Learning Structure-Aware Representations of Dependent Types

Konstantinos Kogkalidis
Orestis Melkonian
Jean-Philippe Bernardy

NEURAL INFORMATION PROCESSING SYSTEMS

Background

Context. Dependent type theories are formal languages used for defining mathematical objects and reasoning about their properties. Dependently-typed programming languages equate proofs with programs, facilitating theorem proving and formal verification. Here's a tiny program in **Agda**, proving that the addition of naturals is commutative:

```
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)
```

Remark 1. Look at all the colors! 🌈
Remark 2. Is proving $(m + n \equiv n + \mathrm{m})$ any different to $(x + y \equiv y + x)$? 🤔

Motivation. If dependently-typed programs are proofs, and representing programs is essential to automating program synthesis, then ***representing dependently-typed programs*** is key to ***automating theorem proving*** (ATP).

Two major issues in the literature:

- **Resource Uniformity.**
  Many ATP models/resources/interfaces for Coq, Lean.
  None for Agda.
- **No Structural Fidelity.**
  Most ATP resources/frameworks today treat proofs as glorified text.
  Gone are all the colors. Names suddenly matter.

Contributions (tl;dr)

- **Machine Learning, for Agda.**
  We develop a package to faithfully extract the skeleton structure of dependently-typed program-proofs from type-checked Agda files. We apply the algorithm on Agda's public library ecosystem and release the result as a massive, highly elaborated ATP dataset.
- **Representation Learning for Dependent Types.**
  Capitalizing on this new resource, we present a representation learning model for expressions involving dependent types. Contra prior work, the model is structure-faithful, being invariant to α-renaming, superficial syntactic sweetening, scope permutation, irrelevant definitions, *etc.*

Dataset

Problem Generation. Left-to-right language modeling assumes proving is a linear process. Truth begs to differ; the statement below is valid syntax:

+-comm zero (suc n) = {!!}

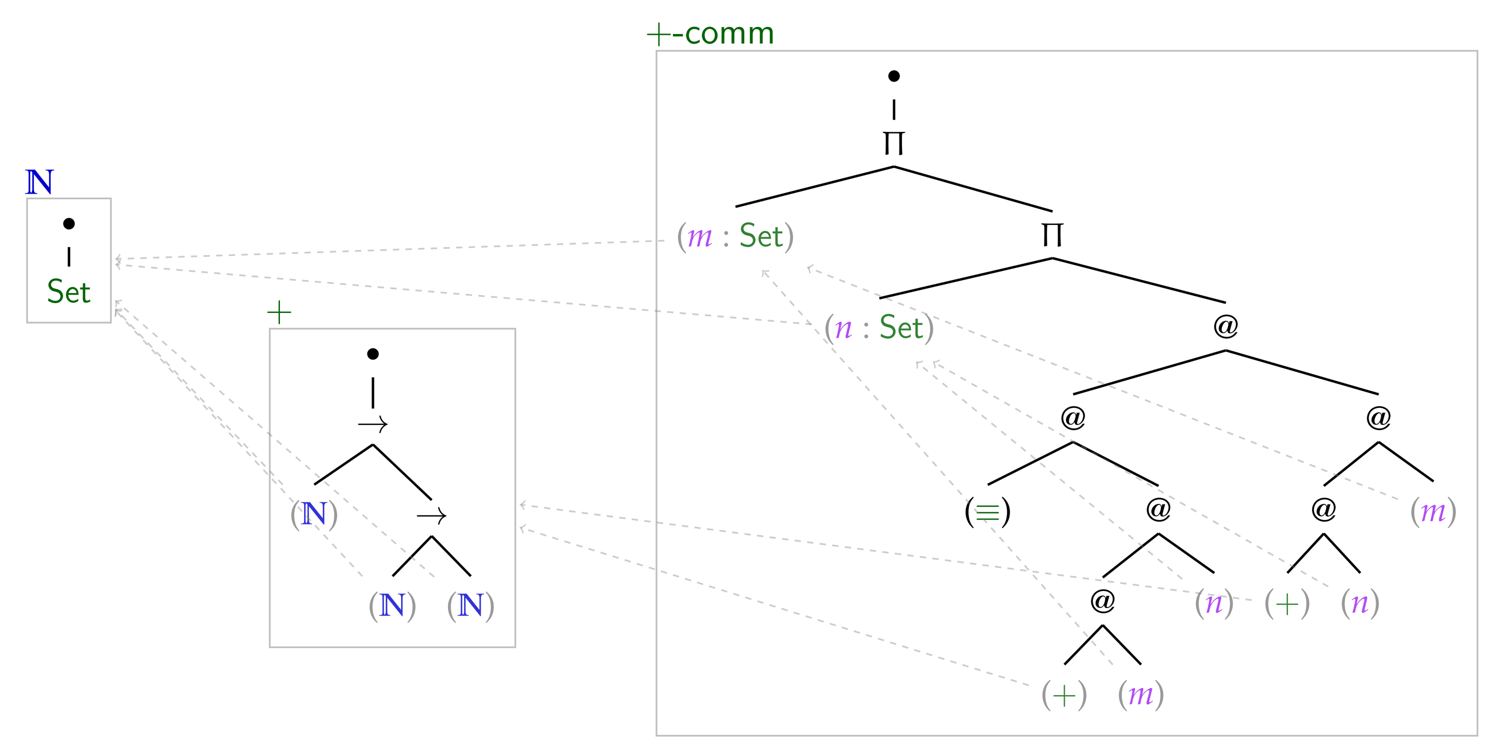
Remark 3. Proofs can have *holes*: unfilled parts deferred for later. 🧀

We use Agda's type-checker to find *all possible holes* in all written proofs. For each hole, we record the *goal type* (the type of the hole) and the *typing context* (all proven premises currently available). Ground truth corresponds to a selection (and arrangement) of the context (how to fill the hole).

Remark 4. Correct **premise selection** goes a long way towards ATP. 🧩

We export the extracted problems not *only* as **strings**, but *also* as **structures**. The export preserves and specifies all type information available to the checker, including **references** and **token structure at the subtype level**.

Post-tokenization, this is what the *types* of ℕ, + and +-comm look like:



Remark 5. Note the AST and referencing structures 🎄🎄🎄

Remark 6. Contrast with the tokenization of GPT-4o below. 📏

data		ℕ	:	Set	<newline>	_	+	_	:	ℕ	→		ℕ	<newline>	+-	comm	:	(	m	n	:		ℕ	)	→	m	+	n		≡	m	+	m	<newline>

Representation Learning

We build representations for lemmas and holes on the basis of their types.

Architecture. We use a fully-attentive bidirectional Transformer encoder, where full attention is restricted to tokens within the same type, augmented with various representational adjustments.

1. **Tree PE.** We use positional encodings that employ an inductive parameterization of the group structure of binary branching trees. These relieve the model from having to "parse" the type's symbolic sequentialization.
2. **Variable Binding.** We resolve nominal indexing, and represent variable references by the representation of the reference's path relative to the binder.
3. **Scope Referencing.** We organize lemmas into a POSET according to their dependency levels. We then build representations in dependency-sorted minibatches, and represent lemma references by the representations of their referents. (here: ℕ < + < +-comm)
4. **Efficient Attention.** We use linear attention combined with a Taylor-approximation of the exponential map to efficiently avoid the quadratic explosion – without losing expressivity.

Training. We train with infoNCE in a premise selection setup using a subset of Agda's standard library, and evaluate in proximal and distant domains.

Model	Average and 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**
- (4)	47.0 / 36.2	37.1 / 29.2	26.8 / 17.0	21.4 / 14.4
- (1)	44.5 / 34.1	30.7 / 24.0	24.8 / 15.5	18.8 / 12.3
- (2)	35.8 / 25.9	25.5 / 19.1	19.7 / 11.6	17.7 / 11.0
Transformer	10.9 / 3.7	8.5 / 4.5	9.4 / 3.9	5.8 / 0.9

Remark 7. Structural adjustments » architectural adjustments. ⚖️

Learn more

For more details, take a look at:

- agda.readthedocs.io
  for an intro to Agda
- github.com/omelkonian/agda2train
  for the proof extraction code
- github.com/konstantinosKokos/quill
  for the Python interface and neural engine
- arxiv.org/abs/2402.02104
  for prose, figures, tables with numbers, etc.

cost EUROPEAN COOPERATION IN SCIENCE & TECHNOLOGY

Funded by the European Union