Important info

- All information on Canvas
- Demonstrator: Steven Cheung
- Assessment: 50% CA + 50% Exam
- CA = weekly exercise sheets (formative) + 1 class test (summative)
- Daily drop-in lab 5-6pm

Weekly syllabus (tentative)

1. Introduction
2. Induction
3. Equality
4. Agda automation
5. Existential types
6. Vectors and finite sets
7. Dependent equality (Martin Escardó)
8. Record and co-patterns (Noam Zeilberger)
9. Heterogeneous data types
10. Class test no lectures (covering weeks 1-5)
11. Univalent maths (Benedikt Ahrens)

The Curry-Howard Isomorphism

AFP :: W1 :: L1

Dan R. Ghica

A true isomorphism

structure-preserving

propositions are types
proofs are programs
simplification is evaluation

Two unrelated (?) discoveries (inventions) in the 1930s

Alonzo Church
The lambda-calculus

Gerhard Gentzen
Natural deduction

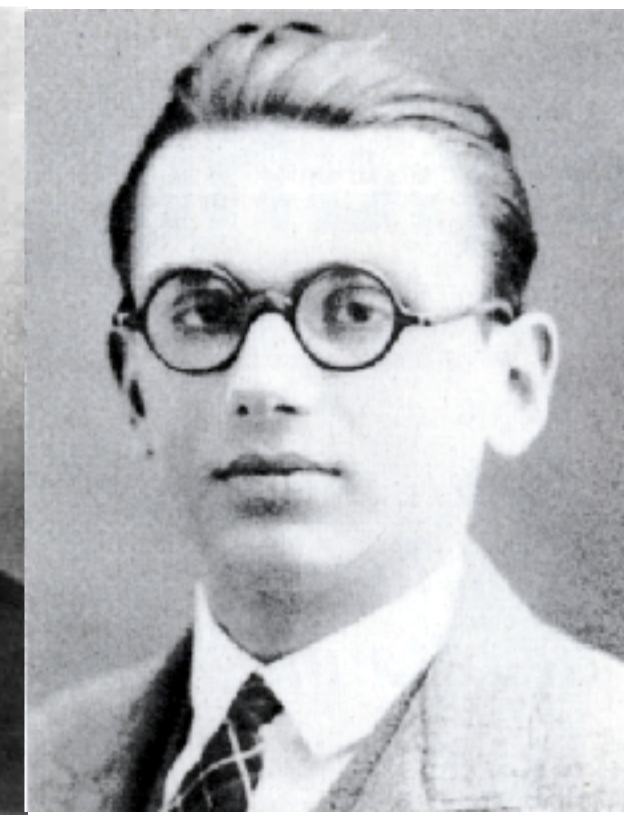
Church: Computation

From maths to computation via logic

Aristotle

Russell

Hilbert



Gödel

But what does it mean “to calculate”

al Khwarizmi

Church

Gödel

Turing

Gentzen:
Logic and consistency

$$\exists x. P(x)$$

$$\forall x. P(x)$$

$$P \mathbin{\&} Q$$

$$P \lor Q$$

$$P \supset Q$$

$$\bot$$

$$\begin{tikzcd}
\text{Natural Deduction} \arrow[r, bend left] & \text{Sequent Calculus} \arrow[l, bend left]
\end{tikzcd}$$

Natural deduction

$$\dfrac{\begin{matrix}\vdots & & \vdots\\ A & & B\end{matrix}}{A\ \&\ B} \qquad \dfrac{\begin{matrix}\vdots\\ A\ \&\ B\end{matrix}}{A} \qquad \dfrac{\begin{matrix}\vdots\\ A\ \&\ B\end{matrix}}{B}$$

Sequent calculus

$$\dfrac{\Gamma\vdash A \qquad \Gamma\vdash B}{\Gamma\vdash A\ \&\ B} \qquad \dfrac{\Gamma\vdash A\ \&\ B}{\Gamma\vdash A} \qquad \dfrac{\Gamma\vdash A\ \&\ B}{\Gamma\vdash B}$$

Proof of consistency

- rules come in pairs: introduction and elimination
- key results: normalisation and sub-formula property
- consistency follows

Haskell Curry: Propositions as types

$$f : A \to B$$

can be read as

$$A \supset B$$

"If a function has an argument of type A then it produces a result of type B.

propositions ⇔ types

functions ⇔ proofs

`A & B ⇔ 'a * 'b`

$$\dfrac{\begin{matrix}\vdots \\ A\end{matrix} \quad \begin{matrix}\vdots \\ B\end{matrix}}{A \,\&\, B} \qquad \dfrac{\begin{matrix}\vdots \\ A \,\&\, B\end{matrix}}{A} \qquad \dfrac{\begin{matrix}\vdots \\ A \,\&\, B\end{matrix}}{B}$$

$$\dfrac{\texttt{p : 'a} \quad \texttt{q : 'b}}{\texttt{(p, q) : 'a * 'b}} \qquad \dfrac{\texttt{p : 'a * 'b}}{\texttt{? : 'a}} \qquad \dfrac{\texttt{p : 'a * 'b}}{\texttt{? : 'b}}$$

$A \,\&\, B \Leftrightarrow$ 'a * 'b

$$\frac{\begin{matrix}\vdots & \vdots\\ A & B\end{matrix}}{A \,\&\, B} \qquad \frac{\begin{matrix}\vdots\\ A \,\&\, B\end{matrix}}{A} \qquad \frac{\begin{matrix}\vdots\\ A \,\&\, B\end{matrix}}{B}$$

$$\frac{\texttt{p : 'a} \quad \texttt{q : 'b}}{\texttt{(p, q) : 'a * 'b}} \qquad \frac{\texttt{p : 'a * 'b}}{\texttt{fst p : 'a}} \qquad \frac{\texttt{p : 'a * 'b}}{\texttt{snd p : 'b}}$$

```
# let p = 1;;
val p : int = 1
# let q = "hi";;
val q : string = "hi"
# (p, q);;
- : int * string = (1, "hi")
```

```
# let p = (3.14, 'z');;
val p : float * char = (3.14, 'z')
# fst p;;
- : float = 3.14
# snd p;;
- : char = 'z'
```

```
A v B ⇔ ('a,'b) v where
type ('a, 'b) v = L of 'a | R of 'b
```

$$\dfrac{\begin{array}{c}\vdots\\ A\end{array}}{A \lor B} \qquad \dfrac{\begin{array}{c}\vdots\\ B\end{array}}{A \lor B} \qquad \dfrac{\texttt{p : 'a}}{\texttt{? : ('a , 'b)v}}$$

A v B ⇔ ('a,'b) v where

```
type ('a, 'b) v = L of 'a | R of 'b
```

$$\dfrac{\dfrac{\vdots}{A}}{A \lor B} \qquad \dfrac{\dfrac{\vdots}{B}}{A \lor B}$$

$$\dfrac{p : \text{'a}}{L\ p : (\text{'a}\ ,\ \text{'b})v} \qquad \dfrac{p : \text{'b}}{R\ p : (\text{'a}\ ,\ \text{'b})v}$$

```
# type ('a, 'b) v = L of 'a | R of 'b ;;
type ('a, 'b) v = L of 'a | R of 'b
# let p = "hello!";;
val p : string = "hello!"
# L p;;
- : (string, 'a) v = L "hello!"
# R p;;
- : ('a, string) v = R "hello!"
```

$$A \vee B \Leftrightarrow (\text{'a},\text{'b})\ \text{v where}$$

```
type ('a, 'b) v = L of 'a | R of 'b
```

$$\dfrac{\begin{array}{c}\vdots \\ A \vee B \qquad A \supset C \qquad B \supset C\end{array}}{C}$$

$$\dfrac{p : (\text{'a},\text{'b})\text{v} \qquad q : \text{'a} \to \text{'c} \qquad r : \text{'b} \to \text{'c}}{? : \text{'c}}$$

$$A \vee B \Leftrightarrow (\text{'a},\text{'b})\ v \text{ where}$$

```
type ('a, 'b) v = L of 'a | R of 'b
```

$$\frac{A \vee B \qquad A \supset C \qquad B \supset C}{C}$$

$$\frac{p : (\text{'a},\text{'b})v \qquad q : \text{'a} \to \text{'c} \qquad r : \text{'b} \to \text{'c}}{\text{match } p \text{ with } L\ x \to q\ x \mid R\ x \to r\ x : \text{'c}}$$

```
# let p = L 1;;
val p : (int, 'a) v = L 1
# let q = string_of_int;;
val q : int → string = <fun>
# let r = string_of_float;;
val r : float → string = <fun>
# match p with L x → q x | R x → r x ;;
- : string = "1"
# let p = R 2.3;;
val p : ('a, float) v = R 2.3
# match p with L x → q x | R x → r x ;;
- : string = "2.3"
```

$$A \supset B \iff \text{'a} \to \text{'b}$$

$$\dfrac{\begin{matrix}[A] \\ \vdots \\ B\end{matrix}}{A \supset B} \qquad\qquad \dfrac{A \supset B \quad A}{B}$$

$$\dfrac{p : \text{'b}}{\texttt{fun (x:'a)} \to p : \text{'a} \to \text{'b}} \qquad \dfrac{p : \text{'a} \to \text{'b} \quad q : \text{'a}}{p\ q : \text{'b}}$$

```
# let x = 1;;
val x : int = 1
# (string_of_int x) ^ "!";;
- : string = "1!"
# fun x → (string_of_int x) ^ "!";;
- : int → string = <fun>
```

```
# let p = string_of_int;;
val p : int → string = <fun>
# let q = 7;;
val q : int = 7
# p q;;
- : string = "7"
```

⊥ ⇔ empty where

type empty

$$\frac{\perp}{A}$$

ex nihilo
quodlibet

$$\frac{\texttt{p : empty}}{\texttt{? : 'a}}$$

$\perp \Leftrightarrow$ `empty` where

`type empty`

$$\frac{\perp}{A}$$

$$\frac{\texttt{p : empty}}{\texttt{(match x with []) p : 'a}}$$

not possible in OCaml, Haskell
possible in camlp4, Agda

$\perp \Leftrightarrow$ exn

$$\frac{\perp}{A}$$

raise: exn → 'a

```
utop # exception E;;
exception E
utop # let f = function
        | [] → 0
        | x :: xs → raise E
;;
val f : 'a list → int = <fun>
```

$\perp \Leftrightarrow$ exn

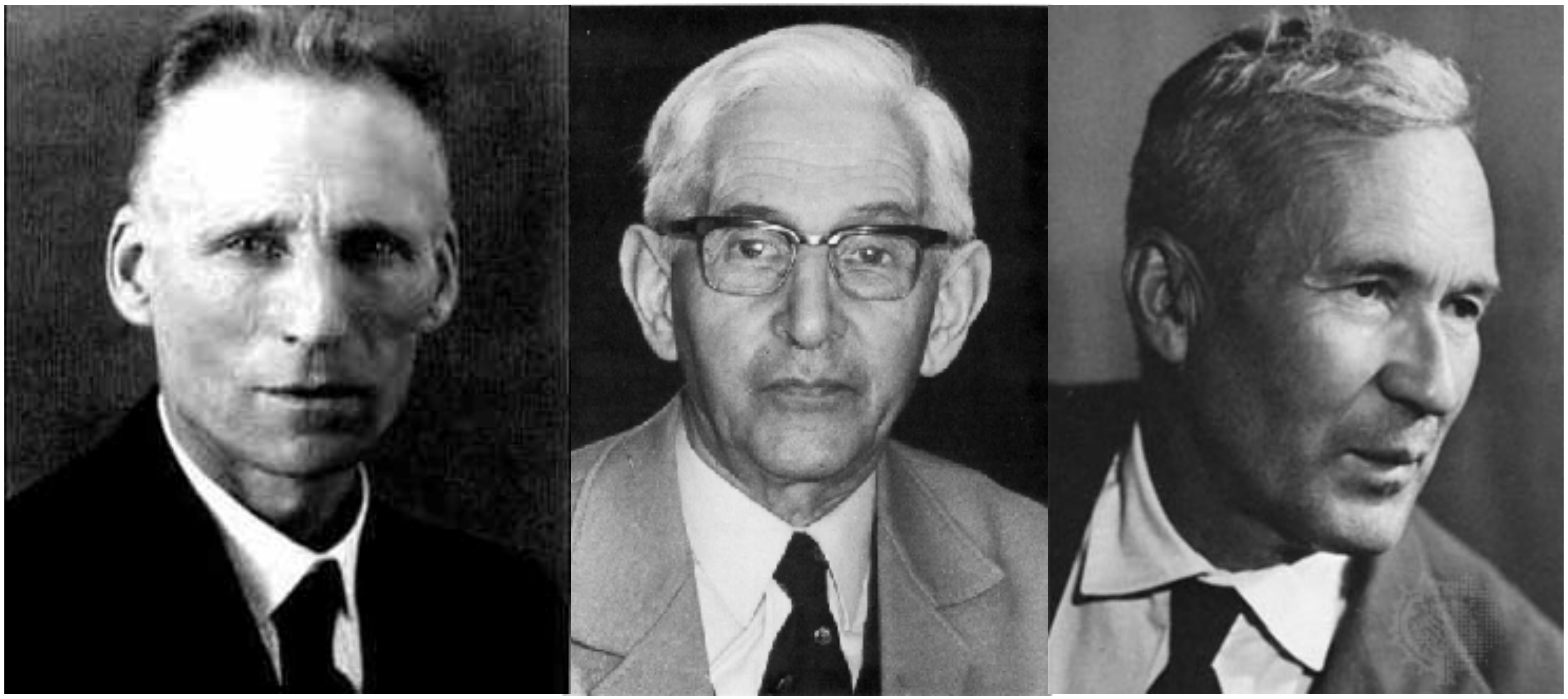
principium
contradictiones /
reductio ad absurdum

$$\frac{(A \supset \perp) \supset \perp}{A}$$

$$\frac{p : (\text{'a} \rightarrow \text{exn}) \rightarrow \text{exn}}{? : \text{'a}}$$

impossible (in general)
“constructive” vs.
“classical”

“True” / “constructively provable”



Brouwer Heyting Kolmogorov

- A proof of $P \,\&\, Q$ is a pair <a, b> where a is a proof of P and b is a proof of Q.
- A proof of $P \vee Q$ is a tagged pair <a, b> where
  a is 0, b a proof of P,
  or a is 1, b a proof of Q.
- A proof of $P \supset Q$ is a function f that converts a proof of P into a proof of Q.
- There is no proof of $\perp$ (the absurdity).
- The formula $\neg P$ is defined as $P \rightarrow \perp$.

Double negation introduction
Triple negation elimination

$$A \supset ((A \supset \bot) \supset \bot) \quad = \quad A \supset \neg\neg A$$

$$(((A \supset \bot) \supset \bot) \supset \bot) \supset (A \supset \neg) \quad = \quad (\neg\neg\neg A) \supset (\neg A)$$

Valid constructive laws.

DNI // TNE

$$A \supset ((A \supset \bot) \supset \bot) \quad = \quad A \supset \neg\neg A$$

$$(((A \supset \bot) \supset \bot) \supset \bot) \supset (A \supset \neg) \quad = \quad (\neg\neg\neg A) \supset (\neg A)$$

```
# let dni a f = f a;;
val dni : 'a → ('a → 'b) → 'b = <fun>
# let tne f a = f (dni a);;
val tne : ((('a → 'b) → 'b) → 'c) → 'a → 'c = <fun>
```

Valid constructive laws.

Exercise DNE⟺LEM

This is not the same A!

Assuming axiom ¬¬A⊃A can we prove A∨¬A ?

Assuming axiom A∨¬A can we prove ¬¬A⊃A ?

Caveat!
OCaml type system is unsound

$\boxed{A \supset B}$

Caveat!
OCaml type system is unsound

$\boxed{A \supset B}$

```
utop # let rec unsound x = unsound x;;
val unsound : 'a → 'b = <fun>
```

or

```
utop # let rec unsound _ = raise E;;
val unsound : 'a → 'b = <fun>
```

(also Haskell etc)

The simple type system is meant primarily as a protection against self-reference. Recursion defeats that!

William Howard

Proof simplification ⇔ Computation

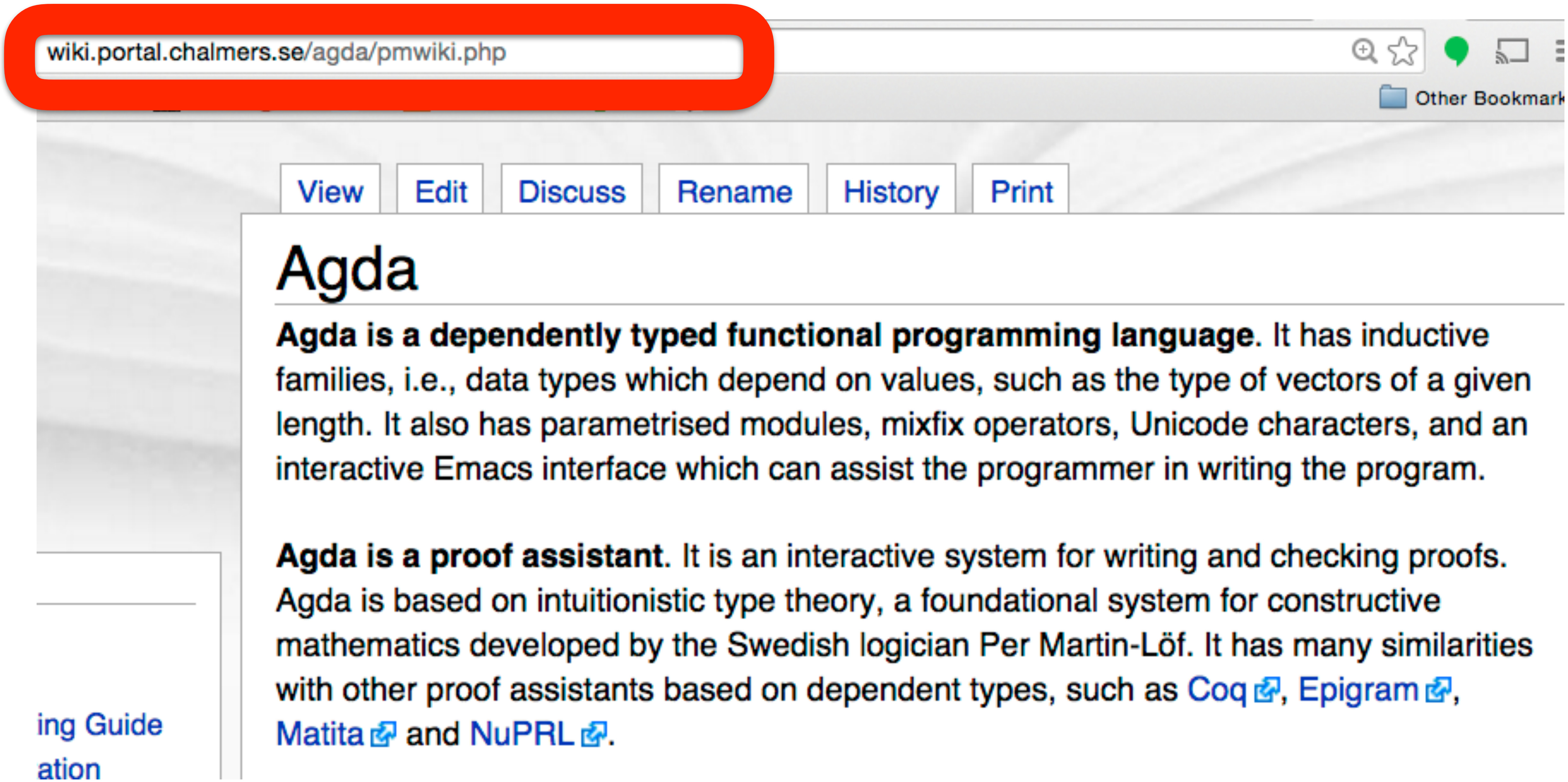
$$\dfrac{\dfrac{\begin{matrix}[A]\\ \vdots\\ B\end{matrix}}{A \supset B} \qquad A}{B} \quad \Rightarrow \quad \begin{matrix}A\\ \vdots\\ B\end{matrix}$$

$$\begin{matrix}(\mathtt{fun}\ x \to f\ x)\ a\\ \Downarrow\\ f\ a\end{matrix}$$

Why is this important

- the key design principle behind “functional” languages
- the key idea in modern PL theory (and design)
- type safety propaganda: “well typed programs don't go wrong”
- behind OCaml, Haskell, Coq, Agda, Rust

Agda



wiki.portal.chalmers.se/agda/pmwiki.php

Other Bookmark

View Edit Discuss Rename History Print

Agda

Agda is a dependently typed functional programming language. It has inductive families, i.e., data types which depend on values, such as the type of vectors of a given length. It also has parametrised modules, mixfix operators, Unicode characters, and an interactive Emacs interface which can assist the programmer in writing the program.

Agda is a proof assistant. It is an interactive system for writing and checking proofs. Agda is based on intuitionistic type theory, a foundational system for constructive mathematics developed by the Swedish logician Per Martin-Löf. It has many similarities with other proof assistants based on dependent types, such as Coq, Epigram, Matita and NuPRL.

ing Guide
ation

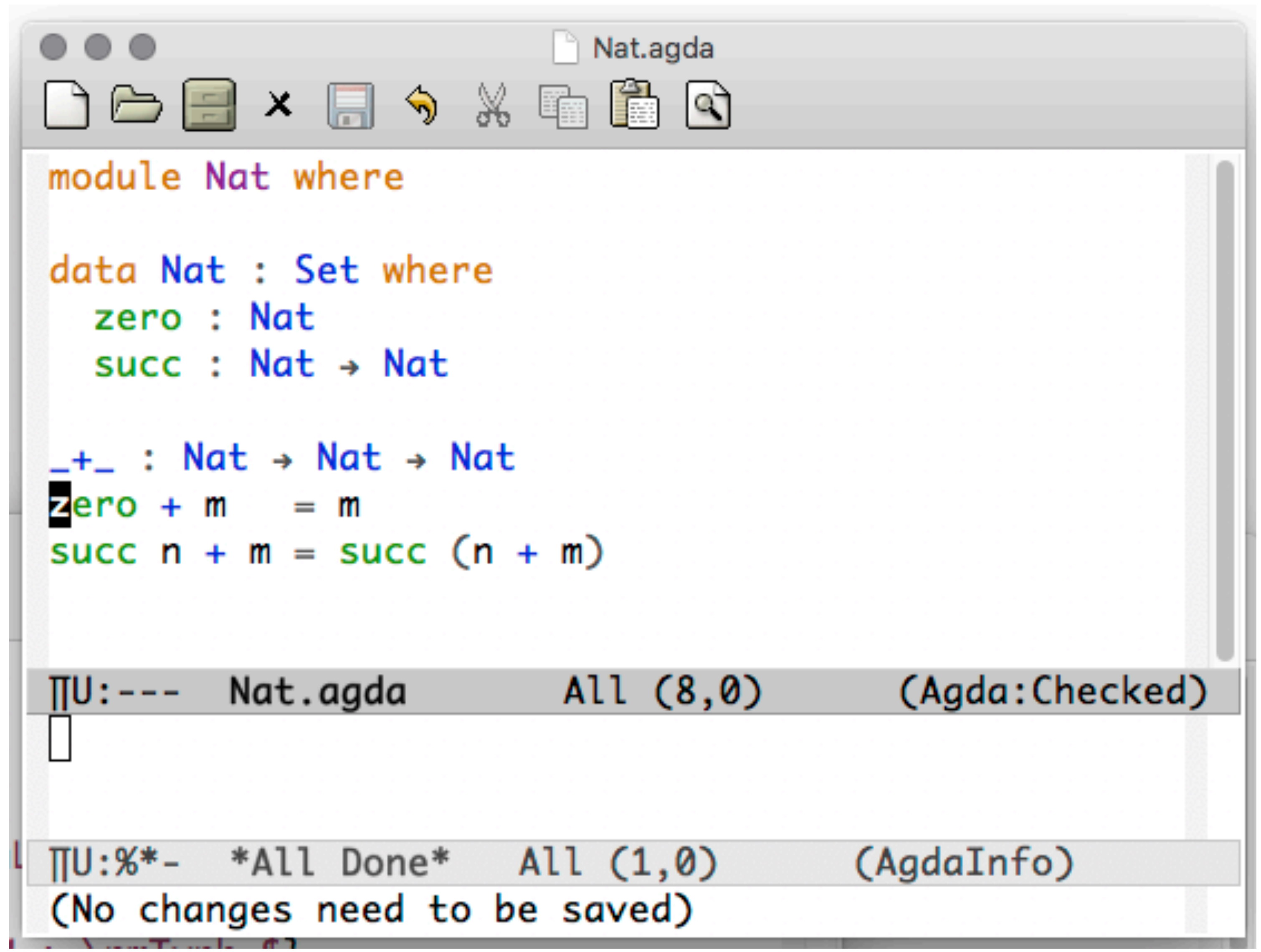
A quick introduction to Agda (I)

module definition
data-type definition
constructors

(infix) function definition
pattern matching

Emacs mode for Agda

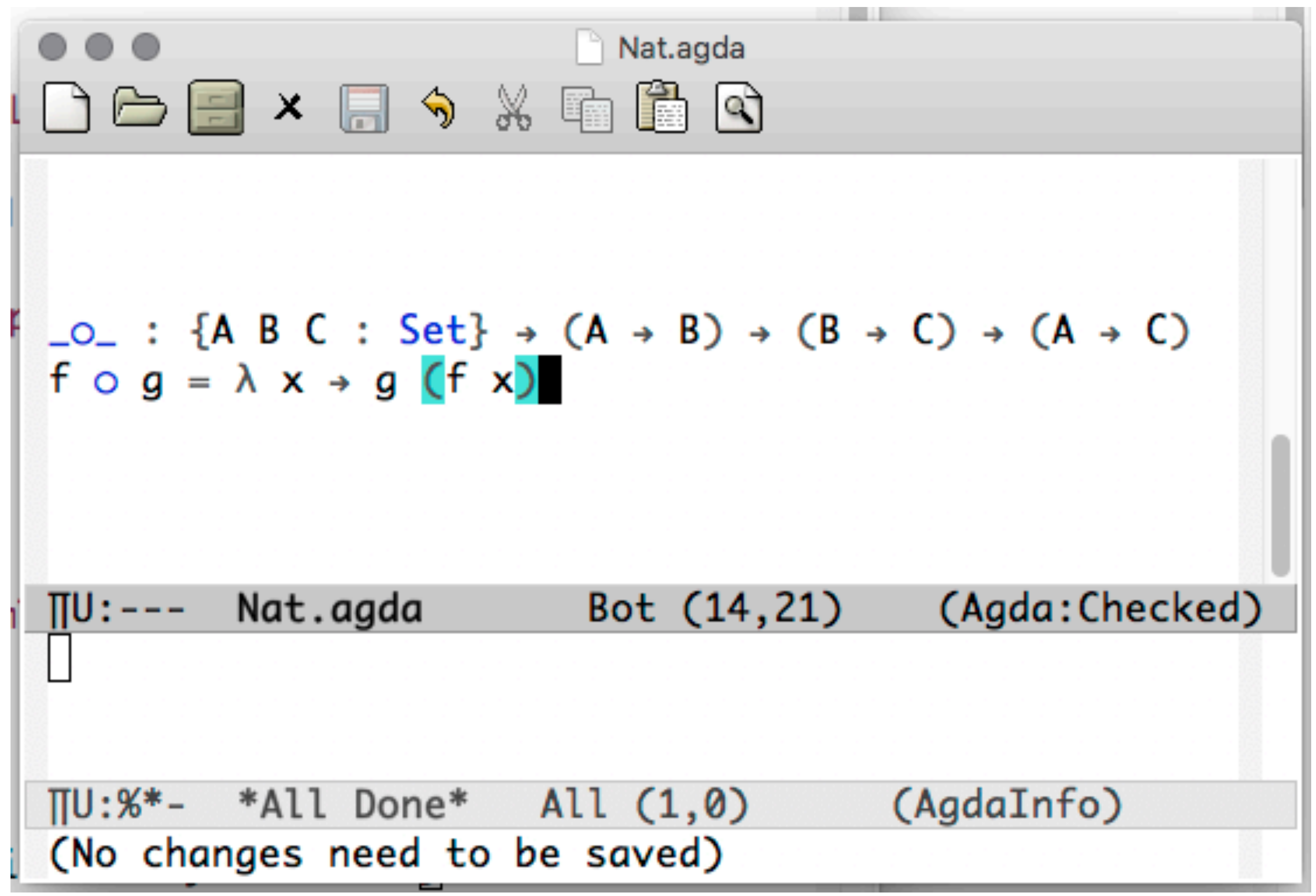
User feedback



To type-check a program : `ctrl+L`

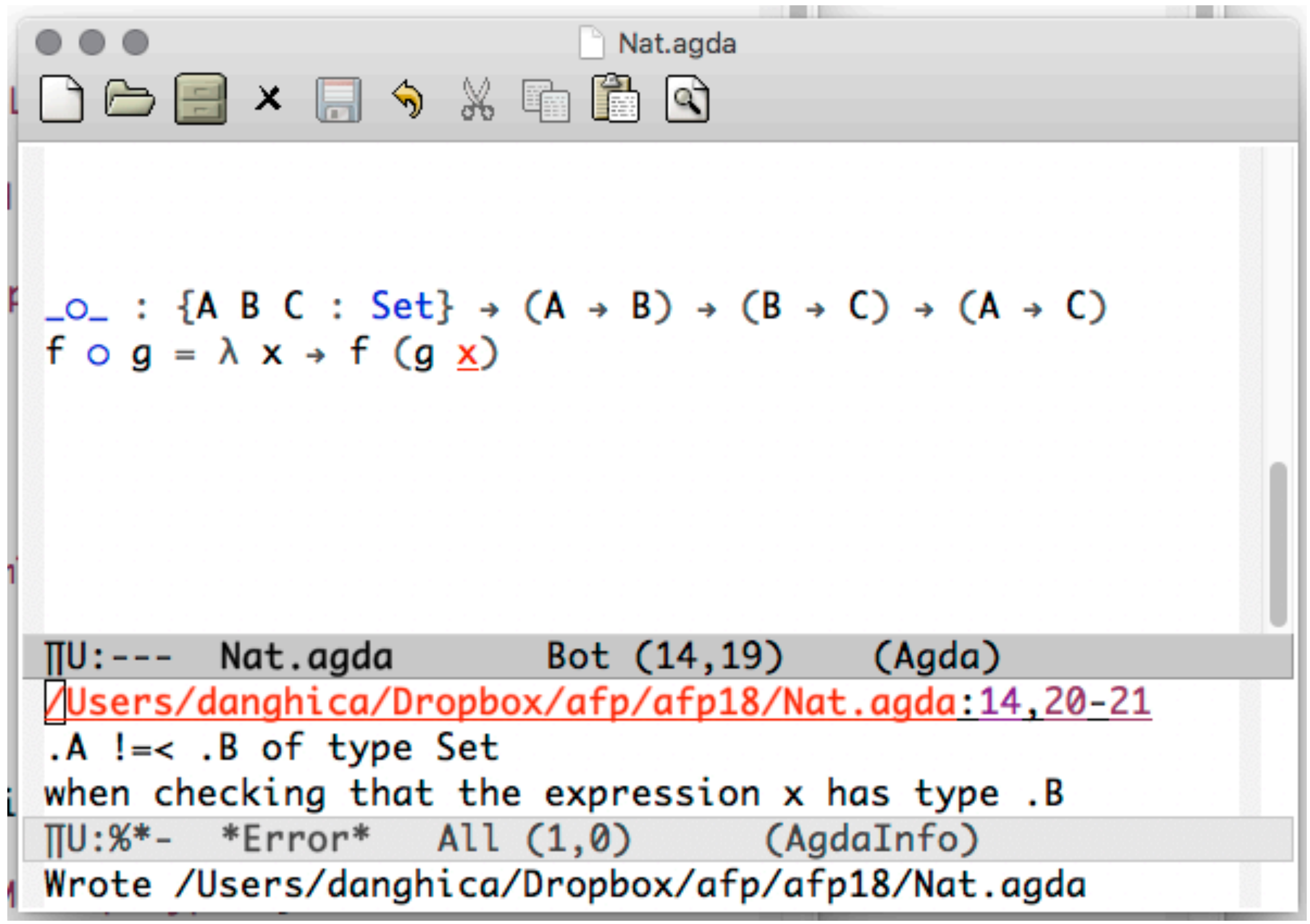
A quick introduction to Agda (II)

polymorphic function
implicit arguments
lambda expressions
unicode identifiers



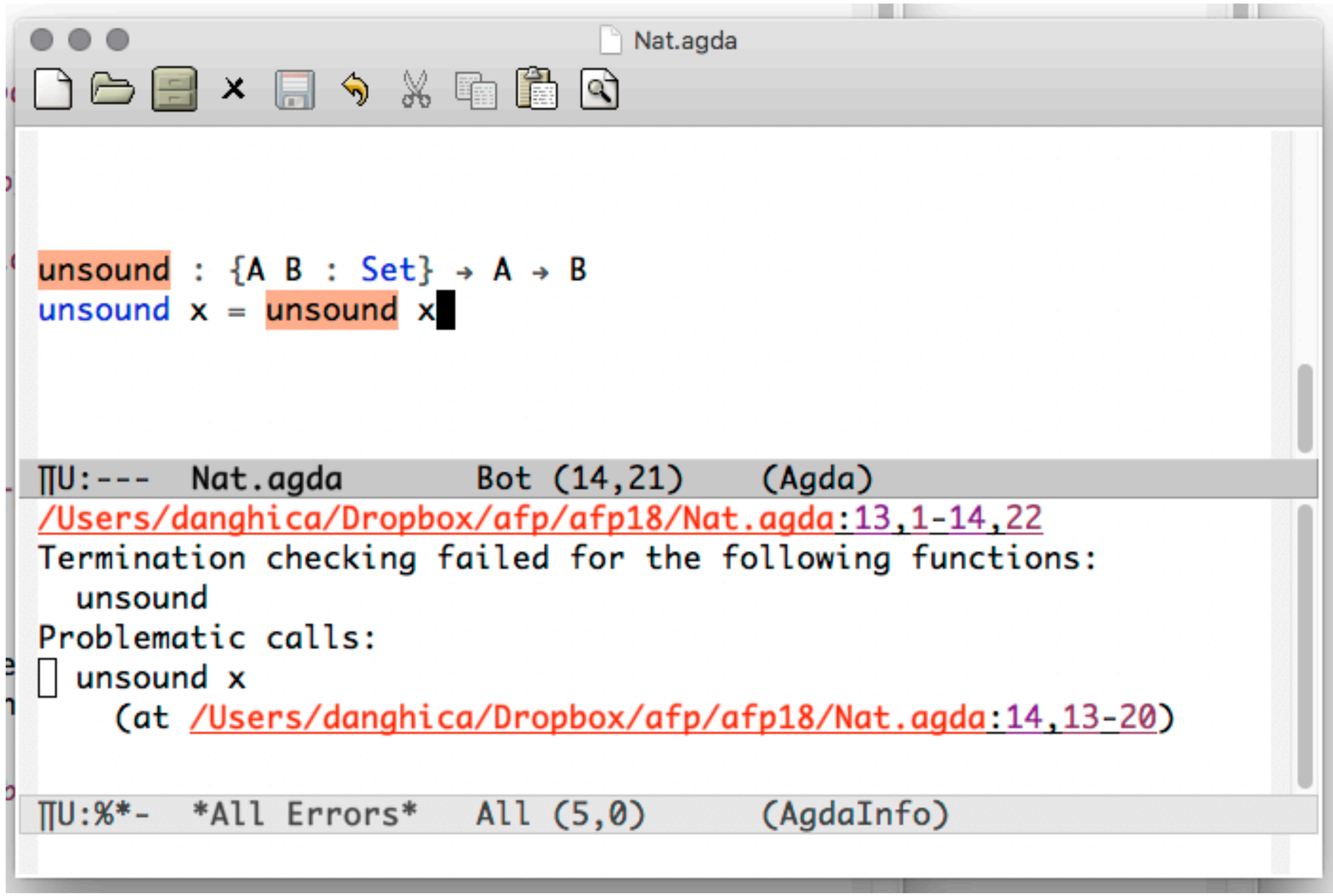
A quick introduction to Agda (III)

type error



A quick introduction to Agda (IV)

termination error



Proving Hilbert's system in Agda

https://www.dropbox.com/s/w5phun4ldnx263a/Hilbert.agda.html?dl=0

- precedence of infix operators
- polymorphic data types
- type-valued functions
- the empty pattern
- 'postulate'

Lab exercise sheet

Prove constructively if possible, classically otherwise:

- Various Hilbert-style axioms
  https://en.wikipedia.org/wiki/List_of_Hilbert_systems
- De Morgan's laws (constructively 3/4)
- DNI and TNE laws
- Equivalence of DNE and LEM
- Further equivalence with Peirce's law
  https://en.wikipedia.org/wiki/Peirce%27s_law

Lab marking scheme

- attendance ... at least 4/10
- completing any easy exercise … at least 6/10
- completing any hard exercise … at least 8/10
- doing something special … 10/10

Note: Lab marks are formative only.