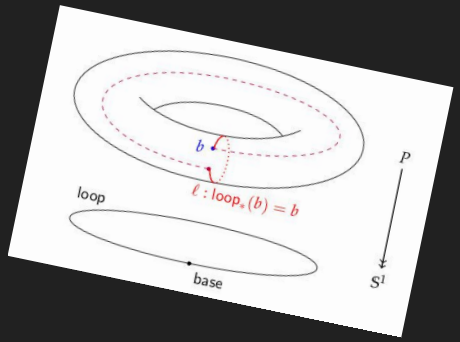
$$\frac{p \quad p \to q}{\therefore q}$$

$$\lambda x.\, x\, y$$



Proofs and Programs Club 2024-25

Chris Purdy

Welcome to PaPC!

The main focus of this club is learning the **Agda** proof assistant.

Along the way we will learn some basics of functional programming, formal logic and type theory.

We meet every week on **Mondays** at **6pm-7/7:30pm**.

Join the Discord to ask questions and see guides for getting started with Agda:

https://discord.gg/8nnqTKCTdK

Agda and other proof assistants

Agda (Norell, U. (2009)) is a *dependently typed* programming language, that can be used as a *proof assistant* (software that assists the user in writing and checking mathematical proofs).

Other proof assistants include Coq, Idris, Lean, Metamath, Isabelle, NuPRL, etc.

The primary field concerned with development of proof assistants and dependently typed programming languages is **type theory**.

Agda and other proof assistants

Proof assistants and type theory are of interest to both computer scientists and mathematicians.

For **computer scientists**:

- Proving correctness of algorithms and protocols
- Correct-by-construction programming

For **mathematicians**:

- Computer-assisted formalisation of proofs and constructions
- A common language for (a large portion of) mathematics
- A more refined notion of equality

Career prospects and summer projects (UROP)

Functional programming, proof assistants and formal methods in general are used widely in industry and academia:

- **Meta** - much of Facebook's abuse detection system Sigma is written in Haskell
- **CompCert** - a C++ compiler that has been formally verified correct using Coq
- **The 4-colour theorem** - a famous and very complicated theorem from graph theory. The proof of this has been formalised in Coq

Learning a bit about these areas is great if you want to write software that is bug-free and/or provably correct.

Career prospects and summer projects (UROP)

Programming language theory, type theory, proof theory and formal logic (areas of theoretical computer science related to Agda and proof assistants) are all active fields of research!

This is what I study for my PhD, and there are plenty of other PhD opportunities out there related to these topics.

If you find that you enjoy learning Agda and related stuff, you may want to talk to us about doing a **UROP summer project**...

Getting started with Agda

Agda has been installed on NoMachine, along with the `agda-mode` VSCode extension.

A template working directory (containing files *required for Agda to work*), and tutorial files are stored in the shared directory:

`/CS/extracurricular/proofsPrograms`

A detailed walkthrough is on the **#announcements** channel of the PaPC Discord.

Agda - programming with natural numbers

```
data Nat : Set where
  zero : Nat
  suc  : Nat → Nat

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

three : Nat
three = suc zero + suc (suc zero)
```

There is a lot going on here:

- `Nat` is the *type* of natural numbers
- Colon (`:`) means “is of type”
- `Set` is the *type* of “all” types
- `zero` and `suc` are *constructors* for `Nat`
- `_+_` is the addition *function* on natural numbers. It is *recursive*.

Comparison to other languages

Agda

```
_+_ : Nat → Nat → Nat
zero    + m = m
(suc n) + m = suc (n + m)
```

Python

```
def plus(n, m):
  if n == 0:
    return m
  else
    return 1 + plus (n - 1, m)
```

Maths

$$0 + m := m$$

$$(1 + n) + m := 1 + (n + m)$$

Types and constructors

```
data Nat : Set where
  zero : Nat
  suc  : Nat → Nat

three : Nat
three = suc (suc (suc zero))
```

One way of defining types in Agda is using the **data** construct.

This lets you specify a type by defining how to **build its elements**.

You can “build” any natural number n by starting with 0, and adding 1, n times.

A “real life” Agda proof

$$\text{Sum from 1 to n} = \frac{n(n+1)}{2}$$

```
prove2*sumn=n*sucn : (n : ℕ) → (sum n * 2) ≡ (n * (1 + n))
prove2*sumn=n*sucn zero = refl
prove2*sumn=n*sucn (suc n) = cong 2+ $ begin
  (n + sum n) * 2               ≡⟨ *-distribr n (sum n) 2 ⟩
  (n * 2) + (sum n * 2)         ≡⟨ cong (_+ (sum n * 2)) (trans (*-comm n 2) (sym (times2 n)))  ⟩
  (n + n) + (sum n * 2)         ≡⟨ cong ((n + n) +_) (prove2*sumn=n*sucn n) ⟩ -- inductive hypothesis
  (n + n) + n * (1 + n)         ≡⟨ cong ((n + n) +_) (distribl* n 1 n) ⟩
  (n + n) + ((n * 1) + (n * n)) ≡⟨ cong ((n + n) +_) (cong (_+ (n * n)) (*rid n)) ⟩
  (n + n) + (n + (n * n))       ≡⟨ trans (sym (+-asso n n (n + (n * n)))) (cong (n +_) (+-asso n n (n * n))) ⟩
  n + ((n + n) + (n * n))       ≡⟨ cong (n +_) (cong (_+ (n * n)) (trans (times2 n) (*-comm 2 n))) ⟩
  n + ((n * 2) + (n * n))       ≡⟨ sym (cong (n +_) (distribl* n 2 n))  ⟩
  n + n * (2 + n)               ∎
```

A *real life* Agda proof

```
square-2 : η (⊗-homo M) (X ⊗₀ A (T Tx) , unit ⊗₀ A (T T1)) ∘ (id ⊗₁ α (T T1) ∘ id ⊗₁ fix-ε ∘ unitorʳ.to)
square-2 = sym-assoc ○ glue▹□ top-tri (glue (⟺ (glue T-ε-sub-square unitʳ-sub-square)) (⟺ pentagonʳ))
  where
    unitaryʳM : F₁ (F M) unitorʳ.from ∘ (η (⊗-homo M) (X ⊗₀ A (T Tx) , unit)) ∘ id ⊗₁ ε M ≈ unitorʳ.from
    unitaryʳM = unitaryʳ M

    top-tri : (η (⊗-homo M) (X ⊗₀ A (T Tx) , unit) ∘ id ⊗₁ ε M) ∘ unitorʳ.to ≈ F₁ (F M) unitorʳ.to
    top-tri = ∘-resp-≈ˡ (⟺ identityˡ ○ ∘-resp-≈ˡ (⟺ (⟺ (homomorphism (F M)) ○ F-resp-≈ (F M) unitor

    unitʳ-sub-square : F₁ (F M) (id ⊗₁ unitorˡ.to {unit}) ∘ η (⊗-homo M) (X ⊗₀ A (T Tx) , unit) ≈ η (⊗-h
    unitʳ-sub-square = ∘-resp-≈ˡ (F-resp-≈ (F M) (⊗-resp-≈ʳ unitor-coherence)) ○ sym-commute (⊗-homo M)

    T-ε-sub-square : (F₁ (F M) (id ⊗₁ (id ⊗₁ fix-ε)))∘ η (⊗-homo M) (X ⊗₀ A (T Tx) , unit ⊗₀ unit) ≈ η (
    T-ε-sub-square = sym-commute (⊗-homo M) (id {X ⊗₀ A (T Tx)} , (id ⊗₁ fix-ε)) ○ ∘-resp-≈ʳ (⊗-resp-≈ˡ

    pentagonʳ : (id {F₀ (F M) (X ⊗₀ A (T Tx))} ⊗₁ (F₁ (F M) (id ⊗₁ fix-ε)) ∘ id ⊗₁ (F₁ (F M) unitorʳ.to)
    pentagonʳ = ⊗-parallel-coalg monoidal (∘-resp-≈ˡ identity²) (assoc ○ commutes (! T1 {unit-coalgʳ}))

square-3 : F₁ (F M) (id {X} ⊗₁ (id {A (T Tx)} ⊗₁ fix-ε)) ∘ F₁ (F M) (id {X} ⊗₁ unitorʳ.to) ≈ (F₁ (F M) (
square-3 = ⟺ (sym-assoc ○ ∘-resp-≈ˡ unitʳ-iso-sub-triangle) ○ (∘-resp-≈ʳ (glue bottom-square middle-squ
  where
    bottom-square : F₁ (F M) (id {X ⊗₀ unit} ⊗₁ (id {A (T Tx)} ⊗₁ fix-ε)) ∘ F₁ (F M) swapInner.from ≈ F₁
    bottom-square = ⟺ (F-lemma {G = F M} (∘-resp-≈ʳ (⊗-resp-≈ˡ (⟺ (identity ⊗))) ○ natural ○ ∘-resp-≈

    middle-square : F₁ (F M) (unitorʳ.to ⊗₁ id {A (T Tx) ⊗₀ unit}) ∘ F₁ (F M) (id {X} ⊗₁ unitorʳ.to) ≈ F
    middle-square = F-lemma {G = F M} (⟺ serialize₁₂ ○ flipʳ unitorʳ.isoʳ (⟺ (flipˡ (⟺ ⊗-distrib-ove

    unitʳ-iso-sub-triangle : F₁ (F M) (unitorʳ.from ⊗₁ id {A (T Tx) ⊗₀ A (T T1)}) ∘ (F₁ (F M) (id {X ⊗₀
    unitʳ-iso-sub-triangle = ⟺ (⟺ identityˡ ○ ∘-resp-≈ˡ (⟺ (identity (F M)) ○ F-resp-≈ (F M) (⟺ (i
```

This is about **5%** of the whole proof