Towards Structure-Aware Neural Representations of Agda Programs

Konstantinos Kogkalidis, Orestis Melkonian, Jean-Philippe Bernardy

Theorem Proving and Machine Learning in the age of LLMs
April 2025, Edinburgh

Publication : Learning Structure-Aware Representations of Dependent Types @ **NeurIPS** 2024
ACK : Funds from **EuroProofNet CA2011** (2 STSMs in 2023)

~~Mandatory (?)~~ Redundant Intro Slide

Automated Theorem Proving

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis.

Automated Theorem Proving in the Times of ML

Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque.

Proving stuff (in Agda): what you write

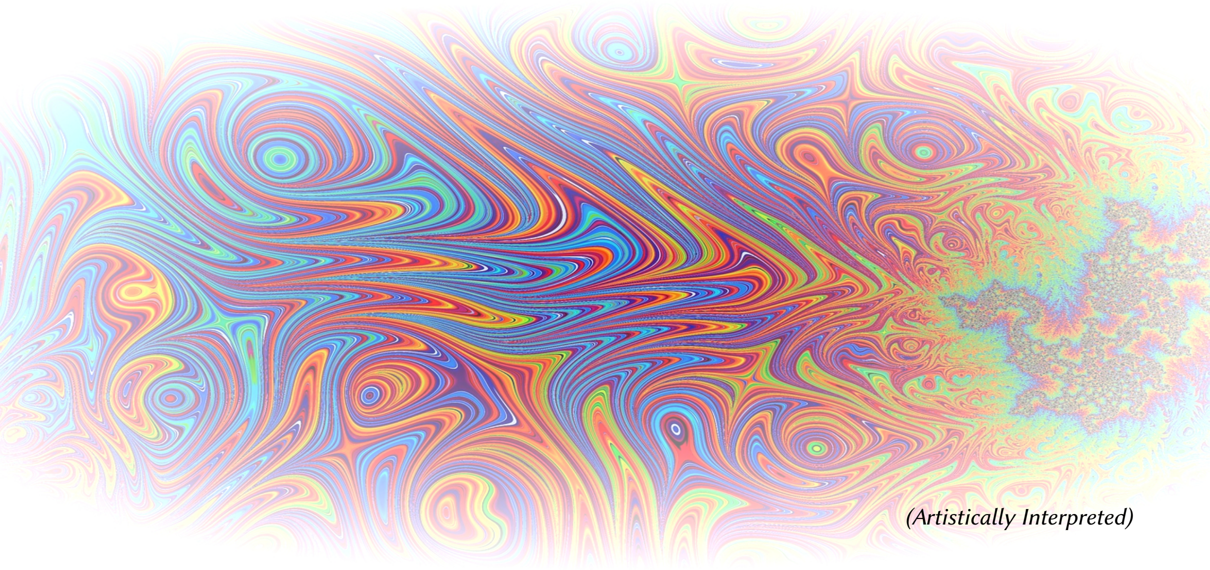
```
open import Relation.Binary.PropositionalEquality using (_≡_; refl; cong; trans)

data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ

_+_ : ℕ → ℕ → ℕ
zero    + n = n
suc m + n = suc (m + n)

+-comm : (m n : ℕ) → m + n ≡ n + m
+-comm zero      zero      = refl
+-comm zero      (suc n) = cong suc (+-comm zero n)
+-comm (suc m) zero      = cong suc (+-comm m zero)
+-comm (suc m) (suc n) = cong suc (trans (+-suc m n) (+-comm (suc m) n))
  where +-suc : ∀ m n → m + suc n ≡ suc (m + n)
        +-suc zero      n = refl
        +-suc (suc m) n = cong suc (+-suc m n)
```

… what Agda shows you

```
open import Relation.Binary.PropositionalEquality using (_≡_; refl; cong; trans)

data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ

_+_ : ℕ → ℕ → ℕ
zero    + n = n
suc m + n = suc (m + n)

+-comm : (m n : ℕ) → m + n ≡ n + m
+-comm zero      zero      = refl
+-comm zero      (suc n) = cong suc (+-comm zero n)
+-comm (suc m) zero      = cong suc (+-comm m zero)
+-comm (suc m) (suc n) = cong suc (trans (+-suc m n) (+-comm (suc m) n))
  where +-suc : ∀ m n → m + suc n ≡ suc (m + n)
        +-suc zero      n = refl
        +-suc (suc m) n = cong suc (+-suc m n)
```

Look at all the colors! 🌈

… what Agda really sees



(Artistically Interpreted)

… what you show the LLM

```
open | import | Relation | .Binary | .Pro | pos | itional | Equality | using | (_ | ≡ | _; | refl | ; | cong | ; | trans
)<newline><newline> | data | | ℕ | : | Set | where | <newline> | | zero | : | | ℕ | <newline> | | suc |
: | | ℕ | → | | ℕ | <newline><newline> | _ | + | _ | : | | ℕ | → | | ℕ | → | | ℕ | <newline> | zero
+ | n | ≡ | n | <newline> | s | uc | m | + | n | ≡ | suc | ( | m | + | n | )<newline><newline> | +- | comm
: | ( | m | n | : | | ℕ | ) | → | m | + | n | | ≡ | n | + | m | <newline> | +- | comm | zero | zero | ≡
refl | <newline> | +- | comm | zero | ( | s | uc | n | ) | ≡ | cong | suc | (+ | - | comm | zero | n | )<newline>
+- | comm | ( | s | uc | m | ) | zero | ≡ | cong | suc | (+ | - | comm | m | zero | )<newline> | +- | comm | (
s | uc | m | ) | ( | s | uc | n | ) | ≡ | cong | suc | ( | trans | (+ | - | s | uc | m | n | ) | (+ | - | comm | (
s | uc | m | ) | n | ))<newline> | | where | +- | s | uc | : | | ∀ | m | n
```

Where did all the colors go? ☁️

Doing things “right”

Doing things right

Contributions:

- *Structured Machine Learning Data for Agda.*
- *Learning to Represent (the Shapes of) Dependent Types.*

Data Extraction

```
open import Relation.Binary.PropositionalEquality using (_≡_; refl; cong; trans)

data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ

_+_ : ℕ → ℕ → ℕ
zero    + n = n
suc m + n = suc (m + n)

+-comm : (m n : ℕ) → m + n ≡ n + m
+-comm zero      zero      = refl
+-comm zero      (suc n) = cong suc (+-comm zero n)
+-comm (suc m) zero      = cong suc (+-comm m zero)
+-comm (suc m) (suc n) = cong suc (trans (+-suc m n) (+-comm (suc m) n))
  where +-suc : ∀ m n → m + suc n ≡ suc (m + n)
            +-suc zero      n = refl
            +-suc (suc m) n = cong suc (+-suc m n)
```

1. We go through **all definitions**.

Data Extraction

```
open import Relation.Binary.PropositionalEquality using (_≡_; refl; cong; trans)

data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ

_+_ : ℕ → ℕ → ℕ
zero    + n = n
suc m + n = suc (m + n)

+-comm : (m n : ℕ) → m + n ≡ n + m
+-comm zero      zero      = refl
+-comm zero      (suc n) = cong suc (+-comm zero n)
+-comm (suc m) zero      = cong suc (+-comm m zero)
+-comm (suc m) (suc n) = cong suc (trans (+-suc m n) (+-comm (suc m) n))
  where +-suc : ∀ m n → m + suc n ≡ suc (m + n)
            +-suc zero      n = refl
            +-suc (suc m) n = cong suc (+-suc m n)
```

1. We go through **all definitions**.

Data Extraction

```
open import Relation.Binary.PropositionalEquality using (_≡_; refl; cong; trans)

data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ

_+_ : ℕ → ℕ → ℕ
zero    + n = n
suc m + n = suc (m + n)

+-comm : (m n : ℕ) → m + n ≡ n + m
+-comm zero      zero      = refl
+-comm zero      (suc n) = cong suc (+-comm zero n)
+-comm (suc m) zero      = cong suc (+-comm m zero)
+-comm (suc m) (suc n) = cong suc (trans (+-suc m n) (+-comm (suc m) n))
  where +-suc : ∀ m n → m + suc n ≡ suc (m + n)
        +-suc zero      n = refl
        +-suc (suc m) n = cong suc (+-suc m n)
```

1. We go through **all definitions**.

Data Extraction

```
open import Relation.Binary.PropositionalEquality using (_≡_; refl; cong; trans)

data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ

_+_ : ℕ → ℕ → ℕ
zero    + n = n
suc m + n = suc (m + n)

+-comm : (m n : ℕ) → m + n ≡ n + m
+-comm zero      zero      = refl
+-comm zero      (suc n) = cong suc (+-comm zero n)
+-comm (suc m) zero      = cong suc (+-comm m zero)
+-comm (suc m) (suc n) = cong suc (trans (+-suc m n) (+-comm (suc m) n))
  where +-suc : ∀ m n → m + suc n ≡ suc (m + n)
        +-suc zero      n = refl
        +-suc (suc m) n = cong suc (+-suc m n)
```

1. We go through **all definitions**.

Data Extraction

```
open import Relation.Binary.PropositionalEquality using (_≡_; refl; cong; trans)

data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ

_+_ : ℕ → ℕ → ℕ
zero    + n = n
suc m + n = suc (m + n)

+-comm : (m n : ℕ) → m + n ≡ n + m
+-comm zero      zero      = refl
+-comm zero      (suc n) = cong suc (+-comm zero n)
+-comm (suc m) zero      = cong suc (+-comm m zero)
+-comm (suc m) (suc n) = cong suc (trans (+-suc m n) (+-comm (suc m) n))
  where +-suc : ∀ m n → m + suc n ≡ suc (m + n)
            +-suc zero      n = refl
            +-suc (suc m) n = cong suc (+-suc m n)
```

1. We go through **all definitions**.
   For each definition, we record:
   - its name

Data Extraction

```
open import Relation.Binary.PropositionalEquality using (_≡_; refl; cong; trans)

data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ

_+_ : ℕ → ℕ → ℕ
zero    + n = n
suc m + n = suc (m + n)

+-comm : (m n : ℕ) → m + n ≡ n + m
+-comm zero      zero      = refl
+-comm zero      (suc n) = cong suc (+-comm zero n)
+-comm (suc m) zero      = cong suc (+-comm m zero)
+-comm (suc m) (suc n) = cong suc (trans (+-suc m n) (+-comm (suc m) n))
  where +-suc : ∀ m n → m + suc n ≡ suc (m + n)
        +-suc zero      n = refl
        +-suc (suc m) n = cong suc (+-suc m n)
```

1. We go through **all definitions**.
   For each definition, we record:
   - its name
   - its type

Data Extraction

```
open import Relation.Binary.PropositionalEquality using (_≡_; refl; cong; trans)

data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ

_+_ : ℕ → ℕ → ℕ
zero    + n = n
suc m + n = suc (m + n)

+-comm : (m n : ℕ) → m + n ≡ n + m
+-comm zero    zero    = refl
+-comm zero    (suc n) = cong suc (+-comm zero n)
+-comm (suc m) zero    = cong suc (+-comm m zero)
+-comm (suc m) (suc n) = cong suc (trans (+-suc m n) (+-comm (suc m) n))
  where +-suc : ∀ m n → m + suc n ≡ suc (m + n)
        +-suc zero    n = refl
        +-suc (suc m) n = cong suc (+-suc m n)
```

1. We go through **all definitions**.
   For each definition, we record:
   - its name
   - its type
   - its term (/proof)

Data Extraction

```
open import Relation.Binary.PropositionalEquality using (_≡_; refl; cong; trans)

data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ

_+_ : ℕ → ℕ → ℕ
zero    + n = n
suc m + n = suc (m + n)

+-comm : (m n : ℕ) → m + n ≡ n + m
+-comm zero      zero      = refl
+-comm zero      (suc n) = cong suc (+-comm zero n)
+-comm (suc m) zero      = cong suc (+-comm m zero)
+-comm (suc m) (suc n) = cong suc (trans (+-suc m n) (+-comm (suc m) n))
  where +-suc : ∀ m n → m + suc n ≡ suc (m + n)
        +-suc zero      n = refl
        +-suc (suc m) n = cong suc (+-suc m n)
```

2. We go through all **subterms**.

Data Extraction

```
open import Relation.Binary.PropositionalEquality using (_≡_; refl; cong; trans)

data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ

_+_ : ℕ → ℕ → ℕ
zero    + n = n
suc m + n = suc (m + n)

+-comm : (m n : ℕ) → m + n ≡ n + m
+-comm zero     zero    = ?
+-comm zero     (suc n) = cong suc (+-comm zero n)
+-comm (suc m) zero     = cong suc (+-comm m zero)
+-comm (suc m) (suc n) = cong suc (trans (+-suc m n) (+-comm (suc m) n))
  where +-suc : ∀ m n → m + suc n ≡ suc (m + n)
        +-suc zero     n = refl
        +-suc (suc m) n = cong suc (+-suc m n)
```

2. We go through all **subterms**.

Data Extraction

```
open import Relation.Binary.PropositionalEquality using (_≡_; refl; cong; trans)

data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ

_+_ : ℕ → ℕ → ℕ
zero    + n = n
suc m + n = suc (m + n)

+-comm : (m n : ℕ) → m + n ≡ n + m
+-comm zero      zero      = refl
+-comm zero      (suc n) = cong suc (+-comm zero ?)
+-comm (suc m) zero      = cong suc (+-comm m zero)
+-comm (suc m) (suc n) = cong suc (trans (+-suc m n) (+-comm (suc m) n))
  where +-suc : ∀ m n → m + suc n ≡ suc (m + n)
        +-suc zero      n = refl
        +-suc (suc m) n = cong suc (+-suc m n)
```

2. We go through all **subterms**.

Data Extraction

```
open import Relation.Binary.PropositionalEquality using (_≡_; refl; cong; trans)

data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ

_+_ : ℕ → ℕ → ℕ
zero    + n = n
suc m + n = suc (m + n)

+-comm : (m n : ℕ) → m + n ≡ n + m
+-comm zero      zero    = refl
+-comm zero      (suc n) = cong suc (+-comm ? n)
+-comm (suc m) zero      = cong suc (+-comm m zero)
+-comm (suc m) (suc n) = cong suc (trans (+-suc m n) (+-comm (suc m) n))
  where +-suc : ∀ m n → m + suc n ≡ suc (m + n)
        +-suc zero      n = refl
        +-suc (suc m) n = cong suc (+-suc m n)
```

2. We go through all **subterms**.

Data Extraction

```
open import Relation.Binary.PropositionalEquality using (_≡_; refl; cong; trans)

data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ

_+_ : ℕ → ℕ → ℕ
zero    + n = n
suc m + n = suc (m + n)

+-comm : (m n : ℕ) → m + n ≡ n + m
+-comm zero      zero      = refl
+-comm zero      (suc n) = cong suc ?
+-comm (suc m) zero      = cong suc (+-comm m zero)
+-comm (suc m) (suc n) = cong suc (trans (+-suc m n) (+-comm (suc m) n))
  where +-suc : ∀ m n → m + suc n ≡ suc (m + n)
            +-suc zero      n = refl
            +-suc (suc m) n = cong suc (+-suc m n)
```

2. We go through all **subterms**.

Data Extraction

```
open import Relation.Binary.PropositionalEquality using (_≡_; refl; cong; trans)

data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ

_+_ : ℕ → ℕ → ℕ
zero    + n = n
suc m + n = suc (m + n)

+-comm : (m n : ℕ) → m + n ≡ n + m
+-comm zero      zero      = refl
+-comm zero      (suc n) = cong suc ?
+-comm (suc m) zero      = cong suc (+-comm m zero)
+-comm (suc m) (suc n) = cong suc (trans (+-suc m n) (+-comm (suc m) n))
  where +-suc : ∀ m n → m + suc n ≡ suc (m + n)
          +-suc zero      n = refl
          +-suc (suc m) n = cong suc (+-suc m n)
```

2. We go through all **subterms**.
   For each subterm, we record:
   - its type (/the goal)

```
goal: n ≡ (n + zero)
```

Data Extraction

```
open import Relation.Binary.PropositionalEquality using (_≡_; refl; cong; trans)

data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ

_+_ : ℕ → ℕ → ℕ
zero    + n = n
suc m + n = suc (m + n)

+-comm : (m n : ℕ) → m + n ≡ n + m
+-comm zero      zero     = refl
+-comm zero      (suc n) = cong suc ?
+-comm (suc m) zero      = cong suc (+-comm m zero)
+-comm (suc m) (suc n) = cong suc (trans (+-suc m n) (+-comm (suc m) n))
  where +-suc : ∀ m n → m + suc n ≡ suc (m + n)
            +-suc zero      n = refl
            +-suc (suc m) n = cong suc (+-suc m n)
```

2. We go through all **subterms**.
   For each subterm, we record:
   - its type (/the goal)
   - its scope

Data Extraction

```
open import Relation.Binary.PropositionalEquality using (_≡_; refl; cong; trans)

data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ

_+_ : ℕ → ℕ → ℕ
zero    + n = n
suc m + n = suc (m + n)

+-comm : (m n : ℕ) → m + n ≡ n + m
+-comm zero      zero      = refl
+-comm zero      (suc n) = cong suc ?
+-comm (suc m) zero      = cong suc (+-comm m zero)
+-comm (suc m) (suc n) = cong suc (trans (+-suc m n) (+-comm (suc m) n))
  where +-suc : ∀ m n → m + suc n ≡ suc (m + n)
        +-suc zero      n = refl
        +-suc (suc m) n = cong suc (+-suc m n)
```

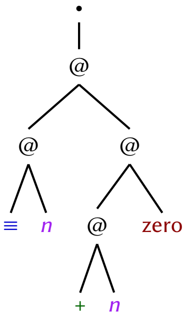
2. We go through all **subterms**.
   For each subterm, we record:
   - its type (/the goal)
   - its scope
   - its context

Data Extraction

3. Each term & type is recorded as both:
   - a pretty string

`goal: n ≡ (n + zero)`

Data Extraction



`goal: n ≡ (n + zero)`

3. Each term & type is recorded as both:
   - a pretty string
   - the underlying AST

Data: TL;DR

Niceties:

- among first ML datasets for Agda
- subterm iteration $\implies$ type-checked data augmentation for free
- extraction explicitly preserving type-structure

Data: TL;DR

Niceties:

- among first ML datasets for Agda
- subterm iteration $\Longrightarrow$ type-checked data augmentation for free
- extraction explicitly preserving type-structure

Numerical[1]:

- 800 modules
- 11.751 definitions
- 67.255 “holes” read: data points

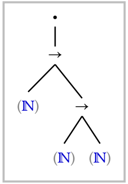
1: passing extracts from `agda-stdlib` 1.7.2

Representation Learning 1

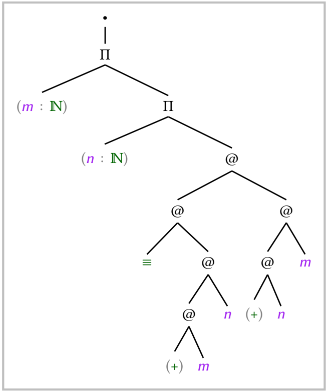
What's to represent?

$\mathbb{N}$

$+$



+-comm

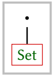


A sequence of ASTs

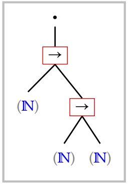
Representation Learning 1

What's to represent?

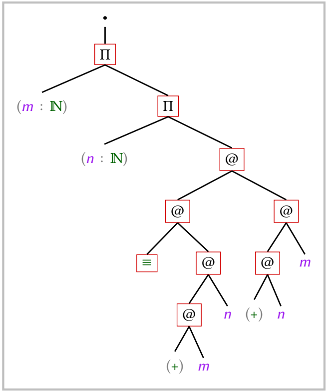
$\mathbb{N}$



$+$



+-comm

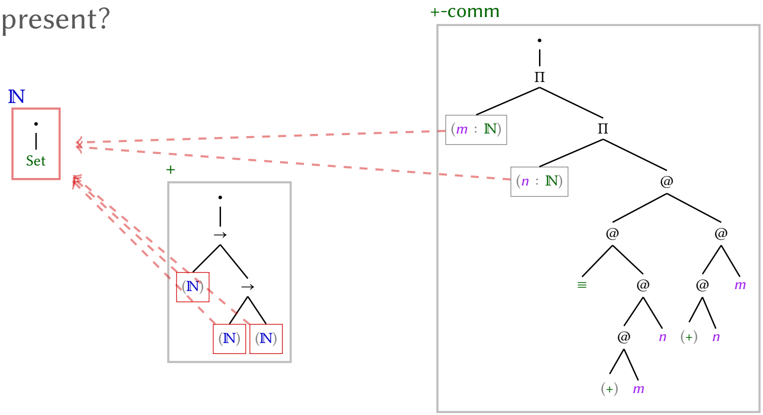


A sequence of ASTs, where nodes are:

- TT primitives

Representation Learning 1

What's to represent?

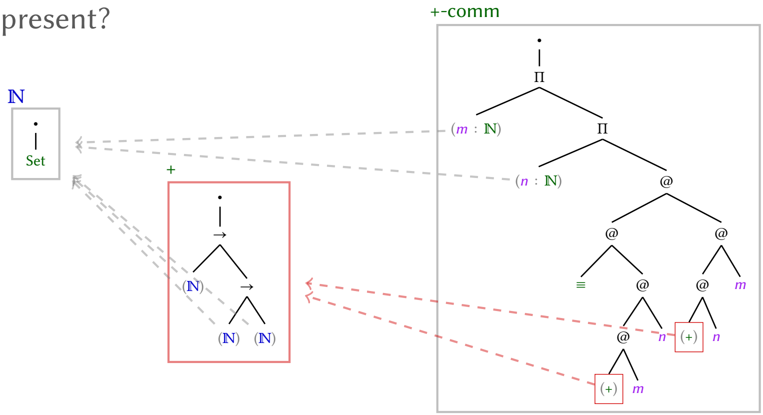


A sequence of ASTs, where nodes are:

- TT primitives
- references to (other) lemmas (inter-AST)

Representation Learning 1

What's to represent?

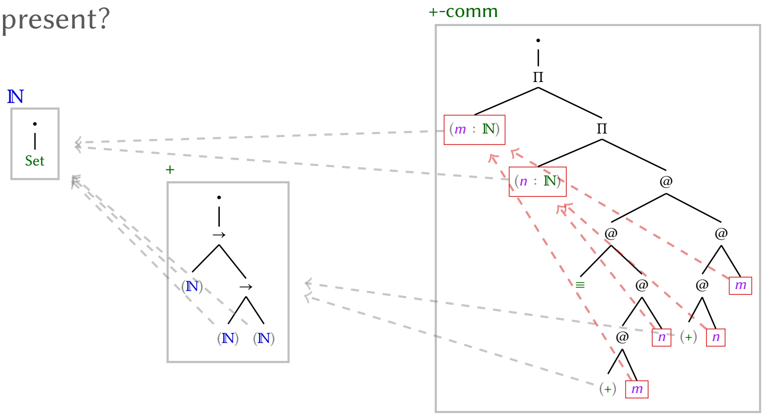


A sequence of ASTs, where nodes are:

- TT primitives
- references to (other) lemmas (inter-AST)

Representation Learning 1

What's to represent?



A sequence of ASTs, where nodes are:

- TT primitives
- references to (other) lemmas (inter-AST)
- references to bound variables (intra-AST)

Representation Learning 2

How to represent it?

Candidate Architectures:

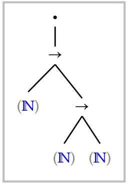
- ~~LLMs~~ -- *just no*
- ~~GNNs~~ -- *too generic, oversmoothing*
- ~~Tree (R)NNs~~ -- *too slow, generally under-performing*
- Full Attention [?] -- *no structural biases*
  -- $\left(\sum_t^T n(t)\right)^2$ *scaling*

Representation Learning 2

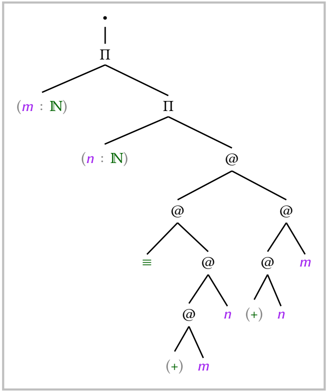
Amending self-attention

ℕ

+



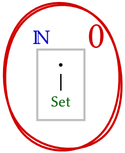
+-comm



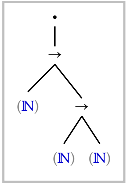
- Treewise Linear (Taylor Series) Attention -- $\#T \times \max_t^T n(t)$ scaling

Representation Learning 2

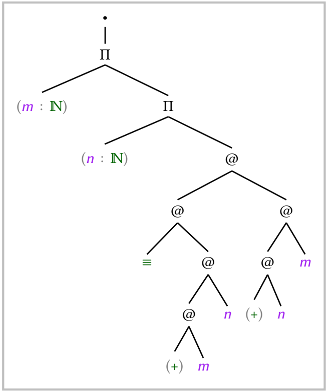
Amending self-attention



+



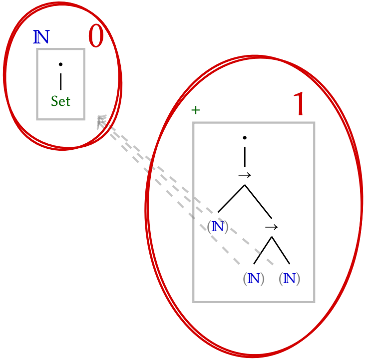
+-comm

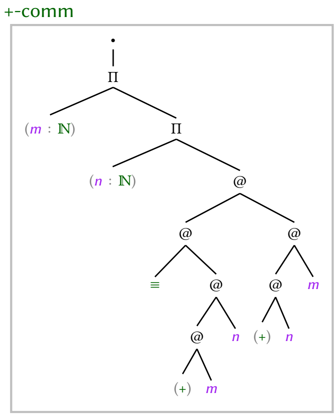


- Treewise <u>Linear</u> (Taylor Series) Attention -- $\#T \times \max_t^T n(t)$ scaling
- Dependency-Level Batching -- explicit scope referencing

Representation Learning 2

Amending self-attention

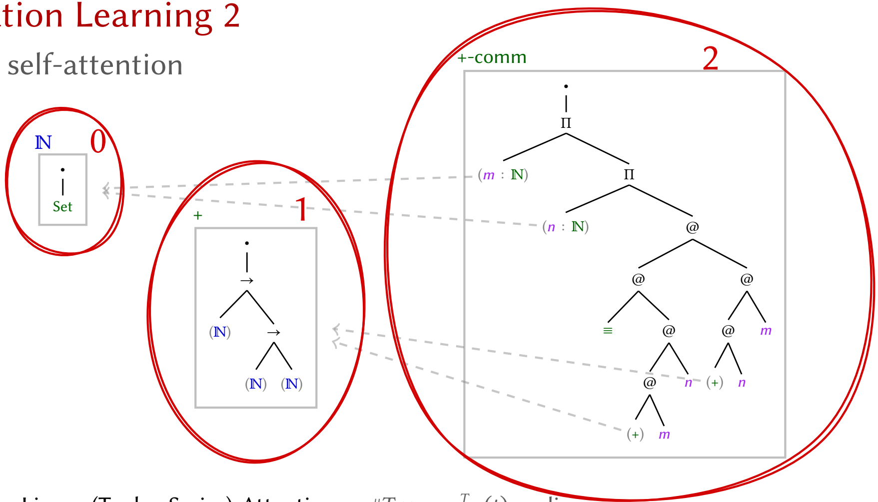




- Treewise Linear (Taylor Series) Attention -- $\#T \times \max_t^T n(t)$ scaling
- Dependency-Level Batching -- explicit scope referencing

Representation Learning 2

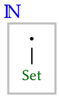
Amending self-attention

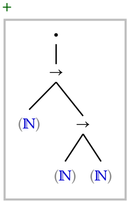


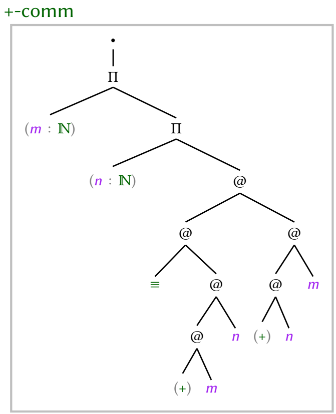
- Treewise Linear (Taylor Series) Attention -- $\#T \times \max_t^T n(t)$ scaling
- Dependency-Level Batching -- explicit scope referencing

Representation Learning 2

Amending self-attention



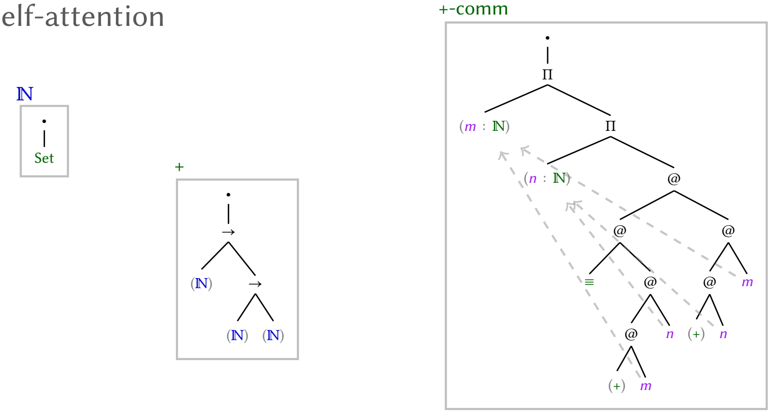




- Treewise Linear (Taylor Series) Attention -- $\#T \times \max_t^T n(t)$ scaling
- Dependency-Level Batching -- explicit scope referencing
- Relative Tree-PE -- proper inductive biases

Representation Learning 2

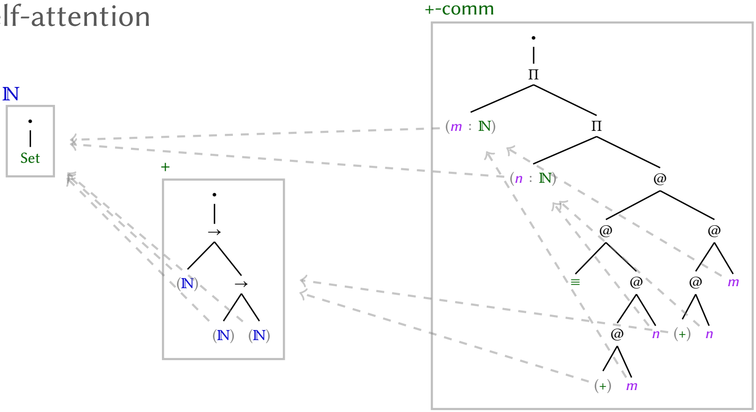
Amending self-attention



- Treewise Linear (Taylor Series) Attention -- $\#T \times \max_t^T n(t)$ scaling
- Dependency-Level Batching -- explicit scope referencing
- Relative Tree-PE -- proper inductive biases
  -- & dynamic variable indexing

Representation Learning 2

Amending self-attention



Representations informed by **type shapes** alone:

- invariance to α-renaming, scope permutations, syntactic distractions, etc.
- ...but a few things get lost in translation

Experimental Setup

Premise Selection
Contextually rank scope entries by their relevance to the current goal.

Quill
Tiny PoC model (6L × 8H × 256D; 1 mil. params; 25MB@FP32) trained for ~8h on a V100

Data

- train on random holes from 85% of `agda-stdlib` (ignoring size outliers)
- eval on unseen proofs from:
  - remaining 15% (split between ID and OOD on the basis of size)
  - `Unimath` & `TypeTopology` (distant domains)

… but does it work?

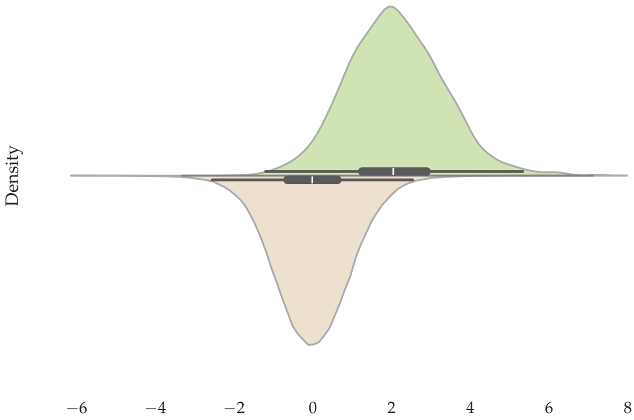
Mandatory table with numbers

	Average / R-Precision			
MODEL	stdlib:ID	stdlib:OOD	Unimath	TypeTopo
QUILL	**50.2** / **40.3**	**38.7** / **31.1**	**27.0** / **17.4**	**22.5** / **15.4**
⋮				
Transformer Baseline[1]	10.9 / 3.7	8.5 / 4.5	9.4 / 3.9	5.8 / 0.9

1: no Tree-PE, no variable indexing resolution, no Taylor expansion in linear attention

… but does it work?

Less obscure visualization



Empirical distribution of selection scores of relevant (green) vs irrelevant (red) lemmas (`stdlib`:ID).

... but does it work?

Findings, TL;DR

- high performance despite limited expressivity & no term exposition
- structure preservation outweights architectural optimizations
- baseline encoder collapses

Suggested takehome messages

🌍 the earth is not flat, proofs are not strings, LLMs are not the (only) answer

🔑 back-and-forth compiler integration is the key to better architectures

Thank you

- PAPER – openreview.net/forum?id=e397soEZh8
  Published manuscript & reviews.
- AGDA2TRAIN – github.com/omelkonian/agda2train
  Data extraction as an Agda compilation backend (in Haskell).
- AGDA-QUILL – github.com/konstantinoskokos/quill
  ML model; ML-facing Python interface for dataset reading & processing.

```
open import Data.List.Relation.Binary.Permutation.Propositional.Properties
open PermutationReasoning


private variable
  -- ℓ : Level
  A B : Set ℓ
  x y : A
  xs ys zs ws : List A
  xss yss : List (List A)

↭-concat⁺ : xss ↭ yss → concat xss ↭ concat yss
↭-concat⁺ = { }0
{-
↭-concat⁺ refl = refl
↭-concat⁺ (prep xs p) = ++⁺ˡ xs (↭-concat⁺ p)
↭-concat⁺ {xss = _ ∷ _ ∷ xss}{_ ∷ _ ∷ yss} (swap xs ys p) = begin
  xs ++ ys ++ concat xss ↭⟨ shifts xs ys ⟩
  ys ++ xs ++ concat xss ↭⟨ ++⁺ˡ ys (++⁺ˡ xs (↭-concat⁺ p)) ⟩
  ys ++ xs ++ concat yss ∎
↭-concat⁺ (trans xs↭ys ys↭zs) = trans (↭-concat⁺ xs↭ys) (↭-concat⁺ ys↭zs)
-}
```

```
REPL.↭-concat⁺
Data.List.Relation.Binary.Permutation.Propositional.Properties.shifts
Data.List.Relation.Binary.Permutation.Propositional.Properties.++⁺ʳ
Data.List.Relation.Binary.Permutation.Propositional.Properties.++⁺ˡ
Data.List.Relation.Binary.Permutation.Propositional._↭_.trans
Data.List.Relation.Binary.Permutation.Propositional.↭-sym
Data.List.Relation.Binary.Permutation.Propositional.Properties.zoom
Data.List.Relation.Binary.Permutation.Propositional.Properties.↭-sym-involutive
Data.List.Relation.Binary.Permutation.Propositional.Properties.shift
Data.List.Relation.Binary.Permutation.Propositional._↭_
Data.List.Relation.Binary.Permutation.Propositional.Properties.++↭ʳ++
Data.List.Base._++_
Data.List.Base._ʳ++_
Data.List.Base.reverseAcc
Data.List.Relation.Binary.Permutation.Propositional.Properties.drop-mid
Data.List.Relation.Binary.Permutation.Propositional.Properties.drop-∷
Data.List.Relation.Binary.Permutation.Propositional.Properties.drop-mid-≡
Data.List.Base.intercalate
Data.List.Relation.Binary.Permutation.Propositional.Properties.++⁺
Data.List.Relation.Binary.Permutation.Propositional.Properties.inject
Data.List.Base.map
Agda.Builtin.List.List
Data.List.Base.concatMap
Data.List.Relation.Binary.Permutation.Propositional.PermutationReasoning.step-prep
Data.List.Relation.Binary.Permutation.Propositional._↭_.prep
Data.List.Relation.Binary.Permutation.Propositional.↭-reflexive
Data.List.Relation.Binary.Permutation.Propositional.Properties.↭-singleton-inv
Data.List.Base.tails
Data.List.Base.inits
Data.List.Base.concat
Agda.Builtin.List.List._∷_
Data.List.Base.ap
Data.List.Base.reverse
Data.List.Relation.Binary.Permutation.Propositional.Properties.↭-reverse
Agda.Builtin.List.List.
Data.List.Relation.Binary.Permutation.Propositional.PermutationReasoning.step-swap
Data.List.Relation.Binary.Permutation.Propositional._↭_.swap
Data.List.Relation.Binary.Permutation.Propositional.Properties.↭-empty-in
```