Lambda, The Ultimate TA (Agda version)

Philip Wadler
(with Wen Kokke and Jeremy Siek)
University of Edinburgh / IOHK
Collège de France, Paris, 12 September 2019

Lambda, the Ultimate TA

Lambda,
The Ultimate TA

Using a Proof Assistant to Teach Programming Language Foundations

ICFP 2009

Benjamin C. Pierce
University of Pennsylvania

automated proof assistant

=

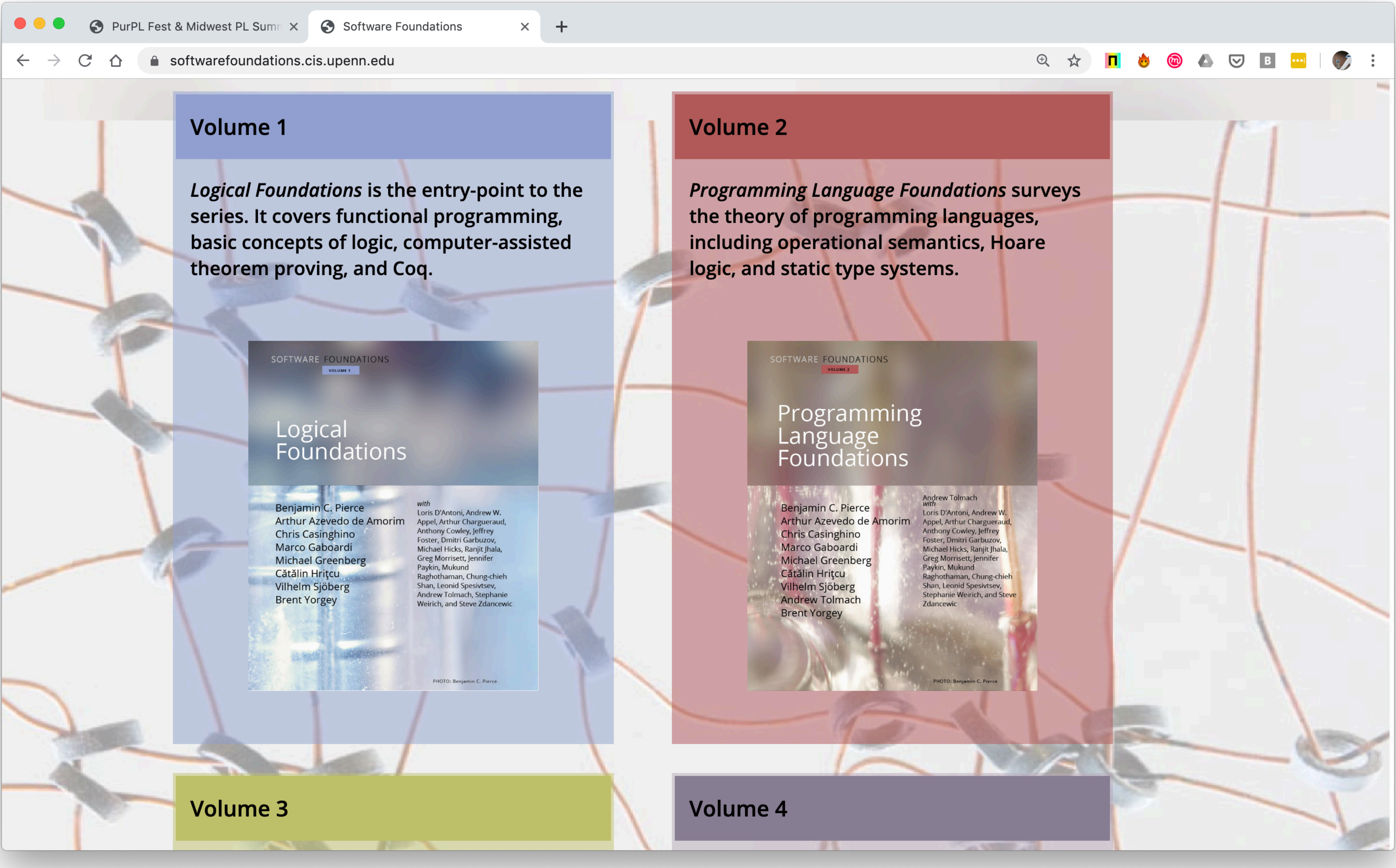
one TA per student

Oops, forgot one thing...

There is one small catch...

- Making up lectures and homeworks takes between one and two orders of magnitude more work for the instructor than a paper-and-pencil presentation of the same material!

Programming Language Foundations in Agda



Volume 1

Logical Foundations is the entry-point to the series. It covers functional programming, basic concepts of logic, computer-assisted theorem proving, and Coq.

SOFTWARE FOUNDATIONS
VOLUME 1

Logical Foundations

Benjamin C. Pierce
Arthur Azevedo de Amorim
Chris Casinghino
Marco Gaboardi
Michael Greenberg
Cătălin Hriţcu
Vilhelm Sjöberg
Brent Yorgey

with
Loris D'Antoni, Andrew W. Appel, Arthur Chargueraud, Anthony Cowley, Jeffrey Foster, Dmitri Garbuzov, Michael Hicks, Ranjit Jhala, Greg Morrisett, Jennifer Paykin, Mukund Raghothaman, Chung-chieh Shan, Leonid Spesivtsev, Andrew Tolmach, Stephanie Weirich, and Steve Zdancewic

PHOTO: Benjamin C. Pierce

Volume 2

Programming Language Foundations surveys the theory of programming languages, including operational semantics, Hoare logic, and static type systems.

SOFTWARE FOUNDATIONS
VOLUME 2

Programming Language Foundations

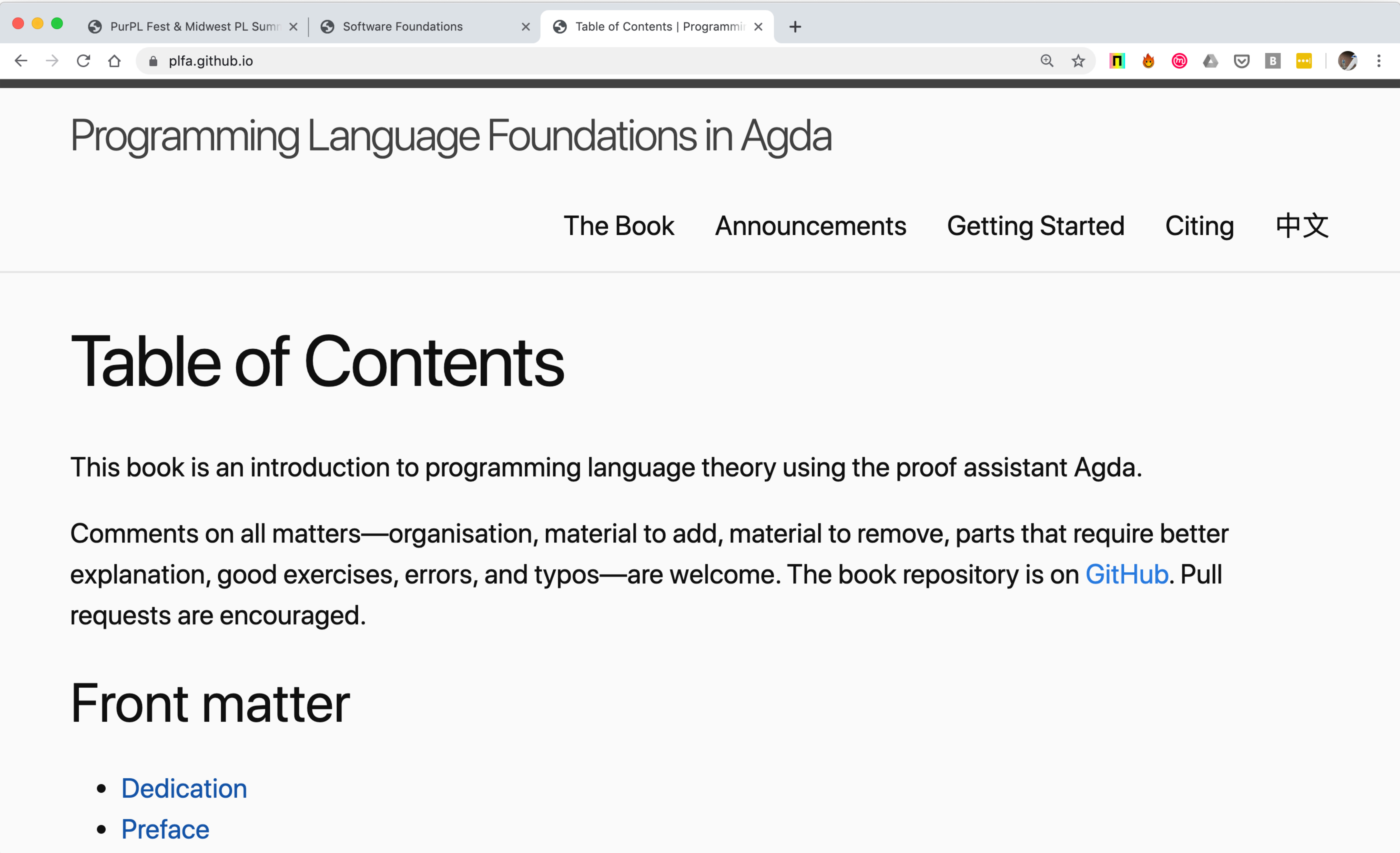
Benjamin C. Pierce
Arthur Azevedo de Amorim
Chris Casinghino
Marco Gaboardi
Michael Greenberg
Cătălin Hriţcu
Vilhelm Sjöberg
Andrew Tolmach
Brent Yorgey

Andrew Tolmach
with
Loris D'Antoni, Andrew W. Appel, Arthur Chargueraud, Anthony Cowley, Jeffrey Foster, Dmitri Garbuzov, Michael Hicks, Ranjit Jhala, Greg Morrisett, Jennifer Paykin, Mukund Raghothaman, Chung-chieh Shan, Leonid Spesivtsev, Stephanie Weirich, and Steve Zdancewic

PHOTO: Benjamin C. Pierce

Volume 3

Volume 4



Programming Language Foundations in Agda

The Book Announcements Getting Started Citing 中文

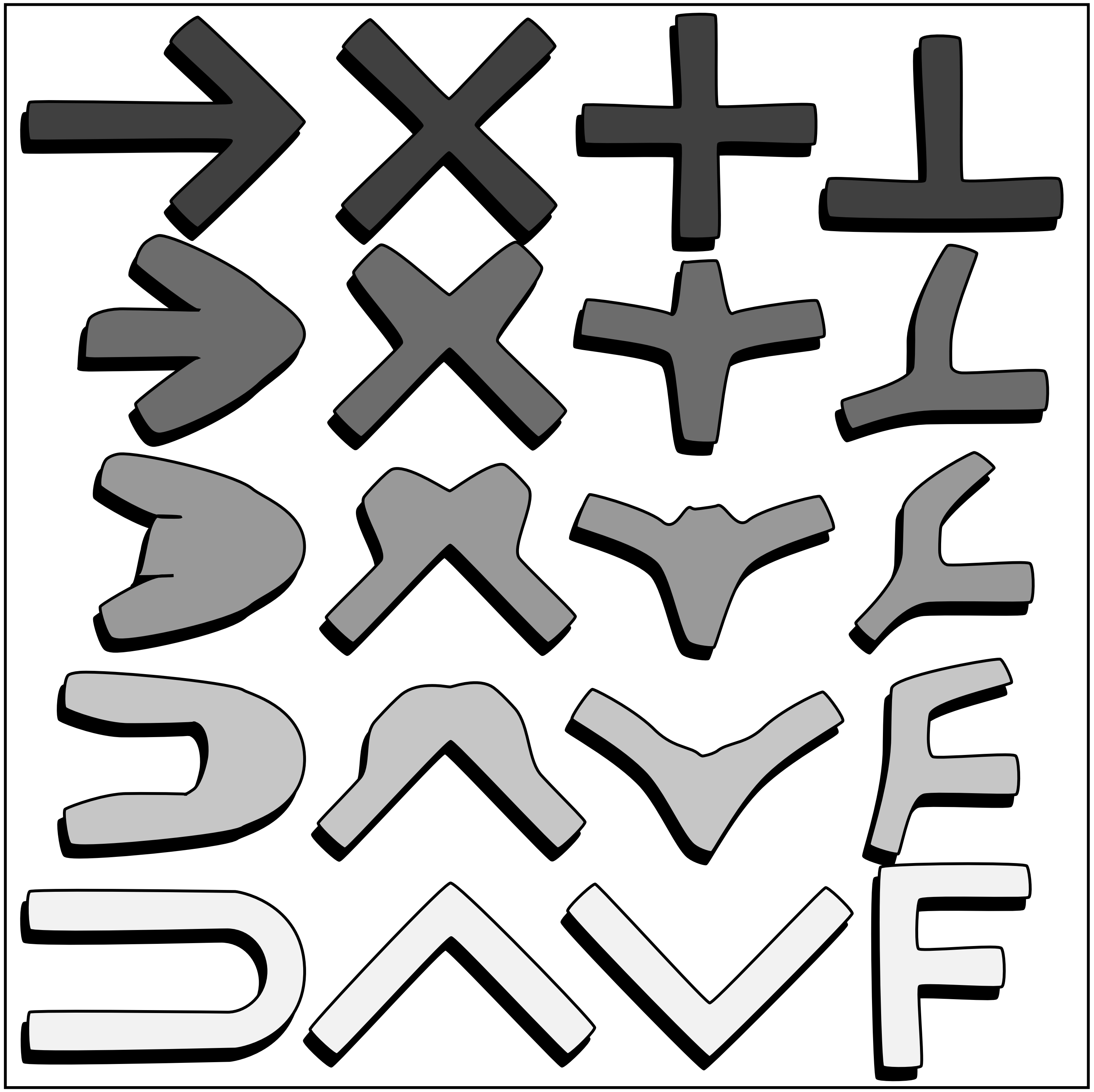
Table of Contents

This book is an introduction to programming language theory using the proof assistant Agda.

Comments on all matters—organisation, material to add, material to remove, parts that require better explanation, good exercises, errors, and typos—are welcome. The book repository is on GitHub. Pull requests are encouraged.

Front matter

- Dedication
- Preface



LC90

The Curry-Howard homeomorphism

(Programming Language) Foundations in Agda

Programming (Language Foundations) in Agda

Coq vs Agda

The troubles with Coq …

- Everything needs to be done twice! Students need to learn both the pair type (terms and patterns) and the tactics for manipulating conjunctions (split and destruct).
- Induction can be mysterious.
- Names vs notations: `subst N x M` *vs* `N[x:=M]`.
- Naming conventions vary widely.
- Propositions as Types present but hidden.

… are absent in Agda

- No tactics to learn. Pairing and conjunction identical.
- Induction is the same as recursion.
- `_[_:=_]` is name for `N [ x := M ]`.
- Standard Library makes a stab at consistency.
- Propositions as Types on proud display.

Agda vs Coq: Simply-Typed Lambda Calculus

Progress

We would like to show that every term is either a value or takes a reduction step. However, this is not true in general. The term

```
`zero · `suc `zero
```

is neither a value nor can take a reduction step. And if `s ⦂ `ℕ ⇒ `ℕ` then the term

```
 s · `zero
```

cannot reduce because we do not know which function is bound to the free variable `s`. The first of those terms is ill-typed, and the second has a free variable. Every term that is well-typed and closed has the desired property.

Progress: If `∅ ⊢ M ⦂ A` then either `M` is a value or there is an `N` such that `M —→ N`.

To formulate this property, we first introduce a relation that captures what it means for a term `M` to make progess.

```
data Progress (M : Term) : Set where

  step : ∀ {N}
    → M —→ N
      ----------
    → Progress M

  done :
      Value M
      ----------
    → Progress M
```

A term `M` makes progress if either it can take a step, meaning there exists a term `N` such that `M —→ N`, or if it is done, meaning that `M` is a value.

If a term is well-typed in the empty context then it satisfies progress.

```
progress : ∀ {M A}
  → ∅ ⊢ M ⦂ A
    ----------
  → Progress M
progress (⊢` ())
progress (⊢ƛ ⊢N)                            =  done V-ƛ
progress (⊢L · ⊢M) with progress ⊢L
... | step L—→L′                            =  step (ξ-·₁ L—→L′)
... | done VL with progress ⊢M
...   | step M—→M′                          =  step (ξ-·₂ VL M—→M′)
...   | done VM with canonical ⊢L VL
...     | C-ƛ _                             =  step (β-ƛ VM)
progress ⊢zero                              =  done V-zero
progress (⊢suc ⊢M) with progress ⊢M
...  | step M—→M′                           =  step (ξ-suc M—→M′)
...  | done VM                              =  done (V-suc VM)
progress (⊢case ⊢L ⊢M ⊢N) with progress ⊢L
... | step L—→L′                            =  step (ξ-case L—→L′)
... | done VL with canonical ⊢L VL
...   | C-zero                              =  step β-zero
...   | C-suc CL                            =  step (β-suc (value CL))
progress (⊢μ ⊢M)                            =  step β-μ
```

We induct on the evidence that `M` is well-typed. Let’s unpack the first three cases.

- The term cannot be a variable, since no variable is well typed in the empty context.
- If the term is a lambda abstraction then it is a value.
- If the term is an application `L · M`, recursively apply progress to the derivation that `L` is well-typed.
    - If the term steps, we have evidence that `L —→ L′`, which by `ξ-·₁` means that our original term steps to `L′ · M`
    - If the term is done, we have evidence that `L` is a value. Recursively apply progress to the derivation that `M` is well-typed.
        - If the term steps, we have evidence that `M —→ M′`, which by `ξ-·₂` means that our original term steps to `L · M′`. Step `ξ-·₂` applies only if we have evidence that `L` is a value, but progress on that subterm has already supplied the required evidence.
        - If the term is done, we have evidence that `M` is a value. We apply the canonical forms lemma to the evidence that `L` is well typed and a value, which since we are in an application leads to the conclusion that `L` must be a lambda abstraction. We also have evidence that `M` is a value, so our original term steps by `β-ƛ`.

The remaining cases are similar. If by induction we have a `step` case we apply a `ξ` rule, and if we have a `done` case then either we have a value or apply a `β` rule. For fixpoint, no induction is required as the `β` rule applies immediately.

Our code reads neatly in part because we consider the `step` option before the `done` option. We could, of course, do it the other way around, but then the `...` abbreviation no longer works, and we will need to write out all the arguments in full. In general, the rule of thumb is to consider the easy case (here `step`) before the hard case (here `done`). If you have two hard cases, you will have to expand out `...` or introduce subsidiary functions.

Progress

The *progress* theorem tells us that closed, well-typed terms are not stuck: either a well-typed term is a value, or it can take a reduction step. The proof is a relatively straightforward extension of the progress proof we saw in the Types chapter. We'll give the proof in English first, then the formal version.

```
Theorem progress : ∀ t T,
  empty |- t ∈ T →
  value t ∨ ∃ t', t ==> t'.
```

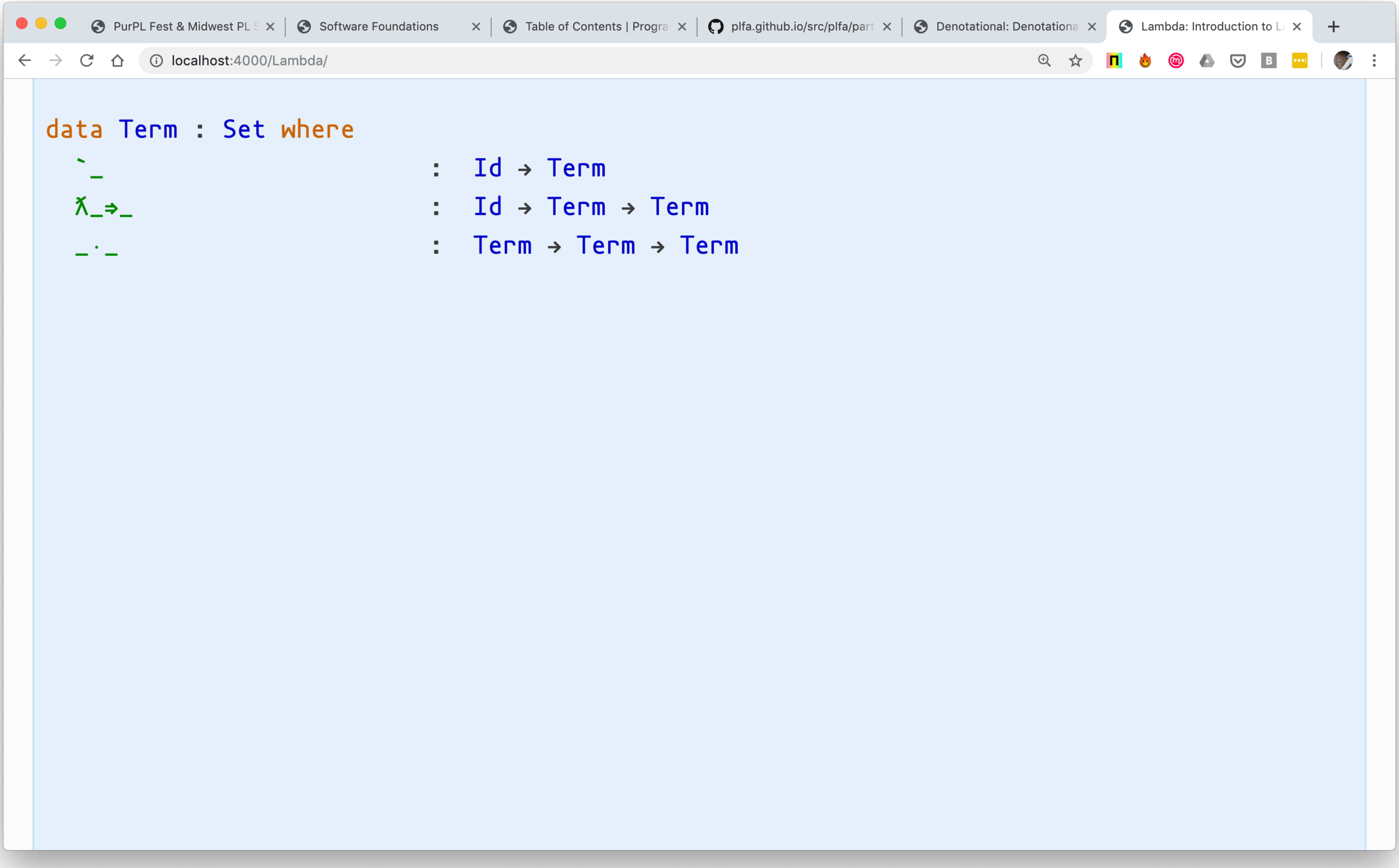
Proof: By induction on the derivation of `|- t ∈ T`.

- The last rule of the derivation cannot be `T_Var`, since a variable is never well typed in an empty context.
- The `T_True`, `T_False`, and `T_Abs` cases are trivial, since in each of these cases we can see by inspecting the rule that `t` is a value.
- If the last rule of the derivation is `T_App`, then `t` has the form $t_1\ t_2$ for some t_1 and t_2, where $|- t_1 \in T_2 \to T$ and $|- t_2 \in T_2$ for some type T_2. By the induction hypothesis, either t_1 is a value or it can take a reduction step.
    - If t_1 is a value, then consider t_2, which by the other induction hypothesis must also either be a value or take a step.
        - Suppose t_2 is a value. Since t_1 is a value with an arrow type, it must be a lambda abstraction; hence $t_1\ t_2$ can take a step by `ST_AppAbs`.
        - Otherwise, t_2 can take a step, and hence so can $t_1\ t_2$ by `ST_App2`.
    - If t_1 can take a step, then so can $t_1\ t_2$ by `ST_App1`.
- If the last rule of the derivation is `T_If`, then $t = \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3$, where t_1 has type `Bool`. By the IH, t_1 either is a value or takes a step.
    - If t_1 is a value, then since it has type `Bool` it must be either `true` or `false`. If it is `true`, then `t` steps to t_2; otherwise it steps to t_3.
    - Otherwise, t_1 takes a step, and therefore so does `t` (by `ST_If`).

```
Proof with eauto.
  intros t T Ht.
  remember (@empty ty) as Gamma.
  induction Ht; subst Gamma...
  - (* T_Var *)
    (* contradictory: variables cannot be typed in an
       empty context *)
    inversion H.

  - (* T_App *)
    (* t = t1 t2.  Proceed by cases on whether t1 is a
       value or steps... *)
    right. destruct IHHt1...
    + (* t1 is a value *)
      destruct IHHt2...
      * (* t2 is also a value *)
        assert (∃ x0 t0, t1 = tabs x0 T11 t0).
        eapply canonical_forms_fun; eauto.
        destruct H1 as [x0 [t0 Heq]]. subst.
        ∃ ([x0:=t2]t0)...

      * (* t2 steps *)
        inversion H0 as [t2' Hstp]. ∃ (tapp t1 t2')...

    + (* t1 steps *)
      inversion H as [t1' Hstp]. ∃ (tapp t1' t2)...

  - (* T_If *)
    right. destruct IHHt1...

    + (* t1 is a value *)
      destruct (canonical_forms_bool t1); subst; eauto.

    + (* t1 also steps *)
      inversion H as [t1' Hstp]. ∃ (tif t1' t2 t3)...
Qed.
```

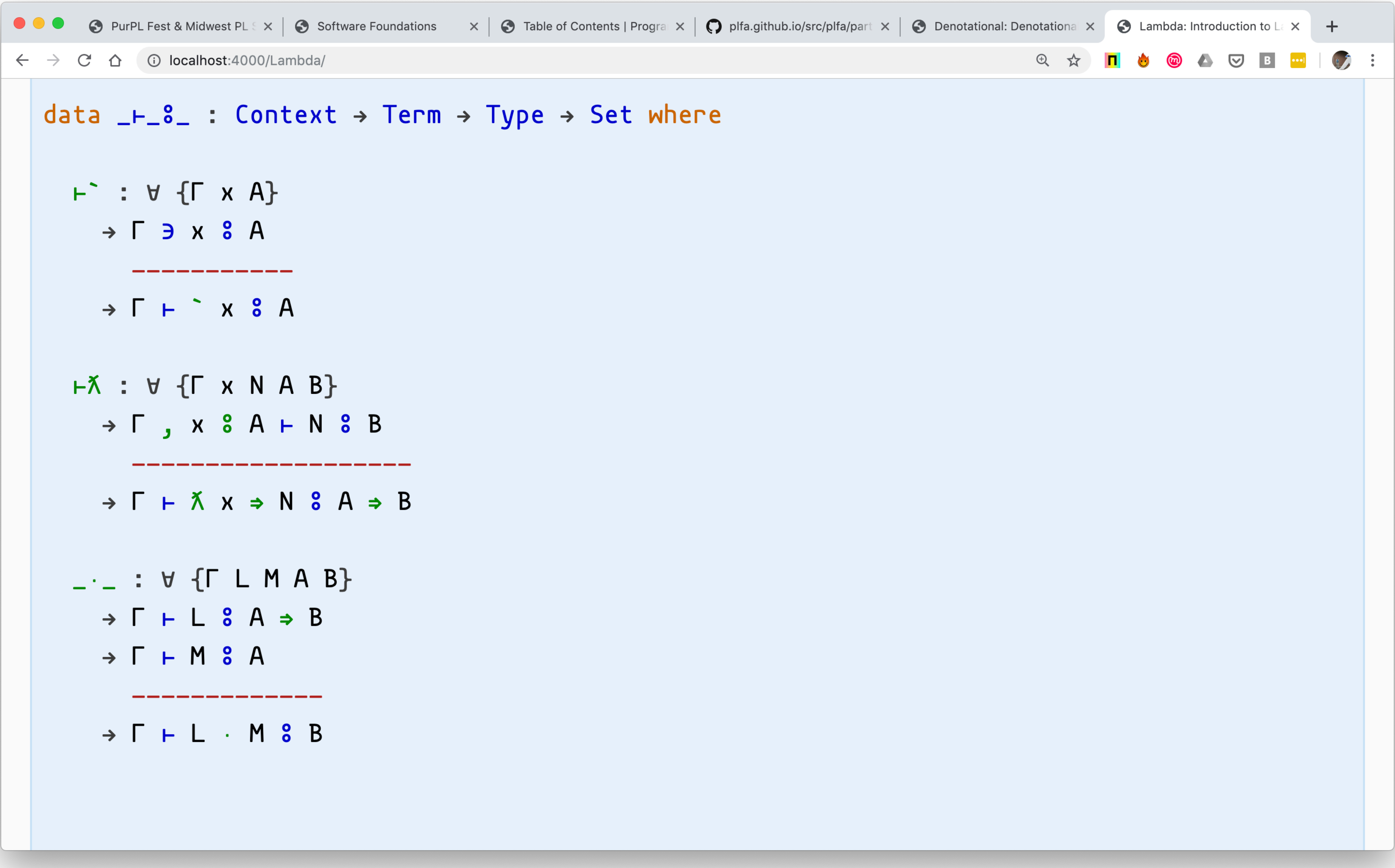
Intrinsically-Typed is Golden

Extrinsic (1)



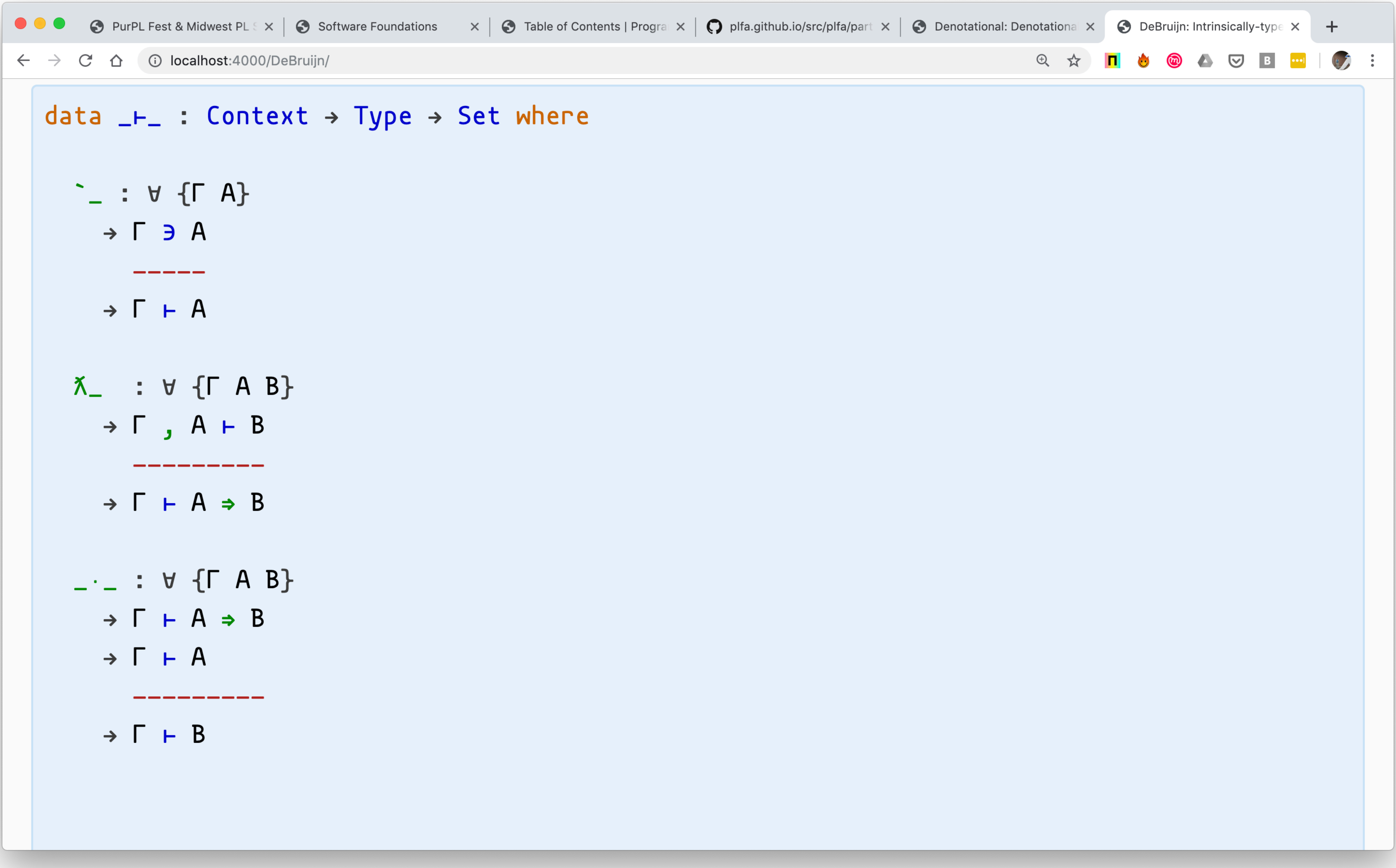
```
data Term : Set where
  `_                        :  Id → Term
  ƛ_⇒_                      :  Id → Term → Term
  _·_                       :  Term → Term → Term
```

Extrinsic (2)



```
data _⊢_⦂_ : Context → Term → Type → Set where

  ⊢` : ∀ {Γ x A}
    → Γ ∋ x ⦂ A
      -----------
    → Γ ⊢ ` x ⦂ A

  ⊢ƛ : ∀ {Γ x N A B}
    → Γ , x ⦂ A ⊢ N ⦂ B
      -------------------
    → Γ ⊢ ƛ x ⇒ N ⦂ A ⇒ B

  _·_ : ∀ {Γ L M A B}
    → Γ ⊢ L ⦂ A ⇒ B
    → Γ ⊢ M ⦂ A
      -------------
    → Γ ⊢ L · M ⦂ B
```

Intrinsic



```
data _⊢_ : Context → Type → Set where

  `_ : ∀ {Γ A}
    → Γ ∋ A
      -----
    → Γ ⊢ A

  ƛ_  : ∀ {Γ A B}
    → Γ , A ⊢ B
      ---------
    → Γ ⊢ A ⇒ B

  _·_ : ∀ {Γ A B}
    → Γ ⊢ A ⇒ B
    → Γ ⊢ A
      ---------
    → Γ ⊢ B
```

	Lines of code, omitting examples
Extrinsic	451
Intrinsic	275

451 / 275 = 1.6

275 / 451 = 0.6

Progress + Preservation
= Animation

Aside: the `normalize` Tactic

When experimenting with definitions of programming languages in Coq, we often want to see what a particular concrete term steps to — i.e., we want to find proofs for goals of the form `t ==>* t'`, where `t` is a completely concrete term and `t'` is unknown. These proofs are quite tedious to do by hand. Consider, for example, reducing an arithmetic expression using the small-step relation `astep`.

The following custom `Tactic Notation` definition captures this pattern. In addition, before each step, we print out the current goal, so that we can follow how the term is being reduced.

```
Tactic Notation "print_goal" :=
  match goal with |- ?x ⇒ idtac x end.

Tactic Notation "normalize" :=
  repeat (print_goal; eapply multi_step ;
            [ (eauto 10; fail) | (instantiate; simpl)]);
  apply multi_refl.
```

The `normalize` tactic also provides a simple way to calculate the normal form of a term, by starting with a goal with an existentially bound variable.

```
Example step_example1''' : ∃ e',
  (P (C 3) (P (C 3) (C 4)))
  ==>* e'.
Proof.
  eapply ex_intro. normalize.
(* This time, the trace is:
       (P (C 3) (P (C 3) (C 4)) ==>* ?e')
       (P (C 3) (C 7) ==>* ?e')
       (C 10 ==>* ?e')
   where ?e' is the variable ``guessed'' by eapply. *)
Qed.
```

Functional Big-step Semantics

Scott Owens[1], Magnus O. Myreen[2], Ramana Kumar[3], and Yong Kiam Tan[4]

[1] School of Computing, University of Kent, UK
[2] CSE Department, Chalmers University of Technology, Sweden
[3] NICTA, Australia
[4] IHPC, A*STAR, Singapore

Testing semantics To test a semantics, one must actually use it to evaluate programs. Functional big-step semantics can do this out-of-the-box, as can many small-step approaches [13,14]. Where semantics are defined in a relational big-

---

13. C. Ellison and G. Rosu. An executable formal semantics of C with applications. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012*, pages 533–544, 2012. `doi:10.1145/2103656.2103719`.
14. C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, J. A. McCarthy, J. Rafkind, S. Tobin-Hochstadt, and R. B. Findler. Run your research: on the effectiveness of lightweight mechanization. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012*, pages 285–296, 2012. `doi:10.1145/2103656.2103691`.

Mechanized Metatheory for the Masses: The POPLMARK Challenge

Brian E. Aydemir[1], Aaron Bohannon[1], Matthew Fairbairn[2], J. Nathan Foster[1],
Benjamin C. Pierce[1], Peter Sewell[2], Dimitrios Vytiniotis[1], Geoffrey
Washburn[1], Stephanie Weirich[1], and Steve Zdancewic[1]

[1] Department of Computer and Information Science, University of Pennsylvania
[2] Computer Laboratory, University of Cambridge

Challenge 2A: Type Safety for Pure $F_{<:}$

Type soundness is usually proven in the style popularized by Wright and Felleisen [51], in terms of *preservation* and *progress* theorems. Challenge 2A is to prove these properties for pure $F_{<:}$.

3.3 THEOREM [PRESERVATION]: If $\Gamma \vdash \mathtt{t} : \mathtt{T}$ and $\mathtt{t} \longrightarrow \mathtt{t}'$, then $\Gamma \vdash \mathtt{t}' : \mathtt{T}$. □

3.4 THEOREM [PROGRESS]: If $\mathtt{t}$ is a closed, well-typed $F_{<:}$ term (i.e., if $\vdash \mathtt{t} : \mathtt{T}$ for some $\mathtt{T}$), then either $\mathtt{t}$ is a value or else there is some $\mathtt{t}'$ with $\mathtt{t} \longrightarrow \mathtt{t}'$. □

Challenge 3: Testing and Animating with Respect to the Semantics

Our final challenge is to provide an implementation of this functionality, specifically for the following three tasks (using the language of Challenge 2B):

1. Given $\mathrm{F}_{<:}$ terms $\mathtt{t}$ and $\mathtt{t}'$, decide whether $\mathtt{t} \longrightarrow \mathtt{t}'$.
2. Given $\mathrm{F}_{<:}$ terms $\mathtt{t}$ and $\mathtt{t}'$, decide whether $\mathtt{t} \longrightarrow^{*} \mathtt{t}' \not\longrightarrow$, where $\longrightarrow^{*}$ is the reflexive-transitive closure of $\longrightarrow$.
3. Given an $\mathrm{F}_{<:}$ term $\mathtt{t}$, find a term $\mathtt{t}'$ such that $\mathtt{t} \longrightarrow \mathtt{t}'$.

Evaluation

By repeated application of progress and preservation, we can evaluate any well-typed term. In this section, we will present an Agda function that computes the reduction sequence from any given closed, well-typed term to its value, if it has one.

The evaluator takes gas and evidence that a term is well-typed, and returns the corresponding steps.

```
eval : ∀ {L A}
  → Gas
  → ∅ ⊢ L ⦂ A
    ---------
  → Steps L
eval {L} (gas zero)    ⊢L                            =  steps (L ∎) out-of-gas
eval {L} (gas (suc m)) ⊢L with progress ⊢L
... | done VL                                        =  steps (L ∎) (done VL)
... | step L—→M with eval (gas m) (preserve ⊢L L—→M)
...    | steps M—↠N fin                               =  steps (L —→⟨ L—→M ⟩ M—↠N) fin
```

```
_ : eval (gas 100) (⊢twoᶜ · ⊢sucᶜ · ⊢zero) ≡
  steps
   ((ƛ "s" ⇒ (ƛ "z" ⇒ ` "s" · (` "s" · ` "z"))) · (ƛ "n" ⇒ `suc ` "n")
   · `zero
   —→⟨ ξ-·₁ (β-ƛ V-ƛ) ⟩
    (ƛ "z" ⇒ (ƛ "n" ⇒ `suc ` "n") · ((ƛ "n" ⇒ `suc ` "n") · ` "z")) ·
     `zero
   —→⟨ β-ƛ V-zero ⟩
    (ƛ "n" ⇒ `suc ` "n") · ((ƛ "n" ⇒ `suc ` "n") · `zero)
   —→⟨ ξ-·₂ V-ƛ (β-ƛ V-zero) ⟩
    (ƛ "n" ⇒ `suc ` "n") · `suc `zero
   —→⟨ β-ƛ (V-suc V-zero) ⟩
    `suc (`suc `zero)
   ∎)
   (done (V-suc (V-suc V-zero)))
_ = refl
```

Agda for Fun and Profit:
System F


Philip Wadler


Manuel Chakravarty


Simon Thompson


Vanessa McHale


Michael Peyton Jones


Kris Jenkins


Roman Kireev


Jann Müller


David Smith


Kenneth MacKenzie


James Chapman


Pablo Lamela Seijas

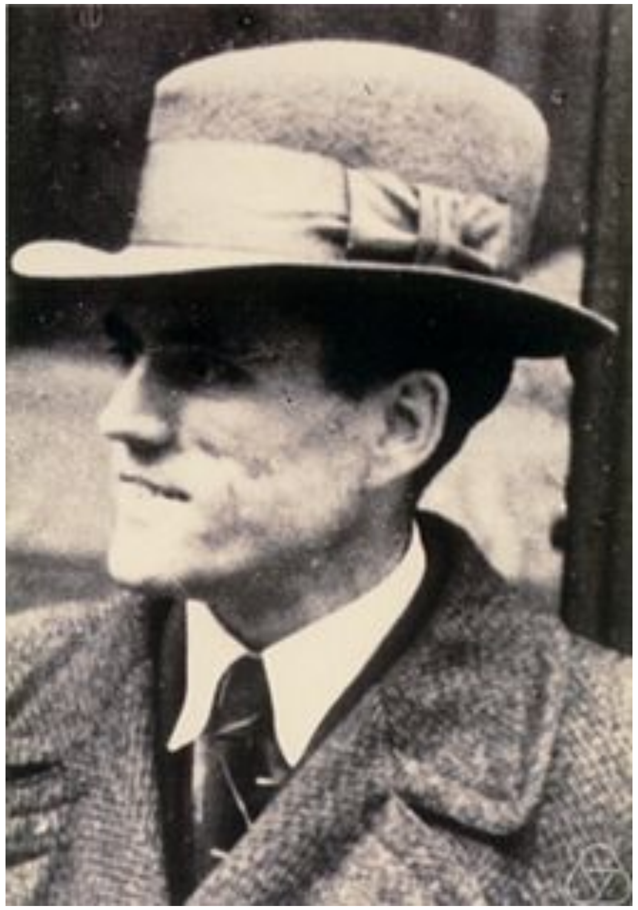

Alexander Nemish


Rebecca Valentine
(Former Member)

Lambda Calculus

Alonzo Church, 1932-40

Natural Deduction

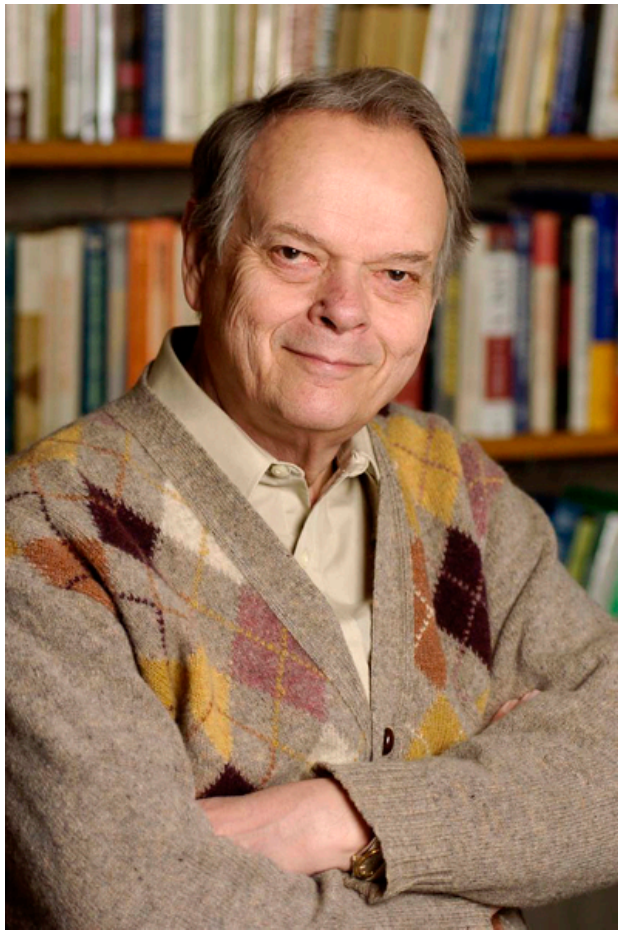


Gerhard Gentzen, 1935

System F

Jean-Yves Girard, 1972

Polymorphic
Lambda Calcululus



John Reynolds, 1974

Plutus Core

Kinds

```
  J,K ::=
    *
    J → K
```

Types

```
  A,B ::=
    X
    A → B
    ∀X.B
    μX.B
    ρ
```

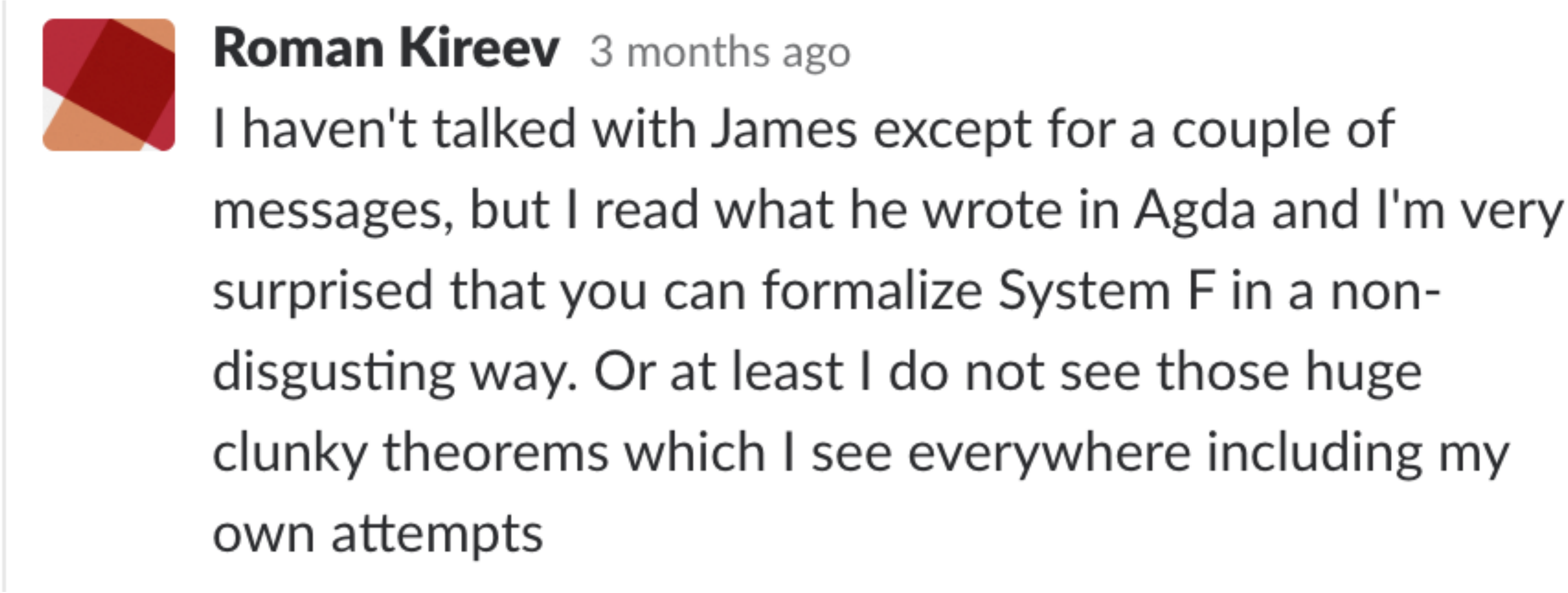
Terms

```
  L,M,N ::=
    x
    λx:A.N
    L M
    ΛX:K.N
    L A
    wrap M
    unwrap M
    ρ
```

Plutus Core in Agda

```
data Kind : Set where
  *   : Kind
  _⇒_ : Kind → Kind → Kind

data _⊢⋆_ : Ctx⋆ → Kind → Set where
  `   : Φ ∋⋆ J
        ------
      → Φ ⊢⋆ J

  ƛ   : Φ ,⋆ K ⊢⋆ J
        -----------
      → Φ ⊢⋆ K ⇒ J

  _·_ : Φ ⊢⋆ K ⇒ J
      → Φ ⊢⋆ K
        ----------
      → Φ ⊢⋆ J

  Π   : Φ ,⋆ K ⊢⋆ *
        -----------
      → Φ ⊢⋆ *

  _⇒_ : Φ ⊢⋆ *
      → Φ ⊢⋆ *
        ------
      → Φ ⊢⋆ *
```

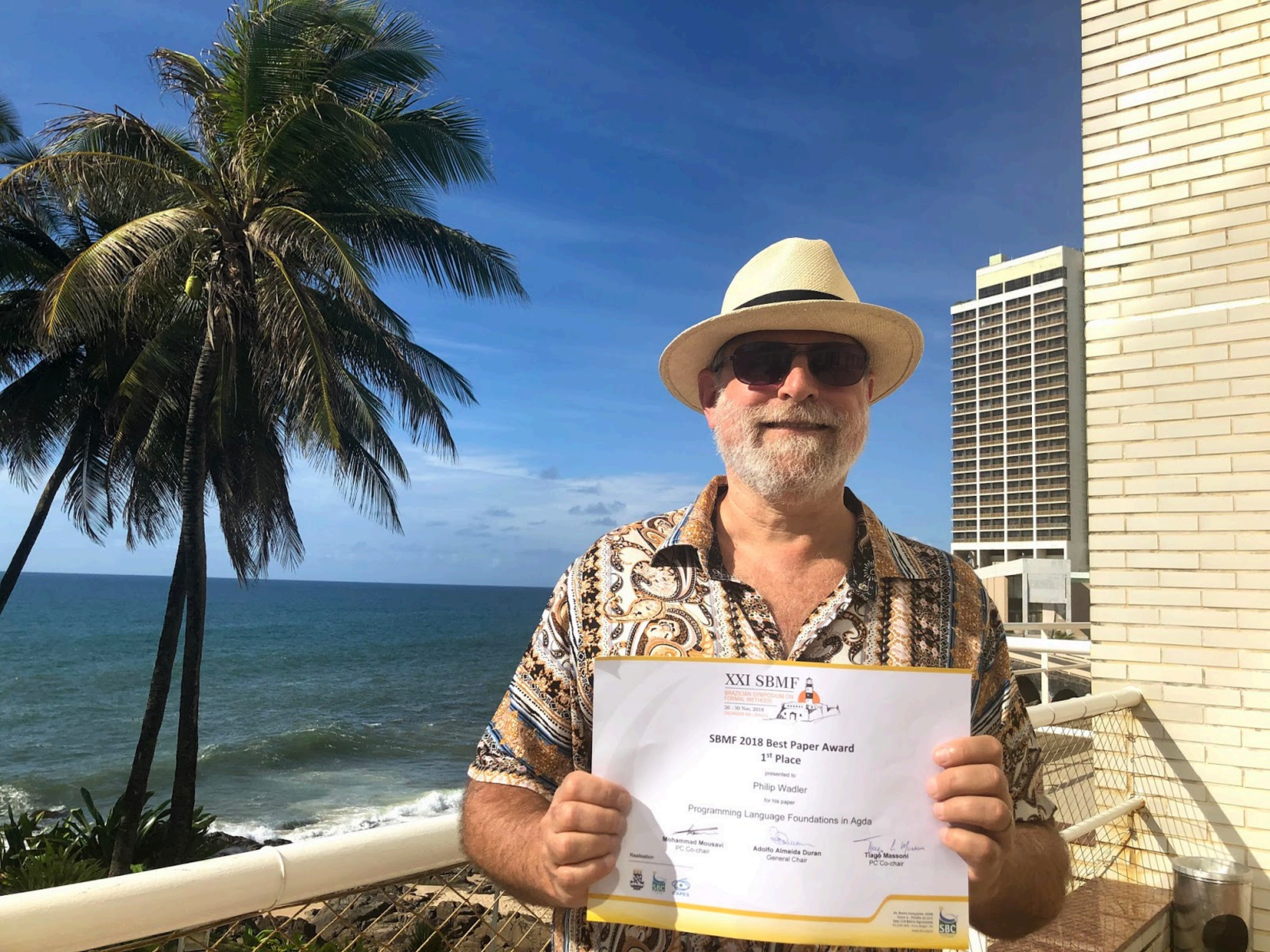
```
data _⊢_ : ∀ Γ → ‖ Γ ‖ ⊢⋆ J → Set where
  `     : Γ ∋ A
          -----
        → Γ ⊢ A

  ƛ     : Γ , A ⊢ B
          ---------
        → Γ ⊢ A ⇒ B

  _·_   : Γ ⊢ A ⇒ B
        → Γ ⊢ A
          ---------
        → Γ ⊢ B

  Λ     : Γ ,⋆ K ⊢ B
          ----------
        → Γ ⊢ Π B

  _·⋆_  : Γ ⊢ Π B
        → (A : ‖ Γ ‖ ⊢⋆ K)
          ---------------
        → Γ ⊢ B [ A ]

  conv  : A ≡β B
        → Γ ⊢ A
          -----
        → Γ ⊢ B
```



Roman Kireev 3 months ago

I haven't talked with James except for a couple of messages, but I read what he wrote in Agda and I'm very surprised that you can formalize System F in a non-disgusting way. Or at least I do not see those huge clunky theorems which I see everywhere including my own attempts

Conclusions



XXI SBMF
BRAZILIAN SYMPOSIUM ON
FORMAL METHODS
26 - 30 Nov, 2018
SALVADOR-BA | BRAZIL

SBMF 2018 Best Paper Award
1st Place

presented to

Philip Wadler

for his paper

Programming Language Foundations in Agda

Mohammad Mousavi
PC Co-chair

Adolfo Almeida Duran
General Chair

Tiago L. Massoni
Tiago Massoni
PC Co-chair

Realisation

CONTRIBUTED ARTICLES

Propositions as Types

By Philip Wadler

Communications of the ACM, December 2015, Vol. 58 No. 12, Pages 75-84
10.1145/2699407

Comments (1)

VIEW AS: SHARE:

Powerful insights arise from linking two fields of study previously thought separate. Examples include Descartes's coordinates, which links geometry to algebra, Planck's Quantum Theory, which links particles to waves, and Shannon's Information Theory, which links thermodynamics to communication. Such a synthesis is offered by the principle of Propositions as Types, which links logic to computation. At first sight it appears to be a simple coincidence—almost a pun—but it turns out to be remarkably robust, inspiring the design of automated proof assistants and programming languages, and continuing to influence the forefronts of computing.

Back to Top

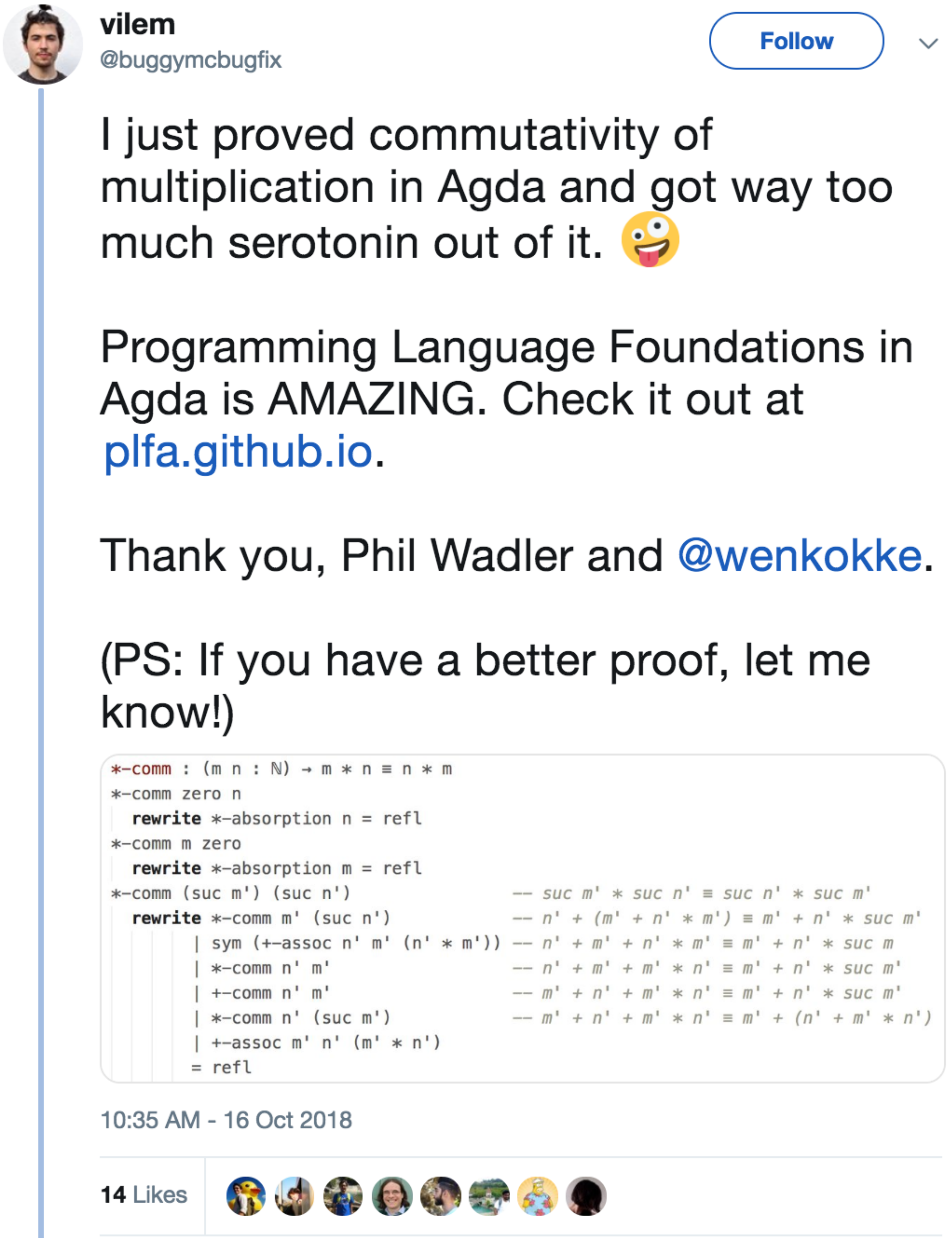
Propositions as Types

Philip Wadler
University of Edinburgh

Strange Loop
St Louis, 25 August 2015

"Propositions as Types" by Philip Wadler

61,321 views



vilem
@buggymcbugfix

I just proved commutativity of multiplication in Agda and got way too much serotonin out of it. 🤪

Programming Language Foundations in Agda is AMAZING. Check it out at plfa.github.io.

Thank you, Phil Wadler and @wenkokke.

(PS: If you have a better proof, let me know!)

```
*-comm : (m n : ℕ) → m * n ≡ n * m
*-comm zero n
  rewrite *-absorption n = refl
*-comm m zero
  rewrite *-absorption m = refl
*-comm (suc m') (suc n')                -- suc m' * suc n' ≡ suc n' * suc m'
  rewrite *-comm m' (suc n')            -- n' + (m' + n' * m') ≡ m' + n' * suc m'
        | sym (+-assoc n' m' (n' * m')) -- n' + m' + n' * m' ≡ m' + n' * suc m
        | *-comm n' m'                  -- n' + m' + m' * n' ≡ m' + n' * suc m'
        | +-comm n' m'                  -- m' + n' + m' * n' ≡ m' + n' * suc m'
        | *-comm n' (suc m')            -- m' + n' + m' * n' ≡ m' + (n' + m' * n')
        | +-assoc m' n' (m' * n')
        = refl
```

10:35 AM - 16 Oct 2018

14 Likes

http://plfa.inf.ed.ac.uk
https://github.com/plfa

Or search for "Kokke Wadler"

Please send your comments and pull requests!