LLM-free Representation Learning of Theorem Structures
(an application in Agda)

Konstantinos Kogkalidis

Deep-Learning Models for Mathematics and Type Theory
April 2025, Gothenburg

Based on:
K. Kogkalidis, O. Melkonian, and J.-P. Bernardy. *Learning structure-aware representations of dependent types.* NeurIPS, 2024.

💶 **from**:
EuroProofNet CA2011 WG5 (2 STSMs @ GU, 2023).

cost
EUROPEAN COOPERATION IN SCIENCE & TECHNOLOGY

Funded by the European Union

Context

...

Proving stuff (in Agda): what you write

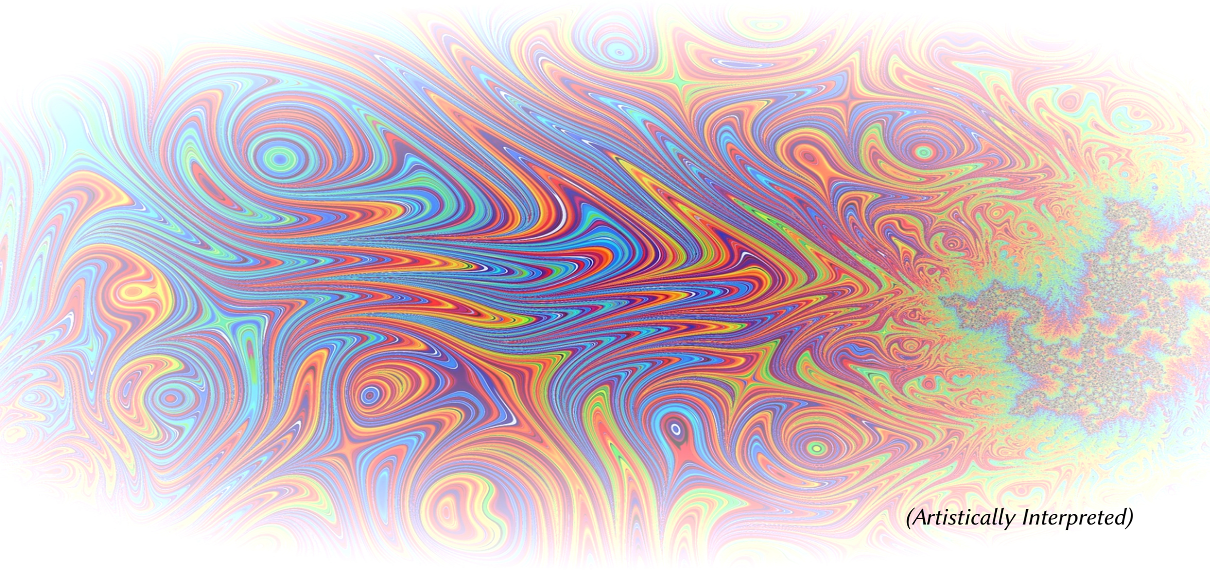
```
open import Relation.Binary.PropositionalEquality using (_≡_; refl; cong; trans)

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? ☁️

... why LLMs then? (take #1)

- text as catch-all modality
  an imperfect/irreversible projection of *arbitrary* structure...
- data scaling enables implicit learning
  exposure to math, logic, code...
- closest thing we have to general machine “intellligence”
  general intelligence subsumes (or at least aids) math problem solving
- ease of integration
  APIs everywhere... (plus: we write sequentially, too)

… why LLMs then? (take #1)

- text as catch-all modality
  an imperfect/irreversible projection of *arbitrary* structure…
- data scaling enables implicit learning
  exposure to math, logic, code…
- closest thing we have to general machine "intellligence"
  general intelligence subsumes (or at least aids) math problem solving
- ease of integration
  APIs everywhere… (plus: we write sequentially, too)

… why LLMs then? (take #1)

- text as catch-all modality
  an imperfect/irreversible projection of *arbitrary* structure…
- data scaling enables implicit learning
  exposure to math, logic, code…
- closest thing we have to general machine "intellligence"
  general intelligence subsumes (or at least aids) math problem solving
- ease of integration
  APIs everywhere… (plus: we write sequentially, too)

… why LLMs then? (take #1)

- text as catch-all modality
  an imperfect/irreversible projection of *arbitrary* structure…
- data scaling enables implicit learning
  exposure to math, logic, code…
- closest thing we have to general machine "intellligence"
  general intelligence subsumes (or at least aids) math problem solving
- ease of integration
  APIs everywhere… (plus: we write sequentially, too)

… why LLMs then? (take #2)

- technoscientific hype, AI zeitgeist
  big tech is now into proof theory…
- low hanging fruit mentality
  LLMs produce results *today*
- hard math made easy
  LLM ≡ universal calculator
- benchmarking culture
  less questions, more answers

… why LLMs then? (take #2)

- technoscientific hype, AI zeitgeist
  big tech is now into proof theory…
- low hanging fruit mentality
  LLMs produce results *today*
- hard math made easy
  LLM $\equiv$ universal calculator
- benchmarking culture
  less questions, more answers

… why LLMs then? (take #2)

- technoscientific hype, AI zeitgeist
  big tech is now into proof theory…
- low hanging fruit mentality
  LLMs produce results *today*
- hard math made easy
  LLM $\equiv$ universal calculator
- benchmarking culture
  less questions, more answers

… why LLMs then? (take #2)

- technoscientific hype, AI zeitgeist
  big tech is now into proof theory…
- low hanging fruit mentality
  LLMs produce results *today*
- hard math made easy
  LLM ≡ universal calculator
- benchmarking culture
  less questions, more answers

… why LLMs then? (take #2)

Andrej Karpathy
@karpathy

There's a new kind of coding I call "vibe ~~coding~~ proving", where you fully give in to the vibes, embrace exponentials, and forget that the ~~code~~ proof even exists. It's possible because the LLMs (e.g. Cursor Composer w Sonnet) are getting too good. Also I just talk to Composer with SuperWhisper so I barely even touch the keyboard. I ask for the dumbest things like "decrease the padding on the sidebar by half" because I'm too lazy to find it. I "Accept All" always, I don't read the diffs anymore. When I get

… why LLMs then? (take #2)

… why LLMs then? (take #2)

… why (not) LLMs then? (take #2)

- technoscientific hype, AI zeitgeist
  big tech is now into proof theory… ⇝ concentration of technoscientific capital/authority
- low hanging fruit mentality
  LLMs produce results *today*
- hard math made easy
  LLM ≡ universal calculator
- benchmarking culture
  less questions, more answers

… why (not) LLMs then? (take #2)

- technoscientific hype, AI zeitgeist
  big tech is now into proof theory… ⇝ concentration of technoscientific capital/authority
- low hanging fruit mentality
  LLMs produce results *today* ⇝ what do we do *tomorrow*?
- hard math made easy
  LLM ≡ universal calculator
- benchmarking culture
  less questions, more answers

… why (not) LLMs then? (take #2)

- technoscientific hype, AI zeitgeist
  big tech is now into proof theory… $\rightsquigarrow$ concentration of technoscientific capital/authority
- low hanging fruit mentality
  LLMs produce results *today* $\rightsquigarrow$ what do we do *tomorrow*?
- hard math made easy
  LLM $\equiv$ universal calculator $\rightsquigarrow$ deskilling, less fundamental alignment between application & theory
- benchmarking culture
  less questions, more answers

… why (not) LLMs then? (take #2)

- technoscientific hype, AI zeitgeist
  big tech is now into proof theory… ⇝ concentration of technoscientific capital/authority
- low hanging fruit mentality
  LLMs produce results *today* ⇝ what do we do *tomorrow*?
- hard math made easy
  LLM ≡ universal calculator ⇝ deskilling, less fundamental alignment between application & theory
- benchmarking culture
  less questions, more answers ⇝ sales pitch aesthetics, knowledge recycling, science as competition, …

… why (not) LLMs then? (take #2)

The main contributions of this work are as follows:

- We identified the hierarchical structure inherent in mathematical reasoning, from foundational definitions to final goals.
- We proposed a new algorithm for better structure learning for LLMs.

Lean is a strongly typed language, which allows all tokens to be naturally unfolded across multiple semantic levels. These levels align with various components of reasoning, with each successive level built upon the foundations of the preceding ones. The categorization of these layers can be delineated

Doing things ~~right~~ better

Search for data efficient, structure-preserving alignment between domain (*problem*) and architecture (*tool*).

Doing things ~~right~~ better

Search for data efficient, structure-preserving alignment between domain (*problem*) and architecture (*tool*).

Here:

- *Structured data extraction from Agda modules.*
- *Learning to represent (the shapes of) the extracted 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)
```

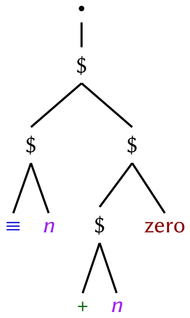
2. We go through all **subterms**.
   For each subterm, we record:
   - its type (/the goal)
   - its scope
   - its context

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 term-structure (type & proof level)

Data: TL;DR

Niceties:

- among first ML datasets for Agda
- subterm iteration $\implies$ type-checked data augmentation for free
- extraction explicitly preserving term-structure (type & proof level)

Numerical[1]:

- 800 modules
- 11.751 definitions
- 67.255 "holes" read: data points
  (w/o subterm iteration: 6.960)

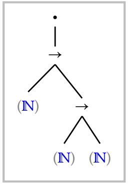
1: for reference – passing extracts from `agda-stdlib` 1.7.2

Representation Learning 1

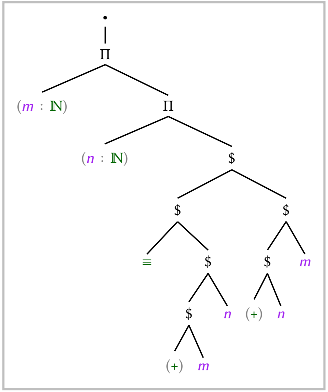
What's to represent?

$\mathbb{N}$

$+$



+-comm

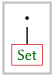


A sequence of ASTs

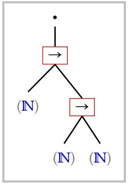
Representation Learning 1

What's to represent?

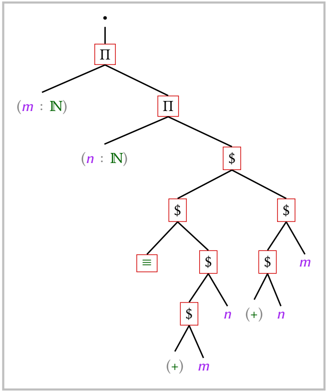
$\mathbb{N}$



$+$



+-comm

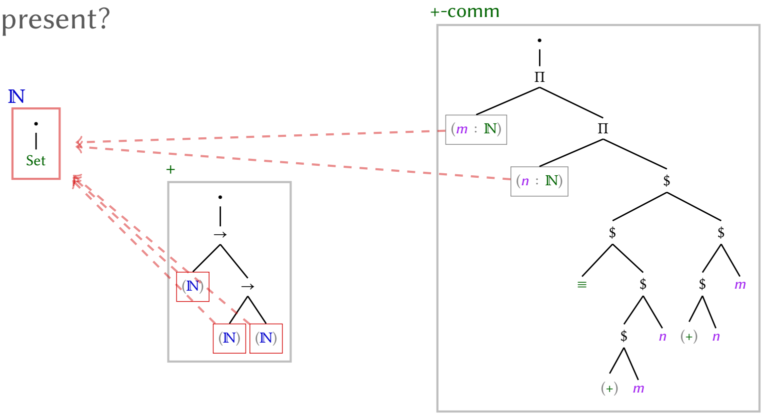


A sequence of ASTs, where nodes are:

- primitive symbols -- Set, Π, →, λ, $, ...

Representation Learning 1

What's to represent?

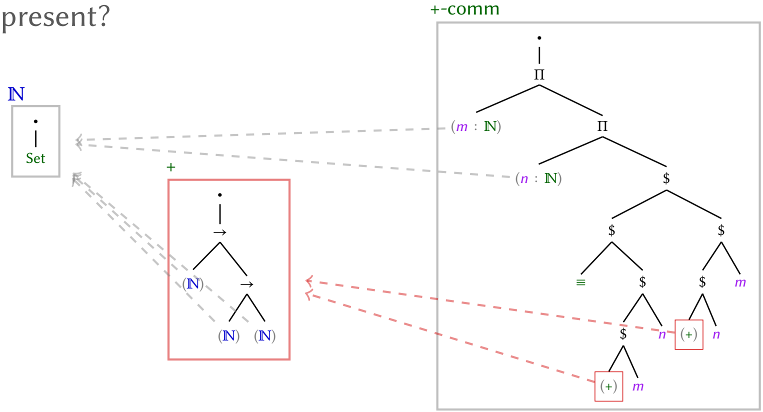


A sequence of ASTs, where nodes are:

- primitive symbols -- Set, Π, →, λ, \$, ...
- references to (other) lemmas (inter-AST) -- ℕ, +, ...

Representation Learning 1

What's to represent?

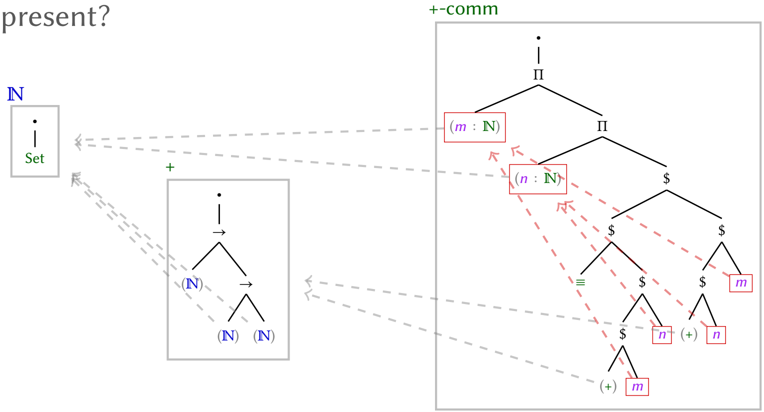


A sequence of ASTs, where nodes are:

- primitive symbols -- Set, Π, →, λ, $, ...
- references to (other) lemmas (inter-AST) -- ℕ, +, ...

Representation Learning 1

What's to represent?



A sequence of ASTs, where nodes are:

- primitive symbols -- Set, Π, →, λ, \$, ...
- references to (other) lemmas (inter-AST) -- ℕ, +, ...
- references to bound variables (intra-AST) -- m, n, ...

Representation Learning 2

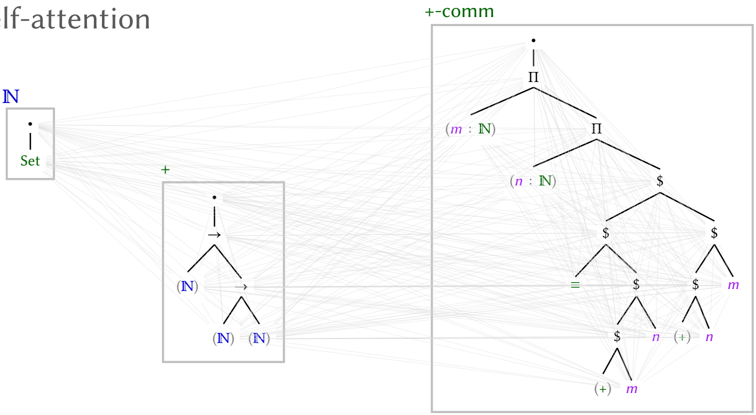
How to represent it?

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

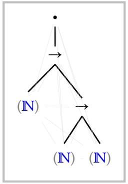


Representation Learning 2

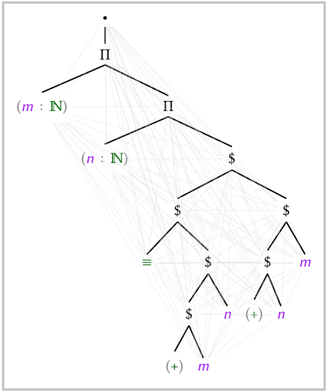
Amending self-attention

$\mathbb{N}$

+



+-comm



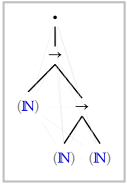
- Per-Tree Attention -- $\sum_t^T n(t)^2$ scaling

Representation Learning 2

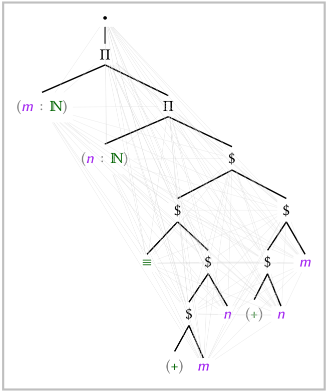
Amending self-attention

$\mathbb{N}$

+



+-comm

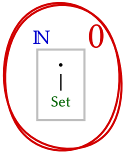


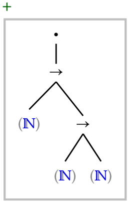
- Per-Tree Attention

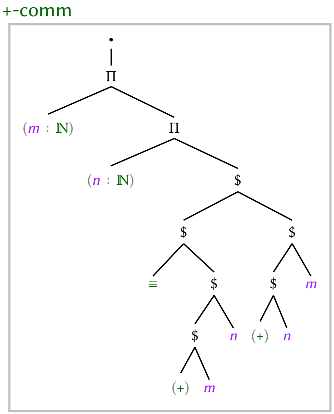
-- $\sum_t^T n(t)^2$ scaling
-- $\sum_t^T n(t)$ if using a linear kernel (here: Taylor series)

Representation Learning 2

Amending self-attention



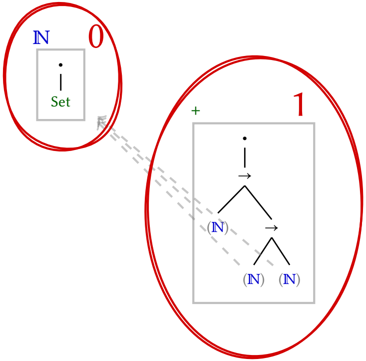


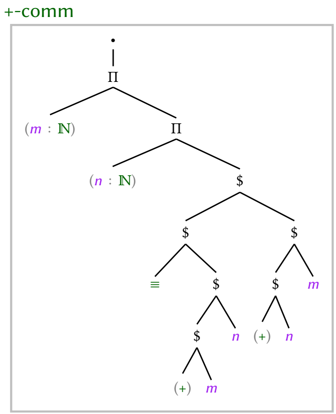


- Per-Tree Attention -- $\sum_t^T n(t)^2$ scaling
  -- $\sum_t^T n(t)$ if using a linear kernel (here: Taylor series)
- Dependency-Level Batching -- explicit scope referencing

Representation Learning 2

Amending self-attention

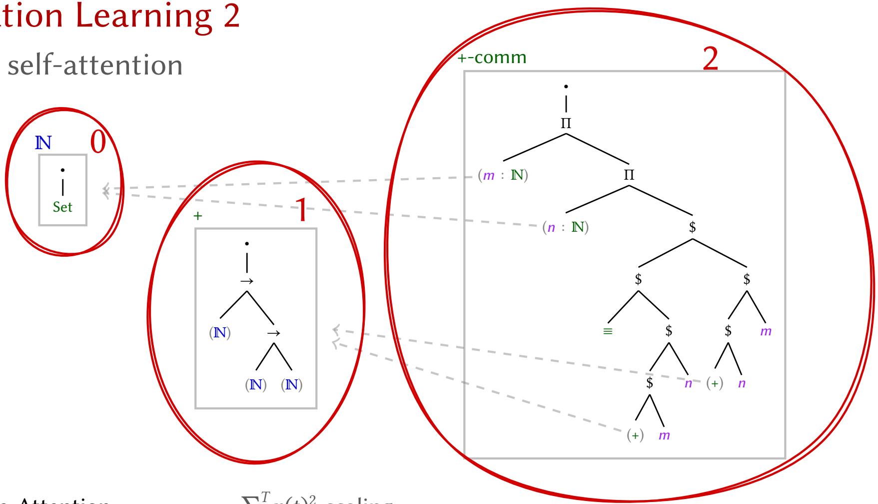




- Per-Tree Attention -- $\sum_t^T n(t)^2$ scaling
  -- $\sum_t^T n(t)$ if using a linear kernel (here: Taylor series)
- Dependency-Level Batching -- explicit scope referencing

Representation Learning 2

Amending self-attention



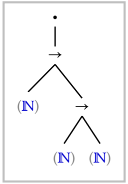
- Per-Tree Attention -- $\sum_t^T n(t)^2$ scaling
  -- $\sum_t^T n(t)$ if using a linear kernel (here: Taylor series)
- Dependency-Level Batching -- explicit scope referencing

Representation Learning 2

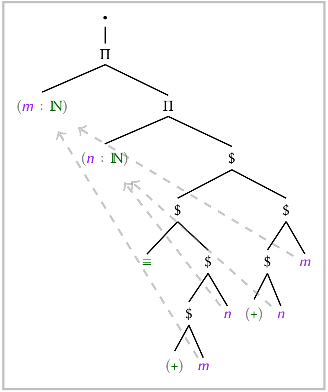
Amending self-attention

$\mathbb{N}$

+



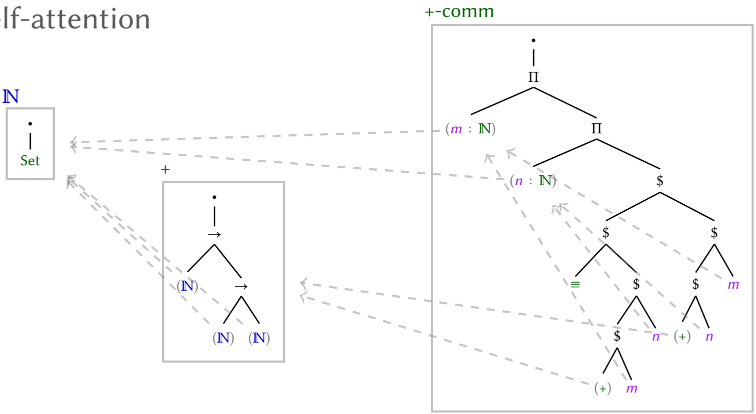
+-comm



- Per-Tree Attention -- $\sum_t^T n(t)^2$ scaling
  -- $\sum_t^T n(t)$ if using a linear kernel (here: Taylor series)
- Dependency-Level Batching -- explicit scope referencing
- Relative Tree-PE -- proper inductive biases

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 single 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)

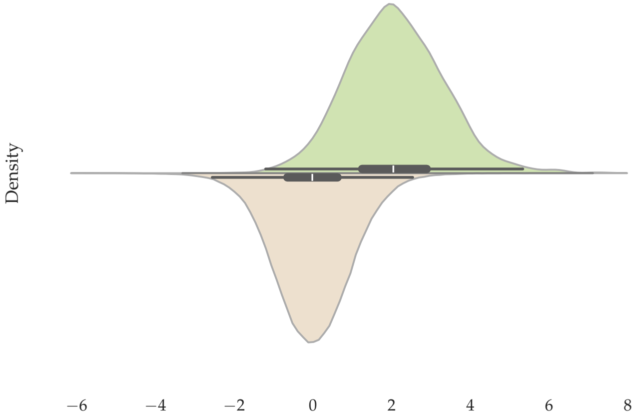
… but does it work?

Mandatory table with numbers

Model	Average / R-Precision			
	`stdlib`:ID	`stdlib`:OOD	`Unimath`	`TypeTopo`
Quill	**50.2** / **40.3**	**38.7** / **31.1**	**27.0** / **17.4**	**22.5** / **15.4**
(no Taylor expansion)	47.0 / 36.2	37.1 / 29.2	26.8 / 17.0	21.4 / 14.4
(no Tree-PE)	44.5 / 34.1	30.7 / 24.0	24.8 / 15.5	18.8 / 12.3
(no variable resolution)	35.8 / 25.9	25.5 / 19.1	19.7 / 11.6	17.7 / 11.0
Transformer Baseline	10.9 / 3.7	8.5 / 4.5	9.4 / 3.9	5.8 / 0.9

… 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)
-}
```

```
1  REPL.↭-concat⁺
2  Data.List.Relation.Binary.Permutation.Propositional.Properties.shifts
3  Data.List.Relation.Binary.Permutation.Propositional.Properties.++⁺ʳ
4  Data.List.Relation.Binary.Permutation.Propositional.Properties.++⁺ˡ
5  Data.List.Relation.Binary.Permutation.Propositional._↭_.trans
6  Data.List.Relation.Binary.Permutation.Propositional.↭-sym
7  Data.List.Relation.Binary.Permutation.Propositional.Properties.zoom
8  Data.List.Relation.Binary.Permutation.Propositional.Properties.↭-sym-involutive
9  Data.List.Relation.Binary.Permutation.Propositional.Properties.shift
10 Data.List.Relation.Binary.Permutation.Propositional._↭_
11 Data.List.Relation.Binary.Permutation.Propositional.Properties.++↭ʳ++
12 Data.List.Base._++_
13 Data.List.Base._ʳ++_
14 Data.List.Base.reverseAcc
15 Data.List.Relation.Binary.Permutation.Propositional.Properties.drop-mid
16 Data.List.Relation.Binary.Permutation.Propositional.Properties.drop-∷
17 Data.List.Relation.Binary.Permutation.Propositional.Properties.drop-mid-≡
18 Data.List.Base.intercalate
19 Data.List.Relation.Binary.Permutation.Propositional.Properties.++⁺
20 Data.List.Relation.Binary.Permutation.Propositional.Properties.inject
21 Data.List.Base.map
22 Agda.Builtin.List.List
23 Data.List.Base.concatMap
24 Data.List.Relation.Binary.Permutation.Propositional.PermutationReasoning.step-prep
25 Data.List.Relation.Binary.Permutation.Propositional._↭_.prep
26 Data.List.Relation.Binary.Permutation.Propositional.↭-reflexive
27 Data.List.Relation.Binary.Permutation.Propositional.Properties.↭-singleton-inv
28 Data.List.Base.tails
29 Data.List.Base.inits
30 Data.List.Base.concat
31 Agda.Builtin.List.List._∷_
32 Data.List.Base.ap
33 Data.List.Base.reverse
34 Data.List.Relation.Binary.Permutation.Propositional.Properties.↭-reverse
35 Agda.Builtin.List.List.
36 Data.List.Relation.Binary.Permutation.Propositional.PermutationReasoning.step-swap
37 Data.List.Relation.Binary.Permutation.Propositional._↭_.swap
38 Data.List.Relation.Binary.Permutation.Propositional.Properties.↭-empty-in
```