.

Frank introduction

- Frank is a strict functional programming language with effects and handlers.
- Frank's evaluation order is left-to-right.
- It is inspired by call-by-push-value (CBPV), meaning there is a distinction between values and computations.

Frank introduction

- Frank is a strict functional programming language with effects and handlers.
- Frank's evaluation order is left-to-right.
- It is inspired by call-by-push-value (CBPV), meaning there is a distinction between values and computations.
- Frank unifies functions and handlers.

Frank introduction

- Frank is a strict functional programming language with effects and handlers.
- Frank's evaluation order is left-to-right.
- It is inspired by call-by-push-value (CBPV), meaning there is a distinction between values and computations.
- Frank unifies functions and handlers.
- Frank's handlers are shallow. This allows flexible changing of how effects are handled, which can be advantageous specifically for AD.

Frank introduction

- Frank is a strict functional programming language with effects and handlers.
- Frank's evaluation order is left-to-right.
- It is inspired by call-by-push-value (CBPV), meaning there is a distinction between values and computations.
- Frank unifies functions and handlers.
- Frank's handlers are shallow. This allows flexible changing of how effects are handled, which can be advantageous specifically for AD.
- The type system propagates the needed effects "inwards".

Frank introduction (cont.)

```
state : S -> <State S>X -> [Console]X
state _ x             = print "end"; x
state s <get   -> k> = print "get"; state s (k s)
state _ <put s -> k> = print "put"; state s (k unit)

main : {[Console]Int}
main! = state (print "start"; 1) (2 + (put (get! + get!); get!))
```

Frank introduction (cont.)

```
state : S -> <State S>X -> [Console]X
state _ x             = print "end"; x
state s <get   -> k> = print "get"; state s (k s)
state _ <put s -> k> = print "put"; state s (k unit)

main : {[Console]Int}
main! = state (print "start"; 1) (2 + (put (get! + get!); get!))
```

- Calling context must support the *ability* `[Console]`

Frank introduction (cont.)

```
state : S -> <State S>X -> [Console]X
state _ x             = print "end"; x
state s <get   -> k> = print "get"; state s (k s)
state _ <put s -> k> = print "put"; state s (k unit)

main : {[Console]Int}
main! = state (print "start"; 1) (2 + (put (get! + get!); get!))
```

- Calling context must support the *ability* `[Console]`
- ... which is a snoc-list containing exactly one *instance* `Console`

Frank introduction (cont.)

```
state : S -> <State S>X -> [Console]X
state _ x             = print "end"; x
state s <get   -> k> = print "get"; state s (k s)
state _ <put s -> k> = print "put"; state s (k unit)

main : {[Console]Int}
main! = state (print "start"; 1) (2 + (put (get! + get!); get!))
```

- Calling context must support the *ability* `[Console]`
- ... which is a snoc-list containing exactly one *instance* `Console`
- ... of the *interface* `Console`.

Frank introduction (cont.)

```
state : S -> <State S>X -> [Console]X
state _ x             = print "end"; x
state s <get   -> k> = print "get"; state s (k s)
state _ <put s -> k> = print "put"; state s (k unit)

main : {[Console]Int}
main! = state (print "start"; 1) (2 + (put (get! + get!); get!))
```

- Calling context must support the *ability* `[Console]`
- ... which is a snoc-list containing exactly one *instance* `Console`
- ... of the *interface* `Console`.
- The ability `[Console]` means we can use the *command* `print` defined by the interface `Console`.

Frank introduction (cont.)

```
state : S -> <State S>X -> [Console]X
state _ x             = print "end"; x
state s <get   -> k> = print "get"; state s (k s)
state _ <put s -> k> = print "put"; state s (k unit)

main : {[Console]Int}
main! = state (print "start"; 1) (2 + (put (get! + get!); get!))
```

- Calling context must support the *ability* `[Console]`
- ... which is a snoc-list containing exactly one *instance* `Console`
- ... of the *interface* `Console`.
- The ability `[Console]` means we can use the *command* `print` defined by the interface `Console`.
- The first argument can only be computed using commands from the instances in the ability `[Console]`.

Frank introduction (cont.)

```
state : S -> <State S>X -> [Console]X
state _ x             = print "end"; x
state s <get    -> k> = print "get"; state s (k s)
state _ <put s -> k> = print "put"; state s (k unit)

main : {[Console]Int}
main! = state (print "start"; 1) (2 + (put (get! + get!); get!))
```

- Calling context must support the *ability* `[Console]`
- ... which is a snoc-list containing exactly one *instance* `Console`
- ... of the *interface* `Console`.
- The ability `[Console]` means we can use the *command* `print` defined by the interface `Console`.
- The first argument can only be computed using commands from the instances in the ability `[Console]`.
- The second argument can use commands from the ability `[Console, State S]`.

Frank introduction (cont.)

```
state : S -> <State S>X -> [Console]X
state _ x             = print "end"; x
state s <get   -> k> = print "get"; state s (k s)
state _ <put s -> k> = print "put"; state s (k unit)

main : {[Console]Int}
main! = state (print "start"; 1) (2 + (put (get! + get!); get!))
```

- Calling context must support the *ability* `[Console]`
- ... which is a snoc-list containing exactly one *instance* `Console`
- ... of the *interface* `Console`.
- The ability `[Console]` means we can use the *command* `print` defined by the interface `Console`.
- The first argument can only be computed using commands from the instances in the ability `[Console]`.
- The second argument can use commands from the ability `[Console, State S]`.
- The second argument has access to `State` commands because the *adjustment* `<State S>` extends the ambient ability `[Console]`.

Frank handlers

Frank handlers

`evaluate` the most basic handler which dispatches to builtin arithmetic operations;

`diff` an implementation of forward mode AD;

`reverse` an implementation of reverse mode AD which makes use of the builtin mutable state interface; and

`reversec` an implementation of checkpointed reverse mode AD which extends `reverse`.

Auxiliary definitions

Auxiliary definitions

```
data Nullary = constE Int
data Unary   = negateE
data Binary  = plusE | timesE

interface Smooth X =
      ap0 : Nullary -> X
    | ap1 : Unary   -> X -> X
    | ap2 : Binary  -> X -> X -> X
```

Auxiliary definitions

```
data Nullary = constE Int
data Unary   = negateE
data Binary  = plusE | timesE

interface Smooth X =
      ap0 : Nullary -> X
    | ap1 : Unary   -> X -> X
    | ap2 : Binary  -> X -> X -> X

c : Int -> [Smooth X] X
c i = ap0 (constE i)

n : X -> [Smooth X] X
n x = ap1 negateE x

p : X -> X -> [Smooth X] X
p x y = ap2 plusE x y

t : X -> X -> [Smooth X] X
t x y = ap2 timesE x y
```

Auxiliary definitions

```
data Nullary = constE Int
data Unary   = negateE
data Binary  = plusE | timesE

interface Smooth X =
      ap0 : Nullary -> X
    | ap1 : Unary   -> X -> X
    | ap2 : Binary  -> X -> X -> X

c : Int -> [Smooth X] X
c i = ap0 (constE i)

n : X -> [Smooth X] X
n x = ap1 negateE x

p : X -> X -> [Smooth X] X
p x y = ap2 plusE x y

t : X -> X -> [Smooth X] X
t x y = ap2 timesE x y
```

```
op0 : Nullary ->            [Smooth X] X
op1 : Unary   -> X ->       [Smooth X] X
op2 : Binary  -> X -> X ->  [Smooth X] X
op0 (constE i)      = c i
op1 negateE    x    = n x
op2 plusE      x y  = p x y
op2 timesE     x y  = t x y
```

Auxiliary definitions

```
data Nullary = constE Int
data Unary   = negateE
data Binary  = plusE | timesE

interface Smooth X =
      ap0 : Nullary -> X
    | ap1 : Unary   -> X -> X
    | ap2 : Binary  -> X -> X -> X

c : Int -> [Smooth X] X
c i = ap0 (constE i)

n : X -> [Smooth X] X
n x = ap1 negateE x

p : X -> X -> [Smooth X] X
p x y = ap2 plusE x y

t : X -> X -> [Smooth X] X
t x y = ap2 timesE x y
```

```
op0 : Nullary ->            [Smooth X] X
op1 : Unary   -> X ->       [Smooth X] X
op2 : Binary  -> X -> X ->  [Smooth X] X
op0 (constE i)      = c i
op1 negateE    x    = n x
op2 plusE      x y  = p x y
op2 timesE     x y  = t x y
```

```
der1  : Unary  -> X ->       [Smooth X] X
der2L : Binary -> X -> X ->  [Smooth X] X
der2R : Binary -> X -> X ->  [Smooth X] X
der1  negateE x    = n (c 1)   d/dx(−x) = −1
der2L plusE   x y  = c 1       d/dx(x + y) = 1
der2L timesE  x y  = y         d/dx(x · y) = y
der2R plusE   x y  = c 1       d/dy(x + y) = 1
der2R timesE  x y  = x         d/dy(x · y) = x
```

The `evaluate` handler

The `evaluate` handler

The top level handler to remove `Smooth` effects.

The `evaluate` handler

The top level handler to remove `Smooth` effects.

```
evaluate : <Smooth Int> X -> X
evaluate x = x
evaluate <ap0 (constE i) -> k> = evaluate (k i)
evaluate <ap1 negateE x  -> k> = evaluate (k (-x))
evaluate <ap2 plusE x y  -> k> = evaluate (k (x + y))
evaluate <ap2 timesE x y -> k> = evaluate (k (x * y))
```

The `evaluate` handler

The top level handler to remove `Smooth` effects.

```
evaluate : <Smooth Int> X -> X
evaluate x = x
evaluate <ap0 (constE i) -> k> = evaluate (k i)
evaluate <ap1 negateE x  -> k> = evaluate (k (-x))
evaluate <ap2 plusE x y  -> k> = evaluate (k (x + y))
evaluate <ap2 timesE x y -> k> = evaluate (k (x * y))
```

Let us evaluate $1 + x^3 + (-y^2)$ at $x = 2$ and $y = 4$:

The `evaluate` handler

The top level handler to remove `Smooth` effects.

```
evaluate : <Smooth Int> X -> X
evaluate x = x
evaluate <ap0 (constE i) -> k> = evaluate (k i)
evaluate <ap1 negateE x  -> k> = evaluate (k (-x))
evaluate <ap2 plusE x y  -> k> = evaluate (k (x + y))
evaluate <ap2 timesE x y -> k> = evaluate (k (x * y))
```

Let us evaluate $1 + x^3 + (-y^2)$ at $x = 2$ and $y = 4$:

```
evaluate (p (c 1) (p (t (t 2 2) 2) (n (t 4 4))))
```

The `evaluate` handler

The top level handler to remove `Smooth` effects.

```
evaluate : <Smooth Int> X -> X
evaluate x = x
evaluate <ap0 (constE i) -> k> = evaluate (k i)
evaluate <ap1 negateE x  -> k> = evaluate (k (-x))
evaluate <ap2 plusE x y  -> k> = evaluate (k (x + y))
evaluate <ap2 timesE x y -> k> = evaluate (k (x * y))
```

Let us evaluate $1 + x^3 + (-y^2)$ at $x = 2$ and $y = 4$:

```
   evaluate (p (c 1) (p (t (t 2 2) 2) (n (t 4 4))))
→ evaluate (p (c 1) (p (t (t 2 2) 2) (n (t 4 4))))
```

The `evaluate` handler

The top level handler to remove `Smooth` effects.

```
evaluate : <Smooth Int> X -> X
evaluate x = x
evaluate <ap0 (constE i) -> k> = evaluate (k i)
evaluate <ap1 negateE x  -> k> = evaluate (k (-x))
evaluate <ap2 plusE x y  -> k> = evaluate (k (x + y))
evaluate <ap2 timesE x y -> k> = evaluate (k (x * y))
```

Let us evaluate $1 + x^3 + (-y^2)$ at $x = 2$ and $y = 4$:

```
   evaluate (p (c 1) (p (t (t 2 2) 2) (n (t 4 4))))
→ evaluate (p (c 1) (p (t (t 2 2) 2) (n (t 4 4))))
→ evaluate (p (c 1) (p (t (t 2 2) 2) (n (t 4 4))))
```

The `evaluate` handler

The top level handler to remove `Smooth` effects.

```
evaluate : <Smooth Int> X -> X
evaluate x = x
evaluate <ap0 (constE i) -> k> = evaluate (k i)
evaluate <ap1 negateE x  -> k> = evaluate (k (-x))
evaluate <ap2 plusE x y  -> k> = evaluate (k (x + y))
evaluate <ap2 timesE x y -> k> = evaluate (k (x * y))
```

Let us evaluate $1 + x^3 + (-y^2)$ at $x = 2$ and $y = 4$:

```
   evaluate (p (c 1) (p (t (t 2 2) 2) (n (t 4 4))))
→ evaluate (p (c 1) (p (t (t 2 2) 2) (n (t 4 4))))
→ evaluate (p (c 1) (p (t (t 2 2) 2) (n (t 4 4))))
→ evaluate (p (c 1) (p (t (t 2 2) 2) (n (t 4 4))))
```

The `evaluate` handler

The top level handler to remove `Smooth` effects.

```
evaluate : <Smooth Int> X -> X
evaluate x = x
evaluate <ap0 (constE i) -> k> = evaluate (k i)
evaluate <ap1 negateE x  -> k> = evaluate (k (-x))
evaluate <ap2 plusE x y  -> k> = evaluate (k (x + y))
evaluate <ap2 timesE x y -> k> = evaluate (k (x * y))
```

Let us evaluate $1 + x^3 + (-y^2)$ at $x = 2$ and $y = 4$:

```
   evaluate (p (c 1) (p (t (t 2 2) 2) (n (t 4 4))))
→ evaluate (p (c 1) (p (t (t 2 2) 2) (n (t 4 4))))
→ evaluate (p (c 1) (p (t (t 2 2) 2) (n (t 4 4))))
→ evaluate (p (c 1) (p (t (t 2 2) 2) (n (t 4 4))))
→ evaluate ({x -> (p x (p (t (t 2 2) 2) (n (t 4 4))))} 1)
```

The `evaluate` handler

The top level handler to remove `Smooth` effects.

```
evaluate : <Smooth Int> X -> X
evaluate x = x
evaluate <ap0 (constE i) -> k> = evaluate (k i)
evaluate <ap1 negateE x  -> k> = evaluate (k (-x))
evaluate <ap2 plusE x y  -> k> = evaluate (k (x + y))
evaluate <ap2 timesE x y -> k> = evaluate (k (x * y))
```

Let us evaluate $1 + x^3 + (-y^2)$ at $x = 2$ and $y = 4$:

```
   evaluate (p (c 1) (p (t (t 2 2) 2) (n (t 4 4))))
→ evaluate (p (c 1) (p (t (t 2 2) 2) (n (t 4 4))))
→ evaluate (p (c 1) (p (t (t 2 2) 2) (n (t 4 4))))
→ evaluate (p (c 1) (p (t (t 2 2) 2) (n (t 4 4))))
→ evaluate ({x -> (p x (p (t (t 2 2) 2) (n (t 4 4))))} 1)
→ evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
```

The `evaluate` handler (cont.)

```
evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
```

The `evaluate` handler (cont.)

```
  evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→ evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
```

The `evaluate` handler (cont.)

```
   evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→  evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→  evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
```

The `evaluate` handler (cont.)

```
   evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→  evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→  evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→  evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
```

The `evaluate` handler (cont.)

```
   evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→  evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→  evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→  evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→  evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
```

The `evaluate` handler (cont.)

```
   evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→  evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→  evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→  evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→  evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→  evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
```

The `evaluate` handler (cont.)

```
   evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→ evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→ evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→ evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→ evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→ evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→ evaluate ({x -> (p 1 (p (t x 2) (n (t 4 4))))} (2 * 2))
```

The `evaluate` handler (cont.)

```
   evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→  evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→  evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→  evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→  evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→  evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→  evaluate ({x -> (p 1 (p (t x 2) (n (t 4 4))))} (2 * 2))
→  evaluate ({x -> (p 1 (p (t x 2) (n (t 4 4))))} 4)
```

The `evaluate` handler (cont.)

```
   evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→  evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→  evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→  evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→  evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→  evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→  evaluate ({x -> (p 1 (p (t x 2) (n (t 4 4))))} (2 * 2))
→  evaluate ({x -> (p 1 (p (t x 2) (n (t 4 4))))} 4)
→  evaluate (p 1 (p (t 4 2) (n (t 4 4))))
```

The `evaluate` handler (cont.)

```
   evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→  evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→  evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→  evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→  evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→  evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→  evaluate ({x -> (p 1 (p (t x 2) (n (t 4 4))))} (2 * 2))
→  evaluate ({x -> (p 1 (p (t x 2) (n (t 4 4))))} 4)
→  evaluate (p 1 (p (t 4 2) (n (t 4 4))))
...
```

The `evaluate` handler (cont.)

```
   evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→ evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→ evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→ evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→ evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→ evaluate (p 1 (p (t (t 2 2) 2) (n (t 4 4))))
→ evaluate ({x -> (p 1 (p (t x 2) (n (t 4 4))))} (2 * 2))
→ evaluate ({x -> (p 1 (p (t x 2) (n (t 4 4))))} 4)
→ evaluate (p 1 (p (t 4 2) (n (t 4 4))))
...
→ -7
```

The `diff` handler

The `diff` handler

```
data Dual X = dual X X

v : Dual X -> X
v (dual x _) = x

dv : Dual X -> X
dv (dual _ dx) = dx

diff : <Smooth (Dual X)> Y -> [Smooth X] Y
diff x = x
diff <ap0 n -> k> =
  let r = dual (op0 n) (c 0) in
  diff (k r)
diff <ap1 u (dual x dx) -> k> =
  let r = dual (op1 u x) (t (der1 u x) dx) in
  diff (k r)
diff <ap2 b (dual x dx) (dual y dy) -> k> =
  let r = dual (op2 b x y) (p (t (der2L b x y) dx)
                              (t (der2R b x y) dy)) in
  diff (k r)
```

- The forward mode AD handler.

The `diff` handler

```
data Dual X = dual X X

v : Dual X -> X
v (dual x _) = x

dv : Dual X -> X
dv (dual _ dx) = dx

diff : <Smooth (Dual X)> Y -> [Smooth X] Y
diff x = x
diff <ap0 n -> k> =
  let r = dual (op0 n) (c 0) in
  diff (k r)
diff <ap1 u (dual x dx) -> k> =
  let r = dual (op1 u x) (t (der1 u x) dx) in
  diff (k r)
diff <ap2 b (dual x dx) (dual y dy) -> k> =
  let r = dual (op2 b x y) (p (t (der2L b x y) dx)
                              (t (der2R b x y) dy)) in
  diff (k r)
```

- The forward mode AD handler.
- `Dual` pairs a value with its derivative.

The `diff` handler

```
data Dual X = dual X X

v : Dual X -> X
v (dual x _) = x

dv : Dual X -> X
dv (dual _ dx) = dx

diff : <Smooth (Dual X)> Y -> [Smooth X] Y
diff x = x
diff <ap0 n -> k> =
  let r = dual (op0 n) (c 0) in
  diff (k r)
diff <ap1 u (dual x dx) -> k> =
  let r = dual (op1 u x) (t (der1 u x) dx) in
  diff (k r)
diff <ap2 b (dual x dx) (dual y dy) -> k> =
  let r = dual (op2 b x y) (p (t (der2L b x y) dx)
                              (t (der2R b x y) dy)) in
  diff (k r)
```

- The forward mode AD handler.
- `Dual` pairs a value with its derivative.
- Handled operations calculate

The `diff` handler

```
data Dual X = dual X X

v : Dual X -> X
v (dual x _) = x

dv : Dual X -> X
dv (dual _ dx) = dx

diff : <Smooth (Dual X)> Y -> [Smooth X] Y
diff x = x
diff <ap0 n -> k> =
  let r = dual (op0 n) (c 0) in
  diff (k r)
diff <ap1 u (dual x dx) -> k> =
  let r = dual (op1 u x) (t (der1 u x) dx) in
  diff (k r)
diff <ap2 b (dual x dx) (dual y dy) -> k> =
  let r = dual (op2 b x y) (p (t (der2L b x y) dx)
                              (t (der2R b x y) dy)) in
  diff (k r)
```

- The forward mode AD handler.
- `Dual` pairs a value with its derivative.
- Handled operations calculate
  - the value and

The `diff` handler

```
data Dual X = dual X X

v : Dual X -> X
v (dual x _) = x

dv : Dual X -> X
dv (dual _ dx) = dx

diff : <Smooth (Dual X)> Y -> [Smooth X] Y
diff x = x
diff <ap0 n -> k> =
  let r = dual (op0 n) (c 0) in
  diff (k r)
diff <ap1 u (dual x dx) -> k> =
  let r = dual (op1 u x) (t (der1 u x) dx) in
  diff (k r)
diff <ap2 b (dual x dx) (dual y dy) -> k> =
  let r = dual (op2 b x y) (p (t (der2L b x y) dx)
                              (t (der2R b x y) dy)) in
  diff (k r)
```

- The forward mode AD handler.
- `Dual` pairs a value with its derivative.
- Handled operations calculate
  - the value and
  - the derivative.

The `diff` handler

```
data Dual X = dual X X

v : Dual X -> X
v (dual x _) = x

dv : Dual X -> X
dv (dual _ dx) = dx

diff : <Smooth (Dual X)> Y -> [Smooth X] Y
diff x = x
diff <ap0 n -> k> =
  let r = dual (op0 n) (c 0) in
  diff (k r)
diff <ap1 u (dual x dx) -> k> =
  let r = dual (op1 u x) (t (der1 u x) dx) in
  diff (k r)
diff <ap2 b (dual x dx) (dual y dy) -> k> =
  let r = dual (op2 b x y) (p (t (der2L b x y) dx)
                              (t (der2R b x y) dy)) in
  diff (k r)
```

- The forward mode AD handler.
- `Dual` pairs a value with its derivative.
- Handled operations calculate
  - the value and
  - the derivative.
- Calculations are via another instance of `Smooth`.

The `diff` handler

```
data Dual X = dual X X

v : Dual X -> X
v (dual x _) = x

dv : Dual X -> X
dv (dual _ dx) = dx

diff : <Smooth (Dual X)> Y -> [Smooth X] Y
diff x = x
diff <ap0 n -> k> =
  let r = dual (op0 n) (c 0) in
  diff (k r)
diff <ap1 u (dual x dx) -> k> =
  let r = dual (op1 u x) (t (der1 u x) dx) in
  diff (k r)
diff <ap2 b (dual x dx) (dual y dy) -> k> =
  let r = dual (op2 b x y) (p (t (der2L b x y) dx)
                              (t (der2R b x y) dy)) in
  diff (k r)
```

- The forward mode AD handler.
- `Dual` pairs a value with its derivative.
- Handled operations calculate
  - the value and
  - the derivative.
- Calculations are via another instance of `Smooth`.
- Execution then continues.

The `diff` handler (cont.)

The `diff` handler (cont.)

Let us differentiate $1 + x^3 + (-y^2)$ with respect to x at $x = 2$ and $y = 4$ which should give $3x^2$ at $x = 2$ which is 12:

The `diff` handler (cont.)

Let us differentiate $1 + x^3 + (-y^2)$ with respect to x at $x = 2$ and $y = 4$ which should give $3x^2$ at $x = 2$ which is 12:

```
evaluate (diff (
  p (c 1) (p (t (t (dual 2 1) (dual 2 1)) (dual 2 1))
             (n (t (dual 4 0) (dual 4 0))))
))
```

The `diff` handler (cont.)

Let us differentiate $1 + x^3 + (-y^2)$ with respect to x at $x = 2$ and $y = 4$ which should give $3x^2$ at $x = 2$ which is 12:

```
evaluate (diff (
  p (c 1) (p (t (t (dual 2 1) (dual 2 1)) (dual 2 1))
             (n (t (dual 4 0) (dual 4 0))))
))
```

→

```
evaluate (diff (
  p (c 1) (p (t (t (dual 2 1) (dual 2 1)) (dual 2 1))
             (n (t (dual 4 0) (dual 4 0))))
))
```

The `diff` handler (cont.)

Let us differentiate $1 + x^3 + (-y^2)$ with respect to x at $x = 2$ and $y = 4$ which should give $3x^2$ at $x = 2$ which is 12:

```
evaluate (diff (
  p (c 1) (p (t (t (dual 2 1) (dual 2 1)) (dual 2 1))
             (n (t (dual 4 0) (dual 4 0))))
))
```

→

```
evaluate (diff (
  p (c 1) (p (t (t (dual 2 1) (dual 2 1)) (dual 2 1))
             (n (t (dual 4 0) (dual 4 0))))
))
```

→

```
evaluate (
  let r = dual (op0 (constE 1)) (c 0) in
  diff ({x -> (p x (p (t (t (dual 2 1) (dual 2 1)) (dual 2 1))
                      (n (t (dual 4 0) (dual 4 0))))))} r)
)
```

The `diff` handler (cont.)

```
→ evaluate (
    let r = dual (c 1) (c 0) in
    diff ({x -> (p x (p (t (t (dual 2 1) (dual 2 1)) (dual 2 1))
                        (n (t (dual 4 0) (dual 4 0))))))} r)
  )
```

The `diff` handler (cont.)

```
→ evaluate (
    let r = dual (c 1) (c 0) in
    diff ({x -> (p x (p (t (t (dual 2 1) (dual 2 1)) (dual 2 1))
                        (n (t (dual 4 0) (dual 4 0))))))} r)
  )

→ evaluate (diff (
    p (dual 1 0) (p (t (t (dual 2 1) (dual 2 1)) (dual 2 1))
                    (n (t (dual 4 0) (dual 4 0))))
  ))
```

The `diff` handler (cont.)

```
→ evaluate (
    let r = dual (c 1) (c 0) in
    diff ({x -> (p x (p (t (t (dual 2 1) (dual 2 1)) (dual 2 1))
                        (n (t (dual 4 0) (dual 4 0))))))} r)
  )

→ evaluate (diff (
    p (dual 1 0) (p (t (t (dual 2 1) (dual 2 1)) (dual 2 1))
                    (n (t (dual 4 0) (dual 4 0))))
  ))

...
```

The `diff` handler (cont.)

```
→ evaluate (
    let r = dual (c 1) (c 0) in
    diff ({x -> (p x (p (t (t (dual 2 1) (dual 2 1)) (dual 2 1))
                        (n (t (dual 4 0) (dual 4 0))))))} r)
  )

→ evaluate (diff (
    p (dual 1 0) (p (t (t (dual 2 1) (dual 2 1)) (dual 2 1))
                    (n (t (dual 4 0) (dual 4 0))))
  ))

...

→ dual -7 12
```

Dynamically choosing handlers

Dynamically choosing handlers

Frank's type system guarantees effects are handled, but we can dynamically choose which handler is used.

Dynamically choosing handlers

Frank's type system guarantees effects are handled, but we can dynamically choose which handler is used.

```
lift : X -> [Smooth X, Smooth (Dual X)] (Dual X)
lift x = dual x (<Smooth> (c 0))

d : {(Dual X) -> [Smooth X, Smooth (Dual X)] (Dual X)}
 -> X -> [Smooth X] X
d f x = dv (diff (f (dual x (<Smooth> (c 1)))))
```

Dynamically choosing handlers

Frank's type system guarantees effects are handled, but we can dynamically choose which handler is used.

```
lift : X -> [Smooth X, Smooth (Dual X)] (Dual X)
lift x = dual x (<Smooth> (c 0))

d : {(Dual X) -> [Smooth X, Smooth (Dual X)] (Dual X)}
 -> X -> [Smooth X] X
d f x = dv (diff (f (dual x (<Smooth> (c 1)))))
```

Frank uses adaptors to aid dynamic choice.

Dynamically choosing handlers (cont.)

Dynamically choosing handlers (cont.)

Adaptors allow us to easily nest code which handles effects. Consider

$$\frac{d}{dx}\left(x \cdot \frac{d}{dy}(x+y)\Big|_{y=1}\right)\Big|_{x=1}$$

which equals 1. The `lift` function is required to express this.

Dynamically choosing handlers (cont.)

Adaptors allow us to easily nest code which handles effects. Consider

$$\frac{d}{dx}\left(x \cdot \frac{d}{dy}(x+y)\Big|_{y=1}\right)\Big|_{x=1}$$

which equals 1. The `lift` function is required to express this.

```
evaluate (d {x -> t x (d {y -> p (lift x) y} (c 1))} (c 1))
```

Dynamically choosing handlers (cont.)

Adaptors allow us to easily nest code which handles effects. Consider

$$\frac{d}{dx}\left(x \cdot \frac{d}{dy}(x+y)\Big|_{y=1}\right)\Big|_{x=1}$$

which equals 1. The `lift` function is required to express this.

```
   evaluate (d {x -> t x (d {y -> p (lift x) y} (c 1))} (c 1))
→  evaluate (d {x -> t x (d {y -> p (lift x) y} (c 1))} 1)
```

Dynamically choosing handlers (cont.)

Adaptors allow us to easily nest code which handles effects. Consider

$$\frac{d}{dx}\left(x \cdot \frac{d}{dy}(x+y)\Big|_{y=1}\right)\Big|_{x=1}$$

which equals 1. The `lift` function is required to express this.

```
   evaluate (d {x -> t x (d {y -> p (lift x) y} (c 1))} (c 1))
→  evaluate (d {x -> t x (d {y -> p (lift x) y} (c 1))} 1)
→  evaluate (dv (diff (
     {x -> t x (d {y -> p (lift x) y} (c 1))}
     (dual 1 (<Smooth> (c 1)))
   )))
```

Dynamically choosing handlers (cont.)

Adaptors allow us to easily nest code which handles effects. Consider

$$\frac{d}{dx}\left(x \cdot \frac{d}{dy}(x+y)\Big|_{y=1}\right)\Big|_{x=1}$$

which equals 1. The `lift` function is required to express this.

```
   evaluate (d {x -> t x (d {y -> p (lift x) y} (c 1))} (c 1))
→  evaluate (d {x -> t x (d {y -> p (lift x) y} (c 1))} 1)
→  evaluate (dv (diff (
     {x -> t x (d {y -> p (lift x) y} (c 1))}
     (dual 1 (<Smooth> (c 1)))
   )))
→  evaluate (dv (diff (
     {x -> t x (d {y -> p (lift x) y} (c 1))}
     (dual 1 (<Smooth> (c 1)))
   )))
```

The `reverse` handler

The `reverse` handler

```
data Prop X = prop X (Ref X)

fwd : Prop X -> X   |  deriv : Prop X -> Ref X
fwd (prop x _) = x  |  deriv (prop _ r) = r

reverse : <Smooth (Prop X)> Unit -> [RefState, Smooth X] Unit
reverse x = x
reverse <ap0 n -> k> =
  let r = prop (op0 n) (new (c 0)) in
  reverse (k r)
reverse <ap1 u (prop x dx) -> k> =
  let r = prop (op1 u x) (new (c 0)) in
  reverse (k r);
  write dx (p (read dx) (t (der1 u x) (read (deriv r))))
reverse <ap2 b (prop x dx) (prop y dy) -> k> =
  let r = prop (op2 b x y) (new (c 0)) in
  reverse (k r);
  write dx (p (read dx) (t (der2L b x y) (read (deriv r))));
  write dy (p (read dy) (t (der2R b x y) (read (deriv r))))
```

- The reverse mode AD handler.

The `reverse` handler

```
data Prop X = prop X (Ref X)

fwd : Prop X -> X    | deriv : Prop X -> Ref X
fwd (prop x _) = x   | deriv (prop _ r) = r

reverse : <Smooth (Prop X)> Unit -> [RefState, Smooth X] Unit
reverse x = x
reverse <ap0 n -> k> =
  let r = prop (op0 n) (new (c 0)) in
  reverse (k r)
reverse <ap1 u (prop x dx) -> k> =
  let r = prop (op1 u x) (new (c 0)) in
  reverse (k r);
  write dx (p (read dx) (t (der1 u x) (read (deriv r))))
reverse <ap2 b (prop x dx) (prop y dy) -> k> =
  let r = prop (op2 b x y) (new (c 0)) in
  reverse (k r);
  write dx (p (read dx) (t (der2L b x y) (read (deriv r))));
  write dy (p (read dy) (t (der2R b x y) (read (deriv r))))
```

- The reverse mode AD handler.
- `Prop` pairs a value with a cell holding its derivative.

The `reverse` handler

```
reverse : <Smooth (Prop X)> Unit -> [RefState, Smooth X] Unit
reverse x = x
reverse <ap0 n -> k> =
  let r = prop (op0 n) (new (c 0)) in
  reverse (k r)
reverse <ap1 u (prop x dx) -> k> =
  let r = prop (op1 u x) (new (c 0)) in
  reverse (k r);
  write dx (p (read dx) (t (der1 u x) (read (deriv r))))
reverse <ap2 b (prop x dx) (prop y dy) -> k> =
  let r = prop (op2 b x y) (new (c 0)) in
  reverse (k r);
  write dx (p (read dx) (t (der2L b x y) (read (deriv r))));
  write dy (p (read dy) (t (der2R b x y) (read (deriv r))))
```

- The reverse mode AD handler.
- `Prop` pairs a value with a cell holding its derivative.
- Handled operation calculates

The `reverse` handler

```
reverse : <Smooth (Prop X)> Unit -> [RefState, Smooth X] Unit
reverse x = x
reverse <ap0 n -> k> =
  let r = prop (op0 n) (new (c 0)) in
  reverse (k r)
reverse <ap1 u (prop x dx) -> k> =
  let r = prop (op1 u x) (new (c 0)) in
  reverse (k r);
  write dx (p (read dx) (t (der1 u x) (read (deriv r))))
reverse <ap2 b (prop x dx) (prop y dy) -> k> =
  let r = prop (op2 b x y) (new (c 0)) in
  reverse (k r);
  write dx (p (read dx) (t (der2L b x y) (read (deriv r))));
  write dy (p (read dy) (t (der2R b x y) (read (deriv r))))
```

- The reverse mode AD handler.
- `Prop` pairs a value with a cell holding its derivative.
- Handled operation calculates
  - the value and

The `reverse` handler

```
reverse : <Smooth (Prop X)> Unit -> [RefState, Smooth X] Unit
reverse x = x
reverse <ap0 n -> k> =
  let r = prop (op0 n) (new (c 0)) in
  reverse (k r)
reverse <ap1 u (prop x dx) -> k> =
  let r = prop (op1 u x) (new (c 0)) in
  reverse (k r);
  write dx (p (read dx) (t (der1 u x) (read (deriv r))))
reverse <ap2 b (prop x dx) (prop y dy) -> k> =
  let r = prop (op2 b x y) (new (c 0)) in
  reverse (k r);
  write dx (p (read dx) (t (der2L b x y) (read (deriv r))));
  write dy (p (read dy) (t (der2R b x y) (read (deriv r))))
```

- The reverse mode AD handler.
- `Prop` pairs a value with a cell holding its derivative.
- Handled operation calculates
  - the value and
  - makes ref. for the derivative

The `reverse` handler

```
reverse : <Smooth (Prop X)> Unit -> [RefState, Smooth X] Unit
reverse x = x
reverse <ap0 n -> k> =
  let r = prop (op0 n) (new (c 0)) in
  reverse (k r)
reverse <ap1 u (prop x dx) -> k> =
  let r = prop (op1 u x) (new (c 0)) in
  reverse (k r);
  write dx (p (read dx) (t (der1 u x) (read (deriv r))))
reverse <ap2 b (prop x dx) (prop y dy) -> k> =
  let r = prop (op2 b x y) (new (c 0)) in
  reverse (k r);
  write dx (p (read dx) (t (der2L b x y) (read (deriv r))));
  write dy (p (read dy) (t (der2R b x y) (read (deriv r))))
```

- Calculations are via another instance of `Smooth`.

- The reverse mode AD handler.
- `Prop` pairs a value with a cell holding its derivative.
- Handled operation calculates
  - the value and
  - makes ref. for the derivative

The `reverse` handler

```
reverse : <Smooth (Prop X)> Unit -> [RefState, Smooth X] Unit
reverse x = x
reverse <ap0 n -> k> =
  let r = prop (op0 n) (new (c 0)) in
  reverse (k r)
reverse <ap1 u (prop x dx) -> k> =
  let r = prop (op1 u x) (new (c 0)) in
  reverse (k r);
  write dx (p (read dx) (t (der1 u x) (read (deriv r))))
reverse <ap2 b (prop x dx) (prop y dy) -> k> =
  let r = prop (op2 b x y) (new (c 0)) in
  reverse (k r);
  write dx (p (read dx) (t (der2L b x y) (read (deriv r))));
  write dy (p (read dy) (t (der2R b x y) (read (deriv r))))
```

- Calculations are via another instance of `Smooth`.
- Execution then continues.

- The reverse mode AD handler.
- `Prop` pairs a value with a cell holding its derivative.
- Handled operation calculates
  - the value and
  - makes ref. for the derivative

The `reverse` handler

```
reverse : <Smooth (Prop X)> Unit -> [RefState, Smooth X] Unit
reverse x = x
reverse <ap0 n -> k> =
  let r = prop (op0 n) (new (c 0)) in
  reverse (k r)
reverse <ap1 u (prop x dx) -> k> =
  let r = prop (op1 u x) (new (c 0)) in
  reverse (k r);
  write dx (p (read dx) (t (der1 u x) (read (deriv r))))
reverse <ap2 b (prop x dx) (prop y dy) -> k> =
  let r = prop (op2 b x y) (new (c 0)) in
  reverse (k r);
  write dx (p (read dx) (t (der2L b x y) (read (deriv r))));
  write dy (p (read dy) (t (der2R b x y) (read (deriv r))))
```

- The reverse mode AD handler.
- `Prop` pairs a value with a cell holding its derivative.
- Handled operation calculates
  - the value and
  - makes ref. for the derivative

- Calculations are via another instance of `Smooth`.
- Execution then continues.
- Derivatives are then *propagated* backwards by accumulation.

The `reverse` handler (cont.)

The `reverse` handler (cont.)

To properly calculate derivatives with `reverse`, we require a small helper function which starts the process of back-propagation, which we call `grad` for gradient.

```
grad : {(Prop X) -> [RefState, Smooth X, Smooth (Prop X)] (Prop X)}
    -> X -> [RefState, Smooth X] X
grad f x =
  let z = prop x (new (c 0)) in
  reverse (write (deriv (f z)) (<Smooth> (c 1)));
  read (deriv z)
```

The `reverse` handler (cont.)

To properly calculate derivatives with `reverse`, we require a small helper function which starts the process of back-propagation, which we call `grad` for gradient.

```
grad : {(Prop X) -> [RefState, Smooth X, Smooth (Prop X)] (Prop X)}
    -> X -> [RefState, Smooth X] X
grad f x =
  let z = prop x (new (c 0)) in
  reverse (write (deriv (f z)) (<Smooth> (c 1)));
  read (deriv z)
```

- `grad` creates a cell for our final derivative

The `reverse` handler (cont.)

To properly calculate derivatives with `reverse`, we require a small helper function which starts the process of back-propagation, which we call `grad` for gradient.

```
grad : {(Prop X) -> [RefState, Smooth X, Smooth (Prop X)] (Prop X)}
    -> X -> [RefState, Smooth X] X
grad f x =
  let z = prop x (new (c 0)) in
  reverse (write (deriv (f z)) (<Smooth> (c 1)));
  read (deriv z)
```

- `grad` creates a cell for our final derivative
- evaluates the function `f` with effects handled by `reverse`

The `reverse` handler (cont.)

To properly calculate derivatives with `reverse`, we require a small helper function which starts the process of back-propagation, which we call `grad` for gradient.

```
grad : {(Prop X) -> [RefState, Smooth X, Smooth (Prop X)] (Prop X)}
    -> X -> [RefState, Smooth X] X
grad f x =
  let z = prop x (new (c 0)) in
  reverse (write (deriv (f z)) (<Smooth> (c 1)));
  read (deriv z)
```

- `grad` creates a cell for our final derivative
- evaluates the function `f` with effects handled by `reverse`
- initializes desired derivative to 1

The `reverse` handler (cont.)

To properly calculate derivatives with `reverse`, we require a small helper function which starts the process of back-propagation, which we call `grad` for gradient.

```
grad : {(Prop X) -> [RefState, Smooth X, Smooth (Prop X)] (Prop X)}
    -> X -> [RefState, Smooth X] X
grad f x =
  let z = prop x (new (c 0)) in
  reverse (write (deriv (f z)) (<Smooth> (c 1)));
  read (deriv z)
```

- `grad` creates a cell for our final derivative
- evaluates the function `f` with effects handled by `reverse`
- initializes desired derivative to 1
- returns the calculated derivative

The `reverse` handler (cont.)

```
evaluate (grad ({x -> let y = c 4 in p (c 1) (p (t (t x x) x) (n (t y y)))} 2))
```

The `reverse` handler (cont.)

```
evaluate (grad ({x -> let y = c 4 in p (c 1) (p (t (t x x) x) (n (t y y)))} 2))
```

The `reverse` handler (cont.)

```
evaluate (grad ({x -> let y = c 4 in p (c 1) (p (t (t x x) x) (n (t y y)))} 2))
```

→

```
evaluate (
  let z = prop 2 (new (c 0)) in
  reverse (write (deriv ({x ->
    let y = c 4 in p (c 1) (p (t (t x x) x) (n (t y y)))
  } z)) (<Smooth> (c 1)));
  read (deriv z))
```

The `reverse` handler (cont.)

```
evaluate (grad ({x -> let y = c 4 in p (c 1) (p (t (t x x) x) (n (t y y)))} 2))
```

→
```
evaluate (
  let z = prop 2 (new (c 0)) in
  reverse (write (deriv ({x ->
    let y = c 4 in p (c 1) (p (t (t x x) x) (n (t y y)))
  } z)) (<Smooth> (c 1)));
  read (deriv z))
```

→
```
evaluate (
  reverse (write (deriv ({x ->
    let y = c 4 in p (c 1) (p (t (t x x) x) (n (t y y)))
  } (prop 2 <z>))) (<Smooth> (c 1)));
  read (deriv (prop 2 <z>)))
```

The `reverse` handler (cont.)

```
  evaluate (grad ({x -> let y = c 4 in p (c 1) (p (t (t x x) x) (n (t y y)))} 2))
→ evaluate (
    let z = prop 2 (new (c 0)) in
    reverse (write (deriv ({x ->
      let y = c 4 in p (c 1) (p (t (t x x) x) (n (t y y)))
    } z)) (<Smooth> (c 1)));
    read (deriv z))
→ evaluate (
    reverse (write (deriv ({x ->
      let y = c 4 in p (c 1) (p (t (t x x) x) (n (t y y)))
    } (prop 2 <z>))) (<Smooth> (c 1)));
    read (deriv (prop 2 <z>)))
→ evaluate (reverse (write (deriv (
      let y = c 4 in
      p (c 1) (p (t (t (prop 2 <z>) (prop 2 <z>)) (prop 2 <z>)) (n (t y y)))
    )) (<Smooth> (c 1)));
    read (deriv (prop 2 <z>)))
```

The `reverse` handler (cont.)

```
→ evaluate (reverse (write (deriv (
      p (prop 1 <r2>) (p (t (t (prop 2 <z>) (prop 2 <z>)) (prop 2 <z>))
                         (n (t (prop 4 <r1>) (prop 4 <r1>))))
    )) (<Smooth> (c 1)));
    read (deriv (prop 2 <z>)))
```

The `reverse` handler (cont.)

```
→ evaluate (reverse (write (deriv (
      p (prop 1 <r2>) (p (t (t (prop 2 <z>) (prop 2 <z>)) (prop 2 <z>))
                         (n (t (prop 4 <r1>) (prop 4 <r1>))))
    )) (<Smooth> (c 1)));
    read (deriv (prop 2 <z>)))

→ evaluate (reverse (write (deriv (
    p (prop 1 <r2>) (p (t (t (prop 2 <z>) (prop 2 <z>)) (prop 2 <z>))
                       (n (t (prop 4 <r1>) (prop 4 <r1>))))
    )) (<Smooth> (c 1)));
    read (deriv (prop 2 <z>)))
```

The `reverse` handler (cont.)

```
→ evaluate (reverse (write (deriv (
      p (prop 1 <r2>) (p (t (t (prop 2 <z>) (prop 2 <z>)) (prop 2 <z>))
                         (n (t (prop 4 <r1>) (prop 4 <r1>))))
    )) (<Smooth> (c 1)));
    read (deriv (prop 2 <z>)))

→ evaluate (reverse (write (deriv (
    p (prop 1 <r2>) (p (t (t (prop 2 <z>) (prop 2 <z>)) (prop 2 <z>))
                       (n (t (prop 4 <r1>) (prop 4 <r1>))))
    )) (<Smooth> (c 1)));
    read (deriv (prop 2 <z>)))

→ evaluate (reverse (write (deriv (
      p (prop 1 <r2>) (p (t (prop 4 <r3>) (prop 2 <z>))
                         (n (t (prop 4 <r1>) (prop 4 <r1>))))
    )) (<Smooth> (c 1)));
    write <z> (p (read <z>) (t (der2L timesE 2 2) (read (deriv (prop 4 <r3>)))));
    write <z> (p (read <z>) (t (der2R timesE 2 2) (read (deriv (prop 4 <r3>)))));
    read (deriv (prop 2 <z>)))
```

The `reverse` handler (cont.)

...

The `reverse` handler (cont.)

```
...
→ evaluate (
    reverse (write <r8> (<Smooth> (c 1)));
    write <r2> (p (read <r2>) (t (der2L plusE 1 -8)  (read (deriv (prop -7  <r8>)))));
    write <r7> (p (read <r7>) (t (der2R plusE 1 -8)  (read (deriv (prop -7  <r8>)))));
    write <r4> (p (read <r4>) (t (der2L plusE 8 -16) (read (deriv (prop -8  <r7>)))));
    write <r6> (p (read <r6>) (t (der2R plusE 8 -16) (read (deriv (prop -8  <r7>)))));
    write <r5> (p (read <r5>) (t (der1  negateE 16)  (read (deriv (prop -16 <r6>)))));
    write <r1> (p (read <r1>) (t (der2L timesE 4 4)  (read (deriv (prop 16  <r5>)))));
    write <r1> (p (read <r1>) (t (der2R timesE 4 4)  (read (deriv (prop 16  <r5>)))));
    write <r3> (p (read <r3>) (t (der2L timesE 4 2)  (read (deriv (prop 8   <r4>)))));
    write <z>  (p (read <z>)  (t (der2R timesE 4 2)  (read (deriv (prop 8   <r4>)))));
    write <z>  (p (read <z>)  (t (der2L timesE 2 2)  (read (deriv (prop 4   <r3>)))));
    write <z>  (p (read <z>)  (t (der2R timesE 2 2)  (read (deriv (prop 4   <r3>)))));
    read (deriv (prop 2 <z>)))
```

Aside on checkpointed reverse mode AD

Aside on checkpointed reverse mode AD

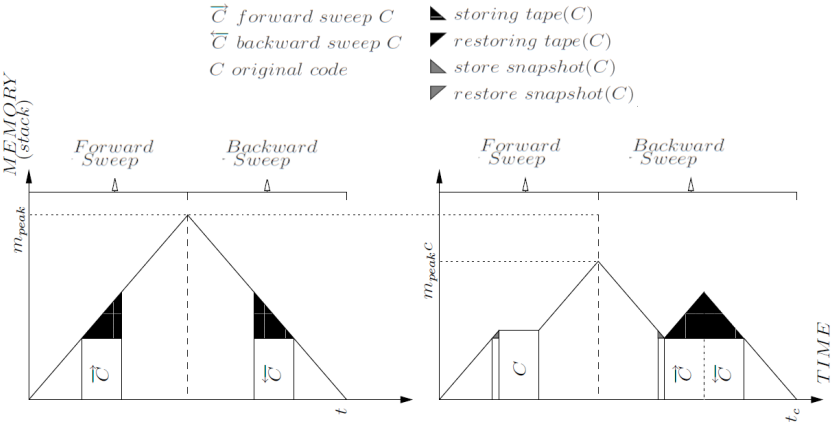

Diagram adapted from 'Enabling user-driven Checkpointing strategies in Reverse-mode Automatic Differentiation' by L. Hascoët and M. Araya-Polo

The reversec handler

```
interface Checkpoint X =
  checkpoint : {[Checkpoint X , Smooth (Prop X)] Prop X} -> Prop X

reversec : <Checkpoint X, Smooth (Prop X)> Unit
        -> [RefState, Smooth X] Unit
reversec x = x
reversec <checkpoint p -> k> =
  let s = new (c 0) in
  let res = <RefState> (evaluatet s (
    <Smooth(s a b -> s b)> p!
  )) in
  let r = prop (fwd res) (new (c 0)) in
  reversec (k r);
  reversec (write
    (deriv (<Smooth(s a b -> s b), RefState> p!))
    (read (deriv r)))
reversec <m> = reversec (<Smooth(s a -> s)> (reverse m!))
```

The `reversec` handler

```
interface Checkpoint X =
  checkpoint : {[Checkpoint X , Smooth (Prop X)] Prop X} -> Prop X

reversec : <Checkpoint X, Smooth (Prop X)> Unit
        -> [RefState, Smooth X] Unit
reversec x = x
reversec <checkpoint p -> k> =
  let s = new (c 0) in
  let res = <RefState> (evaluatet s (
    <Smooth(s a b -> s b)> p!
  )) in
  let r = prop (fwd res) (new (c 0)) in
  reversec (k r);
  reversec (write
    (deriv (<Smooth(s a b -> s b), RefState> p!))
    (read (deriv r)))
reversec <m> = reversec (<Smooth(s a -> s)> (reverse m!))
```

- `checkpoint` command takes a thunk.

The `reversec` handler

```
interface Checkpoint X =
  checkpoint : {[Checkpoint X , Smooth (Prop X)] Prop X} -> Prop X

reversec : <Checkpoint X, Smooth (Prop X)> Unit
        -> [RefState, Smooth X] Unit
reversec x = x
reversec <checkpoint p -> k> =
  let s = new (c 0) in
  let res = <RefState> (evaluatet s (
    <Smooth(s a b -> s b)> p!
  )) in
  let r = prop (fwd res) (new (c 0)) in
  reversec (k r);
  reversec (write
    (deriv (<Smooth(s a b -> s b), RefState> p!))
    (read (deriv r)))
reversec <m> = reversec (<Smooth(s a -> s)> (reverse m!))
```

- `checkpoint` command takes a thunk.
- For a `checkpoint`

The `reversec` handler

```
interface Checkpoint X =
  checkpoint : {[Checkpoint X , Smooth (Prop X)] Prop X} -> Prop X

reversec : <Checkpoint X, Smooth (Prop X)> Unit
        -> [RefState, Smooth X] Unit
reversec x = x
reversec <checkpoint p -> k> =
  let s = new (c 0) in
  let res = <RefState> (evaluatet s (
    <Smooth(s a b -> s b)> p!
  )) in
  let r = prop (fwd res) (new (c 0)) in
  reversec (k r);
  reversec (write
    (deriv (<Smooth(s a b -> s b), RefState> p!))
    (read (deriv r)))
reversec <m> = reversec (<Smooth(s a -> s)> (reverse m!))
```

- `checkpoint` command takes a thunk.
- For a `checkpoint`
  - the thunk runs normally then

The `reversec` handler

```
interface Checkpoint X =
  checkpoint : {[Checkpoint X , Smooth (Prop X)] Prop X} -> Prop X

reversec : <Checkpoint X, Smooth (Prop X)> Unit
        -> [RefState, Smooth X] Unit
reversec x = x
reversec <checkpoint p -> k> =
  let s = new (c 0) in
  let res = <RefState> (evaluatet s (
    <Smooth(s a b -> s b)> p!
  )) in
  let r = prop (fwd res) (new (c 0)) in
  reversec (k r);
  reversec (write
    (deriv (<Smooth(s a b -> s b), RefState> p!))
    (read (deriv r)))
reversec <m> = reversec (<Smooth(s a -> s)> (reverse m!))
```

- `checkpoint` command takes a thunk.
- For a `checkpoint`
  - the thunk runs normally then
  - execution continues before

The `reversec` handler

```
interface Checkpoint X =
  checkpoint : {[Checkpoint X , Smooth (Prop X)] Prop X} -> Prop X

reversec : <Checkpoint X, Smooth (Prop X)> Unit
        -> [RefState, Smooth X] Unit
reversec x = x
reversec <checkpoint p -> k> =
  let s = new (c 0) in
  let res = <RefState> (evaluatet s (
    <Smooth(s a b -> s b)> p!
  )) in
  let r = prop (fwd res) (new (c 0)) in
  reversec (k r);
  reversec (write
    (deriv (<Smooth(s a b -> s b), RefState> p!))
    (read (deriv r)))
reversec <m> = reversec (<Smooth(s a -> s)> (reverse m!))
```

- `checkpoint` command takes a thunk.
- For a `checkpoint`
  - the thunk runs normally then
  - execution continues before
  - the thunk runs with propagation.

The `reversec` handler

```
interface Checkpoint X =
  checkpoint : {[Checkpoint X , Smooth (Prop X)] Prop X} -> Prop X

reversec : <Checkpoint X, Smooth (Prop X)> Unit
        -> [RefState, Smooth X] Unit
reversec x = x
reversec <checkpoint p -> k> =
  let s = new (c 0) in
  let res = <RefState> (evaluatet s (
    <Smooth(s a b -> s b)> p!
  )) in
  let r = prop (fwd res) (new (c 0)) in
  reversec (k r);
  reversec (write
    (deriv (<Smooth(s a b -> s b), RefState> p!))
    (read (deriv r)))
reversec <m> = reversec (<Smooth(s a -> s)> (reverse m!))
```

- `checkpoint` command takes a thunk.
- For a `checkpoint`
  - the thunk runs normally then
  - execution continues before
  - the thunk runs with propagation.
- `Smooth` commands are passed to `reverse`.

The `reversec` handler (cont.)

```
evaluate (gradc ({x ->
  let y = c 2 in
  let z = checkpoint {p x y} in
  let a = checkpoint {let w = checkpoint {t x z} in p w y} in
  p a x
} (c 2)))
```

The `reversec` handler (cont.)

```
  evaluate (gradc ({x ->
    let y = c 2 in
    let z = checkpoint {p x y} in
    let a = checkpoint {let w = checkpoint {t x z} in p w y} in
    p a x
  } (c 2)))
```

→

```
  evaluate (
    reversec (write (deriv (
      let y = c 2 in
      let z = checkpoint {p (prop 2 <z>) y} in
      let a = checkpoint {let w = checkpoint {t (prop 2 <z>) z} in p w y} in
      p a (prop 2 <z>)
    )) (<Smooth> (c 1)));
    read (deriv (prop 2 <z>)))
```

The `reversec` handler (cont.)

```
→ evaluate (
    reversec (<Smooth(s a -> s)> (reverse (write (deriv (
      let y = c 2 in
      let z = checkpoint {p (prop 2 <z>) y} in
      let a = checkpoint {let w = checkpoint {t (prop 2 <z>) z} in p w y} in
      p a (prop 2 <z>)
    )) (<Smooth> (c 1))))));
    read (deriv (prop 2 <z>)))
```

The reversec handler (cont.)

```
→ evaluate (
    reversec (<Smooth(s a -> s)> (reverse (write (deriv (
      let y = c 2 in
      let z = checkpoint {p (prop 2 <z>) y} in
      let a = checkpoint {let w = checkpoint {t (prop 2 <z>) z} in p w y} in
      p a (prop 2 <z>)
    )) (<Smooth> (c 1))))));
    read (deriv (prop 2 <z>)))

→ evaluate (
    reversec (<Smooth(s a -> s)> (reverse (write (deriv (
      let z = checkpoint {p (prop 2 <z>) (prop 2 <r1>)} in
      let a = checkpoint {let w = checkpoint {t (prop 2 <z>) z} in p w (prop 2 <r1>)} in
      p a (prop 2 <z>)
    )) (<Smooth> (c 1))))));
    read (deriv (prop 2 <z>)))
```

The reversec handler (cont.)

```
→ evaluate (
    reversec (<Smooth(s a -> s)> (reverse (write (deriv (
      let z = prop 4 <r2> in
      let a = checkpoint {let w = checkpoint t (prop 2 <z>) z in p w (prop 2 <r1>)} in
      p a (prop 2 <z>)
    )) (<Smooth> (c 1))))));
    reversec (write (deriv (<Smooth(s a b -> s b), RefState>
                              {p (prop 2 <z>) (prop 2 <r1>)}!))
                    (read (deriv (prop 4 <r2>))));
    read (deriv (prop 2 <z>)))
```

The reversec handler (cont.)

$\rightarrow$

```
evaluate (
  reversec (<Smooth(s a -> s)> (
    reverse (write <r4> 1);
    write <r3> (p (read <r3>) (t (der2L plusE 10 2) (read (deriv (prop 12 <r4>)))));
    write <z>  (p (read <z>)  (t (der2R plusE 10 2) (read (deriv (prop 12 <r4>))))));
  reversec (
    write (deriv (<Smooth(s a b -> s b), RefState>
            {let w = checkpoint {t (prop 2 <z>) (prop 4 <r2>)} in p w (prop 2 <r1>)}!))
          (read (deriv (prop 10 <r3>))));
  reversec (
    write (deriv (<Smooth(s a b -> s b), RefState>
            {p (prop 2 <z>) (prop 2 <r1>)}!))
          (read (deriv (prop 4 <r2>))));
  read (deriv (prop 2 <z>)))
```

Utility of effect handler approach

Utility of effect handler approach

- In real world applications, it can be unclear a priori if forward or reverse is better. Shallow handlers could change the mode dynamically based on relevant metrics while executing.

Utility of effect handler approach

- In real world applications, it can be unclear a priori if forward or reverse is better. Shallow handlers could change the mode dynamically based on relevant metrics while executing.
- Nesting automatically works. Frank's effect type system will actually disallow more incorrect (w.r.t. AD) programs than some other type systems.

Utility of effect handler approach

- In real world applications, it can be unclear a priori if forward or reverse is better. Shallow handlers could change the mode dynamically based on relevant metrics while executing.
- Nesting automatically works. Frank's effect type system will actually disallow more incorrect (w.r.t. AD) programs than some other type systems.
- The implementation is very succinct.

Extensions to current work

Extensions to current work

- In Wang et al.'s 'Demystifying differentiable programming: shift/reset the penultimate backpropagator', control flow constructs are taken into account for more efficient AD calculations. The same style of extension can easily be applied here.

Extensions to current work

- In Wang et al.'s 'Demystifying differentiable programming: shift/reset the penultimate backpropagator', control flow constructs are taken into account for more efficient AD calculations. The same style of extension can easily be applied here.
- Effects and handlers have a natural denotational semantics in terms of Lawvere theories and models (equivalently finitary monads and algebras). Thus, the Frank implementation gives insight into possible semantics which relate to previously studied mathematical objects.

Extensions to current work

- In Wang et al.'s 'Demystifying differentiable programming: shift/reset the penultimate backpropagator', control flow constructs are taken into account for more efficient AD calculations. The same style of extension can easily be applied here.
- Effects and handlers have a natural denotational semantics in terms of Lawvere theories and models (equivalently finitary monads and algebras). Thus, the Frank implementation gives insight into possible semantics which relate to previously studied mathematical objects.
- We have shown that modularity for AD can be achieved in Frank. AD literature contains many variations on the basic modes, and quick experimentation is possible with Frank.

Extensions to current work

- In Wang et al.'s 'Demystifying differentiable programming: shift/reset the penultimate backpropagator', control flow constructs are taken into account for more efficient AD calculations. The same style of extension can easily be applied here.
- Effects and handlers have a natural denotational semantics in terms of Lawvere theories and models (equivalently finitary monads and algebras). Thus, the Frank implementation gives insight into possible semantics which relate to previously studied mathematical objects.
- We have shown that modularity for AD can be achieved in Frank. AD literature contains many variations on the basic modes, and quick experimentation is possible with Frank.
- It is simple to extend the exhibited system to work with vectors and matrices (in the style of DiffSharp).

Conclusion

Conclusion

- Effect handlers make AD simple to implement.

Conclusion

- Effect handlers make AD simple to implement.
- The two reverse mode handlers effectively build up ancillary programs as they execute.

Conclusion

- Effect handlers make AD simple to implement.
- The two reverse mode handlers effectively build up ancillary programs as they execute.
- Effect handlers allow a compositional approach to AD algorithms.