Efficient compilation of complex tensor algebra expressions

Martin Sandve Alnæs
Center for Biomedical Computing

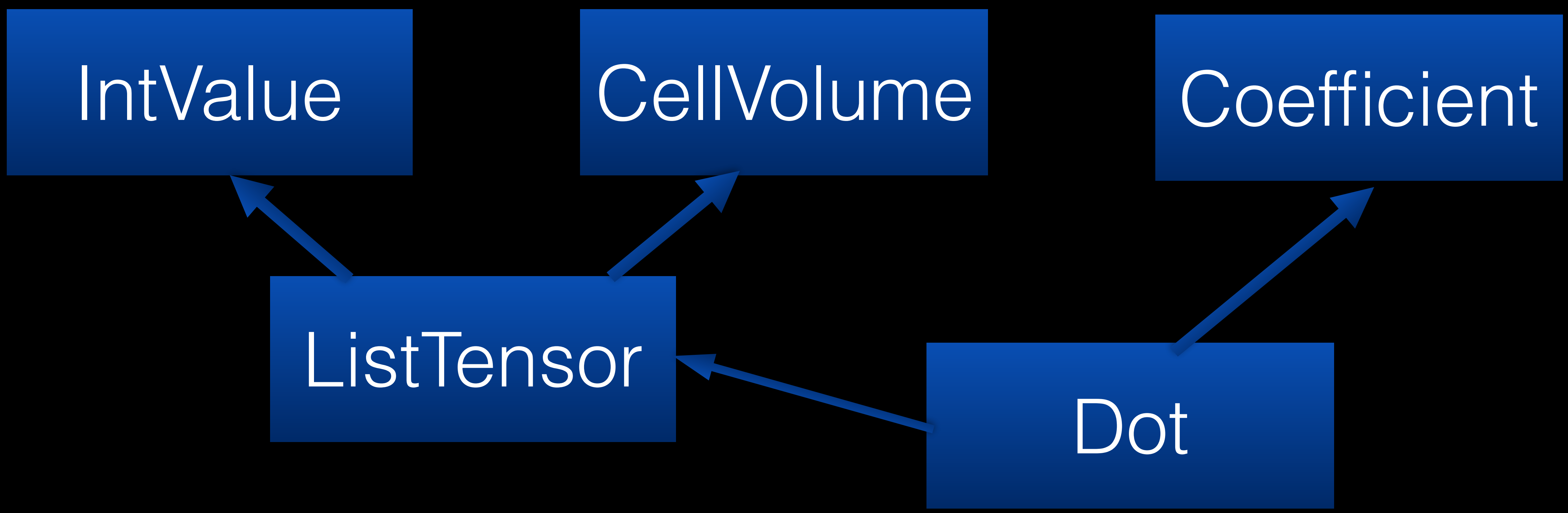
[simula . research laboratory]
- by thinking constantly about it

June 5th, FEniCS 2012

UFL is a DSL, symbolic backend, and frontend to form compilers

- Users recently reported scalabiliy problems when compiling complicated equations
- After some profiling sessions I reduced the memory usage by a factor 10 for one case
- Next I have made an attempt at faster form compiler algorithms, which I will show

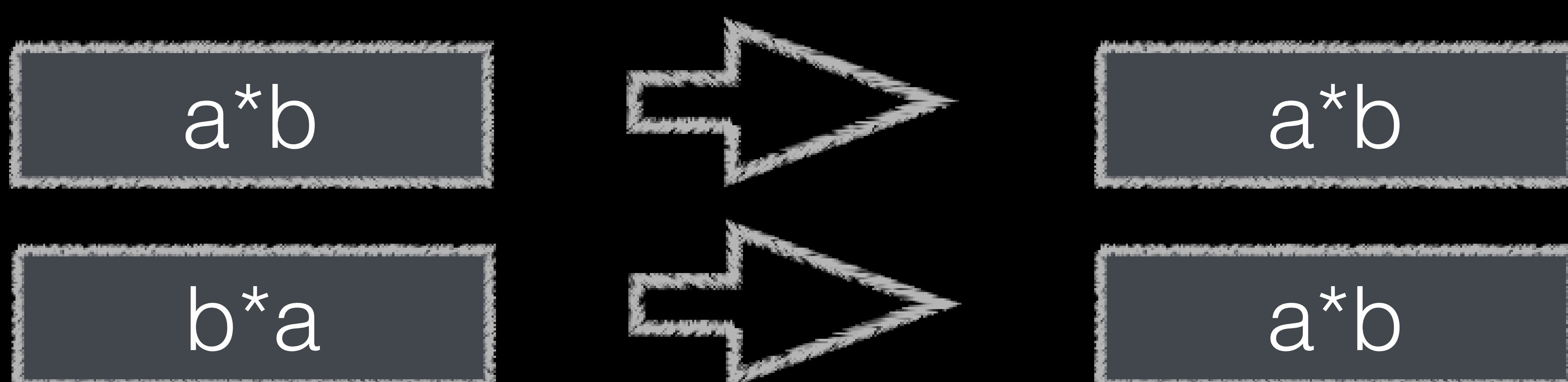
UFL expressions are represented in a symbolic expression tree



e = dot(as_vector((1, cell.volume)),
Coefficient(V))

Some quick design points

- Expr objects are immutable for easy sharing
- Conservative approach to automatic simplifications
- Canonical ordering of sum and product:



UFL simplifies some expressions on construction

1*f	→	f
0*f	→	0
0 + f	→	f
op(c1)	→	c2

Simplifications critical during differentiation algorithm

as_tensor(A[i,j], (i,j)) → A

- d/dx(x * g(y)) = 1 * g + x * 0 -> g

Performance must scale as O(n) with size of expression

- This means almost anything must be O(1)
- In particular __eq__ and __hash__!

Transformations must be safe for floating point computations

- Def eps: 1 + eps > 1
- (1 + eps/2) + eps/2 == 1
- 1 + (eps/2 + eps/2) > 1

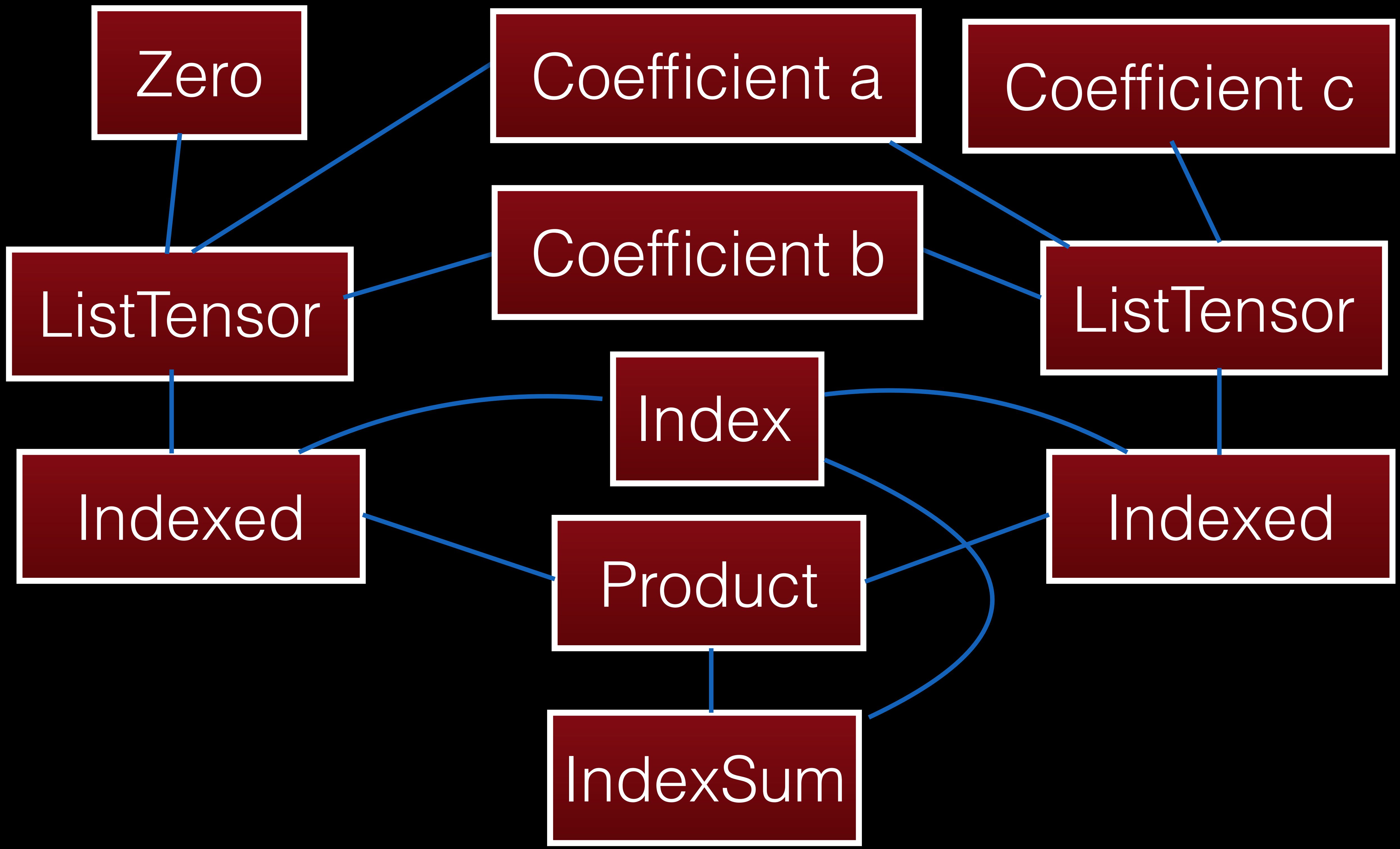
I will take this expression through the compiler algorithms

```
a, b, c = scalar coefficients
u = as_vector((0, a, b))
v = as_vector((c, b, a))
e = dot(u, v)
```

Anticipate result:

```
t = a*b
e = t + t
```

The expression tree after translating dot to index notation



Placing nodes in array

Index	V[i]	Shape	Size
0	0	() + ()	1
1	a	() + ()	1
2	b	() + ()	1
3	c	() + ()	1
4	<0,a,b>	(3,) + ()	3
5	<c,b,a>	(3,) + ()	3
6	u[i]	() + (3,)	3
7	v[i]	() + (3,)	3
8	u[i]*v[i]	() + (3,)	3
9	ISum(V[8],i)	() + ()	1

Scalar subexpressions are assigned unique value numbers

Index	V[i]	Value number
0	0	0
1	a	1
2	b	2
3	c	3
4	<0,a,b>	0,1,2
5	<c,b,a>	3,2,1
6	u[i]	0,1,2
7	v[i]	3,2,1
8	u[i]*v[i]	4,5,6
9	ISum(V[8],i)	7

Scalar subexpressions are reevaluated and placed in a new array

Index	S[i]	Simplifies to
0	0	
1	a	
2	b	
3	c	
4	S[0]*S[3]	0*c = 0
5	S[0]*S[3]	a*b
6	S[0]*S[3]	b*a = a*b
7	S[4]+S[5]+S[6]	a*b + a*b

Throwing away the array only keeping the final expression

```
a*b + a*b
```

Placing nodes in array!

Index	V[i]	Shape	Size
0	a	() + ()	1
1	b	() + ()	1
2	a*b	() + ()	1
3	a*b + a*b	() + ()	1

Analyzing dependencies

Index	V[i]	Dep.	Rev. Dep.
0	a	()	(2,)
1	b	()	(2,)
2	a*b	(0,1)	(3,3)
3	a*b + a*b	(2,2)	()

Final steps in compiler

- Partition final array by dependencies on x,u,v
- Heuristically pick best candidates for subexpressions to place in intermediate variables in generated code
- Format expressions and assignment statements within nested loops
- FEM library specific code generation in separate plugin class, e.g. how to evaluate geometry and coefficients, how to loop over quadrature points and basis functions

Outlook

- bzr branch lp:uflacs
- Can generate dolfin::Expression classes
- Soon SFC can use uflacs to compile forms
- Want to merge algorithms into FFC
- Write plugin class to compile to other FEM libs