Composable Semantics for Language Synthesis

Boyan Beronov, Chris Chen

CPSC 509: Programming Language Principles
University of British Columbia, Vancouver

November 19, 2018
Motivation

What we wish, that we readily believe. (Demosthenes)
Motivation

*What we wish, that we readily believe.*  
*(Demosthenes)*

*Information is not knowledge.*  
*(Albert Einstein)*
### Motivation

<table>
<thead>
<tr>
<th>Motivational Quote</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>What we wish, that we readily believe.</td>
<td>Demosthenes</td>
</tr>
<tr>
<td>Information is not knowledge.</td>
<td>Albert Einstein</td>
</tr>
<tr>
<td>Everything is vague to a degree you do not realize till you have tried to make it precise.</td>
<td>Bertrand Russell</td>
</tr>
<tr>
<td>What we wish, that we readily believe.</td>
<td>(Demosthenes)</td>
</tr>
<tr>
<td>---------------------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>Information is not knowledge.</td>
<td>(Albert Einstein)</td>
</tr>
<tr>
<td>Everything is vague to a degree you do not realize till you have tried to make it precise.</td>
<td>(Bertrand Russell)</td>
</tr>
<tr>
<td>The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise.</td>
<td>(Edsger Dijkstra)</td>
</tr>
<tr>
<td>Motivation</td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td></td>
</tr>
<tr>
<td><strong>What we wish, that we readily believe.</strong>  (Demosthenes)</td>
<td></td>
</tr>
<tr>
<td><strong>Information is not knowledge.</strong>  (Albert Einstein)</td>
<td></td>
</tr>
<tr>
<td><strong>Everything is vague to a degree you do not realize till you have tried to make it precise.</strong>  (Bertrand Russell)</td>
<td></td>
</tr>
<tr>
<td><strong>The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise.</strong>  (Edsger Dijkstra)</td>
<td></td>
</tr>
<tr>
<td><strong>Abstraction is the elimination of the irrelevant and the amplification of the essential.</strong>  (Robert C. Martin)</td>
<td></td>
</tr>
</tbody>
</table>
## Motivation

<table>
<thead>
<tr>
<th>Quote</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>What we wish, that we readily believe.</td>
<td>(Demosthenes)</td>
</tr>
<tr>
<td>Information is not knowledge.</td>
<td>(Albert Einstein)</td>
</tr>
<tr>
<td>Everything is vague to a degree you do not realize till you have</td>
<td>(Bertrand Russell)</td>
</tr>
<tr>
<td>tried to make it precise.</td>
<td></td>
</tr>
<tr>
<td>The purpose of abstraction is not to be vague, but to create a new</td>
<td>(Edsger Dijkstra)</td>
</tr>
<tr>
<td>semantic level in which one can be absolutely precise.</td>
<td></td>
</tr>
<tr>
<td>Abstraction is the elimination of the irrelevant and the amplification</td>
<td>(Robert C. Martin)</td>
</tr>
<tr>
<td>of the essential.</td>
<td></td>
</tr>
<tr>
<td>Civilisation advances by extending the number of important operations</td>
<td>(Alfred North Whitehead)</td>
</tr>
</tbody>
</table>
Levels of abstraction
Marr’s tri-level hypothesis (cognitive science) [1]

<table>
<thead>
<tr>
<th>Computational theory</th>
<th>Representation and algorithm</th>
<th>Hardware implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>What is the goal of the computation, why is it appropriate, and what is the logic of the strategy by which it can be carried out?</td>
<td>How can this computational theory be implemented? In particular, what is the representation for the input and output, and what is the algorithm for the transformation?</td>
<td>How can the representation and algorithm be realized physically?</td>
</tr>
</tbody>
</table>

*Figure 1–4.* The three levels at which any machine carrying out an information-processing task must be understood.
Towards universal interfaces
within and across levels
Towards universal interfaces within and across levels

1. Denotational design
   - Standard vocabulary for abstractions: category theory
   - Denotational design: type class morphism principle
   - Overloading the interpretation: compiling into categories
   - Example: automatic differentiation

2. Component-based semantics
   - Formal specification: kinds of metalanguages
   - Issues: entanglement, evolvability, reusability
   - IR for semantics: fundamental language constructs
Goal:

Denotational design [2]
Denotative programming

Peter Landin recommended “denotative” to replace ill-defined “functional” and “declarative”.

Properties:

- Nested expression structure.
- Each expression denotes something,
- depending only on denotations of subexpressions.

“...gives us a test for whether the notation is genuinely functional or merely masquerading.” (The Next 700 Programming Languages, 1966)
Denotational design

Design methodology for “genuinely functional” programming:

- Precise, simple, and compelling specification.
- Informs *use* and *implementation* without entangling them.
- Standard algebraic abstractions.
- Free of abstraction leaks.
- Laws for free.
- Principled construction of correct implementation.
Automatic differentiation (Part 1):
The computation graph perspective [3]
Derivative

Function of a real variable \( f : \mathbb{R} \to \mathbb{R} \)

Sensitivity of function value w.r.t.
a change in its argument
(the instantaneous rate of change)

Dependent Independent

\[
\frac{dy}{dx} = f'(x) = \dot{y}
\]

\[
\frac{\Delta y}{\Delta x} \to \frac{f(x + \Delta x) - f(x)}{\Delta x} \quad \text{as} \quad \Delta x \to 0
\]

Newton, c. 1665

Leibniz, c. 1675
Derivative

Function of a real variable \( f : \mathbb{R} \to \mathbb{R} \)

General Formulas

1. \( \frac{d}{dx} c = 0 \)
2. \( \frac{d}{dx} [f(x) \pm g(x)] = f'(x) \pm g'(x) \)
3. \( \frac{d}{dx} [f(x)g(x)] = f'(x)g(x) + g'(x)f(x) \)
4. \( \frac{d}{dx} \left[ \frac{f(x)}{g(x)} \right] = \frac{g(x)f'(x) - f(x)g'(x)}{(g(x))^2} \)
5. \( \frac{d}{dx} [f(g(x))] = f'(g(x))g'(x) \)
6. \( \frac{d}{dx} x^n = nx^{n-1} \)

Exponential and Logarithmic Functions

7. \( \frac{d}{dx} e^{f(x)} = f'(x)e^{f(x)} \)
8. \( \frac{d}{dx} a^x = a^x \ln(a) \)
9. \( \frac{d}{dx} \ln(f(x)) = \frac{f'(x)}{f(x)} \)

…

around 15 such rules

Note: the derivative is a linear operator, a.k.a. a **higher-order function** in programming languages \( (\mathbb{R} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R}) \)
Derivatives as code

We can compute the derivatives not just of mathematical functions, but of general programs (with control flow)
Derivatives as code
Symbolic vs automatic differentiation

Often described as opposing techniques:

- **Symbolic:**
  - Apply differentiation rules symbolically.
  - Can duplicate much work.
  - Needs algebraic manipulation.

- **Automatic:**
  - FAD: easy to implement but often inefficient.
  - RAD: efficient but tricky to implement.

My view: *AD is SD done by a compiler.*

Compilers already work symbolically and preserve sharing.
Automatic differentiation

All numerical algorithms, when executed, evaluate to compositions of a finite set of elementary operations with known derivatives
- Called a trace or a Wengert list (Wengert, 1964)
- Alternatively represented as a computational graph showing dependencies

\[ f(a, b): \]
\[ c = a \times b \]
\[ d = \log(c) \]
\[ \text{return } d \]

1.791 \[= f(2, 3)\]
\[ [0.5, 0.333] \[= f'(2, 3) \]
\[ \nabla f(a, b) = (1/a, 1/b) \]
Automatic differentiation

Two main flavors

**Forward** mode

- **Primals**
- **Derivatives**
  (Tangents)

**Reverse** mode (a.k.a. backprop)

- **Primals**
- **Derivatives**
  (Adjoints)

**Nested combinations**
(higher-order derivatives, Hessian–vector products, etc.)
- Forward-on-reverse
- Reverse-on-forward
- ...

Forward mode

\[ f : \mathbb{R}^2 \rightarrow \mathbb{R}^2 \]

In general, forward mode evaluates a Jacobian–vector product \( J_f(x)v \)

So we evaluated:

\[
\begin{bmatrix}
\frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} \\
\frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2}
\end{bmatrix}
\begin{bmatrix}
1 \\
0
\end{bmatrix}
= \begin{bmatrix}
\frac{\partial y_1}{\partial x_1} \\
\frac{\partial y_2}{\partial x_1}
\end{bmatrix}
\]

Can be any \( v \in \mathbb{R}^2 \)
not only unit vectors
Reverse mode

\[ f : \mathbb{R}^2 \rightarrow \mathbb{R}^2 \]

In general, forward mode evaluates a transposed Jacobian–vector product

\[ J^T_f(x)v \]

So we evaluated:

\[
\begin{bmatrix}
\frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} \\
\frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2}
\end{bmatrix}^T
\begin{bmatrix}
1 \\
0
\end{bmatrix} =
\begin{bmatrix}
\frac{\partial y_1}{\partial x_1} \\
\frac{\partial y_1}{\partial x_2}
\end{bmatrix}
\]
Where does the graph come from?

Two main possibilities:

- **Static** computational graphs
  Let the user define the graph as a data structure
  “Define-and-run”

- **Dynamic** computational graphs
  Construct the graph automatically (general-purpose automatic differentiation)
  “Define-by-run”
What to implement

Two main parts

Computational graph
- **Dynamically build the graph**
  Side effect of forward evaluation or “non-standard interpretation”
- **Graph traversal algorithm**
  The API to kickstart the backpropagation: backward, grad, etc.

Derivatives
- **Rules of differentiation**
  For all elementary numerical operations

  Usually implemented on a custom numerical type, using operator overloading
Operator overloading on custom type

```
def eval_and_backprop(fun, x):
    y = fun(x)
    y.backprop(1.)
    return y, x._adjoint

class OpMul():
    def __init__(a, b):
        self.a = a
        self.b = b
    return a * b

def backprop(self, adjoint):
    self.a.backprop(adjoint * b)
    self.b.backprop(adjoint * a)

tape = []

class Number():
    def __init__(value):
        self.value = value
        self._adjoint = 0.

    def backprop(self, adjoint):
        self._adjoint += adjoint

    def __mul__(self, other):
        global tape
        op = OpMul(self, other)
        tape.append(OpMul(self, other))
        return Number(self * other)
```

Global tape (stack)
- Forward: push in the order of evaluation
- Reverse: pop in the reverse order
Correctly handle fan-out

Fan-out: when a node is involved in multiple subsequent operations
- Maintain a fan-out counter per node
- Don’t propagate backward from a node until all derivatives coming to that node have arrived

\[
\frac{\partial f}{\partial v_7} = \frac{\partial f}{\partial v_3} \frac{\partial v_3}{\partial v_7} + \frac{\partial f}{\partial v_8} \frac{\partial v_8}{\partial v_7}
\]
Current Landscape

Currently in progress: frameworks are in transition from 
coarse-grained (module level) backprop 
towards 

fine-grained, general-purpose automatic differentiation

torch7 2011
HIPS autograd 2014

torch-autograd 2015
TensorFlow 2015

PyTorch 2016
TensorFlow eager execution 2017
## DiffSharp

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( f : \mathbb{R} \rightarrow \mathbb{R} )</td>
<td>( f' )</td>
<td>( \mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R} )</td>
<td>X, F</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>diff</td>
<td>( f' )</td>
<td></td>
<td>X, F</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>diff'</td>
<td>( f'_f )</td>
<td>( \mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R} \times \mathbb{R} )</td>
<td>X, F</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>diff2</td>
<td>( f''_f )</td>
<td>( \mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R} )</td>
<td>X, F</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>diff2'</td>
<td>( f''_f )</td>
<td>( \mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R} \times \mathbb{R} )</td>
<td>X, F</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>diff3</td>
<td>( f'''_f )</td>
<td>( \mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R} )</td>
<td>X, F</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>diffn</td>
<td>( f^{(n)}_f )</td>
<td>( \mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R} )</td>
<td>X, F</td>
<td>A</td>
<td>X</td>
</tr>
</tbody>
</table>

\[ f : \mathbb{R}^n \rightarrow \mathbb{R} \]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>grad</td>
<td>( \nabla f )</td>
<td>( \mathbb{R}^n \rightarrow \mathbb{R} \rightarrow \mathbb{R}^n )</td>
<td>X, R</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>gradv</td>
<td>( \nabla f \cdot v )</td>
<td>( \mathbb{R}^n \rightarrow \mathbb{R} \rightarrow \mathbb{R} )</td>
<td>X, R</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>gradw</td>
<td>( \nabla f \cdot v )</td>
<td>( \mathbb{R}^n \rightarrow \mathbb{R} \rightarrow \mathbb{R} )</td>
<td>X, R</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>hessian</td>
<td>( \mathbf{H}_f )</td>
<td>( \mathbb{R}^n \rightarrow \mathbb{R}^n \rightarrow \mathbb{R}^{n \times n} )</td>
<td>X, R-F</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>hessianv</td>
<td>( \nabla \mathbf{H}_f )</td>
<td>( \mathbb{R}^n \rightarrow \mathbb{R}^n \rightarrow \mathbb{R}^{n \times n} )</td>
<td>X, R-F</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>gradhessian</td>
<td>( \nabla \nabla f, \mathbf{H}_f )</td>
<td>( \mathbb{R}^n \rightarrow \mathbb{R}^n \rightarrow \mathbb{R} )</td>
<td>X, R-F</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>gradhessianv</td>
<td>( \nabla \nabla f, \mathbf{H}_f )</td>
<td>( \mathbb{R}^n \rightarrow \mathbb{R}^n \rightarrow \mathbb{R}^{n \times n} )</td>
<td>X, R-F</td>
<td>A</td>
<td>X</td>
</tr>
</tbody>
</table>

\[ f : \mathbb{R}^m \rightarrow \mathbb{R}^m \]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>jacobian</td>
<td>( J_f )</td>
<td>( \mathbb{R}^m \rightarrow \mathbb{R}^m \rightarrow \mathbb{R}^{m \times m} )</td>
<td>X, F/R</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>jacobianv</td>
<td>( \nabla \mathbf{J}_f )</td>
<td>( \mathbb{R}^m \rightarrow \mathbb{R}^m \rightarrow \mathbb{R}^{m \times m} )</td>
<td>X, F/R</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>jacobiantv</td>
<td>( \nabla \mathbf{J}_f \cdot v )</td>
<td>( \mathbb{R}^m \rightarrow \mathbb{R}^m \rightarrow \mathbb{R} )</td>
<td>X, F/R</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>curl</td>
<td>( \nabla \times f )</td>
<td>( \mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R} )</td>
<td>X, F</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>div</td>
<td>( \nabla \cdot f )</td>
<td>( \mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R} )</td>
<td>X, F</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>curlcurl</td>
<td>( \nabla \times (\nabla \times f) )</td>
<td>( \mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R} )</td>
<td>X, F</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>curldiv</td>
<td>( \nabla \times (\nabla \cdot f) )</td>
<td>( \mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R} )</td>
<td>X, F</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>curlcurlv</td>
<td>( \nabla \times (\nabla \times f \cdot v) )</td>
<td>( \mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R} )</td>
<td>X, F</td>
<td>A</td>
<td>X</td>
</tr>
</tbody>
</table>

\[ f : \mathbb{R}^3 \rightarrow \mathbb{R}^3 \]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>curlcurlv</td>
<td>( \nabla \times (\nabla \times f \cdot \nabla \cdot f) )</td>
<td>( \mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R} )</td>
<td>X, F</td>
<td>A</td>
<td>X</td>
</tr>
</tbody>
</table>

\[ f : \mathbb{R}^2 \rightarrow \mathbb{R}^3 \]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>curlcurlv</td>
<td>( \nabla \times (\nabla \times f \cdot \nabla \cdot f) )</td>
<td>( \mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R} )</td>
<td>X, F</td>
<td>A</td>
<td>X</td>
</tr>
</tbody>
</table>
Standard vocabulary:

Algebra & category theory

Type class morphism (TCM) principle [2]
Using standard vocabulary

- We’ve created a domain-specific vocabulary.

- Can we reuse standard vocabularies instead?

- Why would we want to?
  - User knowledge.
  - Ecosystem support (multiplicative power).
  - Laws as sanity check.
  - Tao check.
  - Specification and laws for free, as we’ll see.

- In Haskell, standard type classes.
Example: linear transformations

Assignment:

- Represent linear transformations
- Scalar, non-scalar domain & range, identity and composition

Plan:

- Interface
- Denotation
- Representation
- Calculation (implementation)
Interface and denotation

**Interface:**

\[ \text{type} (\to) :: \star \to \star \to \star \]

\[ \text{scale} :: \text{Num} s \Rightarrow (s :\to s) \]

\[ \hat{id} :: a :\to a \]

\[ (\circ) :: (b :\to c) \to (a :\to b) \to (a :\to c) \]

...\\

**Model:**

\[ \text{type} a \to b \quad -- \text{Linear subset of} \ a \to b \]

\[ \mu :: (a :\to b) \to (a \to b) \]

\[ \mu (\text{scale } s) \equiv \lambda x \to s \times x \]

\[ \mu \hat{id} \equiv id \]

\[ \mu (g \circ f) \equiv \mu g \circ \mu f \]

...
Start with 1D. Recall partial specification:

$$\mu (\text{scale } s) \equiv \lambda x \to s \times x$$

Try a direct data type representation:

```haskell
data (\to) :: * \to * \to *  where
  Scale :: Num s \Rightarrow s \to (s \to s)  -- ...

\mu :: (a \to b) \to (a \to b)
\mu (Scale s) = \lambda x \to s \times x
```

Spec trivially satisfied by $scale = Scale$

Others are more interesting.
Calculate an implementation

Specification:

| \( \mu \widehat{id} \equiv id \) | \( \mu (g \circ f) \equiv \mu g \circ \mu f \) |

Calculation:

<table>
<thead>
<tr>
<th>( id )</th>
<th>( \mu (Scale s) \circ \mu (Scale s') )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \equiv \lambda x \to x )</td>
<td>( \equiv (\lambda x \to s \times x) \circ (\lambda x' \to s' \times x') )</td>
</tr>
<tr>
<td>( \equiv \lambda x \to 1 \times x )</td>
<td>( \equiv \lambda x' \to s \times (s' \times x') )</td>
</tr>
<tr>
<td>( \equiv \mu (Scale 1) )</td>
<td>( \equiv \lambda x' \to ((s \times s') \times x') )</td>
</tr>
<tr>
<td></td>
<td>( \equiv \mu (Scale (s \times s')) )</td>
</tr>
</tbody>
</table>

Sufficient definitions:

| \( \widehat{id} = Scale 1 \) | \( Scale s \circ Scale s' = Scale (s \times s') \) |
Algebraic abstraction

In general,

- Replace ad hoc vocabulary with a standard abstraction.
- Recast semantics as homomorphism.
- Note that laws hold.

What standard abstraction to use for (\(\to\))?
Monoid

Interface:

```haskell
class Monoid m where
    mempty :: m
              -- “mempty”
    mappend :: m -> m -> m
              -- “mappend”
```

Laws:

- \(a \oplus \text{mempty} \equiv a\)
- \(\text{mempty} \oplus b \equiv b\)
- \(a \oplus (b \oplus c) \equiv (a \oplus b) \oplus c\)

class Functor f where
    fmap :: (a → b) → (f a → f b)
**Applicative**

\[
\textbf{class} \quad \text{Functor } f \Rightarrow \text{Applicative } f \ \textbf{where} \\
\quad \text{pure} \quad :: \quad a \rightarrow f \ a \\
\quad (\langle\star\rangle) \quad :: \quad f \ (a \rightarrow b) \rightarrow f \ a \rightarrow f \ b
\]
Category

Interface:

```haskell
class Category k where
  id    :: k a a
  (○)   :: k b c → k a b → k a c
```

Laws:

```haskell
id ○ f        ≡ f
(g ○ id)      ≡ g
(h ○ g) ○ f   ≡ h ○ (g ○ f)
```
Linear transformation category

Linear map semantics:

\[ \mu :: (a \rightarrow b) \rightarrow (a \rightarrow b) \]
\[ \mu (\text{Scale } s) = \lambda x \rightarrow s \times x \]

Specification as homomorphism (no abstraction leak):

\[ \mu \ id \equiv id \]
\[ \mu (g \circ f) \equiv \mu g \circ \mu f \]

Correct-by-construction implementation:

```hs
instance Category (\rightarrow) where
  id = Scale 1
  Scale s \circ Scale s' = Scale (s \times s')
```
Laws for free

\[
\begin{align*}
\mu \ id & \equiv id \\
\mu \ (g \circ f) & \equiv \mu \ g \circ \mu \ f
\end{align*}
\]

\[\Rightarrow\]

\[
\begin{align*}
id \circ f & \equiv f \\
g \circ id & \equiv g \\
(h \circ g) \circ f & \equiv h \circ (g \circ f)
\end{align*}
\]

where equality is *semantic*. Proofs:

\[
\begin{align*}
\mu \ (id \circ f) & \equiv \mu \ id \circ \mu \ f \\
\mu \ (g \circ id) & \equiv \mu \ g \circ \mu \ id \\
\mu \ ((h \circ g) \circ f) & \equiv (\mu \ h \circ \mu \ g) \circ \mu \ f
\end{align*}
\]

\[
\begin{align*}
\equiv id \circ \mu \ f \\
\equiv \mu \ g \circ \mu \ id \\
\equiv \mu \ g \circ id \\
\equiv \mu \ g \\
\equiv \mu \ f
\end{align*}
\]

\[
\begin{align*}
\Rightarrow \\
\equiv \mu \ g \\
\equiv \mu \ f
\end{align*}
\]

Works for other classes as well.
Specifications for free

Semantic type class morphism (TCM) principle:

*The instance’s meaning follows the meaning’s instance.*

That is, the type acts like its meaning.

Every TCM failure is an abstraction leak.

Strong design principle.

Class laws necessarily hold, as we’ll see.
Overloading the interpretation:
Compiling into categories [4]
Overloading

- Alternative interpretation of common vocabulary.
- Laws for modular reasoning.
- Doesn’t apply to lambda, variables, and application.
- Instead, eliminate them.
Eliminating lambda

\[(\lambda p \to k) \quad \rightarrow \quad const \ k\]

\[(\lambda p \to p) \quad \rightarrow \quad id\]

\[(\lambda p \to u \ v) \quad \rightarrow \quad apply \circ ((\lambda p \to u) \triangle (\lambda p \to v))\]

\[(\lambda p \to \lambda q \to u) \quad \rightarrow \quad curry \ (\lambda (p, q) \to u)\]

\[\quad \rightarrow \quad curry \ (\lambda r \to u \ [p := fst \ r, q := snd \ r])\]

Automate via a compiler plugin.
Examples

\[
\text{sqr} :: \text{Num } a \Rightarrow a \rightarrow a \\
\text{sqr } a = a \times a
\]

\[
\text{magSqr} :: \text{Num } a \Rightarrow a \times a \rightarrow a \\
\text{magSqr } (a, b) = \text{sqr } a + \text{sqr } b
\]

\[
\text{cosSinProd} :: \text{Floating } a \Rightarrow a \times a \rightarrow a \times a \\
\text{cosSinProd } (x, y) = (\cos z, \sin z) \text{ where } z = x \times y
\]

After \( \lambda \)-elimination:

\[
\text{sqr} = \text{mulC} \circ (\text{id} \triangle \text{id})
\]

\[
\text{magSqr} = \text{addC} \circ (\text{mulC} \circ (\text{exl} \triangle \text{exl}) \triangle \text{mulC} \circ (\text{exr} \triangle \text{exr}))
\]

\[
\text{cosSinProd} = (\cos C \triangle \sin C) \circ \text{mulC}
\]
Abstract algebra for functions

Interface:

```haskell
class Category k where
  id :: a `k` a
  (○) :: (b `k` c) → (a `k` b) → (a `k` c)
infixr 9 ○
```

Laws:

```plaintext
id ○ f ≡ f
f ○ id ≡ f
(h ○ g) ○ f ≡ h ○ (g ○ f)
```
Products

Interface:

```haskell
class Category k ⇒ Cartesian k where

  type a ×ₖ b

  exl :: (a ×ₖ b) `k` a

  exr :: (a ×ₖ b) `k` b

  (△) :: (a `k` c) → (a `k` d) → (a `k` (c ×ₖ d))

  infixr 3 △
```

Laws:

```
exl ◦ (f △ g) ≡ f
exr ◦ (f △ g) ≡ g
exl ◦ h △ exr ◦ h ≡ h
```
Coproducts

Dual to product.

class Category k ⇒ Cocartesian k where

type a +_k b

inl :: a `k` (a +_k b)
inr :: b `k` (a +_k b)

(ν) :: (a `k` c) → (b `k` c) → ((a +_k b) `k` c)

infixr 2 ν

Laws:

(f ∨ g) ∘ inl ≡ f
(f ∨ g) ∘ inr ≡ g
h ∘ inl ∨ h ∘ inr ≡ h
Exponentials

First-class “functions” (morphisms):

\[
\textbf{class} \ \text{Cartesian} \ k \Rightarrow \text{CartesianClosed} \ k \ \textbf{where}
\]

\[
\begin{align*}
\text{type} & \quad a \Rightarrow_k b \\
\text{apply} & \quad :: \ ((a \Rightarrow_k b) \times_k a) \Rightarrow_k \ b \\
\text{curry} & \quad :: \ ((a \times_k b) \Rightarrow_k c) \rightarrow (a \Rightarrow_k (b \Rightarrow_k c)) \\
\text{uncurry} & \quad :: \ (a \Rightarrow_k (b \Rightarrow_k c)) \rightarrow ((a \times_k b) \Rightarrow_k c)
\end{align*}
\]

Laws:

\[
\begin{align*}
\text{uncurry} \ (\text{curry} \ f) & \quad \equiv f \\
\text{curry} \ (\text{uncurry} \ g) & \quad \equiv g \\
\text{apply} \circ (\text{curry} \ f \circ \text{exl} \triangle \text{exr}) & \quad \equiv f
\end{align*}
\]
Changing interpretations

- We’ve eliminated lambdas and variables

- and replaced them with an algebraic vocabulary.

- What happens if we replace \((\rightarrow)\) with other instances? (Via compiler plugin.)
\[ \text{magSqr} \ (a, b) = \text{sqr} \ a + \text{sqr} \ b \]

\[ \text{magSqr} = \text{addC} \circ (\text{mulC} \circ (\text{exl} \land \text{exl}) \land \text{mulC} \circ (\text{exr} \land \text{exr})) \]
newtype Graph a b = Graph (Ports a -> GraphM (Ports b))

type GraphM = State (PortNum, [Comp])

data Comp = forall a b. Comp (Template a b) (Ports a) (Ports b)

data Template :: * -> * -> * where
  Prim :: String -> Template a b
  Subgraph :: Graph a b -> Template () (a -> b)

instance Category Graph where
  id = Graph return
  Graph g `o` Graph f = Graph (g <=< f)

instance BoolCat Graph where
  notC = genComp "¬"
  andC = genComp "∧"
  orC = genComp "∨"
Interval analysis

\textbf{data} \textit{IFun} \textit{a} \textit{b} = \textit{IFun} (\textit{Interval} \textit{a} \rightarrow \textit{Interval} \textit{b})

\textbf{type family} \textit{Interval} \textit{a}  
\textbf{type instance} \textit{Interval} \textit{Double} = \textit{Double} \times \textit{Double}  
\textbf{type instance} \textit{Interval} \textit{(a \times b)} = \textit{Interval} \textit{a} \times \textit{Interval} \textit{b}  
\textbf{type instance} \textit{Interval} \textit{(a \rightarrow b)} = \textit{Interval} \textit{a} \rightarrow \textit{Interval} \textit{b}

\textbf{instance} \textit{Category \textit{IFun}} \textit{where}  
\textit{id} = \textit{IFun} \textit{id}  
\textit{IFun} \textit{g} \circ \textit{IFun} \textit{f} = \textit{IFun} \textit{(g} \circ \textit{f)}

\textbf{instance} \textit{Cartesian \textit{IFun}} \textit{where}  
\textit{exl} = \textit{IFun} \textit{exl}  
\textit{exr} = \textit{IFun} \textit{exr}  
\textit{IFun} \textit{f} \triangle \textit{IFun} \textit{g} = \textit{IFun} \textit{(f} \triangle \textit{g)}

\textbf{instance} \textit{(Interval} \textit{a} \sim (\textit{a \times a)}, \textit{Num} \textit{a}, \textit{Ord} \textit{a}) \Rightarrow \textit{NumCat} \textit{IFun} \textit{a} \textit{where}  
\textit{addC} = \textit{IFun} (\lambda((\textit{a}_{lo}, \textit{a}_{hi}),(\textit{b}_{lo}, \textit{b}_{hi})) \rightarrow (\textit{a}_{lo} + \textit{b}_{lo}, \textit{a}_{hi} + \textit{b}_{hi}))  
\textit{mulC} = \textit{IFun} (\lambda((\textit{a}_{lo}, \textit{a}_{hi}),(\textit{b}_{lo}, \textit{b}_{hi})) \rightarrow  
\textit{minmax} [\textit{a}_{lo} \times \textit{b}_{lo}, \textit{a}_{lo} \times \textit{b}_{hi}, \textit{a}_{hi} \times \textit{b}_{lo}, \textit{a}_{hi} \times \textit{b}_{hi}]  

\textbf{...}
Interval analysis — example

\[ \lambda(x, y) \rightarrow x + 3 \times y \]
Other examples

- Constraint solving via SMT (with John Wiegley)
- Linear maps
- Incremental evaluation
- Polynomials
- Nondeterministic and probabilistic programming
Shallow embedding

- “Just a library”, but with a suitable host language.
- Easy to implement; but restricts optimization.
- Inherits host language & compiler limitations, e.g., no
  - differentiation or integration
  - incremental evaluation
  - optimization
  - constraint solving
  - novel back-ends, e.g., GPU, circuits, JavaScript
Deep embedding

- Syntactic representation.
- More room for analysis and optimization.
- Harder to implement; redundant with host compiler.
- Requires some vocabulary changes.
Compiling to categories

- Just a library.
- Easy to implement.
- Analysis, optimization, non-standard target architectures.
- Non-standard operations on functions.
Automatic differentiation (Part 2):
The categorical perspective [5]
Current AI revolution runs on large data, speed, and AD, but

- AD algorithm (backprop) is complex and stateful.
- Graph APIs are complex and semantically dubious.

Solutions in this paper:

- AD: Simple, calculated, efficient, parallel-friendly, generalized.
- API: derivative.
What’s a derivative?

\[ \mathcal{D} :: (a \to b) \to (a \to (a \to b)) \]

A local linear (affine) approximation:

\[
\lim_{\varepsilon \to 0} \frac{\|f(a + \varepsilon) - (f a + \mathcal{D} f a \varepsilon)\|}{\|\varepsilon\|} = 0
\]

See *Calculus on Manifolds* by Michael Spivak.
Composition

Sequential:

$((\circ) :: (b \to c) \to (a \to b) \to (a \to c))$

$(g \circ f) \ a = g \ (f \ a)$

$\mathcal{D} (g \circ f) \ a = \mathcal{D} g \ (f \ a) \circ \mathcal{D} f \ a \quad -- \ chain \ rule$

Parallel:

$((\triangle) :: (a \to c) \to (a \to d) \to (a \to c \times d))$

$(f \triangle g) \ a = (f \ a, g \ a)$

$\mathcal{D} (f \triangle g) \ a = \mathcal{D} f \ a \triangle \mathcal{D} g \ a$
Linear functions

Linear functions are their own derivatives everywhere.

\[ \mathcal{D} \text{id} \ a = \text{id} \]
\[ \mathcal{D} \text{fst} \ a = \text{fst} \]
\[ \mathcal{D} \text{snd} \ a = \text{snd} \]

...
Compositionality

Chain rule:

\[ \mathcal{D} (g \circ f) \ a = \mathcal{D} g (f \ a) \circ \mathcal{D} f \ a \quad -- \text{non-compositional} \]

To fix, combine regular result with derivative:

\[ \hat{\mathcal{D}} :: (a \to b) \to (a \to (b \times (a \to b))) \]
\[ \hat{\mathcal{D}} f = f \triangle \mathcal{D} f \quad -- \text{specification} \]

Often much work in common to \( f \) and \( \mathcal{D} f \).
Automatic differentiation

\textbf{newtype} \( D \ a \ b = D \ (a \to b \times (a \to b)) \)

\( \hat{D} :: (a \to b) \to D \ a \ b \)

\( \hat{D} \ f = D \ (f \triangle \hat{D} \ f) \) \quad -- \text{not computable} 

Specification: \( \hat{D} \) preserves \textit{Category} and \textit{Cartesian} structure:

\( \hat{D} \ id = id \)

\( \hat{D} \ (g \circ f) = \hat{D} \ g \circ \hat{D} \ f \)

\( \hat{D} \ exl = exl \)

\( \hat{D} \ exr = exr \)

\( \hat{D} \ (f \triangle g) = \hat{D} \ f \triangle \hat{D} \ g \)

\textit{The game}: solve these equations for the RHS operations.
Solution: simple automatic differentiation

```haskell
newtype D a b = D (a → b × (a → b))

linearD f = D (λa → (f a, f))

instance Category D where
  id = linearD id
  D g ⪯ D f = D (λa → let \((b, f') = f a; (c, g') = g b\) in \(c, g' ⪯ f'\))

instance Cartesian D where
  exl = linearD exl
  expr = linearD expr
  D f ⪰ D g = D (λa → let \((b, f') = f a; (c, g') = g a\) in \((b, c), f' ⪰ g'\))

instance NumCat D where
  negate = linearD negate
  add = linearD add
  mul = D (mul ⪰ (λ(a, b) → λ(da, db) → b * da + a * db))
```
AD example

\[ \text{cosSinProd} \ (x, y) = (\cos \ z, \sin \ z) \text{ where } z = x \ast y \]

\[ \text{cosSinProd} = (\cos \ 	riangle \ \sin) \circ \text{mul} \]
Generalizing AD

newtype \( D \ a \ b = D \ (a \to b \times (a \to b)) \)

\( \text{linearD} \ f = D \ (\lambda a \to (f \ a, f)) \)

instance Category \( D \) where
  \( \text{id} = \text{linearD} \ \text{id} \)
  \( D \ g \odot D \ f = D \ (\lambda a \to \text{let} \ ((b, f') = f \ a; (c, g') = g \ b) \ \text{in} \ (c, g' \circ f')) \)

instance Cartesian \( D \) where
  \( \text{exl} = \text{linearD} \ \text{exl} \)
  \( \text{exr} = \text{linearD} \ \text{exr} \)
  \( D \ f \triangle D \ g = D \ (\lambda a \to \text{let} \ ((b, f') = f \ a; (c, g') = g \ a) \ \text{in} \ ((b, c), f' \triangle g')) \)

Each \( D \) operation just uses corresponding \((\to\circ)\) operation.

Generalize from \((\to)\) to other cartesian categories.
Generalized AD

\textbf{newtype} \(D_{(\sim)}\) \(a\ b = D\ (a \rightarrow b \times (a \sim b))\)

\textit{linearD} \(f\ f' = D\ (\lambda a \rightarrow (f\ a, f'))\)

\textbf{instance} \(\text{Category}\ \(\sim\) \Rightarrow \text{Category}\ D_{(\sim)}\ \text{where}\)
\begin{align*}
\text{id} &= \text{linearD}\ \text{id}\ \text{id} \\
D\ g \circ D\ f &= D\ (\lambda a \rightarrow \text{let} \ \{(b, f') = f\ a; (c, g') = g\ b\} \ \text{in} \ (c, g' \circ f'))
\end{align*}

\textbf{instance} \(\text{Cartesian}\ \(\sim\) \Rightarrow \text{Cartesian}\ D_{(\sim)}\ \text{where}\)
\begin{align*}
\text{exl} &= \text{linearD}\ \text{exl}\ \text{exl} \\
\text{exr} &= \text{linearD}\ \text{exr}\ \text{exr} \\
D\ f \triangle D\ g &= D\ (\lambda a \rightarrow \text{let} \ \{(b, f') = f\ a; (c, g') = g\ a\} \ \text{in} \ ((b, c), f' \triangle g'))
\end{align*}

\textbf{instance} \ldots \Rightarrow \text{NumCat}\ D\ \text{where}\)
\begin{align*}
\text{negate} &= \text{linearD}\ \text{negate}\ \text{negate} \\
\text{add} &= \text{linearD}\ \text{add}\ \text{add} \\
\text{mul} &= ??
\end{align*}
Numeric operations

Specific to (linear) functions:

\[ \text{mul} = D (\text{mul} \triangle (\lambda(a, b) \rightarrow \lambda(da, db) \rightarrow b \ast da + a \ast db)) \]

Rephrase:

\[ \text{scale} :: \text{Multiplicative } a \Rightarrow a \rightarrow (a \circ a) \]
\[ \text{scale } u = \lambda v \rightarrow u \ast v \]
\[ (\triangledown) :: (a \circ c) \rightarrow (b \circ c) \rightarrow ((a \times b) \circ c) \]
\[ f \triangledown g = \lambda(a, b) \rightarrow f \ a + g \ b \]

Now

\[ \text{mul} = D (\text{mul} \triangle (\lambda(a, b) \rightarrow \text{scale } b \triangledown \text{scale } a)) \]
Linear arrow (biproduct) vocabulary

**class** Category \((\sim)\) **where**

- \(id :: a \sim a\)
- \((\circ) :: (b \sim c) \to (a \sim b) \to (a \sim c)\)

**class** Category \((\sim)\) ⇒ Cartesian \((\sim)\) **where**

- \(exl :: (a \times b) \sim a\)
- \(exr :: (a \times b) \sim b\)
- \((\triangle) :: (a \sim c) \to (a \sim d) \to (a \sim (c \times d))\)

**class** Category \((\sim)\) ⇒ Cocartesian \((\sim)\) **where**

- \(inl :: a \sim (a \times b)\)
- \(inr :: b \sim (a \times b)\)
- \((\triangledown) :: (a \sim c) \to (b \sim c) \to ((a \times b) \sim c)\)

**class** ScalarCat \((\sim)\) a **where**

- \(scale :: a \to (a \sim a)\)
Linear transformations as functions

newtype \( a \rightarrow^+ b = \text{AddFun} \ (a \rightarrow b) \)

instance \( \text{Category} \ (\rightarrow^+) \) where
\[
\text{id} = \text{AddFun} \ \text{id} \\
(\circ) = \text{inNew}_2 \ (\circ)
\]

instance \( \text{Cartesian} \ (\rightarrow^+) \) where
\[
\text{exl} = \text{AddFun} \ \text{exl} \\
\text{exr} = \text{AddFun} \ \text{exr} \\
(\triangle) = \text{inNew}_2 \ (\triangle)
\]

instance \( \text{Cocartesian} \ (\rightarrow^+) \) where
\[
\text{inl} = \text{AddFun} \ (, 0) \\
\text{inr} = \text{AddFun} \ (0,) \\
(\nabla) = \text{inNew}_2 \ (\lambda f \ g \ (x, y) \rightarrow f \ x + g \ y)
\]

instance \( \text{Multiplicative} \ s \Rightarrow \text{ScalarCat} \ (\rightarrow^+) \ s \) where
\[
\text{scale} \ s = \text{AddFun} \ (s \ \ast)
\]
Extracting a data representation

- How to extract a matrix or gradient vector?

- Sample over a domain *basis* (rows of identity matrix).

- For *n*-dimensional *domain*,
  - Make *n* passes.
  - Each pass works on *n*-D sparse (“one-hot”) input.
  - Very inefficient.

- For gradient-based optimization,
  - High-dimensional domain.
  - Very low-dimensional (1-D) codomain.
Generalized matrices

\[
\text{newtype } M_s a b = L (V_s b (V_s a s))
\]

\[
applyL :: M_s a b \rightarrow (a \rightarrow b)
\]

Require \(applyL\) to preserve structure. Solve for methods.
Core vocabulary

Sufficient to build arbitrary “matrices”:

\[
\text{scale} :: \ a \rightarrow (\ a \leadsto a) \\
(\nabla) \ :: (a \leadsto c) \rightarrow (b \leadsto c) \rightarrow ((a \times b) \leadsto c) \quad -- \text{horizontal juxt} \\
(\triangle) \ :: (a \leadsto c) \rightarrow (a \leadsto d) \rightarrow (a \leadsto (c \times d)) \quad -- \text{vertical juxt}
\]

Types guarantee rectangularity.
Efficiency of composition

- Arrow composition is associative.

- Some associations are more efficient than others, so
  - Associate optimally.
  - Equivalent to matrix chain multiplication — $O(n \log n)$.
  - Choice determined by types, i.e., compile-time information.

- All-right: “forward mode AD” (FAD).

- All-left: “reverse mode AD” (RAD).

- RAD is much better for gradient-based optimization.
Left-associating composition (RAD)

- CPS-like category:
  - Represent $a \rightsquigarrow b$ by $(b \rightsquigarrow r) \rightarrow (a \rightsquigarrow r)$.
  - Meaning: $f \mapsto (\circ f)$.
  - Results in left-composition.
  - Initialize with $id :: r \rightsquigarrow r$.
  - Construct $h \circ \mathcal{D} f a$ directly, without $\mathcal{D} f a$.

- We’ve seen this trick before:
  - Transforming naive reverse from quadratic to linear.
  - List generalizes to monoids, and monoids to categories.
Continuation category

newtype $\text{Cont}^r_{(\sim \rightarrow)} a b = \text{Cont} \ ((b \sim r) \rightarrow (a \sim r))$

$$\text{cont} :: \text{Category} \ (\sim \rightarrow) \Rightarrow (a \sim b) \rightarrow \text{Cont}^r_{(\sim \rightarrow)} a b$$
$$\text{cont} f = \text{Cont} \ (\circ f)$$

Require $\text{cont}$ to preserve structure. Solve for methods.

We’ll use an isomorphism:

$$\text{join} :: \text{Cocartesian} \ (\sim \rightarrow) \Rightarrow (c \sim a) \times (d \sim a) \rightarrow ((c \times d) \sim a)$$
$$\text{unjoin} :: \text{Cocartesian} \ (\sim \rightarrow) \Rightarrow ((c \times d) \sim a) \rightarrow (c \sim a) \times (d \sim a)$$

$$\text{join} \ (f, g) = f \uplus g$$
$$\text{unjoin} \ h \ = \ (h \circ \text{inl}, h \circ \text{inr})$$
instance Category \( (\sim) \Rightarrow \text{Category } \text{Cont}_{(\sim)}^r \) where
\[
\begin{align*}
id &= \text{Cont } id \\
\text{Cont } g \circ \text{Cont } f &= \text{Cont } (f \circ g)
\end{align*}
\]

instance Cartesian \( (\sim) \Rightarrow \text{Cartesian } \text{Cont}_{(\sim)}^r \) where
\[
\begin{align*}
exl &= \text{Cont } (\text{join } \circ \text{inl}) \\
exr &= \text{Cont } (\text{join } \circ \text{inr}) \\
(\triangle) &= \text{inNew}_2 (\lambda f \ g \rightarrow (f \triangle g) \circ \text{unjoin})
\end{align*}
\]

instance Cocartesian \( (\sim) \Rightarrow \text{Cocartesian } \text{Cont}_{(\sim)}^r \) where
\[
\begin{align*}
inl &= \text{Cont } (\text{exl } \circ \text{unjoin}) \\
inr &= \text{Cont } (\text{exr } \circ \text{unjoin}) \\
(\triangledown) &= \text{inNew}_2 (\lambda f \ g \rightarrow \text{join } \circ (f \triangledown g))
\end{align*}
\]

instance ScalarCat \( (\sim) \ a \Rightarrow \text{ScalarCat } \text{Cont}_{(\sim)}^r \ a \) where
\[
\begin{align*}
scale \ s &= \text{Cont } (\text{scale } s)
\end{align*}
\]
Reverse-mode AD without tears

\[ D_{\text{Cont}^r_{M_S}} \]
Vector space dual: $u \to s$, with $u$ a vector space over $s$.

If $u$ has finite dimension, then $u \to s \cong u$.

For $f :: u \to s$, $f = \text{dot } v$ for some $v :: u$.

Gradients are derivatives of functions with scalar codomain.

Represent $a \to b$ by $(b \to s) \to (a \to s)$ by $b \to a$.

Ideal for extracting gradient vector. Just apply to $1$ ($id$).
newtype $Dual_{(\sim)} a b = Dual (b \sim a)$

$asDual :: Cont^{s}_{(\sim)} a b \rightarrow Dual_{(\sim)} a b$

$asDual (Cont f) = Dual (dot^{-1} \circ f \circ dot)$

where

$dot :: u \rightarrow (u \rightarrow s)$

$dot^{-1} :: (u \rightarrow s) \rightarrow u$

Require $asDual$ to preserve structure. Solve for methods.
**Duality (solution)**

```haskell
newtype Dual (~) a b = Dual (b ~> a)

instance Category (~>) ⇒ Category Dual(~) where
  id   = Dual id
  (⊙) = inNew2 (flip (⊙))

instance Cocartesian (~>) ⇒ Cartesian Dual(~) where
  exl  = Dual inl
  exr  = Dual inr
  (∆) = inNew2 (∇)

instance Cartesian (~>) ⇒ Cocartesian Dual(~) where
  inl  = Dual exl
  inr  = Dual exr
  (∇) = inNew2 (∆)

instance ScalarCat (~>) s ⇒ ScalarCat Dual(~) s where
  scale s = Dual (scale s)
```
Backpropagation

\[ D_{Dual_{\rightarrow^t}} \]
RAD example (dual vector)
RAD example (dual function)
RAD example (dual vector)
RAD example (dual function)
RAD example (matrix)
What happened?

But it’s the same thing we had before!
What happened?

*But it’s the same thing we had before!*

Yes, but:

- Constructed from general-purpose components
- Without a language embedding
- Using only the **host language compiler**

By improving the level of abstraction, we achieve:

- **Explicit & computable** (algebraic) statement of essential structure
- **Minimal** specification & implementation
- **Reduction** of accidental complexity
- Code **reusability** for related tasks
- **Interoperability** with other interpretations
Conclusions

- Simple AD algorithm, specializing to forward, reverse, mixed.
- No graphs, tapes, tags, partial derivatives, or mutation.
- Parallel-friendly and low memory use.
- Calculated from simple, regular algebra problems.
- Generalizes to derivative categories other than linear maps.
- Differentiate regular Haskell code (via plugin).
- More details in my ICFP 2018 paper.
Reflections: recipe for success

Key principles:

- Capture main concepts as first-class values.
- Focus on abstract notions, not specific representations.
- Calculate efficient implementation from simple specification.

Not previously applied to AD (afaik).

Quandary: Most programming languages poor for function-like things.

Solution: Compiling to categories.
Component-based semantics
Let’s talk about metalanguages

Inspired by Peter D Mosses’s 2017 SLE Keynote [6]

How do we formally specify languages? We can use...

1. BNF to specify **Syntax**.
2. Type systems to specify **Static Semantics**.
3. Operational, denotational, and other semantic systems to specify **Dynamic Semantics**.

We will concern ourselves mostly with dynamic semantics for this talk.
Given a formal semantics ⇒
Because all is right with the world

We can reason about the *meaning* of our programs.

1. Is some implementation equivalent to some specification (given any preconditions)?

2. But... the analysis comes at a cost: our theorems are only valid for the particular language in question.
What if you *don’t* have a formal semantics?

Because reasons

Chances are, the language you’re using doesn’t have a formal specification, and if it did, it’s

1. the result of an archaeological expedition or “autopsy” (Mosses)
2. incomplete
   *In this paper, we formally specify a subset of...*
3. out of date
   *In this paper, we show a $\rightarrow$ between the set of graduate students and the set of formal specifications of subsets of...*
What reasons?

With regards to operational semantics

Semantics are defined in terms of the abstract syntax, so

1. As a language evolves, maintainers need to update an ever more complex set of semantic rules (entanglement)

2. It’s often impossible to share semantic rules between two languages
An example

In big-step SOS

\[
\frac{(\lambda x.t_2) \; t_1 \Downarrow v}{\text{let } x = t_1 \text{ in } t_2 \Downarrow v}
\]

\[
\left(\text{let}\sigma\right) \quad \frac{\rho \vdash \langle (\lambda x.t_2) \; t_1, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}{\rho \vdash \langle \text{let } x = t_1 \text{ in } t_2, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}
\]

As our languages grow in complexity, so too do our rule definitions. We would like a way to separate out *incidental* from *essential* complexity. Moreover, what if we had *clean* rules that were composable?
Denotational and Hybrid Approaches

And monadic, and...

In denotational continuation-passing style:

\[ E : Expr \rightarrow Env \rightarrow K \rightarrow C \]

\[ E[\text{let } x = e_1 \text{ in } e_2]_\rho K = E[e_1]_\rho (\lambda v. E[e_2]_\rho([v/x])_\kappa) \]

In Action Semantics, a hybrid approach where semantic entities are partitioned into *actions, data, and yielders*:

\[ \text{evaluate } \_ :: \ Exp \rightarrow \]

\[ \text{action [giving a value] [using current bindings]} \]

\[ \text{evaluate } [\text{“let” } x:Id “=” e_1:Exp “in” e_2:Exp] = \]

furthermore (evaluate e_1 then bind x to the given value)

hence evaluate e_2
In 2005, Iversen and Mosses [7] published their work on developing a Constructive Action Semantics for a core subset of ML. Their approach was to define a Basic Abstract Syntax of language features, and translate ML constructs down to BAS. BAS itself was specified in Action Semantics.

\[
\text{exp2bas} :: \text{Expression} \rightarrow \text{BAS} \\
\text{exp2bas}(\text{let } x = e_1 \text{ in } e_2) = \\
\quad \text{local}(\text{bind-val}(\text{val}(x), \text{exp2bas}(e_1)), \text{exp2bas}(e_2))
\]
In 2009, Mosses [8] proposed that rather than providing a set of abstract language features, one could construct simpler, readily composable language constructs built on top of either Action Semantics or Modular Structural Operational Semantics [9], a variant formulated by Mosses in 1999.

These *fundamental language constructs* are referred to as *funcons*. 
Layered specifications
From Peter D Mosses's 2017 SLE Keynote [6]
Specifying a Language Component in CBS

1. Syntax

\[
E : exp ::= \ldots | \text{'let' } id \text{'='} exp \text{'in' } exp | \ldots
\]

2. Semantics

\[
\text{eval}[: \text{exp}] \Rightarrow \text{values}
\]

3. Rules

\[
\text{eval}[\text{'let' } x \text{'='} e_1 \text{'in' } e_2] = \text{scope}(\text{bind}(x, \text{eval}[e_1], \text{eval}[e_2]))
\]

In this example, \textit{scope} and \textit{bind} are funcons.
Funcons [10]

Funcons are a library of language constructs, partitioned into values and computations.

1. **Primitive values**: none, atoms, bools, ints...
2. **Composite values**: algebraic datatypes, tuples, lists, pointers, references...
3. **Abstract values**: closures, functions...
4. **Normal computation**: control flow, sequential ordering, binding...
5. **Abnormal computation**: failing, throwing...

Funcons are intended to follow the *UNIX Philosophy* of small, well defined, composable elements.
The CBS framework is available (in \( \beta \)) at https://plancomps.github.io/CBS-beta/

The IDE is built on top of Spoofax/Eclipse for

1. Editing
2. Browsing
3. Interpreting Programs

There is extensive documentation.
In Conclusion

1. Currently, to provide a *formal dynamic semantics* for a language, one has to go all the way down to sets.

2. *Component-based semantics* is intended to make the process of specifying a language easier, by meeting us halfway.

3. But more than that; it provides an exciting *target for analysis*, because theorems that hold for funcons also hold for any languages that use those funcons. *We can tear down the silos that separate us.*
References


