The Differentiable Curry

Martin Abadi, Dan Belov, Gordon Plotkin, Richard Wei, **Dimitrios Vytiniotis**
DeepMind and Google Brain

thanks to the many from the [Swift For Tensorflow] and [JAX] teams

The Diff… …rry

Artificial Exponentials* for Cartesian Closure

Martin Aba… …n, Richard Wei, **Dimitrios Vytiniotis**

DeepMind and Google Brain

* Term due to Conal Elliott

Two starting ideas for this work

Reverse-Mode AD in a Functional Framework: Lambda the Ultimate Backpropagator

BARAK A. PEARLMUTTER
Hamilton Institute
and
JEFFREY MARK SISKIND
Purdue University

We show that reverse-mode AD (Automatic Differentiation)—a generalized gradient-calculation operator—can be incorporated as a first-class function in an augmented lambda calculus, and therefore into a functional-programming language. Closure is achieved, in that the new operator can be applied to any expression in the augmented language, yielding an expression in that language. This requires the resolution of two major technical issues: (a) how to transform nested lambda expressions, including those with free-variable references, and (b) how to support self application of the AD machinery. AD transformations preserve certain complexity properties, among them that the reverse phase of the reverse-mode AD transformation of a function have the same temporal complexity as the original untransformed function. First-class unrestricted AD operators increase the expressive power available to the numeric programmer, and may have significant practical implications for the construction of numeric software that is robust, modular, concise, correct, and efficient.

Categories and Subject Descriptors: D.3.2.a [**Programming Languages**]: Language Classifications—*Applicative (functional) languages*; G.1.4.b [**Numerical Analysis**]: Quadrature and Numerical Differentiation—*Automatic differentiation*

General Terms: Experimentation, Languages, Performance

Additional Key Words and Phrases: closures, derivatives, forward-mode AD, higher-order AD, higher-order functional languages, Jacobian, program transformation, reflection

The Simple Essence of Automatic Differentiation

*Extended version**

Conal Elliott
Target
conal@conal.net

March, 2018

Abstract

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.

This paper: AD and Higher-Order Functions

```
func lstmCell(w : Params, state : Tensor, input : Tensor) -> Tensor { ... }

func rnn(xs : Array<Tensor>, cell_fn) {
  func go(idx, state) {
      if (idx < xs.length) {
            return go(idx+1, cell_fn(state, xs[idx]))
      else return state
  }
  return loss_fn(go(0, 0.0))
}

model = ... // init parameters
for xs in minibatch {
  grads = grad (λps. rnn(xs, λ h x. lstmCell(ps, h, x)) (model)
  update(model, along: grads)
}
```

Function arguments (higher-order functions)

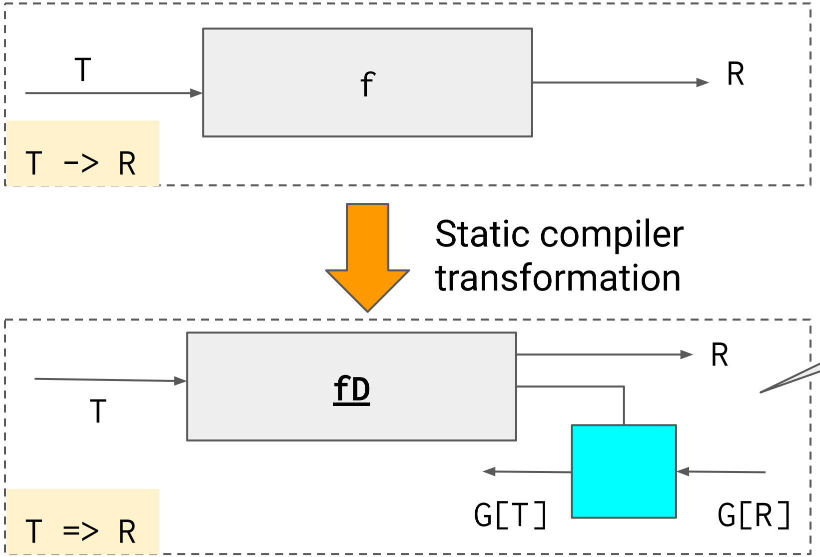
Partial application, capturing differentiable variables

AD possible today even in **production languages:**

https://www.tensorflow.org/swift

We will show how to do combinator-style AD, and prove something about what we did.

AD by lifting primitives equipped with pullbacks



`mult(x,y) = x*y`

Pullback of f, G[T] sometimes called “co-tangent” of T

```
multD(x,y) =
  (x*y, \g->(g*y, g*x))
```

(**fD** : `T ~> R`) can be applied, or passed to other functions, as if it was an ordinary function `T -> R`

NB: lots of other ways of describing this transformation with different tradeoffs.

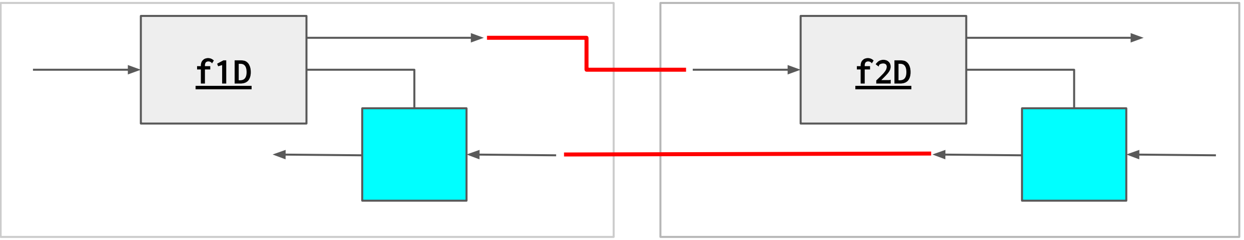
Reverse-mode AD in one slide

AD = composition of primitive pullbacks (chain rule)

```
f1, f2 : Float => Float

func g(x:Float) : Float {
  let v        = f1(x);
  let r        = f2(v);
  return r;
}
```

```
func gD(x:Float) {
  let (v, pb_f1)        = f1D(x);
  let (r, pb_f2)        = f2D(v);
  return (r, \gt ->
     let gv = pb_f2(gt)
     let gx = pb_f1(gv)
     return gx
     })
}
```



Looks like a very “systematic” translation, let’s translate all programs to diagrams!

Recipe for AD: compile first to CCC algebra

```
id : T => T

(f : S => T) o (g : T => R) : S => R

prod(f1 : G => A, f2 : G => B) : G => (A,B)

proj_left : (A, B) => A

proj_right : (A, B) => B

curry(f : (T, S) => R) : T => (S => R)

eval : (T, T => R) => R
```

```
func f(x, w, b) =
  let r1 = mult(x,w)
      r2 = add(r1,b)
  in r2
```

An “ordinary” program

A categorical program

```
prod(proj_left o mult,
          proj_right) o add
```

NB: Nothing specific to AD: it’s all vanilla lambda calculus and category theory.

Then implement T => S and combinators

```
id : T => T
(f : S => T) o (g : T => R) : S => R
prod(f1 : G => A, f2 : G => B) : G => (A,B)
proj_left : (A, B) => A
proj_right : (A, B) => B
curry(f : (T, S) => R) : T => (S => R)
eval : (T, T => R) => R
```

How to define type (`T => S`)

We need (`T => S`) to satisfy at least:

1. Given $(h : T^{FO} \Rightarrow S^{FO})$ we can extract the *mathematical* $\mathrm{vjp}(h) : T^{FO} \to (S^{FO}, (S^{FO} \to T^{FO}))$
2. Ensure the implementation of the combinators respects CCC laws (more on this in a bit)

Why only for first-order (FO) types?

T^{FO} ::= Float | Vector | (S^{FO}, T^{FO})

A **compromise,** but useful for differentiating end-to-end programs.

Substantial work on "true" derivatives for h-o types:

- Categorical Models for Simply Typed Resource Calculi
- In-progress work by Conal Elliott
- The differential lambda calculus, ccc semantics in: The convenient space of global analysis

Start with the intuitive definition

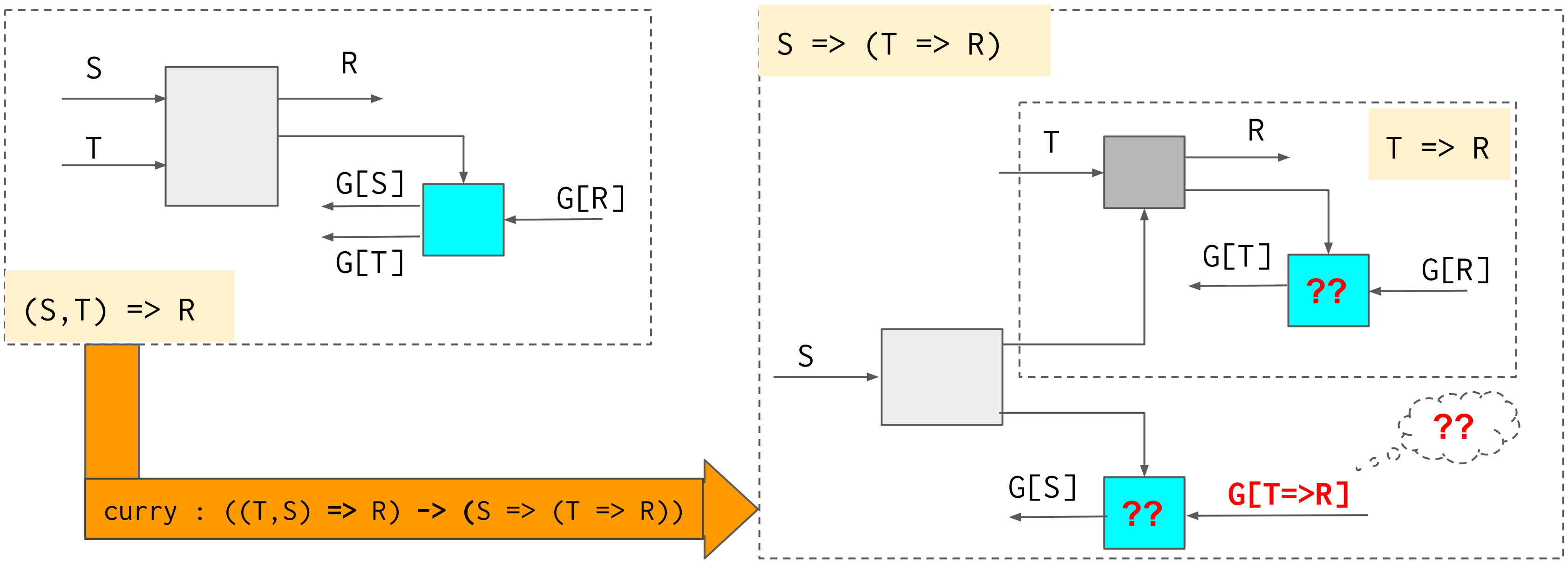
```
T => S          ≜          T -> (S, G[S] -> G[T])
```

where

```
G[Float] = Float,
G[(T1,T2)] = (G[T1], G[T2])
```

Frequently used notion of “pullback” linear map, operator G[T] is often called the “cotangent” space of T.

Main bulk of paper: how to implement curry



So that the implementation validates Req. 2 set previously!

Results (I): a simply-typed curry

```
curry :: ((T,S) => R) -> (T => (S => R))
curry f = new_f
  where
    new_f :: T -> (S => R, G[S=>R] -> G[T])
    new_f t =
       let new_g :: S -> (R, G[R] -> G[S])
           new_g s =
              let (r,pullback) = f(t,s)
              in (r, \gr -> snd (pullback gr))
           new_pb :: G[S=>R] -> G[T]
           new_pb ss_grs = List.sum $
             List.map (\(s,gr) -> fst (snd (f(t,s)) ss_grs
       in (new_g, new_pb)
```

```
G[S => R] = AdditiveMap (S, G[R])
```

Thm: we get a CCC

```
eval :: (T => S, T) => S
eval = ...
```

```
(.) :: (T => S) -> (S => R) -> (T => R)
(.) = ...
```

```
id :: (T => T)
id =
```

```
proj_left :: ((T,S) => T)
proj_left = ...
```

```
proj_right :: ((T,S) => S)
proj_right = ...
```

```
prod :: (X => A) -> (Y => B) -> ((X,Y) => (A,B))
prod = …
```

Thm: for `f:(T,S) => R, h : T => S => R`

- `(prod (curry f) id) . eval ≅ f`
- `curry ((tuple h id) . eval) ≅ h`

Corollary: AD respects **equational reasoning about programs**
Corollary: compiler transformations preserve AD results

CCC theorems (back in lambda-calculus speak)

<table>
<tr>
<td><pre>f :: (Float, Float) => Float

foo1 (a, b) =
  let g = λxb → f (a, xb)
  in g b

foo2 (a, b) = f (a, b)</pre></td>
<td><pre>foo1 (f, g) x =
  let y1 = f x
      y2 = g x
  in y1

foo2 (f, g) x = f x</pre></td>
<td><pre>foo1 f x =
  let y1 = f x
      y2 = f x
  in y1 + y2

foo2 f x = let y = f x
           in (y + y)</pre></td>
</tr>
<tr>
<td>Partial applications</td>
<td>Forgetting results</td>
<td>Summing results</td>
</tr>
</table>

`vjp(foo1) ≅ vjp(foo2)`

- Both forward-, and backward equivalent
- Need a notion of ≅ that respects 0 and +

Results (II): an efficient curry via dependent types

A closure `f : T -> S` is really an object `Closure<T,S>` containing:

- An Environment `Env` of captured variables
- A static code pointer: `Env -> T -> S`

Key idea: every function has a ***different*** sensitivity, depending on the environment it captured when allocated.

```
T1 => T2 =
  exists Δ. (x : T1) ->
    Σ (y : T2). G[y : T2] -> (Δ, G[x : T1])

G [ v : T1 => T2 ] =
   case v of
   | exists Δ _ => Δ
```

Coq

`G[T=>S]` **becomes dependent** `G[f:T=>S]`

```
G[\x -> y + x] = Float
G[\x -> y + z + x] = (Float, Float)
```

Thm: we get a weak CCC
Open: do we get a strong CCC?

* Idea first appears in Pearlmutter & Siskind classic “Lambda the ultimate back-propagator” [TOPLAS’08] (no proofs)

Not just theory, curry ***is*** a Swift IL (SIL) instruction

```
struct LinLayer {
   Tensor w;
   func call(x:Tensor):Tensor { return (x*w); }
}
… use site …
linlayer.call(inputs);

================================================
⇒ in the Swift IL (SIL) (simplifiing)
================================================
func func_1(x: Tensor, self : LinLayer) : Tensor {
    return (x * self.w);
}
… use site …
h = papply(func_1, linlayer) // Tensor => Tensor
r = h(inputs)
```

If we have differentiated `func_1` then we want **papply**`(func_1,linlayer)` to **return** a (=>) value

Moreover, for training: we need to backpropagate back through to `linlayer`, i.e need a **differentiable** partial application

```
papply : (((T,S) => R), S) => (T => R)
```

Dependent types? Swift is not dependently-typed …

```
curry :: ((T,S) => R) -> (T => (S => R))
curry (exists D. f) = pack () new_f
  where
    new_f :: (t:T) -> ((g : S => R), G[g:S=>R] -> (D, G[t:T])
    new_f t =
       let g :: (s:S) -> (r:R, G[r:R] -> ((D,G[t:T]), G[s:S])
           g s =
              let (r, pullback) = f(t,s)
              in (r, \gr -> let (cte,(ctt,cts)) = pullback gr
                            in ((cte,ctt), cts))
           new_pb :: G[g:S=>R] -> (D, G[t:T])
           new_pb env = env   // Magic (but type-correct)!
       in (pack [..] g, new_pb)
```

Proof guides the implementation of higher-order functions in Swift for **efficiency**, **memory safety**, and **correctness**.

```
G[S => T] = AnyDerivative // An “opaque” type with 0 and +
S => T = (S -> (T, G[T] -> (AnyDerivative,G[S]))

curry :: ((T,S) => R) -> (T => (S => R))
curry (exists D. f) = pack () new_f
  where                                              AnyDerivative
    new_f :: (t:T) -> ((g : S => R), G[g:S=>R] -> (D, G[t:T])
    new_f t =
       let g :: (s:S) -> (r:R, G[r:R] -> ((D,G[t:T]), G[s:S])
           g s =
             let (r, pullback) = f(t,s)             AnyDerivative
             in (r, \gr -> let (cte,(ctt,cts)) = pullback gr
                               in ((cte,ctt), cts))
           new_pb :: G[g:S=>R] -> (D, G[t:T])
           new_pb env = env
       in (pack [..] g, new_pb)          AnyDerivative
```

Swift

Artificial exponentials

Not truly higher-order

- Cannot do anything **useful** with `vjp(h : (A => (B => C))` or `vjp(h : (A => B) => C)`

- But the loss is small, end-to-end programs are first-order, only intermediates are higher-order!

- Cartesian closure enough to guarantee same behaviour as fully inlined program

Hence we call the result of curry an **“artificial exponential”**. It has no direct meaning as a derivative, but enables closure computationally!

The bigger picture and future work

Nothing really about AD! Bigger picture is this:

- Start with a CCC category $\mathcal{C}$
- Define a (possibly dependent) pairing of each object with an **affine space** in a category of **affine spaces and linear maps**, call that $\mathcal{LMC}$
- We give a construction that runs $\mathcal{C}$ forward and returns backward (or forward, similar techniques are applicable) arrows in the $\mathcal{LMC}$, given the primivites.

AD just one application: dynamic symbolic analysis (with **sets** and **union** of various sorts) might be another, forward or backward provenance analyses etc.

Future work!

Thanks!

- A combinator-based differentiation strategy
- A curry cooked two ways, correct for FO programs
- “Artificial exponentials” and cartesian closure for ensuring conservative extension to higher-order types
- Ideas being implemented in experimental Swift

Paper Draft Here

A call for careful formal treatment of AD: stability under program transformations, perturbation confusion, HO-AD etc.