Automatic differentiation in Coconut

Tom Ellis
Simon Peyton Jones
Andrew Fitzgibbon

Atoms

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

Terms

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

Types

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

Figure 1. Syntax of the language

1 The language

The syntax of our intermediate 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. AWF: at some point we should say where this restriction is needed
- 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.)
- Conditionals are are a language construct. SPJ: Treating "if" as a function just didn't work; in particular ∇if needed a linear-map version of "if" and once we have that we might as well build "if" in. Anyway, conditionals are very fundamental, so it's unsurprising.
- Let-bindings are non-recursive. For now, at least, top-level functions are also non-recursive. SPJ: I think that top-level recursive functions might be OK, but I don't want to think about that yet.
- 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.

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 $[m_1; m_2]$ is short for $VCat(e_1, e_2)$.

We allow ourselves to write vector indexing 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 $Vec\ T$, 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

$$
\begin{array}{rcll}
(+) & :: & \mathit{Field}\ t \Rightarrow (t,t) \to t & \\
(*) & :: & \mathit{Field}\ t \Rightarrow (t,t) \to t & \\
\pi_{1,2} & :: & (t_1,t_2) \to t_1 & \text{Selection} \\
\pi_{2,2} & :: & (t_1,t_2) \to t_2 & \text{..ditto..} \\
\mathit{build} & :: & (\mathbb{N},\ \mathbb{N} \to t) \to \mathit{Vec}\ t & \text{Vector build} \\
\mathit{index} & :: & (\mathbb{N},\ \mathit{Vec}\ t) \to t & \text{Indexing} \\
\mathit{sum} & :: & \mathit{Field}\ t \Rightarrow \mathit{Vec}\ t \to t & \text{Sum a vector} \\
\mathit{size} & :: & \mathit{Vec}\ t \to \mathbb{N} & \text{Size of a vector}
\end{array}
$$

Derivatives of built-in functions

$$
\begin{array}{rcl}
\nabla + & :: & \mathit{Field}\ t \Rightarrow (t,t) \to ((t,t) \multimap t) \\
\nabla +(x,y) & = & \mathbf{1} \bowtie \mathbf{1} \\
\\
\nabla * & :: & \mathit{Field}\ t \Rightarrow (t,t) \to ((t,t) \multimap t) \\
\nabla *(x,y) & = & \mathcal{S}(y) \bowtie \mathcal{S}(x) \\
\\
\nabla \pi_{1,2} & :: & (t,t) \to ((t,t) \multimap t) \\
\nabla \pi_{1,2}(x) & = & \mathbf{1} \bowtie \mathbf{0} \\
\\
\nabla \mathit{index} & :: & (\mathbb{N},\ \mathit{Vec}\ t) \to ((\mathbb{N},\ \mathit{Vec}\ t) \multimap t) \\
\nabla \mathit{index}(i,v) & = & \mathbf{0} \bowtie \mathcal{B}'(\mathit{size}(v), \lambda j.\ \text{if } i = j \text{ then } \mathbf{1} \text{ else } \mathbf{0}) \\
\\
\nabla \mathit{sum} & :: & \mathit{Field}\ t \Rightarrow \mathit{Vec}\ t \to (\mathit{Vec}\ t \multimap t) \\
\nabla \mathit{sum}(v) & = & \mathcal{B}'(\mathit{size}(v),\ \lambda i.\mathbf{1}) \\
\\
\ldots & &
\end{array}
$$

Figure 2. Built-in functions

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

- $\mathit{build} :: (\mathbb{N},\ \mathbb{N} \to t) \to \mathit{Vec}\ t$ for vector construction.
- $\mathit{index} :: (\mathbb{N},\ \mathit{Vec}\ t) \to t$ for indexing. Informally we allow ourselves to write $v[i]$ instead of $\mathit{index}(i,v)$.
- $\mathit{sum} :: \mathit{Field}\ t \Rightarrow \mathit{Vec}\ t \to t$ to add up the elements of a vector. We specifically do not have a general, higher order, fold operator; we say why in Section 3.4. TE: I believe that for a vector v of size n, $\mathit{sum}(v)$ is the same as $\mathcal{B}'(n, \mathit{const}\ \mathit{id})\ v$. This may or may not be useful in reducing the size of the base language, should we want to do that. SPJ: I don't think so! $\mathcal{B}'$ is a linear map, so you can't apply it to v. Maybe you mean $\mathcal{B}'(n, \mathit{const}\ \mathit{id}) \odot v$? But that (Figure 4) is defined using *sum*!
- $\mathit{size} :: \mathit{Vec}\ t \to \mathbb{N}$ takes the size of a vector.
- Arithmetic functions $(*), (+)$ etc are overloaded to work over vectors, always elementwise.

SPJ: Do we need scan? Or (specialising to $(+)$) cumulative sum?

2 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).
- 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 2.1).
- 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
- Linear maps satisfy *laws* given in Figure 4. Note that $(\circ)$ and $\oplus$ behave like multiplication and addition respectively.
- **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.

2.1 Matrix interpretation of linear maps

A linear map $m :: \mathbb{R}^m \multimap \mathbb{R}^n$ is isomorphic to an 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 $([;])$ is vetical juxtaposition, and so on. These matrix interpretations are all tiven in the final column of Figure 3.

2.2 Lambdas and linear maps

Notice the similarity between the type of $([;])$ 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 $\mathcal{L}'$:

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

	Operator		Type	Matrix interpretation where $s = \mathbb{R}^m$, and $t = \mathbb{R}^n$
Apply	$(\odot)$	:	$(s \multimap t) \to (s \to t)$	Matrix/vector multiplication
Compose	$(\circ)$	:	$(s \multimap t,\ r \multimap s) \to (r \multimap t)$	Matrix/matrix multiplication
Sum	$(\oplus)$	:	$\mathit{Field}\ t \Rightarrow (s \multimap t,\ s \multimap t) \to (s \multimap t)$	Matrix addition
Zero	$\mathbf{0}$	:	$\mathit{Field}\ t \Rightarrow s \multimap t$	Zero matrix
Unit	$\mathbf{1}$	:	$s \multimap s$	Identity matrix (square)
Scale	$\mathcal{S}(\cdot)$	:	$\mathit{Field}\ s \Rightarrow s \to (s \multimap s)$	
Pair	$([\cdot;\cdot])$	:	$\mathit{Field}\ s \Rightarrow (s \multimap t_1,\ s \multimap t_2) \to (s \multimap (t_1, t_2))$	Vertical juxtaposition
Join	$(\bowtie)$	:	$\mathit{Field}\ s \Rightarrow (t_1 \multimap s,\ t_2 \multimap s) \to ((t_1, t_2) \multimap s)$	Horizontal juxtaposition
Transpose	$\cdot^\top$	:	$(s \multimap t) \to (t \multimap s)$	Matrix transpose
NB: We expect to have only $\mathcal{L}/\mathcal{L}'$ or $\mathcal{B}/\mathcal{B}'$, 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 $\mathcal{L}$
Build	$\mathcal{B}$	:	$(\mathbb{N},\ \mathbb{N} \to (s \multimap t)) \to (s \multimap \mathit{Vec}\ t)$	
BuildT	$\mathcal{B}'$	:	$(\mathbb{N},\ \mathbb{N} \to (t \multimap s)) \to (\mathit{Vec}\ t \multimap s)$	Transpose of $\mathcal{B}$

Figure 3. Operations over linear maps

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

For this reason

2.3 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. AWF: No, perfectly fine to ask if a float is 1 — the nearby floats are far away, and there's no other float f such that $\mathcal{S}(f) = \mathbf{1}$. SPJ: For the purposes of this paper I prefer having $\mathbf{1}$ as well; unity plays such a key role!
- Do these laws fully define linear maps?

Notes

- In practice we allow n-ary versions of $m \bowtie n$ and $[m; n]$.

3 Automatic differentiation

To perform source-to-source AD of a function f, we follow the plan outlined in Figure 5. Specifically, starting with a function definition `f(x) = e`:

- Construct the full Jacobian ∇f, and transposed full Jacobian $\nabla^\top f$, using the tranformations in Figure 5[1].
- Optimise these two definitions, using the laws of linear maps in Figure 4.
- Construct the forward derivative f' and reverse derivative $f^{\backprime}$, as shown in Figure 5[2].
- 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 let $\nabla x = e$ in b, inline ∇x at each of its occurences 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 3.3, it requires that *build* appears applied to a lambda.

[1] We consider ∇f and $\nabla^\top f$ 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.

[2] Again f' and $f^{\backprime}$ are new names, derived from f

Semantics of linear maps

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

Rules for transposition of linear maps

$$
\begin{aligned}
(m_1 \circ m_2)^\top &= m_2^\top \circ m_1^\top \qquad \text{Note reversed order!} \\
([m_1; m_2])^\top &= m_1^\top \bowtie m_2^\top \\
(m_1 \bowtie m_2)^\top &= m_1^\top [;^\top m_2] \\
(m_1 \oplus m_2)^\top &= m_1^\top \oplus m_2^\top \\
\mathbf{0}^\top &= \mathbf{0} \\
\mathbf{1}^\top &= \mathbf{1} \\
\mathcal{S}(k)^\top &= \mathcal{S}(k) \\
(m^\top)^\top &= m \\
\mathcal{B}(n,\ \lambda i.m)^\top &= \mathcal{B}'(n,\ \lambda i.m^\top) \\
\mathcal{B}'(n,\ \lambda i.m)^\top &= \mathcal{B}(n,\ \lambda i.m^\top) \\
\mathcal{L}(\lambda i.m)^\top &= \mathcal{L}'(\lambda i.m^\top) \\
\mathcal{L}'(\lambda i.m)^\top &= \mathcal{L}(\lambda i.m^\top)
\end{aligned}
$$

Laws for linear maps

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

Figure 4. Laws for linear maps

Original function	$f : S \to T$ $f(x) = e$
Full Jacobian	$\nabla f : S \to (S \multimap T)$ $\nabla f(x) = \text{let } \nabla x = \mathbf{1} \text{ in } \nabla_S [\![e]\!]$
Transposed Jacobian	$\nabla^\top f : S \to (T \multimap S)$ $\nabla^\top f(x) = (\nabla f(x))^\top$
Forward derivative	$f' : (S, S) \to T$ $f'(x, dx) = \nabla f(x) \odot dx$
Reverse derivative	$f` : (S, T) \to S$   $f`(x, dr) = \nabla^\top f(x) \odot dr$

Differentiation of an expression

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

$$
\begin{aligned}
\nabla_S [\![k]\!] &= \mathbf{0} \\
\nabla_S [\![x]\!] &= \nabla x \\
\nabla_S [\![f(e)]\!] &= \nabla f(e) \circ \nabla_S [\![e]\!] \\
\nabla_S [\![(e_1, e_2)]\!] &= [\nabla_S [\![e_1]\!]; \nabla_S [\![e_2]\!]] \\
\nabla_S [\![build(e_n, \lambda i.e)]\!] &= \mathcal{B}(e_n, \lambda i. \nabla_S [\![e]\!]) \\
\nabla_S [\![\lambda i.\, e]\!] &= \mathcal{L}(\lambda i.\, \nabla_S [\![e]\!]) \\
\nabla_S [\![\text{let } x{=}e_1 \text{ in } e_2]\!] &= \text{let } x = e_1 \text{ in} \\
&\quad\ \text{let } \nabla x = \nabla_S [\![e_1]\!] \text{ in} \\
&\quad\ \nabla_S [\![e_2]\!]
\end{aligned}
$$

Figure 5. Automatic differentiation

- We give the full Jacobian for some built-in functions in Figure 5, including for conditionals (∇if).

3.1 Forward and reverse AD

Consider

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

Just running the algorithm above on f gives

$$
\begin{aligned}
f(x) &= p(q(r(x))) \\
\nabla f(x) &= \nabla p \circ (\nabla q \circ \nabla r) \\
f'(x, dx) &= (\nabla p \circ (\nabla q \circ \nabla r)) \odot dx \\
&= \nabla p \odot ((\nabla q \circ \nabla r) \odot dx) \\
&= \nabla p \odot (\nabla q \odot (\nabla r \odot dx)) \\
\nabla^\top f(x) &= (\nabla^\top r \circ \nabla^\top q) \circ \nabla^\top p \\
f`(x, dr) &= ((\nabla^\top r \circ \nabla^\top q) \circ \nabla^\top p) \odot dr \\
&= (\nabla^\top r \circ \nabla^\top q) \odot (\nabla^\top p \odot dr) \\
&= \nabla^\top r \odot (\nabla^\top q \odot (\nabla^\top p \odot dr))
\end{aligned}
$$

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. TE: I was also puzzled by this. Conal's claim is suspicious to me, but firstly it's very cool and secondly it's Conal, so I want it to be true and I still hope it is.

3.2 Avoiding duplication

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

$$\begin{array}{rl} & \nabla_S[\![sqrt(x + (y * z))]\!] \\ = & \nabla sqrt(x + (y * z)) \circ \nabla_S[\![x + (y * z)]\!] \\ = & \nabla sqrt(x + (y * z)) \circ \nabla{+}(x, y * z) \\ & \circ ([\nabla_S[\![x]\!]; \nabla_S[\![y * z]\!]]) \\ = & \nabla sqrt(x + (y * z)) \circ \nabla{+}(x, y * z) \\ & \circ ([\nabla x; (\nabla{*}(y, z) \circ ([\nabla y; \nabla z]))]) \end{array}$$

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

TE: Yes, although when I say "AD" I mean something that is distinct from what I mean by "symbolic differentiation". In particular by "AD" I mean something that preserves sharing in a way that symbolic differentation doesn't. Perhaps between us we should pin down some terminology. SPJ: I don't understand this. Perhaps you can make it precise?

TE: Consider $exp(exp(x))$. I consider its "symbolic derivative" to be $exp(exp(x))exp(x)$ and its "forward automatic derivative" to be $let\ y = exp(x)\ in\ exp(y)y$. In other words, taking proper care of sharing is what makes AD AD and not just any old form of symbolic differentiation, in my personal nomenclature at least. Does that make it any clearer what I mean? AWF: For me, "AD" very specifically implies a second argument to the function. That's how you detect it's AD. I.e. $f'(x, dx) = ...$ is forward mode, and $f`(x, df) = ...$ is reverse mode. There's a lot of chat about what AD really is, and those who want to avoid such chat often now say "algorithmic differentiation", to mean "all this stuff". The real claim we want to explore is this: "Forward mode is good for functions with small inputs and large outputs, e.g. $\mathbb{R} \mapsto \mathbb{R}^n$, and reverse mode is for $\mathbb{R}^n \mapsto \mathbb{R}$.

3.3 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 $\nabla index$ are correct!

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

Direct path. Figure 5 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 5; there is no $\nabla build$. 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 $\mathcal{B}'$, but then I needed its transposition, so just using $\mathcal{B}'$ 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!

3.4 General folds

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

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

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

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

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.

4 Implementation

The implementation differs from this document as follows:

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

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

  This is treated as equivalent to

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

5 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 }
    ((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)

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