Type Arithmetic for Tensor Typing

Alfonso L. Castaño

13th May, 2021

State of tensor typing

Variadics

```
def create_tensor(shape: Ts) -> Tensor[Ts]
def matmul(x: Tensor[A,B], y: Tensor[B,C]) -> Tensor[A,C]
t1 = create_tensor(2,3) # Tensor[2,3]
t2 = create_tensor(3,4) # Tensor[3,4]
t3 = matmul(t1,t2) # Tensor[2,4]
x,_ = t1.shape # x : Literal[2]
```

Notation

- `Tensor[2]` represents `Tensor[Literal[2]]`
- Ts represents a variadic

Use cases: Basic transformations

```
def concat(t1: Tensor[N], t2: Tensor[M]) -> Tensor[N + M]
def duplicate(t: Tensor[N]) -> Tensor[N * 2]
def split(t: Tensor[N], start: A, step: S) -> Tensor[(N-A)//S]

a : Tensor[3]
b : Tensor[2]
c : Tensor[15]

c = concat(a,b) # Tensor[5]
d = duplicate(a) # Tensor[6]
e = split(c,0,3) # Tensor[5]
```

Use cases: Complex transformations

```
def conv2d(t: Tensor[N, C_in, H_in, W_in])
    -> Tensor[
        N,
        C_out,
        1 + ((H_in + 2 * P - D * (K - 1) - 1) // S),
        1 + ((W_in + 2 * P - D * (K - 1) - 1) // S)
    ]: ...

x: Tensor[20,16,50,100]
conv = nn.Conv2d(16, 33, 3, stride=2)
y = conv(x) # Tensor[20, 33, 24, 49]
```

Use cases: Fixed size containers

```
def append(t: FixedList[N], element : Any) -> Tensor[N + 1]
def _getitem_(t: FixedList[N], start: A) -> Tensor[N - A]

l4: FixedList[4]
l5 = append(l4, "foo") # FixedList[5]
l3 = l5[2:] # FixedList[3]
```

Use cases: Arithmetic on literals

The foundation of type arithmetic

```
x : Literal[2] = 2
y : Literal[3] = 3
xy : Literal[6] = x*y

def f(x : X, y : Y) -> X*Y:
    return x*y

x = f(2,3) # x : Literal[5]
def f(x : X, x : Y) -> Tensor[X*Y]:
    size = x*y # size : X*Y
    return np.create((size))
```

Out of scope

Dependent Typing

```
x : Literal[5]
arr : List[???] (forall e in arr, e == 3)
y = x + arr[0] # Literal[8]
```

Refinement types

```
def remove(x: Tensor[N], k : <=N)
x : Tensor[2]
x.pop(3) # Error
```

Const generics / constexpr

```
x : Literal[5]
y = const_eval_hash(x) # Literal[458305]
```

Key Ideas

Focus on basic operators +, -, *, //

Add integrated support inside the type system

Working with them should be as simple as possible

Equality

Equality matters

Basic arithmetic properties are not obvious for the type checker.

```
def fn(a : Tensor[A], b : Tensor[B]):
```

```
c1 = concat(a,b) # T[A+B]
c2 = concat(b,a) # T[B+A]
c1 + c2 # A+B != B+A

c1 = concat(a,a) # T[A+A]
c2 = duplicate(a) # T[2*A]
c1 + c2 # A+A != 2*A
```

```
c1 = duplicate(a) # T[2*A]
c2 = half(c1) # T[(2*A)//2]
a + c2 # A != (2*A)//2

a1 = a.append(1).pop() # T[A+1-1]
a + a1 # A != A+1-1
```

Equivalence of arithmetic expressions

Canonical representation: Expressions are normalized so equivalent expressions have the same representation. For example:

```
(x+1)(x-1)              -> ... -> -1 + x^2
(5x-4x)+x*(x-1)-1       -> ... -> -1 + x^2
```

Automatic reasoning on arithmetic properties

The type checker, not the programmer, should take care of equality checks.

Avoid theorem proving or dependent typing

Type arithmetic should not add any overhead to the programmer.

More details in previous presentation: Link

Fraction normalization

GCD on multivariate polynomials is complex and expensive.

```
(x^2-1)/x+1 -> (x+1)(x-1)/x-1 -> x+1
```

For simplicity we could limit ourselves to the intersection of the unknown terms (x,y,z) and the GCD of the constant part.:

(**2x**y + **4x**z)/(**6x** + **4x**^{2}) -> (y + 2z)/(3 + 2x)

Domain of type arithmetic

Where is arithmetic defined?

```
A = ???

class Tensor(Generic[A]): ...
```

Where is arithmetic defined?

```
A = IntVar('A') # Literal[...| -1 | 0 | 1 | 2 |...]

class Tensor(Generic[A]): ...
```

Where is arithmetic defined?

```
A = IntVar('A') # Literal[...| -1 | 0 | 1 | 2 |...]

class Tensor2D(Generic[A,B]): ...

x : Tensor2D[???,5] = read_csv('...', ncols=5)
```

Where is arithmetic defined?

```
A = IntVar('A') # Literal[...| -1 | 0 | 1 | 2 |...]

class Tensor2D(Generic[A,B]): ...

x : Tensor2D[A,5] = read_csv('...', ncols=5)
```

Where is arithmetic defined?

```
A = IntVar('A') # Literal[...| -1 | 0 | 1 | 2 |...]

class Tensor2D(Generic[A,B]): ...

x : Tensor2D[A,5] = read_csv('...', ncols=5)

# Error: Unbound type variable A
```

Where is arithmetic defined?

```
A = TypeVar('A',bound=AnyNum) # AnyNum: Literal[...| -1 | 0 | 1 | 2 |...]

class Tensor2D(Generic[A,B]): ...

x : Tensor2D[AnyNum,5] = read_csv('...', ncols=5)
```

Where is arithmetic defined?

```
A = TypeVar('A', bound=int)

class Tensor2D(Generic[A,B]): ...

x : Tensor2D[int, 5] = read_csv('...', ncols=5)
```

Where is arithmetic defined?

```
A = TypeVar('A', bound=int)

class Tensor2D(Generic[A,B]): ...

x : Tensor2D[int, 5] = read_csv('...', ncols=5)
```

Int + ?	Int * ?	Int // ?
------------------------	------------------------	------------------------
Int	Int	Int

Potential extensions: Product & Length

```python
def flatten(t: Tensor[Shape]) -> Tensor[Prod[Shape]]
def _len_(t: Tuple[Ts]) -> Len[Ts]

x : Tensor[1,2,4,1]
x.flatten() # Tensor[8]
len(x) # Literal[4]
```

Potential extensions: Product & Length

```
def flatten(t: Tensor[Shape]) -> Tensor[Prod[Shape]]
def _len_(t: Tuple[Ts]) -> Len[Ts]

x : Tensor[1,2,4,1]
x.flatten() # Tensor[8]
len(x) # Literal[4]

def argmin(x : Tensor[Ts1,A,Ts2], axis:Len[Ts1])
argmin(x, 1) # Tensor[1,4,1]

def flatten2(x: Tensor[Ts1,Ts2], axis: Len[Ts2]) -> Tensor[Ts1,Prod[Ts2]]
flatten2(x,3) # Tensor[1,8]
```

Potential extension: Broadcast

```
def broadcast(x: Tensor[Ts1], y: Tensor[Ts2]) -> Tensor[BC[Ts1,Ts2]]
x : Tensor[  1,2]
y : Tensor[4,3,1]
broadcast(x,y) # Tensor[4,3,2]
```

Recap

- Type arithmetic is particularly useful for working with n-dim matrices and for making literals more useful.
- Built in support for basic arithmetic operators (+, -, *, //).
- Provide automatic equality checking on those operators.
- Use “int” to represent unknown numbers.
- Potentially extensible to other operators.

Powerful & Simple type arithmetic is possible

Thanks!

Semantics

Literal[x] + Literal[y]

Literal[x + y]

Literal[x] * Literal[y]

Literal[x * y]

Literal[x] // Literal[y]

Literal[x // y]

Any + ?

Any

Any * ?

Any

Any // ?

Any

Int + ?

Int

Int * ?

Int

Int // ?

Int

Canonical Representation

Equivalent expressions must have the same representation

Addition & Multiplication

Expression: `(x+1)(x-1)`

Expand -> `x^2 + x - x - 1`

Group -> `x^2 + 0x - 1`

Erase zeros -> `x^2 - 1`

Total order -> `-1 + x^2`

Expression: `x*(x-1) -1 + (5x-4x)`

Expand -> `x^2 -x -1 + 5x -4x`

Group -> `x^2 -1 + 0x`

Erase zeros -> `x^2 - 1`

Total order -> `-1 + x^2`

Addition & Multiplication

Expression: `(x+1)(x-1)`

Expand -> `x^2 + x - x - 1`

Group -> `x^2 + 0x - 1`

Erase zeros -> `x^2 - 1`

Total order -> `-1 + x^2`

Expression: `x*(x-1) -1 + (5x-4x)`

Expand -> `x^2 -x -1 + 5x -4x`

Group -> `x^2 -1 + 0x`

Erase zeros -> `x^2 - 1`

Total order -> `-1 + x^2`

Any expression with addition and multiplication can be normalized to an ordered list of monomials.

Integer Division: What is valid?

```
N//2 + N//2 ?= N
```

```
N//2 * N//8 ?= N^2//16
```

```
N//2 + N//2 ?= 2 * (N//2)
```

```
(2 * N)//2 ?= N
```

Integer Division: (A//B) * B != A

```
N//2 + N//2 != N
1//2 + 1//2 != 1
```

```
N//2 * N//8 != N^2//16
7//2 * 7//8 != 49//16
```

```
N//2 + N//2 == 2 * (N//2)
1//2 + 1//2 == 2 * (1//2)
```

```
(2 * N)//2 == N
(2 * 1)//2 == 1
```

Reduce Operators on Variadics

Unary operators, can be abstracted as a variable

```
2 + Length[Ts] ->
    y = normalize Length[Ts];
    2 + y
```

Requires extraction of type variables

```
Length[A,Ts] -> 1 + Length[Ts]
Prod[A,Ts] ->  A * Prod[Ts]
```

Current state of Type Arithmetic

Right now available in Pyre:

Type arithmetic operators: *Add*, *Multiply*, *Divide*

Reduce operators on variadics: *Product*, *Length*

Automatic equality checking of arithmetic expressions

Implemented but requires standardization:

Arithmetic on literals and type variables

Implemented but requires further research:

Type level Broadcasting