New language features

Charisee Chiw

March 28, 2018

Our work has made it easier and faster to add to the Diderot language. When we add an operator to the language we try to leverage our existing work. This includes a concise representation in an expressive IR, the generic implementation of operators, and a robust testing model. In the following, we illustrate the process of extending Diderot by providing examples.

1 Design and Implementation

1.1 Composition

```
field#k(d0)[σ] F0;
field#k(d1)[d0] F1;
field#k(d1)[σ] H = F0 ∘ F1;
tensor[d1] pos;
tensor[σ] out = H(pos);
```

Representation We represent the field composition operator with the generic EIN operator.

$$\underset{init}{\Longrightarrow} H = \lambda(F,G)\Big\langle F_\alpha \circ [\langle G_\beta \rangle_{\hat\beta}] \Big\rangle_{\hat\alpha}(F0,F1) \qquad \text{where } \hat\alpha = \sigma \text{ and } \hat\beta = [d0]$$

The field terms F and G represent fields in the composition. F and G have separate index spaces. F is bound by α and G is bound by β.

Normalization The probe of a composition $(e_1 \circ [\langle e_2 \rangle_{\hat\beta}])(x)$ is rewritten depending on the structure of the outer term e_1. When the outer term is a constant the result does not depend on the composition operation.

$$(c \circ e_c)(x) \xrightarrow[rule]{} c$$

Similarily, when the outer term is a non-field:

$$\begin{array}{llll} (\delta_\alpha \circ e_c)(x) & \xrightarrow[rule]{} \delta_\alpha & (\mathcal{E}_\alpha \circ e_c)(x) & \xrightarrow[rule]{} \mathcal{E}_\alpha \\ (Z_\alpha \circ e_c)(x) & \xrightarrow[rule]{} Z_\alpha & (\mathbf{lift}_d(e) \circ e_c)(x) & \xrightarrow[rule]{} e \end{array}$$

The probe operator is pushed past arithmetic operators:

$$\begin{array}{ll} (\odot_1 e \circ e_c)(x) & \xrightarrow[rule]{} \odot_1\,(e \circ e_c)(x) \\ (\sum_{\hat\alpha} e \circ e_c)(x) & \xrightarrow[rule]{} \sum_{\hat\alpha}(e \circ e_c)(x) \end{array}$$

The probe is distributed:

$$\begin{array}{ll} ((e_a - e_b) \circ e_c)(x) & \xrightarrow[rule]{} (e_a \circ e_c)(x) - (e_b \circ e_c)(x) \\ ((e_a * e_b * e_s) \circ e_c)(x) & \xrightarrow[rule]{} (e_a \circ e_c)(x) * (e_b \circ e_c)(x) * (e_s \circ e_c)(x) \end{array}$$

The derivative of a field composition is applied by using the chain rule.

$$\nabla(F \circ G) \longrightarrow_{direct-style} (\nabla F \circ G) \bullet (\nabla G)$$

The derivative of a field composition of two fields is represented in the EIN IR as

$$\nabla_j(F_\alpha \circ [\langle G_{i\beta} \rangle_{\hat{i\beta}}]) \xrightarrow[rule]{} \sum_{\hat k}((\nabla_k F_\alpha \circ [\langle G_{i\beta} \rangle_{\hat{i\beta}}]) * (\nabla_j G_{k\beta}))$$

Generally we use the rewrite rule to apply the rewrite between two EIN expressions:

$$\nabla_j(e_1 \circ [\langle e_2\rangle_{\hat{i\beta}}]) \xrightarrow[rule]{} \sum_{\hat{k}}((\nabla_k e_1 \circ [\langle e_2\rangle_{\hat{i\beta}}]) * (\nabla_j e_{2[i/k]}))$$

Flatten composition operator

$$(a \circ [\langle b\rangle_{\hat{m}}]) \circ e_c \xrightarrow[rule]{} a \circ [\langle b\rangle_{\hat{m}}, e_c]$$

$$a \circ [\langle b \circ e_c\rangle_{\hat{m}}] \xrightarrow[rule]{} a \circ [\langle b\rangle_{\hat{m}}, e_c]$$

Split After being normalized the probed composition operator is split into several probes.

$$\text{out} = \lambda F, G, x\Big\langle F_\alpha \circ [\langle G_\beta\rangle_{\hat{\beta}}](x)\Big\rangle_\alpha (F0, F1, x) \underset{split}{\Longrightarrow} \begin{array}{rl} \text{t}_0 = & \lambda G, x\langle G_\beta(x)\rangle_{\hat{\beta}}(F1, x) \\ \text{out} = & \lambda F, x\langle F_\alpha(x)\rangle_\alpha(F0, \text{t}_0) \end{array}$$

1.2 Concat

A user can define new tensors by concatenating tensors together. A Diderot program

```
tensor[d1]S;
tensor[d2]T;
tensor[d1,d2]M = [S,T];
```

A user can refer to components of tensor fields by using the slice operation as shown in the following code. A Diderot program

```
field#k(d)[d1,d2]A;
field#k(d)[d1,d2]B;
field#k(d)[d1]F = A[:,0];
field#k(d)[d1]G = B[:,1];
```

We would like to provide a way to define new tensors fields by concatenating components together. Using the tensor field variables F and G defined earlier in the program the Diderot code should support the line

```
field#k(d)[d1,d1]H = [F, G];
```

We illustrate the structure of H below.

$$H = \begin{bmatrix} F_0 & F_1 \\ G_0 & G_1 \end{bmatrix}$$

Representation We can use EIN expressions as building blocks to represent field concatenation. In EIN each field term is represented by an expression and it is enabled with a delta function

$$\underset{init}{\Longrightarrow} H = \lambda F, G.\langle F_j\delta_{0i} + G_j\delta_{1i}\rangle_{i:2,j:2}(\text{F,G})$$

After substitution the new EIN operator would be

$$\underset{subst}{\Longrightarrow} H = \lambda A, B.\langle A_{j0}\delta_{0i} + B_{j1}\delta_{1i}\rangle_{\hat{i}\hat{j}}(\text{A,B}).$$

In the compiler we choose to create generic versions of an EIN operator that can be instantiated to certain types.

$$\begin{array}{l} \lambda F, G.\langle F_\alpha\delta_{0i} + G_\alpha\delta_{1i}\rangle_{i:2\hat{\alpha}}(\text{F,G}) \\ \lambda F, G, H.\langle F_\alpha\delta_{0i} + G_\alpha\delta_{1i} + H_\alpha\delta_{2i}\rangle_{i:3\hat{\alpha}}(\text{F,G,H}) \end{array}$$

To implement this operator we need to add to cases to the Diderot typechecker and add the generic representations but not much else. Since we are solely using existing EIN expressions to represent this computation, we can rely on the existing code to handle the EIN syntax.

1.3 Abs, Max, and Min

Implementation lift max and min operators to the field level and represents the magnitude of scalar fields with absolute expressions. Extra syntax is created to support differentiation.

Design Need to add new syntax to support differentiation.

cond	=	$e > e \mid e < e$	conditional
e	=	$\text{Max}(a,b) \mid \text{Min}(a,b)$	Binary EIN operators
	$\mid$	$\text{if}(cond, e, e)$	If wrapper returns tensor-valued expression
	$\mid$	$\text{Abs}(e)$	Absolute function
	$\mid$	$\text{Sgn}(e)$	Returns Sign (-1, 0, 1)

Differentiation rules Differentiation of an absolute expression:

$$\frac{\partial}{\partial x_\alpha}\text{abs}(e) \rightarrow (\frac{\partial}{\partial x_\alpha}e) * (\text{Sgn}(e))$$

Differentiation creates an if wrapper expressions.

$$\frac{\partial}{\partial x_\alpha}\text{Max}(a,b) \rightarrow if(a > b, \frac{\partial}{\partial x_\alpha}a, \frac{\partial}{\partial x_\alpha}b)$$

$$\frac{\partial}{\partial x_\alpha}\text{Min}(a,b) \rightarrow if(a < b, \frac{\partial}{\partial x_\alpha}a, \frac{\partial}{\partial x_\alpha}b)$$

Differentiation of an If wrapper is pushed to leaves.

$$\frac{\partial}{\partial x_\alpha}\text{If}(cond, c, d) \rightarrow if(cond, \frac{\partial}{\partial x_\alpha}c, \frac{\partial}{\partial x_\alpha}d)$$

other rules Otherwise, Max and Min are treated like other binary operators. The following pushes the probes to the leaves.

$$(\text{Max}(a,b))(x) \rightarrow \text{Max}(a(x), b(x))$$

1.4 Defining fields with polynomials

It is natural to define a field with a polynomial expression.

$$F = x^3$$

In the surface language we added function poly(). The first argument is a variable and the second argument is a field definition.

```
vec2 x;
ofield#2(2)[2] V = poly(x, x);
ofield#2(2)[] S = poly(x, x²);
```

This allows the programmer to differentiate this type of field.

```
ofield#1(2)[2] GS = ∇S;
```

We illustrate the expected structure below:

$$V = \begin{bmatrix} V_0 \\ V_1 \end{bmatrix} \qquad \nabla \otimes V = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad S = V_0^2 + V_1^2 \qquad \nabla S = \begin{bmatrix} 2 * V_0 \\ 2 * V_1 \end{bmatrix}$$

Representation .
As an ongoing example: A field F is defined by taking the cube of the input variable.

```
real p;
ofield#1(2)[] F = poly(p, p³);
tensor[] out = F(pos);
```

$$\underset{init}{\Longrightarrow} \begin{array}{l} \text{F} = \lambda()\langle PolyWrap_p(p^3)\rangle() \\ \text{out} = \lambda(F,x)\langle F(x)\rangle(\text{F,x}) \end{array}$$

Substitution creates:
$\underset{subst}{\Longrightarrow}$ out=$\lambda(x)\langle PolyWrap_p(p^3)(x)\rangle$ (x)

Replace polynomial variable .
The variable p represents a vector of length 2, where p= [X,Y].
The term P_0 represents the 0th component of the vector, or X. In the following, we will use the terms X and Y, in place of P_0 and P_1.

The polynomial variable is instantiated with the position.
$\Rightarrow$ out=$\lambda(p)\langle e\rangle$(x) where e=$p*p*p$.
The EIN term (p) is replaced with an EIN term that represents the vector components. In the 2-d case there are two terms indexed with a constant index in $p = X\delta_{0i} + Y\delta_{1i}$

Occurrances for P are replaced inside the expression:
$\rightarrow (P_0\delta_{0i} + P_1\delta_{1i}) * (P_0\delta_{0i} + P_1\delta_{1i}) * (P_0\delta_{0i} + P_1\delta_{1i})$.
$= (X\delta_{0i} + Y\delta_{1i}) * (X\delta_{0i} + Y\delta_{1i}) * (X\delta_{0i} + Y\delta_{1i})$.

Normalization .
Similar terms are collected:

$$P_0 * P_0 \rightarrow {P_0}^2 \quad \text{or} \quad X * X \rightarrow X^2$$

The differentiation operator is distributed over the EIN term, as usual, and pushed to a polynomial term

$$\frac{\partial}{\partial x_i}({P_0}^2 + e) \rightarrow \frac{\partial}{\partial x_i}{P_0}^2 + \frac{\partial}{\partial x_i}e \quad \text{or} \quad \frac{\partial}{\partial x_i}(X^2 + e) \rightarrow \frac{\partial}{\partial x_i}X^2 + \frac{\partial}{\partial x_i}e$$

Evaluation .
During the evaluation the variable index in a differentiation operator is bound to a number. An EIN term such as $\frac{\partial}{\partial x_i}{P_c}^n$ is evaluated.
When i and c are both 0:

$$\frac{\partial}{\partial x_0}X \rightarrow 1 \quad \frac{\partial}{\partial x_0}X^2 \rightarrow 2*X \quad \frac{\partial}{\partial x_0}X^3 \rightarrow 3*X^2$$

When i and c are both 1:

$$\frac{\partial}{\partial x_1}Y \rightarrow 1 \quad \frac{\partial}{\partial x_1}Y^2 \rightarrow 2*Y \quad \frac{\partial}{\partial x_1}Y^3 \rightarrow 3*Y^2$$

When i and c are not the same

$$\frac{\partial}{\partial x_1}X^n \rightarrow 0 \quad \frac{\partial}{\partial x_1}Y^n \rightarrow 0$$

2 Testing results

XT1 One was in the creation of the EIN operator for the concat operator. The indices on the terms in the concat EIN operator were switched.

XT2 The bug arose when computing the determinant of the concatenation of a field.

```
field#k(d)[2]F,G;
field#k(d)[]H = det(concat(F,G));
```

The bug was caused by the rewriting system. Our rewriting system applies index-based rewrites to reduce EIN expressions. A specific index rewrite is applied when the index in the delta term matches an index in tensor (or field) term. The rewrite checked if two indices were equal and did not distinguish between variable and constant indices. It is mathematically incorrect to reduce constant indices, because they are not equivalent to variable indices.

Comp-B1 Mistake in index scope when using substitution.

```
field#k(2)[2,2] F0;
field#k(2)[2] F1;
field#k(2)[2] F2;
field#k(2)[2,2] G = (F0 ∘ F1) • F2;
```

There was a mistake in the substitution method. The scope of the composition indices were handled incorrectly. The following is the expected and observed representation of the computation in the EIN IR.
Expected: $e = \sum_{\hat{j}} A_{ij} \circ [\langle B_i \rangle_{\hat{\beta}}] * C_j$
Observed: $e = \sum_{\hat{k}} A_{ik} \circ [\langle B_i \rangle_{\hat{\beta}}] * C_j$
in $\lambda(A, B, C)\langle e \rangle_{\beta}$(F0,F1,F2).

Comp-B2 Missing cases in split method.
Probes of a composition are handled differently before reconstruction.
$\sum F(x)$ and $\sum(\text{Comp}(F, G, -))(x)$.
Missing case in method leads to a compile time error. Additionally (Comp(Comp -)-)

Comp-B3 Differentiate a composition
The jacobian of a field composition:

```
field#k(d1)[d]  F0;
field#k(d)[d1]  F1;
field#k(d1)[d,d1]  G = ∇⊗  (F0  ∘  F1);
```

is represented as $\langle \nabla_j(\text{Comp}(A_i, B_i, i)) \rangle_{ij}$
In accordance with the chain rule ((f ∘ g)' = (f' ∘ g) · g') the rewriting system multiplies the inner derivative (g') with a new composition operation (f' ∘ g). In practice, the implementation does a point-wise multiplication when it should do an inner product.
Expected: $\sum_{\hat{k}}(\nabla_k A_i \circ [\langle B_i \rangle_{\hat{\beta}}] * \nabla_j G_k)$
Observed: $\nabla_j A_{\beta} \circ [\langle B_i \rangle_{\hat{i}}] * (\nabla_j G_i)$
in $\lambda(A, B)\langle e \rangle_{ij}$(F0,F1).

XT6 Using If Wrapper with other field operators

```
field #4(2)[]G = compose(minF((F0),(F1)),(F2*0.1));
```

Field operators need to be applied to the leaves in if wrapper. The composition is a field operator so it needed to be pushed to the leaves.

$$(\text{If}(c, e3, e4)) \circ es \rightarrow \text{If}(c, e3 \circ es, e4 \circ es)$$