Working notes on Automatic differentiation

September 2, 2019 — not for circulation

Tom Ellis
Simon Peyton Jones
Andrew Fitzgibbon

1 The language

This paper is about automatic differentiation of functions, so we must be precise about the language in which those functions are written.

The syntax of our language is given in Figure 1. Note that

- Variables are divided into *functions*, f, g, h; and *local variables*, x, y, z, which are either function arguments or let-bound.
- The language has a first order sub-language. Functions are defined at top level; functions always appear in a call, never (say) as an argument to a function; in a call $f(e)$, the function f is always a top-level-defined function, never a local variable.
- Functions have exactly one argument. If you want more than one, pass a pair.
- Pairs are built-in, with selectors $\pi_{1,2}, \pi_{2,2}$. In the real implementation, pairs are generalised to n-tuples, and we often do so informally here.
- Conditionals are are a language construct.
- Let-bindings are non-recursive. For now, at least, top-level functions are also non-recursive.
- Lambda expressions and applications are are present, so the language is higher order. AD will only accept a subset of the language, in which lambdas appear only as an argument to *build*. But the *output* of AD may include lambdas and application, as we shall see.

Atoms

f, g, h	::=	Function	
x, y, z	::=	Local variable (lambda-bound or let-bound)	
k	::=	Literal constants	

Terms

pgm	::=	$def_1 \ldots def_n$	
def	::=	$f(x) = e$	
e	::=	k	Constant
	|	x	Local variable
	|	$f(e)$	Function call
	|	(e_1, e_2)	Pair
	|	$\lambda x.\, e$	Lambda
	|	$e_1\ e_2$	Application
	|	$\text{let } x{=}e_1 \text{ in } e_2$	
	|	$\text{if } b \text{ then } e_1 \text{ else } e_2$	

Types

τ	::=	$\mathbb{N}$	Natural numbers
	|	$\mathbb{R}$	Real numbers
	|	(τ_1, τ_2)	Pairs
	|	$Vec\ n\ \tau$	Vectors
	|	$\tau_1 \to \tau_2$	Functions
	|	$\tau_1 \multimap \tau_2$	Linear maps

Figure 1. Syntax of the language

1.1 Built in functions

The language has built-in functions shown in Figure 2.

We allow ourselves to write functions infix where it is convenient. Thus $e_1 + e_2$ means the call $+(e1, e2)$, which applies the function $+$ to the pair (e_1, e_2). (So, like all other functions, $(+)$ has one argument.) Similarly the linear map $m_1 \times m_2$ is short for $\times(e_1, e_2)$.

We allow ourselves to write vector indexing $ixR(i, a)$ using square brackets, thus $a[i]$.

Multiplication and addition are overloaded to work on any suitable type. On vectors they work element-wise; if you want dot-product you have to program it.

1.2 Vectors

The language supports one-dimensional vectors, of type $Vec\ n\ T$, whose elements have type T (Figure 1). A matrix can be represented as a vector of vectors.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Built-in functions

$$
\begin{array}{rcll}
(+) & :: & (\mathbb{R}, \mathbb{R}) \to \mathbb{R} & \\
(*) & :: & (\mathbb{R}, \mathbb{R}) \to \mathbb{R} & \\
\pi_{1,2} & :: & (t_1, t_2) \to t_1 & \text{Selection} \\
\pi_{2,2} & :: & (t_1, t_2) \to t_2 & \text{..ditto..} \\
build & :: & (n :: \mathbb{N},\ \mathbb{N} \to t) \to Vec\ n\ t & \text{Vector build} \\
ixR & :: & (\mathbb{N},\ Vec\ n\ t) \to t & \text{Indexing (NB arg order)} \\
sum & :: & Vec\ n\ t \to t & \text{Sum a vector} \\
sz & :: & Vec\ n\ t \to \mathbb{N} & \text{Size of a vector}
\end{array}
$$

Derivatives of built-in functions

$$
\begin{array}{rcl}
\partial + & :: & (\mathbb{R}, \mathbb{R}) \to ((\mathbb{R}, \mathbb{R}) \multimap \mathbb{R}) \\
\partial + (x, y) & = & \mathbf{1} \bowtie \mathbf{1} \\
\\
\partial * & :: & (\mathbb{R}, \mathbb{R}) \to ((\mathbb{R}, \mathbb{R}) \multimap \mathbb{R}) \\
\partial * (x, y) & = & \mathcal{S}(y) \bowtie \mathcal{S}(x) \\
\\
\partial \pi_{1,2} & :: & (t, t) \to ((t, t) \multimap t) \\
\partial \pi_{1,2}(x) & = & \mathbf{1} \bowtie \mathbf{0} \\
\\
\partial ixR & :: & (\mathbb{N},\ Vec\ n\ t) \to ((\mathbb{N},\ Vec\ n\ t) \multimap t) \\
\partial ixR(i, v) & = & \mathbf{0} \bowtie \mathcal{H}(build(sz(v), \lambda j.\ \text{if } i = j \text{ then } \mathbf{1} \text{ else } \mathbf{0})) \\
\\
\partial sum & :: & Vec\ n\ \mathbb{R} \to (Vec\ n\ \mathbb{R} \multimap \mathbb{R}) \\
\partial sum(v) & = & lmhcatvbuild(sz(v),\ \lambda i.1) \\
\\
& \ldots &
\end{array}
$$

Figure 2. Built-in functions

Vectors are supported by the following built-in functions (Figure 2):

- $build :: (\mathbb{N},\ \mathbb{N} \to t) \to Vec\ n\ t$ for vector construction.
- $ixR :: (\mathbb{N},\ Vec\ n\ t) \to t$ for indexing. Informally we allow ourselves to write $v[i]$ instead of $ixR(i, v)$.
- $sum :: Vec\ n\ \mathbb{R} \to \mathbb{R}$ to add up the elements of a vector. We specifically do not have a general, higher order, fold operator; we say why in Section 4.1.
- $sz :: Vec\ n\ t \to \mathbb{N}$ takes the size of a vector.
- Arithmetic functions $(*), (+)$ etc are overloaded to work over vectors, always elementwise.

2 Linear maps and differentiation

If $f : S \to T$, then its derivative ∂f has type

$$\partial f : S \to (S \multimap T)$$

where $S \multimap T$ is the type of *linear maps* from S to T. That is, at some point $p : S$, $\partial f(p)$ is a linear map that is a good approximation of f at p.

By "a good approximation of f at p" we mean this:

$$\forall p : S.\ f(p + \delta_p) \approx f(p) + \partial f(p) \odot \delta_p$$

Here the operation $(\odot)$ is linear-map application: it takes a linear map $S \multimap T$ and applies it to an argument of type S, giving a result of type T (Figure 3).

The linear maps from S to T are a subset of the functions from S to T. We characterise linear maps more precisely in Section 2.1, but a good intuition can be had for functions $g : \mathbb{R}^2 \to \mathbb{R}$. This function defines a curvy surface $z = g(x, y)$. Then a linear map of type $\mathbb{R}^s \multimap \mathbb{R}$ is a plane, and $\partial g(p_x, p_y)$ is the plane that best approximates g near (p_x, p_y), that is a tangent plane passing through $z = g(p_x, p_y)$

2.1 Linear maps

A *linear map*, $m : S \multimap T$, is a function from S to T, satisfying these two properties:

$$
\begin{array}{llrcl}
(LM1) & \forall x, y : S & m \odot (x + y) & = & m \odot x + m \odot y \\
(LM2) & \forall k : \mathbb{R}, x : S & k * (m \odot x) & = & m \odot (k * x)
\end{array}
$$

Here $(\odot) : (s \multimap t) \to (s \to t)$ is an operator that applies a linear map $(s \multimap t)$ to an argument of type s. The type $s \multimap t$ is a type in the language (Figure 1).

Linear maps can be *built and consumed* using the operators in (see Figure 3). Indeed, you should think of linear maps as an *abstract type*; that is, you can *only* build or consume linear maps with the operators in Figure 3. We might *represent* a linear map in a variety of ways, one of which is as a matrix (Section 2.5).

2.1.1 Semantics of linear maps

The *semantics* of a linear map is completely specified by saying what ordinary function it corresponds to; or, equivalently, by how it behaves when applied to an argument by $(\odot)$. The semantics of each form of linear map are given in Figure 4

2.1.2 Properties of linear maps

Linear maps satisfy *properties* given in Figure 4. Note that $(\circ)$ and $\oplus$ behave like multiplication and addition respectively.

These properties can readily be proved from the semantics. To prove two linear maps are equal, we must simply prove that they give the same result when applied to any argument. So, to prove that $\mathbf{0} \circ m = m$, we choose an arbitrary x and reason thus:

$$
\begin{array}{cll}
& (\mathbf{0} \circ m) \odot x & \\
= & \mathbf{0} \odot (m \odot x) & \{\text{semantics of } (\circ)\} \\
= & 0 & \{\text{semantics of } \mathbf{0}\} \\
= & \mathbf{0} \odot x & \{\text{semantics of } \mathbf{0} \text{ backwards}\}
\end{array}
$$

Note that the property

$$(m_1 \bowtie m_2) \circ (n_1 \times n_2) = (m_1 \circ n_1) \oplus (m_2 \circ n_2)$$

is the only reason we need the linear map $(\oplus)$.

	Operator		Type	Matrix interpretation where $s = \mathbb{R}^m$, and $t = \mathbb{R}^n$
Apply	$(\odot)$	:	$(s \multimap t) \to \delta s \to \delta t$	Matrix/vector multiplication
Reverse apply	$(\odot_R)$	:	$\delta t \to (s \multimap t) \to \delta s$	Vector/matrix multiplication
Compose	$(\circ)$	:	$(s \multimap t,\ r \multimap s) \to (r \multimap t)$	Matrix/matrix multiplication
Sum	$(\oplus)$	:	$(s \multimap t,\ s \multimap t) \to (s \multimap t)$	Matrix addition
Zero	$\mathbf{0}$	:	$s \multimap t$	Zero matrix
Unit	$\mathbf{1}$	:	$s \multimap s$	Identity matrix (square)
Scale	$\mathcal{S}(\cdot)$	:	$\mathbb{R} \to (s \multimap s)$	
VCat	$(\times)$	:	$(s \multimap t_1,\ s \multimap t_2) \to (s \multimap (t_1, t_2))$	Vertical juxtaposition
VCatV	$\mathcal{V}(\cdot)$	:	$Vec\ n\ (s \multimap t) \to (s \multimap Vec\ n\ t)$	...vector version
HCat	$(\bowtie)$	:	$(t_1 \multimap s,\ t_2 \multimap s) \to ((t_1, t_2) \multimap s)$	Horizontal juxtaposition
HCatV	$\mathcal{H}(\cdot)$	:	$Vec\ n\ (t \multimap s) \to (Vec\ n\ t \multimap s)$	...vector version
Transpose	$\cdot^{\top}$	:	$(s \multimap t) \to (t \multimap s)$	Matrix transpose
NB: We expect to have only $\mathcal{L}/\mathcal{L}'$ but not both				
Lambda	$\mathcal{L}$	:	$(\mathbb{N} \to (s \multimap t)) \to (s \multimap (\mathbb{N} \to t))$	
TLambda	$\mathcal{L}'$	:	$(\mathbb{N} \to (t \multimap s)) \to ((\mathbb{N} \to t) \multimap s)$	Transpose of $\mathcal{L}$

Figure 3. Operations over linear maps

Theorem: $\forall (m : S \multimap T).\ m \odot 0 = 0$. That is, all linear maps pass through the origin. **Proof**: property (LM2) with $k = 0$. Note that the function $\lambda x.x + 4$ is not a linear map; its graph is a staight line, but it does not go through the origin.

2.2 Vector spaces

Given a linear map $m : S \multimap T$, we expect both S and T to be a *vector space with dot product* (aka inner product space[1]). A vector space with dot product V has:

- *Vector addition* $(+_V) : V \to V \to V$.
- *Zero vector* $0_V : V$.
- *Scalar multiplication* $(*_V) : \mathbb{R} \to V \to V$
- *Dot-product* $(\bullet_V) : V \to V \to \mathbb{R}$.

We omit the V subscripts when it is clear which $(*)$, $(+)$, $(\bullet)$ or 0 is intended.

These operations must obey the laws of vector spaces

$$\begin{aligned}
v_1 + (v_2 + v_3) &= (v_1 + v_2) + v_3 \\
v_1 + v_2 &= v_2 + v_1 \\
v + 0 &= 0 \\
0 * v &= 0 \\
1 * v &= v \\
r_1 * (r_2 * v) &= (r_1 * r_2) * v \\
r * (v_1 + v_2) &= (r * v_1) + (r * v_2) \\
(r_1 + r_2) * v &= (r_1 * v) + (r_2 * v)
\end{aligned}$$

[1] https://en.wikipedia.org/wiki/Vector_space

2.2.1 Building vector spaces

What types are vector spaces? Look the syntax of types in Figure 1.

- The real numbers $\mathbb{R}$ is a vector space, using the standard $+$ and $*$ for reals; and $\bullet_{\mathbb{R}} = *$.
- If V is a vector space then $Vec\ n\ V$ is a vector space, with
  - $v_1 + v_2$ is vector addittion
  - $r * v$ multiplies each element of the vector v by the real r.
  - $v_1 \bullet v_2$ is a the usual vector dot-product.

  We often write $Vec\ n\ \mathbb{R}$ as $\mathbb{R}^N$.
- If V_1 and V_2 are vector spaces, then the product space (V_1, V_2) is a vector space
  - $(v_1, v_2) + (w_1, w_2) = (v_1 + w_1, v_2 + w_2)$.
  - $r * (v_1, v_2) = (r * v_1, r * v_2)$
  - $(v_1, v_2) \bullet (w_1, w_2) = (v_1 \bullet w_1) + (v_2 \bullet w_2)$.

In all cases the necessary properties of the operations (associativity, distribution etc) are easy to prove.

2.3 Transposition

For any linear map $m : S \multimap T$ we can produce its transpose $m^{\top} : T \multimap S$. Despite its suggestive type, the transpose is *not* the inverse of m! (In the world of matrices, the transpose of a matrix is not the same as its inverse.)

Semantics of linear maps

$$
\begin{aligned}
(m_1 \circ m_2) \odot x &= m_1 \odot (m_2 \odot x) \\
(m_1 \times m_2) \odot x &= (m_1 \odot x, m_2 \odot x) \\
(m_1 \bowtie m_2) \odot (x_1, x_2) &= (m_1 \odot x_1) + (m_2 \odot x_2) \\
(m_1 \oplus m_2) \odot x &= (m_1 \odot x) + (m_2 \odot x) \\
\mathbf{0} \odot x &= 0 \\
\mathbf{1} \odot x &= x \\
\mathcal{S}(k) \odot x &= k * x \\
\mathcal{V}(m) \odot x &= build(sz(m), \lambda i. m[i] \odot x) \\
\mathcal{H}(m) \odot x &= \Sigma_i\, (m[i] \odot x[i]) \\
\mathcal{L}(f) \odot x &= \lambda i. (f\ i) \odot x \\
\mathcal{L}'(f) \odot g &= \Sigma_i (f\ i) \odot g(i)
\end{aligned}
$$

Properties of linear maps

$$
\begin{aligned}
\mathbf{0} \circ m &= \mathbf{0} \\
m \circ \mathbf{0} &= \mathbf{0} \\
\mathbf{1} \circ m &= m \\
m \circ \mathbf{1} &= m \\
m \oplus \mathbf{0} &= m \\
\mathbf{0} \oplus m &= m \\
m \circ (n_1 \bowtie n_2) &= (m \circ n_1) \bowtie (m \circ n_2) \\
(m_1 \times m_2) \circ n &= (m_1 \circ n) \times (m_2 \circ n) \\
(m_1 \bowtie m_2) \circ (n_1 \times n_2) &= (m_1 \circ n_1) \oplus (m_2 \circ n_2) \\
\mathcal{S}(k_1) \circ \mathcal{S}(k_2) &= \mathcal{S}(k_1 * k_2) \\
\mathcal{S}(k_1) \oplus \mathcal{S}(k_2) &= \mathcal{S}(k_1 + k_2)
\end{aligned}
$$

Figure 4. Linear maps: semantics and properties

Definition 2.1. Given a linear map $m : S \multimap T$, its *transpose* $m^\top : T \multimap S$ is defined by the following property:

$$(TP) \quad \forall s{:}S,\ t{:}T.\ \ (m^\top \odot t) \bullet s = t \bullet (m \odot s)$$

This property *uniquely* defines the transpose, as the following theorem shows:

Theorem 2.2. *If m_1 and m_2 are linear maps satisfying*

$$\forall s\, t.\ (m_1 \odot s) \bullet t = (m_2 \odot s) \bullet t$$

then $m_1 = m_2$

Proof. It is a property of dot-product that if $v_1 \bullet x = v_2 \bullet x$ for every x, then $v_1 = v_2$. (Just use a succession of one-hot vectors for x, to pick out successive components of v_1 and v_2.) So (for every t):

$$
\begin{aligned}
&\forall s\, t.\ (m_1 \odot s) \bullet t = (m_2 \odot s) \bullet t \\
\Rightarrow\ &\forall s.\ m_1 \odot s = m_2 \odot s
\end{aligned}
$$

and that is the definition of extensional equality. So m_1 and m_2 are the same linear maps. □

Laws for transposition of linear maps

$$
\begin{aligned}
(m_1 \circ m_2)^\top &= m_2^\top \circ m_1^\top && \text{Note reversed order!} \\
(m_1 \times m_2)^\top &= m_1^\top \bowtie m_2^\top \\
(m_1 \bowtie m_2)^\top &= m_1^\top \times m_2^\top \\
(m_1 \oplus m_2)^\top &= m_1^\top \oplus m_2^\top \\
\mathbf{0}^\top &= \mathbf{0} \\
\mathbf{1}^\top &= \mathbf{1} \\
\mathcal{S}(k)^\top &= \mathcal{S}(k) \\
(m^\top)^\top &= m \\
\mathcal{V}(v)^\top &= \mathcal{H}(map\ (\cdot)^\top\ v) \\
\mathcal{H}(v)^\top &= \mathcal{V}(map\ (\cdot)^\top\ v) \\
\mathcal{L}(\lambda i.m)^\top &= \mathcal{L}'(\lambda i.m^\top) \\
\mathcal{L}'(\lambda i.m)^\top &= \mathcal{L}(\lambda i.m^\top)
\end{aligned}
$$

Laws for reverse-application

$$r \odot_R m = m^\top \odot r \qquad \text{By definition}$$

$$
\begin{aligned}
r \odot_R (m_1 \circ m_2) &= (r \odot_R m_1) \odot_R m_2 \\
(r_1, r_2) \odot_R (m_1 \times m_2) &= (r_1 \odot_R m_1) + (r_2 \odot_R m_2) \\
r \odot_R (m_1 \bowtie m_2) &= (r \odot_R m_1,\ r \odot_R m_2) \\
r \odot_R (m_1 \oplus m_2) &= (r \odot_R m_1) + (r \odot_R m_2) \\
r \odot_R \mathbf{0} &= 0 \\
r \odot_R \mathbf{1} &= r \\
r \odot_R \mathcal{S}(k) &= k * r \\
r \odot_R m^\top &= m \odot r \\
r \odot_R \mathcal{V}(v) &= \Sigma_i\, (r[i] \odot_R v[i]) \\
r \odot_R \mathcal{H}(v) &= build(sz(v), \lambda i. r \odot_R m[i])
\end{aligned}
$$

Figure 5. Laws for transposition

Figure 5 has a collection of laws about transposition. These identies are readily proved using the above definition. For example, to prove that $(m_1 \circ m_2)^\top = m_2^\top \circ m_1^\top$ we may reason as follows:

$$
\begin{aligned}
&((m_2^\top \circ m_1^\top) \odot t) \bullet s \\
&= (m_2^\top \odot (m_1^\top \odot t)) \bullet s && \text{Semantics of } (\circ) \\
&= (m_1^\top \odot t) \bullet (m_2 \odot s) && \text{Use (TP)} \\
&= t \bullet (m_1 \odot (m_2 \odot s)) && \text{Use (TP) again} \\
&= t \bullet ((m_1 \circ m_1) \odot s) && \text{Semantics of } (\circ)
\end{aligned}
$$

And now the property follows by Theorem 2.2.

2.4 Reverse linear-map application

Rather than transpose the linear map (which is a rather boring operation), just replacing one operator with another, it's easier to define a reverse-application operator for linear maps:

$$(\odot_R) : \delta t \to (s \multimap t) \to \delta s$$

It is defined by the following property:

$$(RP) \quad \forall s{:}\delta S,\ t{:}\delta T.\ \ (t \odot_R m) \bullet s = t \bullet (m \odot s)$$

2.5 Matrix interpretation of linear maps

A linear map $m{:}\mathbb{R}^M \multimap \mathbb{R}^N$ is isomorphic to a matrix $\mathbb{R}^{N \times M}$ with N rows and M columns.

Many of the operators over linear maps then have simple matrix interpetations; for example, composition of linear maps ($\circ$) is matrix multiplication, pairing ($\times$) is vetical juxtaposition, and so on. These matrix interpretations are all given in the final column of Figure 3.

You might like to check that matrix transposition satisfies property (TP).

When it comes to implementation, we do not want to *represent* a linear map by a matrix, becuase a linear map $\mathbb{R}^M \multimap \mathbb{R}^N$ is an $N \times M$ matrix, which is enormous if $N = M = 10^6$, say. The function might be very simple (perhaps even the identity function) and taking 10^{12} numbers to represent it is plain silly. So our goal will be to *avoid realising linear maps as matrices.*

2.6 Optimisation

In optimisation we are usually given a function $f : \mathbb{R}^N \to \mathbb{R}$, where N can be large, and asked to find values of the input that maximises the output. One way to do this is by *gradient descent*: start with a point p, make a small change to $p{+}\delta_p$, and so on. From p we want to move in the direction of maximum slope. (How *far* to move in that direction is another matter — indeed no one knows — but we will concentrate on the *direction* in which to move.)

Suppose $\delta(i, N)$ is the one-hot N-vector with 1 in the i'th position and zeros elsewhere. Then $\delta_p[i] = \partial f(p) \odot \delta(i, N)$ describes how fast the output of f changes for a change in the i'th input. The direction of maximum slope is just the vector

$$\delta_p = (\delta_p[1]\ \delta_p[2]\ \ldots\ \delta_p[N])$$

How can we compute this vector? We can simply evaluate $\partial f(p) \odot \delta(i, N)$ for each i. But that amounts to running f N times, which is bad if N is large (say 10^6).

Suppose that we somehow had access to $\partial_R f$. Then we can use property (TP), setting $\delta_f = 1$ to get

$$\forall \delta_p.\ \partial f(p) \odot \delta_p = (\partial_R f(p) \odot 1) \bullet \delta_p$$

Then

$$\begin{array}{rcl} \delta_p[i] & = & \partial f(p) \odot \delta(i, N) \\ & = & (\partial_R f(p) \odot 1) \bullet \delta(i, N) \\ & = & (\partial_R f(p) \odot 1)[i] \end{array}$$

That is $\delta_p[i]$ is the i'th component of $\partial_R f(p) \odot 1$, so $\delta_p = \partial_R f(p) \odot 1$.

That is, $\partial_R f(p) \odot 1$ is the N-vector of maximum slope, the direction in which to move if we want to do gradient descent starting at p. And *that* is why the transpose is important.

Original function $\quad f : S \to T$, $\quad f(x) = e$

Full Jacobian $\quad \partial f : S \to (S \multimap T)$, $\quad \partial f(x) = \text{let } \partial x{=}\mathbf{1} \text{ in } \nabla_S[\![e]\!]$

Forward derivative $\quad f' : (S, S) \to T$, $\quad f'(x, dx) = \partial f(x) \odot dx$

Reverse derivative $\quad f'_R : (S, T) \to S$, $\quad f'_R(x, dr) = dr \odot_R \partial f(x)$

Differentiation of an expression

$$\text{If } e : T \text{ then } \nabla_S[\![e]\!] : S \multimap T$$

$$\begin{array}{rcl} \nabla_S[\![k]\!] & = & \mathbf{0} \\ \nabla_S[\![x]\!] & = & \partial x \\ \nabla_S[\![f(e)]\!] & = & \partial f(e) \circ \nabla_S[\![e]\!] \\ \nabla_S[\![(e_1, e_2)]\!] & = & \nabla_S[\![e_1]\!] \times \nabla_S[\![e_2]\!] \\ \nabla_S[\![\text{let } x{=}e_1 \text{ in } e_2]\!] & = & \text{let } x = e_1 \text{ in} \\ & & \text{let } \partial x = \nabla_S[\![e_1]\!] \text{ in} \\ & & \nabla_S[\![e_2]\!] \\ \nabla_S[\![build(e_n, \lambda i.e)]\!] & = & \mathcal{V}(build(e_n, \lambda i.\nabla_S[\![e]\!])) \\ \nabla_S[\![\lambda i.\, e]\!] & = & \mathcal{L}(\lambda i.\, \nabla_S[\![e]\!]) \end{array}$$

Figure 6. Automatic differentiation

2.7 Lambdas and linear maps

Notice the similarity between the type of ($\times$) and the type of $\mathcal{L}$; the latter is really just an infinite version of the latter. Their semantics in Figure 4 are equally closely related.

The tranpsositions of these two linear maps, ($\bowtie$) and $\mathcal{L}'$, are similarly related. *But*, there is a problem with the semantics of $\mathcal{L}'$:

$$\mathcal{L}'(f) \odot g \quad = \quad \Sigma_i (f\ i) \odot g(i)$$

This is an *infinite sum*, so there is something fishy about this as a semantics.

2.8 Questions about linear maps

- Do we need $\mathbf{1}$? After all $\mathcal{S}(1)$ does the same job. But asking if $k = 1$ is dodgy when k is a float.
- Do these laws fully define linear maps?

Notes

- In practice we allow n-ary versions of $m \bowtie n$ and $m{\times}n$.

3 AD as a source-to-source transformation

To perform source-to-source AD of a function f, we follow the plan outlined in Figure 6. Specifically, starting with a function definition `f(x) = e`:

- Construct the full Jacobian ∂f, and transposed full Jacobian $\partial_R f$, using the tranformations in Figure 6[2].
- Optimise these two definitions, using the laws of linear maps in Figure 4.
- Construct the forward derivative f' and reverse derivative f'_R, as shown in Figure 6[3].
- Optimise these two definitions, to eliminate all linear maps. Specifically:
  - Rather than *calling* ∂f (in, say, f'), instead *inline* it.
  - Similarly, for each local let-binding for a linear map, of form $\text{let} \quad \partial x = e \ \text{ in } b$, inline ∂x at each of its occurrences in b. This may duplicate e; but ∂x is a function that may be applied (via $\odot$) to many different arguments, and we want to specialise it for each such call. (I think.)
  - Optimise using the rules of $(\odot)$ in Figure 4.
  - Use standard Common Subexpression Elimination (CSE) to recover any lost sharing.

Note that

- The transformation is fully compositional; each function can be AD'd independently. For example, if a user-defined function f calls another user-defined function g, we construct ∂g as described; and then construct ∂f. The latter simply calls ∂g.
- The AD transformation is *partial*; that is, it does not work for every program. In particular, it fails when applield to a lambda, or an application; and, as we will see in Section 4, it requires that *build* appears applied to a lambda.
- We give the full Jacobian for some built-in functions in Figure 6, including for conditionals (∂if).

3.1 Forward and reverse AD

Consider

```
f(x) = p( q( r( x )))
```

[2] We consider ∂f and $\partial_R f$ to be the names of two new functions. These names are derived from, but distinctd from f, rather like f' or f_1 in mathematics.

[3] Again f' and f'_R are new names, derived from f

Just running the algorithm above on f gives

$$
\begin{array}{rcl}
f(x) & = & p(q(r(x))) \\
\partial f(x) & = & \partial p \circ (\partial q \circ \partial r) \\
f'(x, dx) & = & (\partial p \circ (\partial q \circ \partial r)) \odot dx \\
& = & \partial p \odot ((\partial q \circ \partial r) \odot dx) \\
& = & \partial p \odot (\partial q \odot (\partial r \odot dx)) \\
\partial_R f(x) & = & (\partial_R r \circ \partial_R q) \circ \partial_R p \\
f'_R(x, dr) & = & ((\partial_R r \circ \partial_R q) \circ \partial_R p) \odot dr \\
& = & (\partial_R r \circ \partial_R q) \odot (\partial_R p \odot dr) \\
& = & \partial_R r \odot (\partial_R q \odot (\partial_R p \odot dr))
\end{array}
$$

In *"The essence of automatic differentiation"* Conal says (Section 12)

> The AD algorithm derived in Section 4 and generalized in Figure 6 can be thought of as a family of algorithms. For fully right-associated compositions, it becomes forward mode AD; for fully left-associated compositions, reverse-mode AD; and for all other associations, various mixed modes.

But the forward/reverse difference shows up quite differently here: it has nothing to do with *right-vs-left association*, and everything to do with *transposition.*

This is mysterious. Conal is not usually wrong. I would like to understand this better.

4 AD for vectors

Like other built-in functions, each built-in function for vectors has has its full Jacobian versions, defined in Figure 2. You may enjoy checking that ∂sum and ∂ixR are correct!

For *build* there are two possible paths, and it's not yet clear which is best

Direct path. Figure 6 includes a rule for $\nabla_S[\![build(e_n, \lambda i.e)]\!]$.

But *build* is an exception! It is handled specially by the AD transformation in Figure 6; there is no $\partial build$. Moreover the AD transformation only works if the second argument of the build is a lambda, thus $build(e_n, \lambda i.e)$. I tried dealing with build and lambdas separately, but failed (see Section ??).

I did think about having a specialised linear map for indexing, rather than using $\mathcal{H}()$, but then I needed its transposition, so just using $\mathcal{H}()$ seemed more economical. On the other hand, with the fucntions as I have them, I need the grotesquely delicate optimisation rule

$$
\begin{array}{l}
sum(build(n, \lambda i.\ \text{if } i == e_i \text{ then } e \text{ else } 0)) \\
\quad = \quad \text{let } i = e_i \text{ in } b \\
\text{if } i \notin e_i
\end{array}
$$

I hate this!

4.1 General folds

We have $sum :: Vec\ n\ \mathbb{R} \to \mathbb{R}$. What is ∂sum? One way to define its semantics is by applying it:

$$\begin{array}{rcl} \partial sum & :: & Vec\ n\ \mathbb{R} \to (Vec\ n\ \mathbb{R} \multimap \mathbb{R}) \\ \partial sum(v) \odot dv & = & sum(dv) \end{array}$$

That is OK. But what about product, which multiplies all the elements of a vector together? If the vector had three elements we might have

$$\begin{array}{l} \partial product([x_1, x_2, x_3]) \odot [dx_1, dx_2, dx_3] \\ \quad = (dx_1 * x_2 * x_3) + (dx_2 * x_1 * x_3) + (dx_3 * x_1 * x_2) \end{array}$$

This looks very unattractive as the number of elements grows. Do we need to use product?

This gives the clue that taking the derivative of *fold* is not going to be easy, maybe infeasible! Much depends on the particular lambda it appears. So I have left out product, and made no attempt to do general folds.

5 Avoiding duplication

5.1 ANF and CSE

We may want to ANF-ise before AD to avoid gratuitous duplication. E.g.

$$\begin{array}{ll} & \nabla_S[\![sqrt(x + (y * z))]\!] \\ = & \partial sqrt(x + (y * z)) \circ \nabla_S[\![x + (y * z)]\!] \\ = & \partial sqrt(x + (y * z)) \circ \partial + (x, y * z) \\ & \circ (\nabla_S[\![x]\!] \times \nabla_S[\![y * z]\!]) \\ = & \partial sqrt(x + (y * z)) \circ \partial + (x, y * z) \\ & \circ (\partial x \times (\partial * (y, z) \circ (\partial y \times \partial z))) \end{array}$$

Note the duplication of $y * z$ in the result. Of course, CSE may recover it.

5.2 Tupling: basic version

A better (and well-established) path is to modify $\partial f : S \to (S \multimap T)$ so that it returns a pair:

$$\overline{\partial f} : \forall a.(a \multimap S, S) \to (a \multimap T, T)$$

That is $\overline{\partial f}$ returns the "normal result" T as well as a linear map.

5.3 Polymorphic tupling: forward mode

Everything works much more compositionally if $\overline{\partial f}$ also *takes* a linear map as its input. The new transform is shown in Figure 8. Note that there is no longer any code duplications, even without ANF or CSE.

In exchange, though, all the types are a bit more complicated. So we regard Figure 6 as canonical, to be used when working thiungs out, and Figure 8 as a (crucial) implementation strategy.

The crucial property are these:

$$(CP) \quad \overline{\partial f}(e) \mathbin{\overline{\odot}} dx \quad = \quad f'(e \mathbin{\overline{\odot}} dx)$$

Crucial because suppose we have

$$\mathrm{f(x)\ =\ g(h(x))}$$

Then, we can transform as follows, using (CP) twice, on lines marked (†):

$$\begin{array}{rcll} \overline{\partial f}(\overline{x}) & = & \overline{\partial g}(\ \overline{\partial h}(\ \overline{x}\ )\ ) & \\ f'(x, dx) & = & \overline{\partial g}(\ \overline{\partial h}(\ x, \mathbf{1}\ )\ ) \mathbin{\overline{\odot}} dx & \\ & = & g'(\ \overline{\partial h}(\ x, \mathbf{1}\ ) \mathbin{\overline{\odot}} dx\ ) & (\dagger) \\ & = & g'(\ h'(\ (x, \mathbf{1}) \mathbin{\overline{\odot}} dx\ )) & (\dagger) \\ & = & g'(\ h'(\ x, \mathbf{1} \odot dx\ )) & \\ & = & g'(\ h'(\ x, dx\ )) & \end{array}$$

Why is (CP) true? It follows from a more general property of $\overline{\partial f}$:

$$\begin{array}{l} \forall f : S \to T,\ x : S,\ m_1 : A \multimap S,\ m_2 : B \multimap A,\ db : \delta B. \\ \quad \overline{\partial f}(x, m_1) \mathbin{\overline{\odot}} (m_2 \odot db) = \overline{\partial f}(x, m_1 \circ m_2) \mathbin{\overline{\odot}} db \end{array}$$

$$\begin{array}{l} \forall f : S \to T,\ x : S,\ m_1 : S \multimap A,\ m_2 : A \multimap B,\ dr : \delta T. \\ \quad m_2 \odot (\overline{\partial_R f}(x, m_1) \mathbin{\overline{\odot}} dr) = \overline{\partial_R f}(x, m_2 \circ m_1) \mathbin{\overline{\odot}} dr \end{array}$$

Now we can prove our claim as follows

$$\begin{array}{ll} & f'(e \mathbin{\overline{\odot}} dx) \\ = & \{\text{by defn of } (\ \overline{\odot}\ )\} \\ & f'(\pi_1(e),\ \pi_2(e) \odot dx) \\ = & \{\text{by defn of } f'\} \\ & \overline{\partial f}(\pi_1(e),\ \mathbf{1}) \mathbin{\overline{\odot}} (\pi_2(e) \odot dx) \\ = & \{\text{by crucial property}\} \\ & \overline{\partial f}(\pi_1(e),\ \pi_2(e)) \mathbin{\overline{\odot}} dx \\ = & \overline{\partial f}(e) \mathbin{\overline{\odot}} dx \end{array}$$

5.4 Polymorphic tupling: reverse mode

It turns out that things work quite differently for reverse mode. For a start the equivalent of (CP) for reverse-mode would look like this:

$$\overline{\partial_R f}(e) \mathbin{\overline{\odot}} dr = f'_R(e \mathbin{\overline{\odot}} dr)$$

But this is not even well-typed!

How did we use (CP)? Supppose f is defined in terms of g and h:

$$\mathrm{f(x)\ =\ g(h(x))}$$

Then we want f' to be defined in terms of g' and h'. That is, we want a *compositional* method, where we can create the code for f' without looking at the code for g or h, simpply by calling g and h's derived functions. And that's just what we achieved:

$$f'(x, dx) = g'(h'(x, dx))$$

Original function $f : S \to T$

$f(x) = e$

Full Jacobian $\overline{\partial f} : S \to (T, S \multimap T)$

$\overline{\partial f}(x) = \text{let } \overline{\partial x} = (x, \mathbf{1}) \text{ in } \overline{\nabla}_S[\![e]\!]$

Forward derivative $f' : (S, \delta S) \to (T, \delta T)$

$f'(x, dx) = \overline{\partial f}(x) \mathbin{\overline{\odot}} dx$

Reverse derivative $f'_R : (S, \delta T) \to (T, \delta S)$

$f'_R(x, dfr) = dr \mathbin{\overline{\odot}_R} \overline{\partial f}(x)$

Differentiation of an expression

$$\text{If } e : T \text{ then } \overline{\nabla}_S[\![e]\!] : (S \multimap T, T)$$

$$\begin{array}{rcl}
\overline{\nabla}_S[\![k]\!] & = & (k, \mathbf{0}) \\
\overline{\nabla}_S[\![x]\!] & = & \overline{\partial x} \\
\overline{\nabla}_S[\![(e_1, e_2)]\!] & = & \overline{\nabla}_S[\![e_1]\!] \mathbin{\overline{\times}} \overline{\nabla}_S[\![e_2]\!] \\
\overline{\nabla}_S[\![f(e)]\!] & = & \text{let } a = \overline{\nabla}_S[\![e]\!] \text{ in} \\
 & & \text{let } r = \overline{\partial f}(\pi_1(a)) \text{ in} \\
 & & (\pi_1(r),\ \pi_2(r) \circ \pi_2(a)) \\
\overline{\nabla}_S[\![\text{let } x{=}e_1 \text{ in } e_2]\!] & = & \text{let } \overline{\partial x} = \nabla_S[\![e_1]\!] \text{ in } \overline{\nabla}_S[\![e_2]\!] \\
\overline{\nabla}_S[\![build(e_n, \lambda i.e)]\!] & = & \text{let } p = \Phi(build(e_n, \lambda i.\overline{\nabla}_S[\![e]\!])) \text{ in} \\
 & & (\pi_1(p),\ \mathcal{V}(\pi_2(p)))
\end{array}$$

Modified linear-map operations

$$\begin{array}{rcl}
(\ \overline{\odot}\ ) & : & (r, s \multimap t) \to \delta s \to \delta t \\
(v, m) \mathbin{\overline{\odot}} ds & = & m \odot ds \\
 & & \\
(\ \overline{\odot}_R\ ) & : & \delta t \to (r, s \multimap t) \to \delta s \\
dr \mathbin{\overline{\odot}_R} vm & = & dr \mathbin{\overline{\odot}} vm \\
 & & \\
(\ \overline{\times}\ ) & : & ((t_1, s \multimap t_1),\ (t_2, s \multimap t_2)) \to ((t_1, t_2),\ s \multimap (t_1, t_2)) \\
(t_1, m_1) \mathbin{\overline{\times}} (t_2, m_2) & = & ((t_1, t_2), m_1 \times m_2) \\
 & & \\
(\ \overline{\bowtie}\ ) & : & ((t_1, t_1 \multimap s),\ (t_2, t_2 \multimap s)) \to ((t_1, t_2),\ (t_1, t_2) \multimap s) \\
(t_1, m_1) \mathbin{\overline{\bowtie}} (t_2, m_2) & = & ((t_1, t_2), m_1 \bowtie m_2) \\
 & & \\
\Phi & : & Vec\ n\ (a, b) \to (Vec\ n\ a, Vec\ n\ b) \\
 & & \\
.^{\overline{\top}} & : & (r, s \multimap t) \to (r, t \multimap s)
\end{array}$$

Derivatives of built-in functions

$$\begin{array}{rcl}
\overline{\partial +} & :: & (\mathbb{R}, \mathbb{R}) \to ((\mathbb{R}, \mathbb{R}) \multimap \mathbb{R}, \mathbb{R}) \\
\overline{\partial +}(x, y) & = & (\mathbf{1} \bowtie \mathbf{1},\ x + y) \\
 & & \\
\overline{\partial *} & :: & (\mathbb{R}, \mathbb{R}) \to ((\mathbb{R}, \mathbb{R}) \multimap \mathbb{R}, \mathbb{R}) \\
\overline{\partial *}(x, y) & = & (\mathcal{S}(y) \bowtie \mathcal{S}(x),\ x * y)
\end{array}$$

Figure 7. Automatic differentiation: tupling

$$
\begin{array}{ll}
\textbf{Original function} & f : S \to T \\
& f(x) = e \\
\textbf{Full Jacobian} & \overline{\partial f} : \forall a.\,(S, a \multimap S) \to (T, a \multimap T) \\
& \overline{\partial f}(\overline{x}) = \overline{\nabla}_a\llbracket e \rrbracket \\
\textbf{Transposed Jacobian} & \overline{\partial_R f} : \forall a.\,(S, S \multimap a) \to (T, T \multimap a) \\
& \overline{\partial_R f}(\overline{x}) = (\overline{\partial f}(\overline{x}))^{\overline{\top}} \\
\textbf{Forward derivative} & f' : (S, \delta S) \to (T, \delta T) \\
& f'(x, dx) = \overline{\partial f}(x, \mathbf{1}) \;\overline{\odot}\; dx \\
\textbf{Reverse derivative} & f'_R : (S, \delta T) \to (T, \delta S) \\
& f'_R(x, dr) = \overline{\partial_R f}(x, \mathbf{1}) \;\overline{\odot}\; dr
\end{array}
$$

Differentiation of an expression

$$
\text{If } e : T \text{ then } \overline{\nabla}_a\llbracket e \rrbracket : (T, a \multimap T)
$$

$$
\begin{array}{rcl}
\overline{\nabla}_a\llbracket k \rrbracket & = & (k, \mathbf{0}) \\
\overline{\nabla}_a\llbracket x \rrbracket & = & \overline{x} \\
\overline{\nabla}_a\llbracket f(e) \rrbracket & = & \overline{\partial f}(\ \overline{\nabla}_a\llbracket e \rrbracket\ ) \\
\overline{\nabla}_a\llbracket (e_1, e_2) \rrbracket & = & \overline{\nabla}_a\llbracket e_1 \rrbracket \;\overline{\times}\; \overline{\nabla}_a\llbracket e_2 \rrbracket \\
\overline{\nabla}_a\llbracket \text{let } x{=}e_1 \text{ in } e_2 \rrbracket & = & \text{let } \overline{x}{=}\overline{\nabla}_a\llbracket e_1 \rrbracket \text{ in } \overline{\nabla}_a\llbracket e_2 \rrbracket
\end{array}
$$

Modified linear-map operations

$$
\begin{array}{rcl}
(\ \overline{\odot}\ ) & : & (r, s \multimap t) \to \delta s \to (r, \delta t) \\
(v, m) \;\overline{\odot}\; ds & = & (v, m \odot ds) \\
\\
(\ \overline{\times}\ ) & : & ((t_1, s \multimap t_1),\ (t_2, s \multimap t_2)) \to ((t_1, t_2),\ s \multimap (t_1, t_2)) \\
(t_1, m_1) \;\overline{\times}\; (t_2, m_2) & = & ((t_1, t_2), m_1 \times m_2) \\
\\
(\ \overline{\bowtie}\ ) & : & ((t_1, t_1 \multimap s),\ (t_2, t_2 \multimap s)) \to ((t_1, t_2),\ (t_1, t_2) \multimap s) \\
(t_1, m_1) \;\overline{\bowtie}\; (t_2, m_2) & = & (t_1 + t_2, m_1 \bowtie m_2) \\
\\
.^{\overline{\top}} & : & (t, s \multimap t) \to (t, t \multimap s)
\end{array}
$$

Derivatives of built-in functions

$$
\begin{array}{rcl}
\overline{\partial +} & :: & \forall a.((\mathbb{R}, \mathbb{R}), a \multimap (\mathbb{R}, \mathbb{R})) \to (\mathbb{R}, a \multimap \mathbb{R}) \\
\overline{\partial +}((x, y), m) & = & (x + y,\ (\mathbf{1} \bowtie \mathbf{1}) \circ m) \\
\\
\overline{\partial *} & :: & \forall a.((\mathbb{R}, \mathbb{R}), a \multimap (\mathbb{R}, \mathbb{R})) \to (\mathbb{R}, a \multimap \mathbb{R}) \\
\overline{\partial *}((x, y), m) & = & (x * y,\ (\mathcal{S}(y) \bowtie \mathcal{S}(x)) \circ m)
\end{array}
$$

Figure 8. Automatic differentiation: polymorphic tuples

But for reverse mode, this plan is much less straightforward. Look at the types:

$$
\begin{array}{rcl}
f & : & R \to T \\
g & : & S \to T \\
h & : & R \to S \\
f'_R & : & (R, \delta T) \to (T, \delta R) \\
g'_R & : & (S, \delta T) \to (T, \delta S) \\
h'_R & : & (R, \delta S) \to (S, \delta R)
\end{array}
$$

How can we define f'_R by calling g'_R and h'_R? It would have to look something like this

$$
\begin{array}{rcl}
f'_R(r, dt) & = & \text{letrec}\ \ (t, ds) = g'_R(s, dt) \\
& & \qquad\quad\ (s, dr) = h'_R(r, ds) \\
& & \text{in } (t, dr)
\end{array}
$$

We can't call g'_R before h'_R, nor the other way around. That's why there is a letrec! Even leaving aside how we generate this code, We'd need lazy evaluation to execute it.

The obvious alternative is to change f''s interface. Currently we have

$$f'_R : (R, \delta T) \to (T, \delta R)$$

Instead, we can take that R value, but return a function $\delta T \to \delta R$, thus:

$$f'_R : R \to (T, \delta T \to \delta R)$$

But that commits to returning a *function*, with its fixed, built-in representation. Instead, let's return linear map:

$$f'_R : R \to (T, \delta T \multimap \delta R)$$

Now we can re-interpret the retuned linear map as some kind of record (trace) of all the things that f did. And if we insist on our compositional account we really must *manifest* that data structure, and later apply it to a value of type δT to get a value of type δR. We could represent those linear maps as:

- A matrix
- A function closure that, when called, applies the linear map to an argument
- A syntax tree whose nodes are the constructors of the linear map type. When applying the linear map, we interpret taht syntax tree.

Finally, notice that this final version of f' is exactly $\overline{\partial_R f}$, just specialised with an input linear map of $\mathbf{1}$. So we may as well just use $\overline{\partial_R f}$, which *already* compositionally calls $\overline{\partial_R g}$ and $\overline{\partial_R h}$.

TL;DR: for reverse mode, we must simply compile $\overline{\partial_R f}$.

Notice that we can get quite a bit of optimisation by inlining $\overline{\partial_R g}$ into $\overline{\partial_R f}$, and so on. The more inlining the better. If we inline everything we'll elminate all intermediate linear maps.

6 Implementation

The implementation differs from this document as follows:

- Rather than pairs, the implementation supports n-ary tuples. Similary the linear maps $(\times)$ and $\bowtie$ are n-ary.
- Functions definitions can take n arguments, thus

  ```
  f(x,y,z) = e
  ```

  This is treated as equivalent to

  $$\begin{aligned} f(t) \;=\; \mathrm{let}\;\; & x = \pi_{1,3}(t) \\ & y = \pi_{2,3}(t) \\ & z = \pi_{3,3}(t) \\ \mathrm{in}\;\; & e \end{aligned}$$

7 Fold

8 Demo

You can run the prototype by saying `ghci Main`.

The function `demo :: Def -> IO ()` runs the prototype on the function provided as example. Thus:

```
bash$ ghci Main

*Main> demo ex2

----------------------------
Original definition
----------------------------
fun f2(x)
  = let { y = x * x }
    let { z = x + y }
    y * z

----------------------------
Anf-ised original definition
----------------------------
fun f2(x)
  = let { y = x * x }
    let { z = x + y }
    y * z

----------------------------
The full Jacobian (unoptimised)
----------------------------
fun Df2(x)
  = let { Dx = lmOne() }
    let { y = x * x }
    let { Dy = lmCompose(D*(x, x), lmVCat(Dx, Dx)) }
    let { z = x + y }
    let { Dz = lmCompose(D+(x, y), lmVCat(Dx, Dy)) }
    lmCompose(D*(y, z), lmVCat(Dy, Dz))

----------------------------
The full Jacobian (optimised)
----------------------------
fun Df2(x)
  = let { y = x * x }
    lmScale( (x + y) * (x + x) + (x + y) * (x + x) )

----------------------------
Forward derivative (unoptimised)
----------------------------
fun f2'(x, dx)
  = lmApply(let { y = x * x }
            lmScale( (x + y) * (x + x) +
                     (x + y) * (x + x) ),
            dx)

----------------------------
Forward-mode derivative (optimised)
----------------------------
fun f2'(x, dx)
  = let { y = x * x }
```

Typing rules for fold

$$\begin{array}{rcl} t &:& (a,b) \\ e &:& a \\ acc &:& a \\ v &:& \text{Vec } b \\ \text{fold } (\lambda t.e)\ acc\ v &:& a \end{array}$$

Typing rules for lmFold

$$\begin{array}{rcl} t &:& (a,b) \\ e &:& a \\ e' &:& (s,(a,b)) \multimap a \\ acc &:& a \\ v &:& \text{Vec } b \\ \text{lmFold } (\lambda t.e)\ (\lambda t.e')\ acc\ v &:& (s,(a,\text{Vec } b)) \multimap a \end{array}$$

Typing rules for FFold and RFold

$$\begin{array}{rcl} t &:& (a,b) \\ t_{dr} &:& ((a,b),\delta a) \\ t_{dt} &:& ((a,b),(\delta a,\delta b)) \\ e &:& a \\ e_{dr} &:& (\delta s,(\delta a,\delta b)) \\ e_{dt} &:& \delta a \\ acc &:& a \\ v &:& \text{Vec } b \\ dr &:& \delta a \\ d_{acc} &:& \delta a \\ d_v &:& \text{Vec } \delta b \\ \text{FFold } (\lambda t.e)\ acc\ v\ (\lambda t_{dt}.e_{dt})\ d_{acc}\ d_v &:& \delta a \\ \text{RFold } (\lambda t.e)\ (\lambda t_{dr}.e_{dr})\ acc\ v\ dr &:& (\delta s,(\delta a,\text{Vec } \delta b)) \end{array}$$

Figure 9. Rules for fold

```
    ((x + y) * (x + x) + (x + y) * (x + x)) * dx

----------------------------
Forward-mode derivative (CSE'd)
----------------------------
fun f2'(x, dx)
  = let { t1 = x + x * x }
    let { t2 = x + x }
    (t1 * t2 + t1 * t2) * dx

----------------------------
Transposed Jacobian
----------------------------
fun Rf2(x)
  = lmTranspose( let { y = x * x }
                 lmScale( (x + y) * (x + x) +
                         (x + y) * (x + x) ) )
```

```
----------------------------
Optimised transposed Jacobian
----------------------------
fun Rf2(x)
  = let { y = x * x }
    lmScale( (x + y) * (x + x) +
             (x + y) * (x + x) )

----------------------------
Reverse-mode derivative (unoptimised)
----------------------------
fun f2`(x, dr)
  = lmApply(let { y = x * x }
            lmScale( (x + y) * (x + x) +
                     (x + y) * (x + x) ),
            dr)
```

Differentiation of fold

$$\text{If } e : T \text{ then } \nabla_s[\![e]\!] : s \multimap T$$

$$\begin{array}{rcl}
\nabla_s[\![\text{fold } (\lambda t.e)\ acc\ v]\!] & = & \text{lmFold } (\lambda t.e)\ (\lambda t.e')\ acc\ v \circ p \\
\text{where } p & : & s \multimap (s, (a, \text{Vec } b)) \\
p & = & \mathbf{1}_s \times (\nabla_s[\![acc]\!] \times \nabla_s[\![v]\!]) \\
e' & = & \text{let } \nabla x = \nabla x \circ (\mathbf{1}_s \bowtie \mathbf{0}_s^{(a,b)}) \\
 & & \ldots \text{ for each } x \text{ ocurring free in } \lambda t.e \\
 & & \text{let } \nabla t = \mathbf{0}_{(a,b)}^{s} \bowtie \mathbf{1}_{(a,b)} \\
 & & \text{in } \nabla_{(s,(a,b))}[\![e]\!]
\end{array}$$

Applying an lmFold

$$\begin{array}{rcl}
\text{lmFold } (\lambda t.e)\ (\lambda t.e')\ acc\ v \odot dx & = & \text{FFold } (\lambda t.e)\ acc\ v\ (\lambda t_{dt}.e_{dt})\ d_{acc}\ d_v \\
\text{where } e_{dt} & = & \text{let } t = \pi_1(t_{dt}) \\
 & & \text{let } dt = \pi_2(t_{dt}) \\
 & & \text{in } e' \odot (ds, dt) \\
ds & = & \pi_1(dx) \\
d_{acc} & = & \pi_1(\pi_2(dx)) \\
d_v & = & \pi_2(\pi_2(dx)) \\
 & & \\
dx \odot_R \text{lmFold } (\lambda t.e)\ (\lambda t.e')\ acc\ v & = & \text{RFold } (\lambda t.e)\ (\lambda t_{dr}.e_{dr})\ acc\ v\ dx \\
\text{where } e_{dr} & = & \text{let } t = \pi_1(t_{dr}) \\
 & & \text{let } dr = \pi_2(t_{dr}) \\
 & & \text{in } dr \odot_R e'
\end{array}$$

Figure 10. Rules for fold

```
def FFold dA ((f  : F)  (acc  : A)  (v  : Vec n B)
              (f_ : F_) (dacc : dA) (dv : Vec n dB))
  = FFold_recursive(0, f, acc, v f_, dacc, dv)

def FFold_recursive dA ((i : Integer) (f  : F)  (acc  : A)  (v  : Vec n B)
                                      (f_ : F_) (dacc : dA) (dv : Vec n dB))
  = if i == n
    then dacc
    else let fwd_f = f_((acc, v[i]), (dacc, dv[i]))
         in FFold_recursive(i + 1, f, f(acc, v[i]), v, f_, fwd_f, dv)
```

Figure 11. Forward mode derivative for fold

```
----------------------------
Reverse-mode derivative (optimised)
----------------------------
fun f2`(x, dr)
  = let { y = x * x }
    ((x + y) * (x + x) +
     (x + y) * (x + x)) * dr

----------------------------
```

```
Reverse-mode derivative (CSE'd)
----------------------------

fun f2`(x, dr)
  = let { t1 = x + x * x }
    let { t2 = x + x }
    (t1 * t2 + t1 * t2) * dr
```

```
def RFold (S, (dA, Vec n dB)) ((f : F) (f_ : F_) (acc : A) (v : Vec n B) (dr : dA))
  = let (ds, dv, da) = RFold_recursive(f, f_, 0, v, acc, dr)
    in (s, (da, dv))

def RFold_recursive (S, Vec n dB, dA) ((f : F) (f_ : F_) (i : Integer) (v : Vec n B)
                                       (acc : A) (dr : dA))
  = if i == n
    then (0, 0, dr)
    else let (r_ds, r_dv, r_dacc) = RFold_recursive(f, f_, i + 1, v, f(acc, v[i]), dr)
             (f_ds, (f_dacc, f_db)) = f_((acc, v[i]), r_dacc)
         in (r_ds + f_ds, r_dv + deltaVec(i, f_db), f_dacc)
```

Figure 12. Reverse mode derivative for fold