Implementing Tensor Calc functions in Diderot

Charisee Chiw

March 31, 2018

1 Overview

Shorthand for code .
Path = https://github.com/cchiw
Exs = *Path*/latte/
DATm = *Path*/DATm
Diderot_Dev = *Path*/Diderot-Dev
Vis15 = vis15

Shorthand for Text .
[Doc] = *Exs*/writeup/paper.pdf
[dissertation] = Chiw'17 dissertation
[AST paper] = Chiw'17 ICSE-AST paper

2 New and extended syntax

2.1 Matrix Inverse

Functionality: Matrix Inverse
Syntax: "inv()"
field#k(d)[i, j] $\rightarrow$ field#k(d)[i, j]
tensor[i, j] $\rightarrow$ tensor[i, j]
Branch: *Diderot_Dev*
Text: Design chapter in [dissertation]
Issues: None
Examples: Check out programs in *Exs*/fn_matrixInverse
DATm command: python3 cte.py 1 8

2.2 Composition

2.2.1 Overview

Functionality: Field Composition
Syntax: "compose" and "$\circ$"
field#k(d_1)[α]$\times$ field#k(d_0)[d_1] $\rightarrow$ field#k(d_1)[α]
Branch: *Diderot_Dev*
Text: EIN IR design, rewriting rules, and resolved bugs listed in [Doc]
Issues: none
Examples: Check out programs in *Exs*/fn_composition

- *function name syntax* can use "compose" (B*/observ.diderot)
- *unicode syntax* can use "$\circ$"(X1/observ.diderot,*Exs*/fn_composition/X2/t.diderot)
- *chains composition operator* can apply composition operator to other operators and to itself (X2/*.diderot)
- *solved bugs* Copies of the programs with solved bugs are in *Path*/B*

DATm command: python3 cte.py 1 36

2.2.2 Design and Implementation

```
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.

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

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

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

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

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

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.

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

2.2.3 Testing results

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).

DATm Command: python3 cte.py 4 36 17 10 13

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 -)-)
label:-

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).

2.3 Clerp and Clamp

Functionality: Clerp. Clamp and Lerp all in one
Syntax: "clerp()"
tensor$[i]$ × tensor$[i]$ × real → tensor$[i]$
tensor$[i]$ × tensor$[i]$ × real× real× real → tensor$[i]$
Branch: *Diderot_Dev* & *Vis15*
Text: none
Issues: none
Examples: Check out *Exs*/fn_clerp/clerp3.diderot

Functionality: Apply clamp to arbituary-sized tensors
Syntax: "clamp()"
tty = tensor$[\alpha]$
tty × tty × tty → tty
Branch: *Diderot_Dev* & *Vis15*
Text: none
Issues: none
Examples: Check out *Exs*/fn_clerp/clamp.diderot

2.4 Swap and Selection

Functionality: Field assignment based on integer selector
Syntax: "swap()"

fty = field#k(d)[α]
int × fty × fty → fty
int × fty × fty× .... → fty
int × fty × fty × fty × fty × fty × fty → fty
Selection-id The first argument is an integer that serves to select a field. *i.e.* id=1 chooses the first field argument
Field Arguments The function accepts 2-6 field arguments.

Branch: *Diderot_Dev*
Text: none
Issues: Selection id is clamped when outside range (*i.e.* id=-6 → id=1) instead of throwing an error
Examples: Check out programs in *Exs*/fn_selection

- *Set Selection id* in Update method (X2)
- *Set Selection id* in Global Initialization (X1)

Future work: Hard-coded code generation needs to be more general to support n field arguments

2.5 Concatenation

2.5.1 Overview

Functionality: Concatenation
Syntax: "concat()"

field#k(d)[α] × field#k(d)[α] → field#k(d)[2,α]

Branch: *Diderot_Dev*
Text: Future Work chapter in [dissertation]. Details provided in [Doc]
Issues: None
Examples: Check out programs in *Exs*/fn_concatenation
DATm command: python3 cte.py 1 37
Future work Use syntax "[","]"

2.5.2 Design and Implementation

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.

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

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.

2.5.3 Testing results

concat-B1 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.

concat-B2 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.

2.6 Max and Min

2.6.1 Overview

Functionality: Maximal and Minimum between fields
Syntax: MaxF(), MinF()
    fty = field#k(d)[α]
    fty $\times$ fty $\rightarrow$ fty
Branch: *Diderot_Dev*
Text: none
Issues: syntax "Max()" instead of "MaxF()
Examples: Check out programs in *Exs*/fn_min_max

2.6.2 Design and Implementation

Design Need to add new syntax to support differentiation of max and min.

cond	=	$e > e \mid e < e$	conditional
e	=	Max(a, b) | Min(a, b)	Binary EIN operators
	|	if($cond, e, e$)	If wrapper returns tensor-valued expression
	|	Abs(e)	Absolute function
	|	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.

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

2.6.3 Testing results

maxmix_B1 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.

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

2.7 Other

Unsupported functionality:
- radial basis functions
- absolute value
- sign (positive/negative)
- if statement