Programming Programming Language Foundations in Agda in Agda

by Wen Kokke

You: "What's an 'Agda'?"

Me: "It's a proof assistant!"

You: "What's a proof assistant?"

Me: "Uh..."

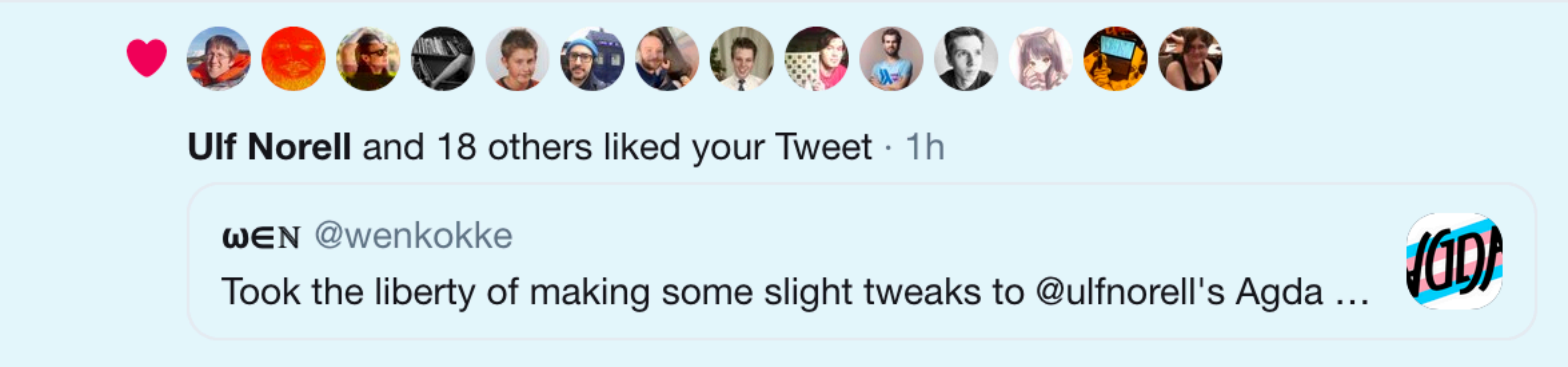
ACL2, Agda, Agda 2, Albatross, ALF, Aquarius, ATS, Automath, a blackboard, Blodwen, Cambridge LCF, Cayenne, Cedille, Clam, Clam 2, Clam 3, CLIN, Coq, Dafny, Dependent ML, DISCOUNT, Epigram, EQP, ETPS, F*, Haskell, HOL Light, HOL4, HOL88, HOL90, Idris, IMPS, InKa, Intercal, Isabelle, Jape, KeY, LambdaClam, LCF77, Lean, LEGO, Logic Theorist, Mace, Mace4, Matita, Metamath, me, MINLOG, Mizar, MKRP, NQTHM, NuPRL, OLEG, OMEGA, OSHL, Otter, Peers, Peers-mcd.a, Peers-mcd.b, Peers-mcd.c, Peers-mcd.d, PhoX, PLTP, PRESS, ProCom, Prover9, PRV, PVS, RDL, SCUNAC, SETHEO, SNARK, SASyLF, TPS, Twelf, Tutch, Typelab, Yarrow

Coq

by Bruno Barras, Yves Bertot, Pierre Castéran, Thierry Coquand, Jean-Christophe Filliâtre, Hugo Herbelin, Gérard Pierre Huet, Chetan Murthy, and Christine Paulin-Mohring (and at least 133 other contributors)

AGDA

by Andreas Abel, Stevan Andjelkovic, Marcin Benke, James Chapman, Jesper Cockx, Jean-Philippe Bernardy, Nils Anders Danielsson, Dominique Devriese, Péter Diviánszky, Olle Fredriksson, Samuel Gélineau, Daniel Gustafsson, Patrik Jansson, Alan Jeffrey, Fredrik Lindblad, Stefan Monnier, Darin Morrison, Guilhem Moulin, Fredrik Nordvall Forsberg, Ulf Norell, Nicolas Pouillard, Andrés Sicard-Ramírez, Wouter Swierstra, Makoto Takeyama, Andrea Vezzosi, and Nobuo Yamashita (and at least 74 other contributors)



Ulf Norell and 18 others liked your Tweet · 1h

ωEN @wenkokke

Took the liberty of making some slight tweaks to @ulfnorell's Agda …

AGDA

by Andreas Abel, Stevan Andjelkovic, Marcin Benke, James Chapman, Jesper Cockx, Jean-Philippe Bernardy, Nils Anders Danielsson, Dominique Devriese, Péter Diviánszky, Olle Fredriksson, Samuel Gélineau, Daniel Gustafsson, Patrik Jansson, Alan Jeffrey, Fredrik Lindblad, Stefan Monnier, Darin Morrison, Guilhem Moulin, Fredrik Nordvall Forsberg, Ulf Norell, Nicolas Pouillard, Andrés Sicard-Ramírez, Wouter Swierstra, Makoto Takeyama, Andrea Vezzosi, and Nobuo Yamashita (and at least 74 other contributors)

You: "Why use proof assistant in my class?"

Me: "This... is all I've ever known???"

im baby

Lambda,
The Ultimate TA

Using a Proof Assistant to Teach Programming Language Foundations

ICFP 2009

Benjamin C. Pierce
University of Pennsylvania

My List

Logic

- Inductively defined relations
- Inductive proof techniques

Functional Programming

- programs as data, polymorphism, recursion, ...

PL Theory

- Precise description of program structure and behavior
  - operational semantics
  - lambda-calculus
- Program correctness
  - Hoare Logic
- Types

Oops, forgot one thing...

- The difficulty with teaching many of these topics is that they presuppose the ability to read and write mathematical proofs.
- In a course for arbitrary computer science students, this appears to be a really bad assumption.

automated proof assistant

=

one TA per student

You: "This... is all you've ever known?"

Me: "Y-y-yes?"

(2014)
I was taught from
Software Foundations,
and learned Coq and Agda
in the same course,
one after the other,
taught by this guy.

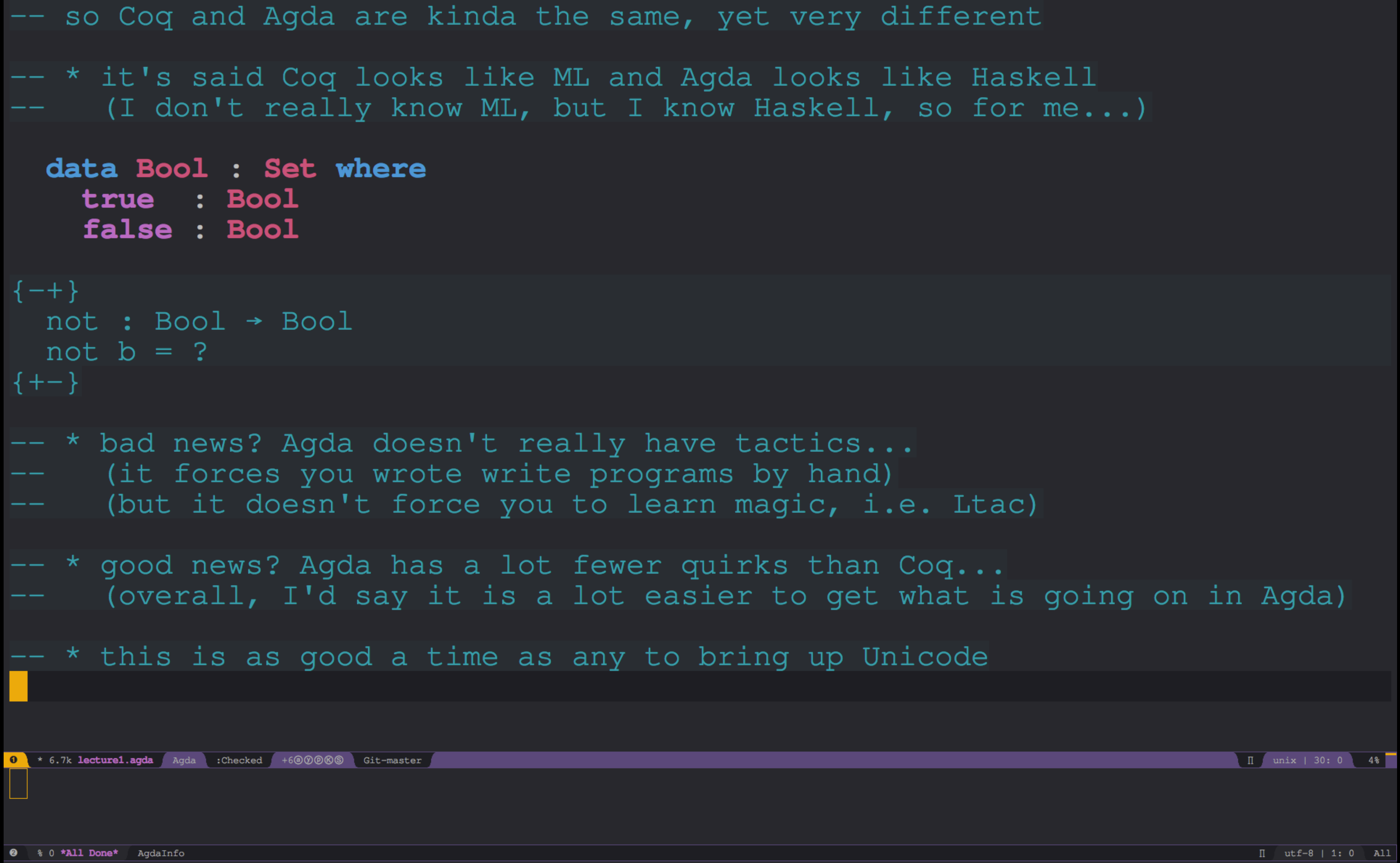
(2014–2016)
FORMALISED SEVERAL CALCULI USING AGDA, MOSTLY SUBSTRUCTURAL, WITH THIS GUY.

(2016)
Taught Software foundations, with this guy.

He did most of the lecturing, tbh.

(2017)
Taught Software Foundations,
with this guy.

I gave several additional lectures on Agda!



```
-- so Coq and Agda are kinda the same, yet very different

-- * it's said Coq looks like ML and Agda looks like Haskell
--   (I don't really know ML, but I know Haskell, so for me...)

  data Bool : Set where
    true  : Bool
    false : Bool

{-+}
  not : Bool → Bool
  not b = ?
{+-}

-- * bad news? Agda doesn't really have tactics...
--   (it forces you wrote write programs by hand)
--   (but it doesn't force you to learn magic, i.e. Ltac)

-- * good news? Agda has a lot fewer quirks than Coq...
--   (overall, I'd say it is a lot easier to get what is going on in Agda)

-- * this is as good a time as any to bring up Unicode
```

(2018)
Taught Programming Language Foundations in Agda!

You: "Hold on. Why Agda? Isn't Software Foundations, just like, fine???"

Me: "Uh... good question!"

My Troubles with Coq...

- everything is done twice,
  once in Gallina, once in Ltac:

  `pair` and `match` vs. `split` and `destruct`

- everything is done four times,
  'cuz names and notation are different things:

  `(prod A B)` is written `(A * B)`
  `(pair A B)` is written `(A , B)`

My Troubles with Coq...

- it's not even just four times! 😫

  `split`
  vs. `apply pair`
  vs. `constructor 1`
  vs. `constructor`
  vs. `auto`

- this landscape of spurious equivalences burdens and confuses the students!

…DISAPPEAR WITH AGDA!

- Agda doesn't have tactics
- everything is done once

  _×_ and _,_
- no distinction between name and notation, the name for the product type is _×_

My Troubles with Coq...

- Ltac code is imperative
  you're manipulating an invisible proof stack
- to understand Ltac you have to step through
- Ltac is not readable

...DISAPPEAR WITH AGDA!

- Agda doesn't have tactics

- wait, is that fair?

My Troubles with Coq...

"For me, if [induction] was the only thing they got out of this course, that would be okay."
— Benjamin Pierce

- induction can be confusing
- induction does the same as destruct, but gives you this random other data... sometimes?
- induction interacts with intros

…DISAPPEAR WITH AGDA!

- in Agda, <u>induction</u> is <u>recursion</u>

You: "Okay. I'm convinced. Let's talk PLFA."

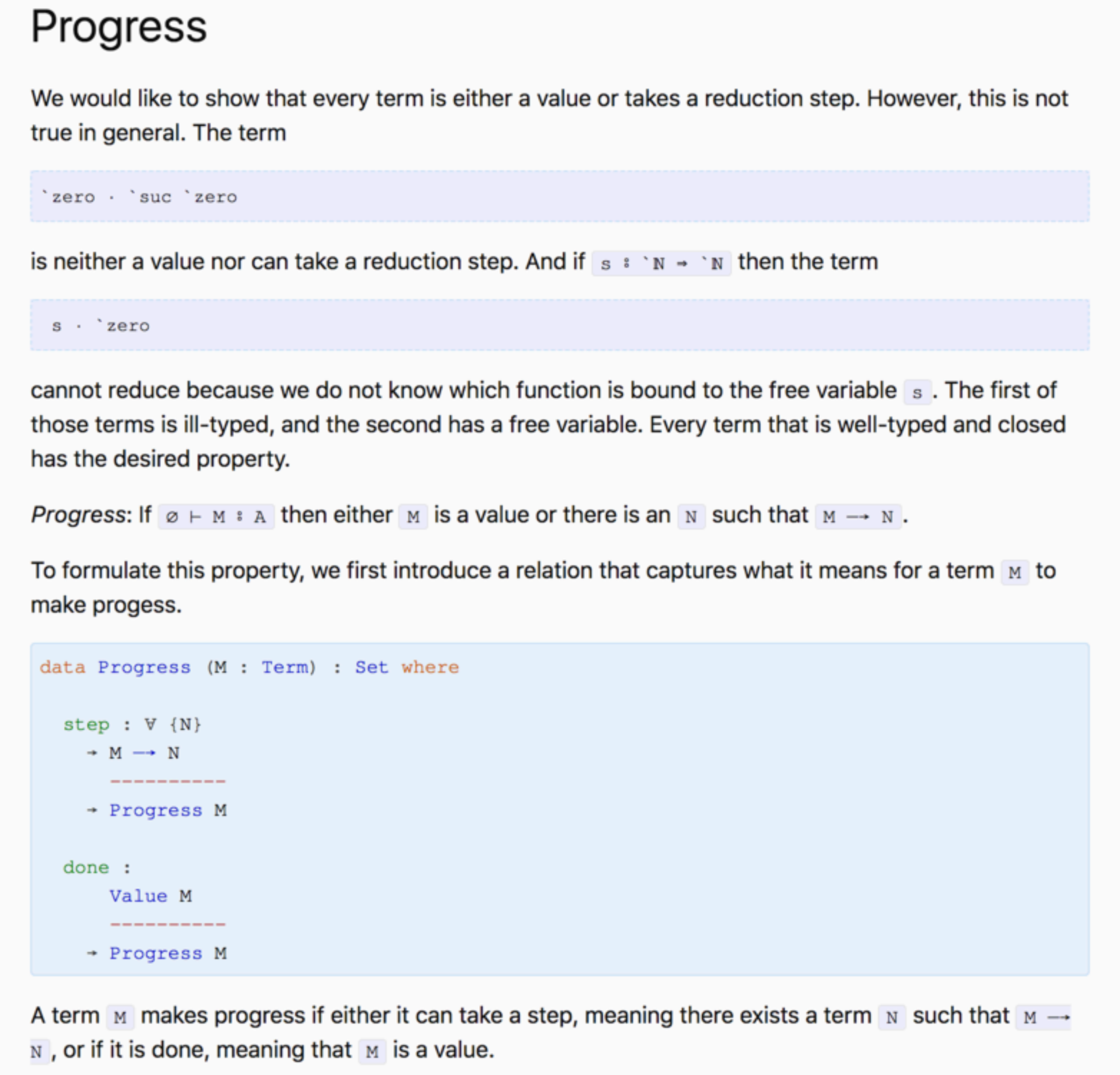
PLFA

by Marko Dimjašević, Wen Kokke, Jeremy Siek, Zbigniew Stanasiuk, Philip Wadler, and Yasu Watanabe (and 32 other contributors)

How most of PLFA was produced:

Our Concerns with Agda…

- is Agda stable enough?
- does the lack of automation blow up proof size?



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
```

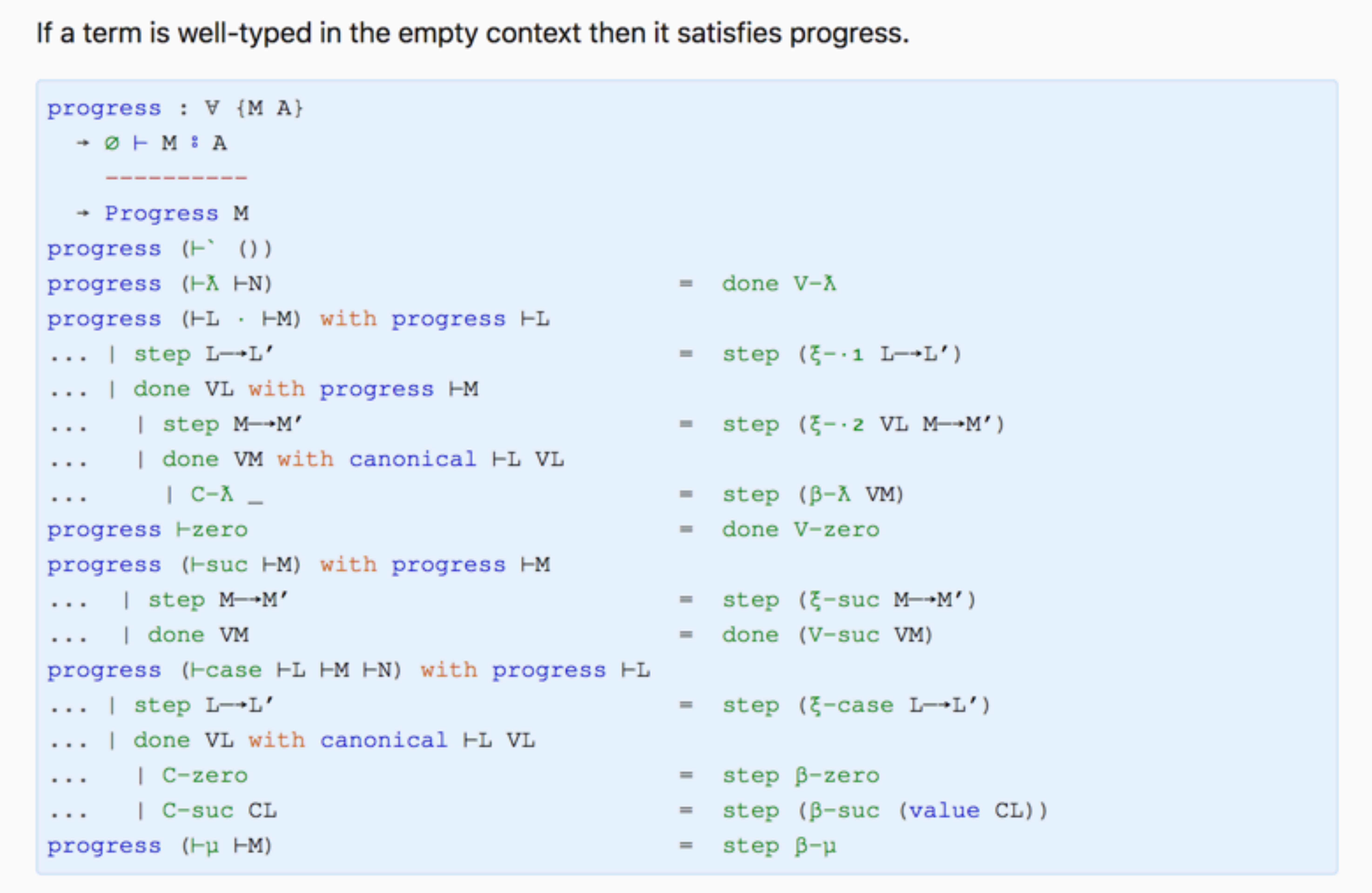
cannot reduce because we do not know which function is bound to the free variable `s`. The first of those terms is ill-typed, and the second has a free variable. Every term that is well-typed and closed has the desired property.

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.

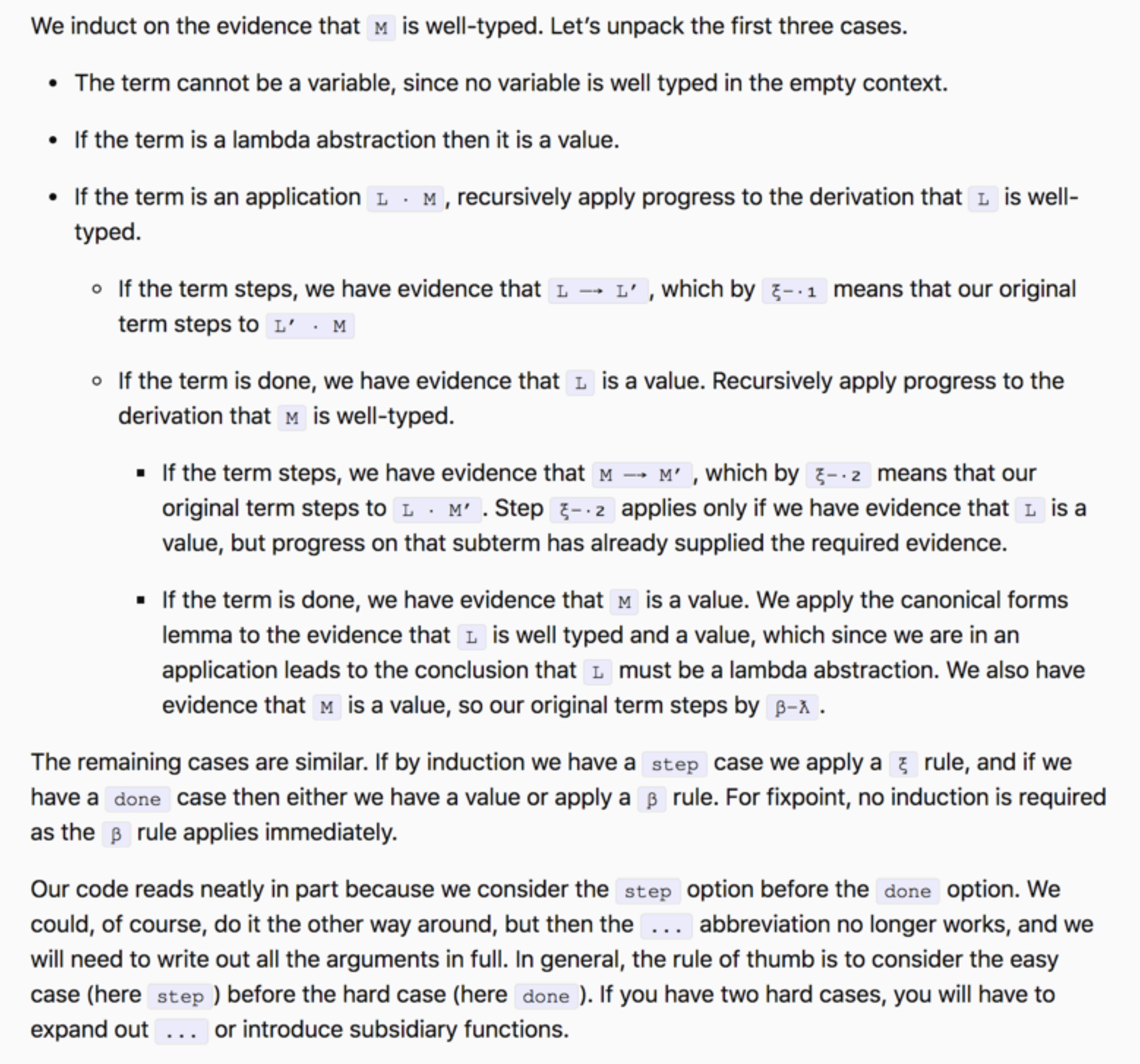
```
data Progress (M : Term) : Set where

  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
```

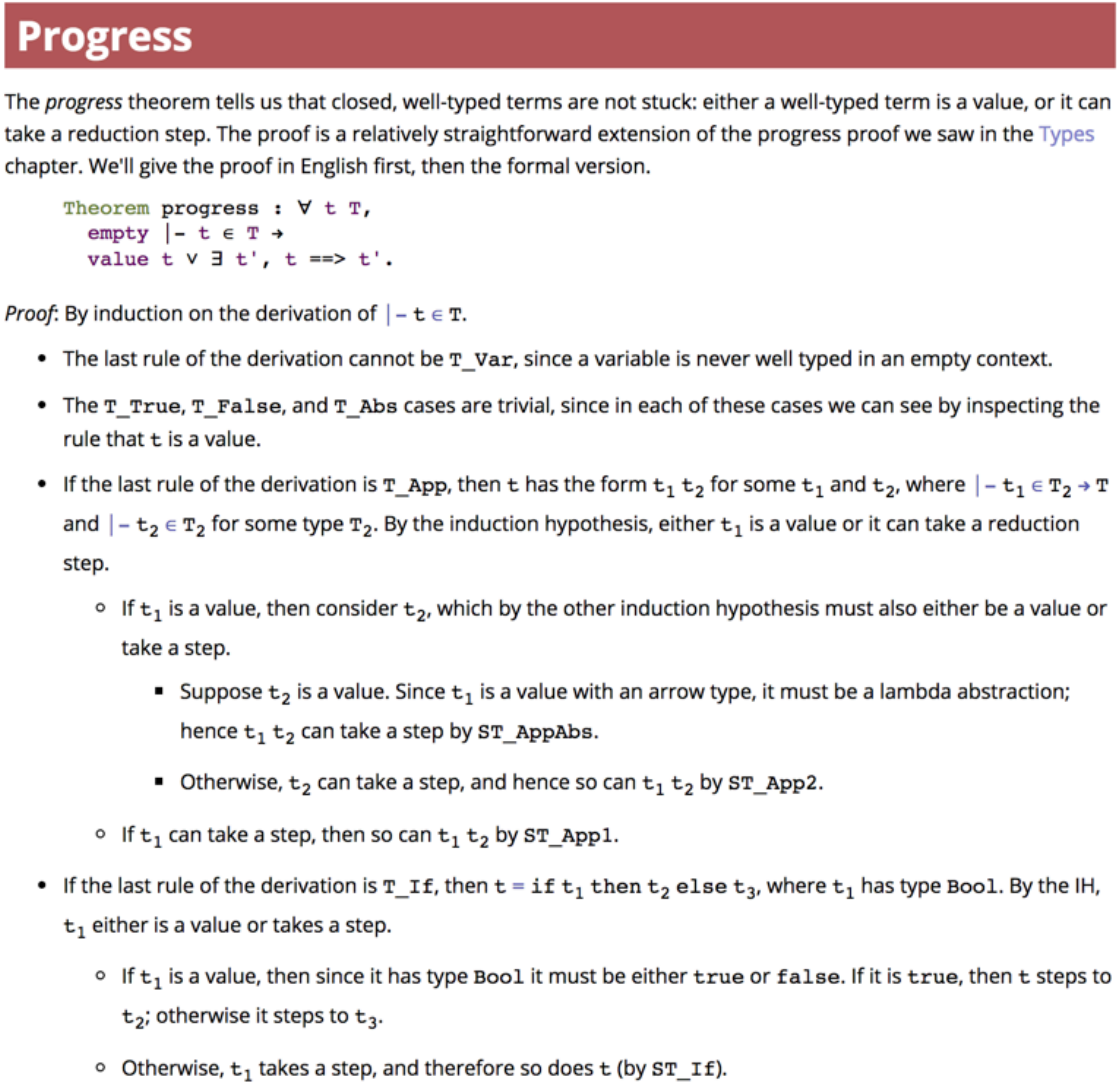


We induct on the evidence that `M` is well-typed. Let's unpack the first three cases.

- 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` = `if` t_1 `then` t_2 `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.
```

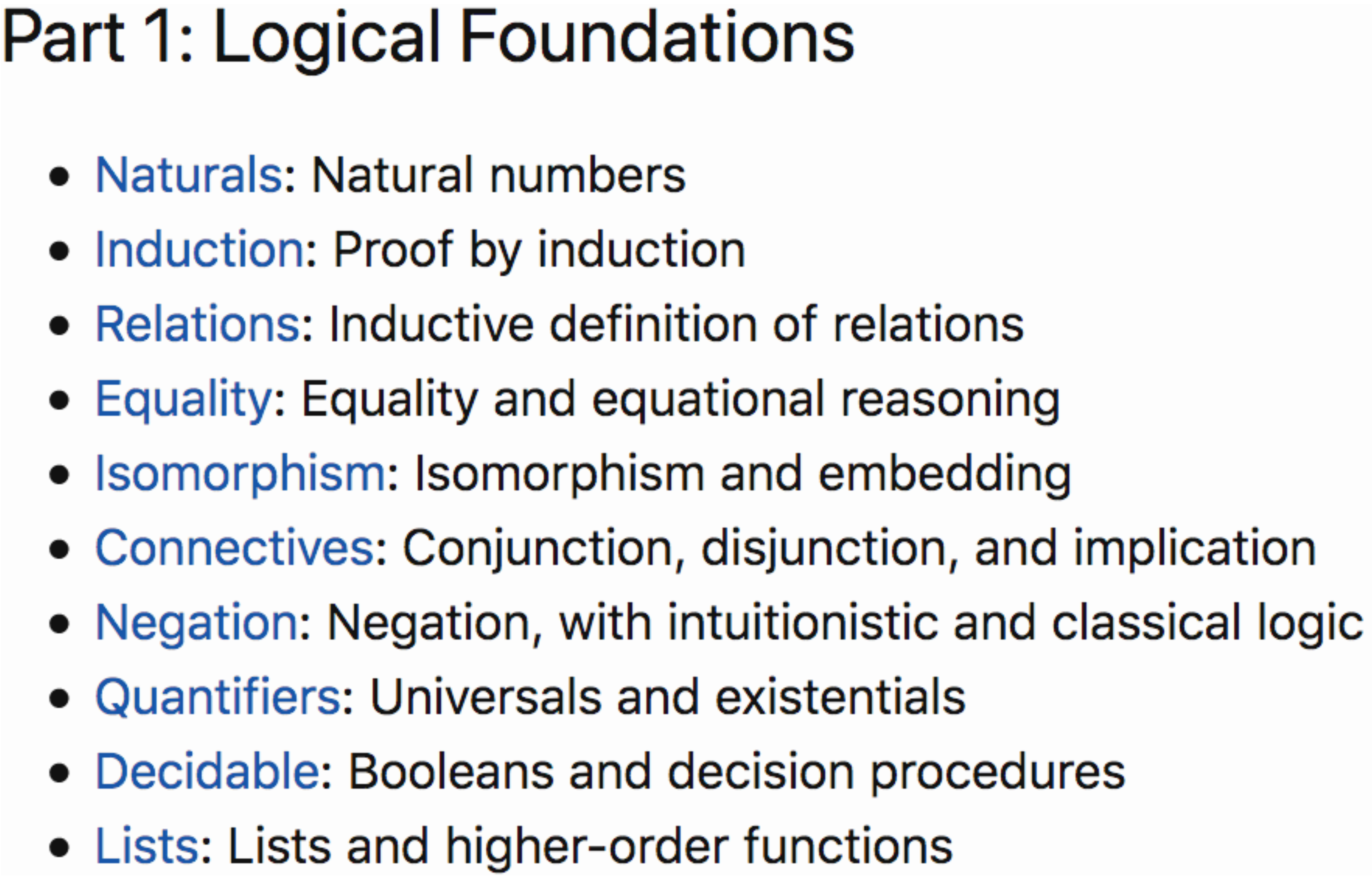
You: "How does PLFA compare to SF?"

Me: "Uh, we're pretty close, actually..."

New Syllabus

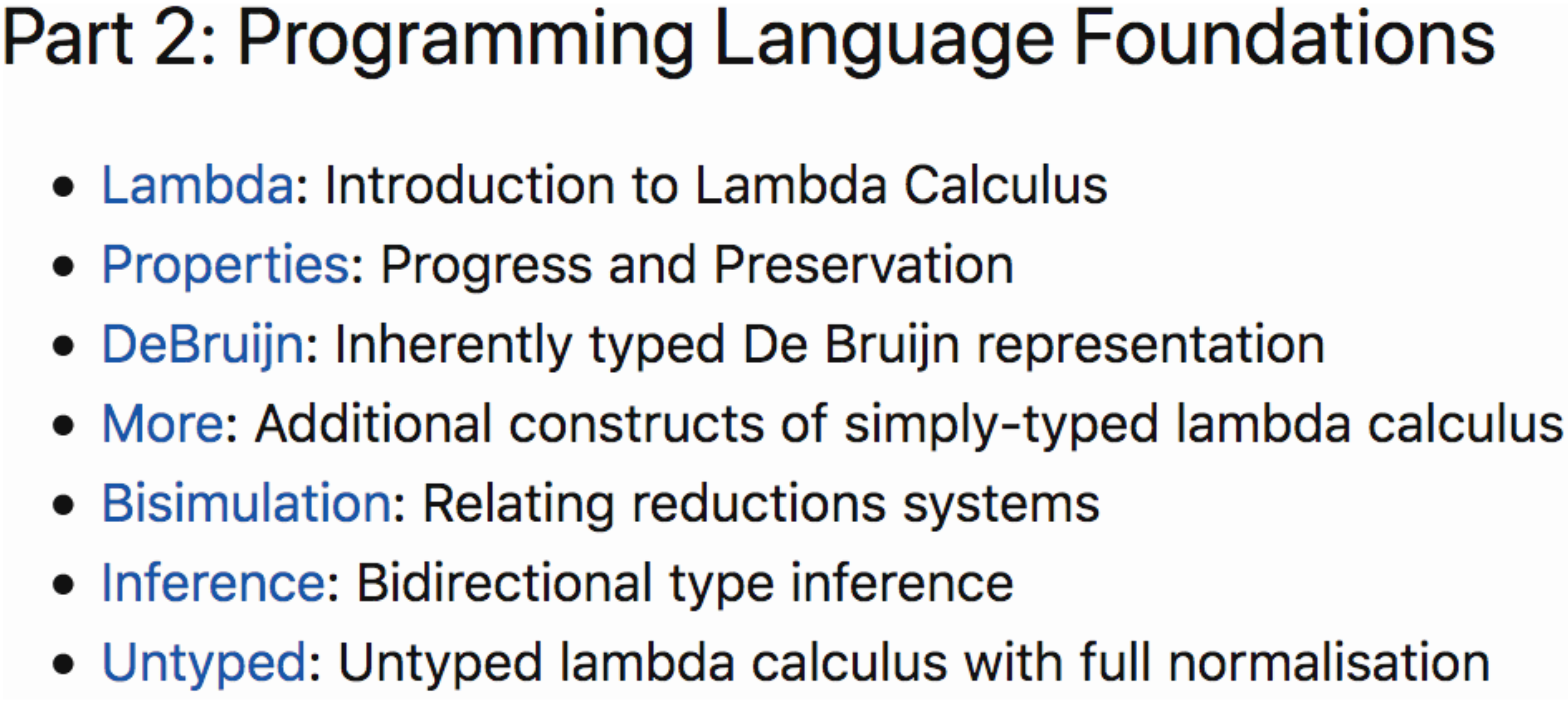
- inductive definitions
- operational semantics
- ~~untyped λ-calculus~~
- simply typed λ-calculus
- ~~references and exceptions~~
- records and subtyping
- ~~Featherweight Java~~
- functional programming
- logic (and Curry-Howard)
- `while` programs
- program equivalence
- Hoare Logic
- Coq

- ~~Coq~~
- ~~while programs~~
- ~~Hoare Logic~~
- ~~records and subtyping~~
- Agda
- untyped λ-calculus
- deBruijn indices
- bidirectional typing



Part 1: Logical Foundations

- Naturals: Natural numbers
- Induction: Proof by induction
- Relations: Inductive definition of relations
- Equality: Equality and equational reasoning
- Isomorphism: Isomorphism and embedding
- Connectives: Conjunction, disjunction, and implication
- Negation: Negation, with intuitionistic and classical logic
- Quantifiers: Universals and existentials
- Decidable: Booleans and decision procedures
- Lists: Lists and higher-order functions



Part 2: Programming Language Foundations

- Lambda: Introduction to Lambda Calculus
- Properties: Progress and Preservation
- DeBruijn: Inherently typed De Bruijn representation
- More: Additional constructs of simply-typed lambda calculus
- Bisimulation: Relating reductions systems
- Inference: Bidirectional type inference
- Untyped: Untyped lambda calculus with full normalisation

You: "Okay. What are some fundamental differences?"

Cultural Differences
Booleans vs. Decidable

Progress and Preservation equals Evaluation

Is Coq The Ultimate TA?

Pros:

- Can really build everything we need from scratch
- Curry-Howard
  - Proving = programming
- Good automation

Cons:

- Curry-Howard
  - Proving = programming → deep waters
  - Constructive logic can be confusing to students
- Annoyances
  - Lack of animation facilities
  - “User interface”
    - Notation facilities
    - Choice of variable names

> My Coq proof scripts do not have the conciseness and elegance of Jérôme Vouillon's. Sorry, I've been using Coq for only 6 years...
>
> – Leroy (2005)

Aside: the `normalize` Tactic

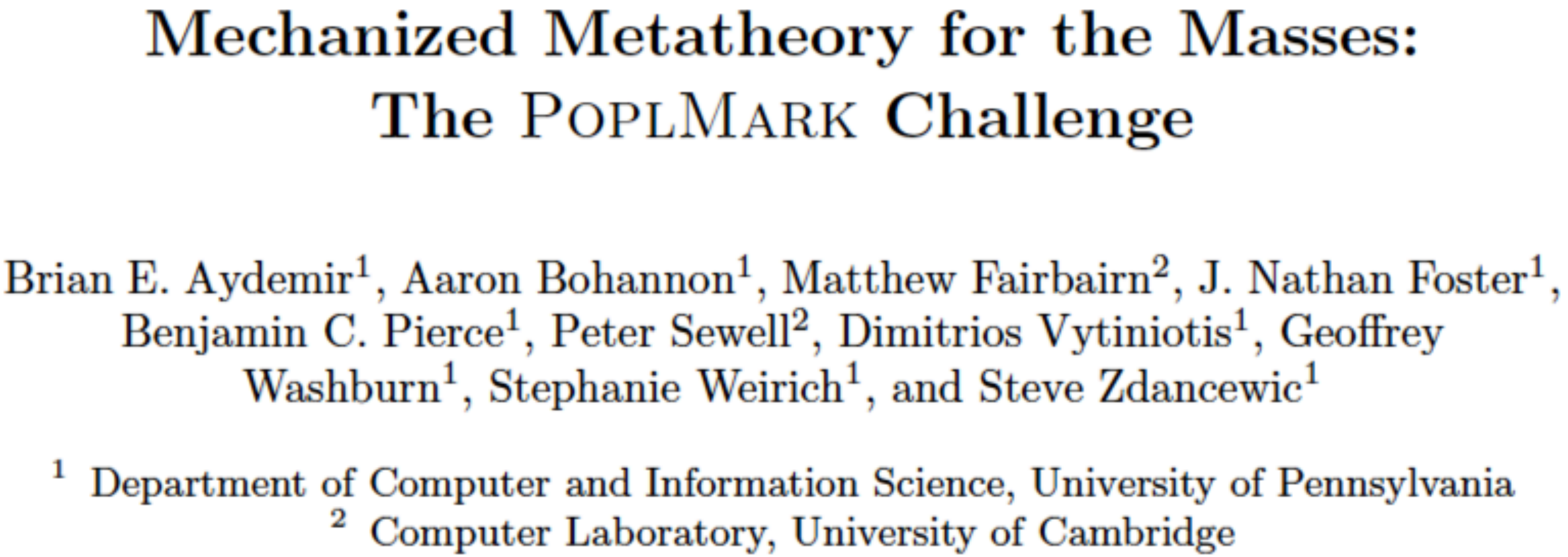
When experimenting with definitions of programming languages in Coq, we often want to see what a particular concrete term steps to — i.e., we want to find proofs for goals of the form `t ==>* t'`, where `t` is a completely concrete term and `t'` is unknown. These proofs are quite tedious to do by hand.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



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 $\mathtt{t} \longrightarrow \mathtt{t}'$. □

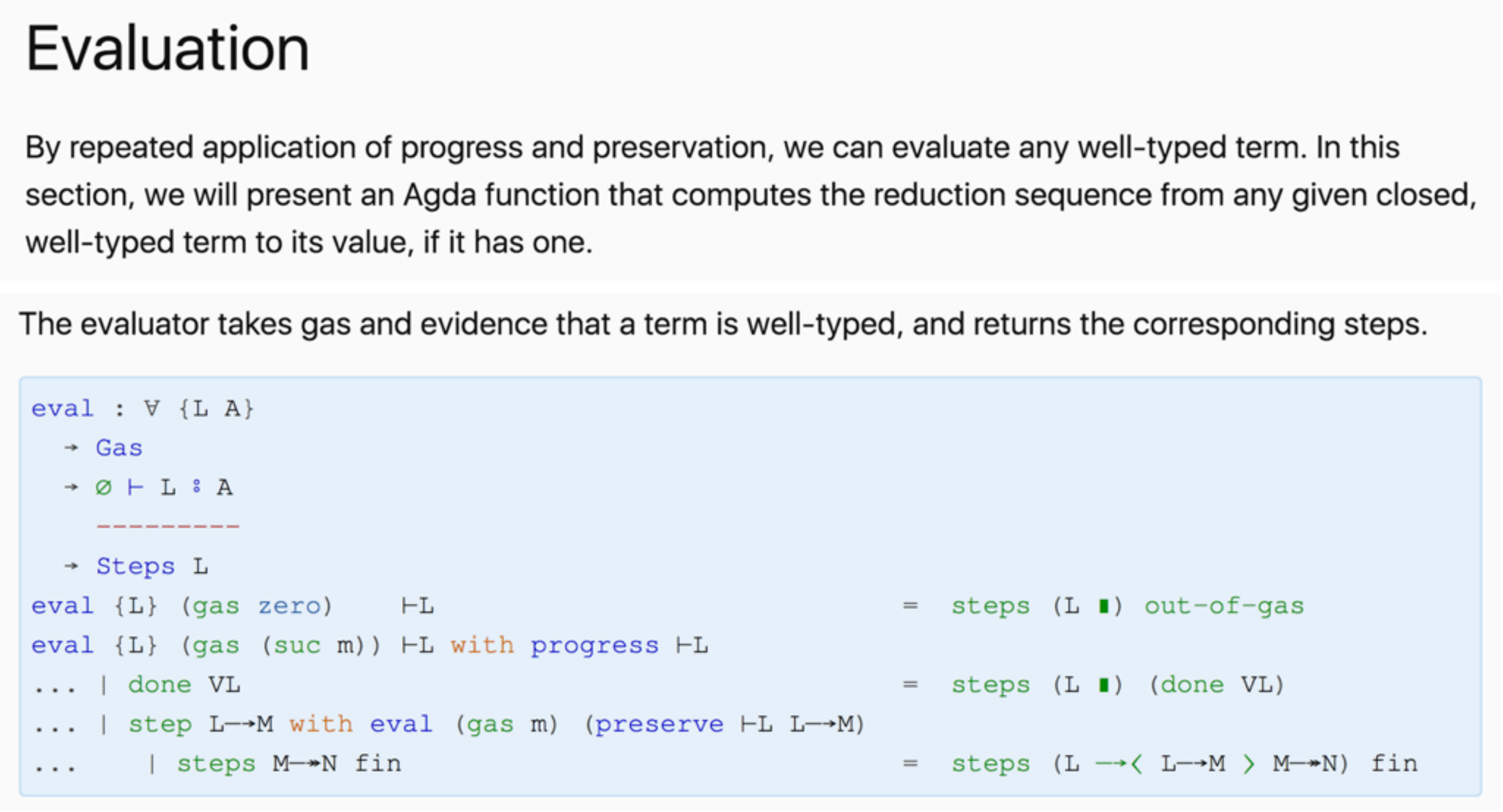
Challenge 3: Testing and Animating with Respect to the Semantics

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 $\mathtt{t} \longrightarrow \mathtt{t}'$.

How to Animate a Language

- repeatedly apply progress and preservation: it's evaluation!
- progress proof is an evaluation strategy: determines which step you take
- reservations about non-confluent systems

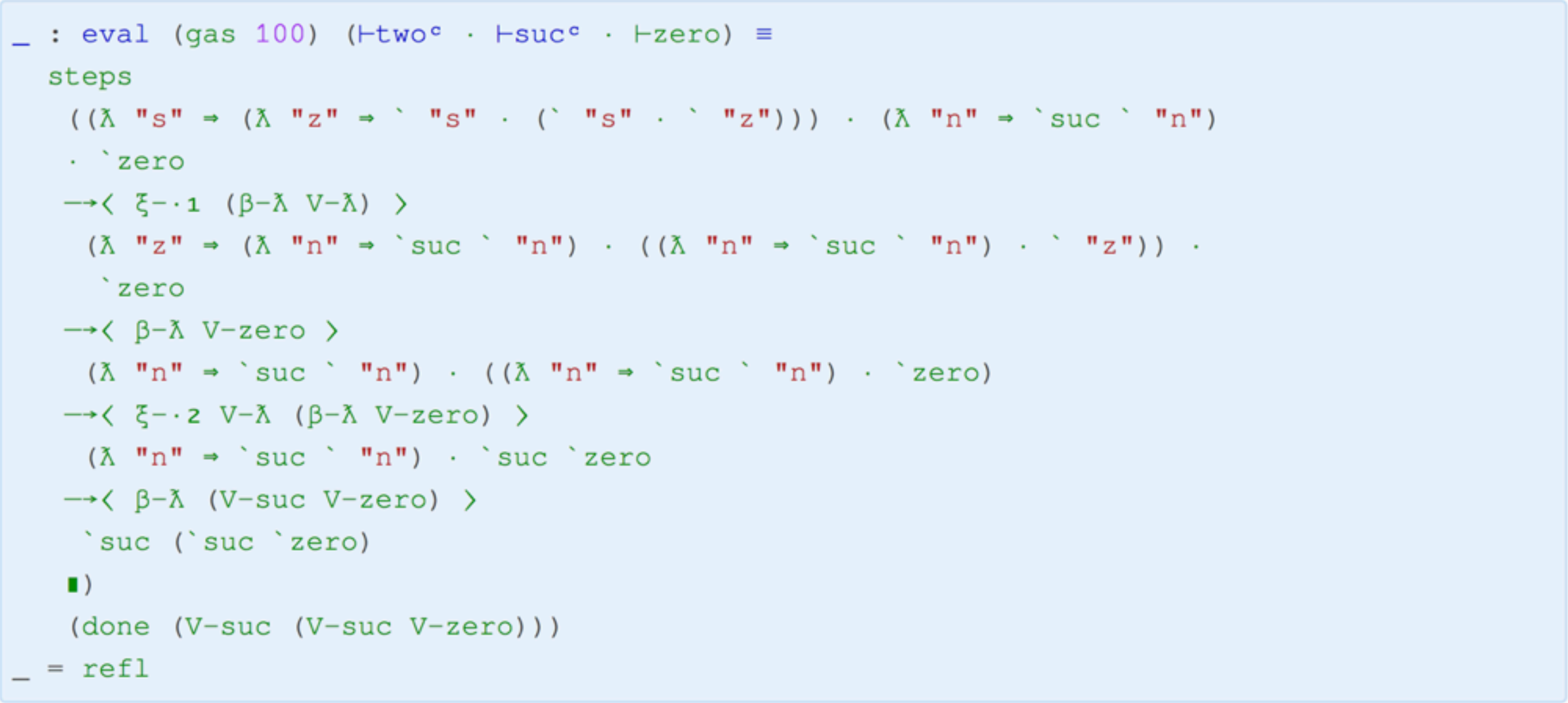


Evaluation

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
```

Inherently-Typed Terms & deBruijn Indices

The POPLMark Tarpit

- Dealing carefully with variable binding is hard; doing it formally is even harder
- What to do?
  - DeBruijn indices?
  - Locally Nameless?
  - Switch to Isabelle? Twelf?
  - Finesse the problem!

Type-and-Scope Safe Programs and Their Proofs

Guillaume Allais
gallais@cs.ru.nl
Radboud University,
The Netherlands

James Chapman Conor McBride
{james.chapman,conor.mcbride}@strath.ac.uk
University of Strathclyde, UK

James McKinna
james.mckinna@ed.ac.uk
University of Edinburgh, UK

Abstract

We abstract the common type-and-scope safe structure from computations on λ-terms that deliver, e.g., renaming, substitution, evaluation, CPS-transformation, and printing with a name supply. By exposing this structure, we can prove generic simulation and fusion lemmas relating operations built this way. This work has been fully formalised in Agda.

Categories and Subject Descriptors D.2.4 [*Software / Program Verification*]: Correctness Proofs; D.3.2 [*Language Classifications*]: Applicative (functional) languages; F.3.2 [*Semantics of Programming Languages*]: Denotational semantics, Partial evaluation

```
ren : (∀σ. Var σ Γ → Var σ Δ) → (∀σ. Tm σ Γ → Tm σ Δ)
ren ρ (`var v)  = `var (ρ v)
ren ρ (f `$ t)  = ren ρ f `$ ren ρ t
ren ρ (`λ b)    = `λ (ren ((su ∘ ρ) −, ze) b)

sub : (∀σ. Var σ Γ → Tm σ Δ) → (∀σ. Tm σ Γ → Tm σ Δ)
sub ρ (`var v)  = ρ v
sub ρ (f `$ t)  = sub ρ f `$ sub ρ t
sub ρ (`λ b)    = `λ (sub ((ren su ∘ ρ) −, `var ze) b)
```

Figure 1. Renaming and Substitution for the STλC

Cheap Tricks for Substitution

A Cheap Solution

- Observation: If we only ever substitute closed terms, then capture-incurring and capture-avoiding substitution behave the same.
- Second observation [Tolmach]: Replacing the standard weakening+permutation with a "context invariance" lemma makes this presentation *very* clean.

```
Reserved Notation "'[' x ':=' s ']' t" (at level 20).

Fixpoint subst (x : string) (s : tm) (t : tm) : tm :=
  match t with
  | var x' ⇒
      if eqb_string x x' then s else t
  | abs x' T t1 ⇒
      abs x' T (if eqb_string x x' then t1 else ([x:=s] t1))
  | app t1 t2 ⇒
      app ([x:=s] t1) ([x:=s] t2)
  | tru ⇒
      tru
  | fls ⇒
      fls
  | test t1 t2 t3 ⇒
      test ([x:=s] t1) ([x:=s] t2) ([x:=s] t3)
  end

where "'[' x ':=' s ']' t" := (subst x s t).
```

Technical note: Substitution becomes trickier to define if we consider the case where `s`, the term being substituted for a variable in some other term, may itself contain free variables. Since we are only interested here in defining the `step` relation on *closed* terms (i.e., terms like `\x:Bool. x` that include binders for all of the variables they mention), we can sidestep this extra complexity, but it must be dealt with when formalizing richer languages.

For example, using the definition of substitution above to substitute the *open* term `s = \x:Bool. r`, where `r` is a *free* reference to some global resource, for the variable `z` in the term `t = \r:Bool. z`, where `r` is a bound variable, we would get `\r:Bool. \x:Bool. r`, where the free reference to `r` in `s` has been "captured" by the binder at the beginning of `t`.

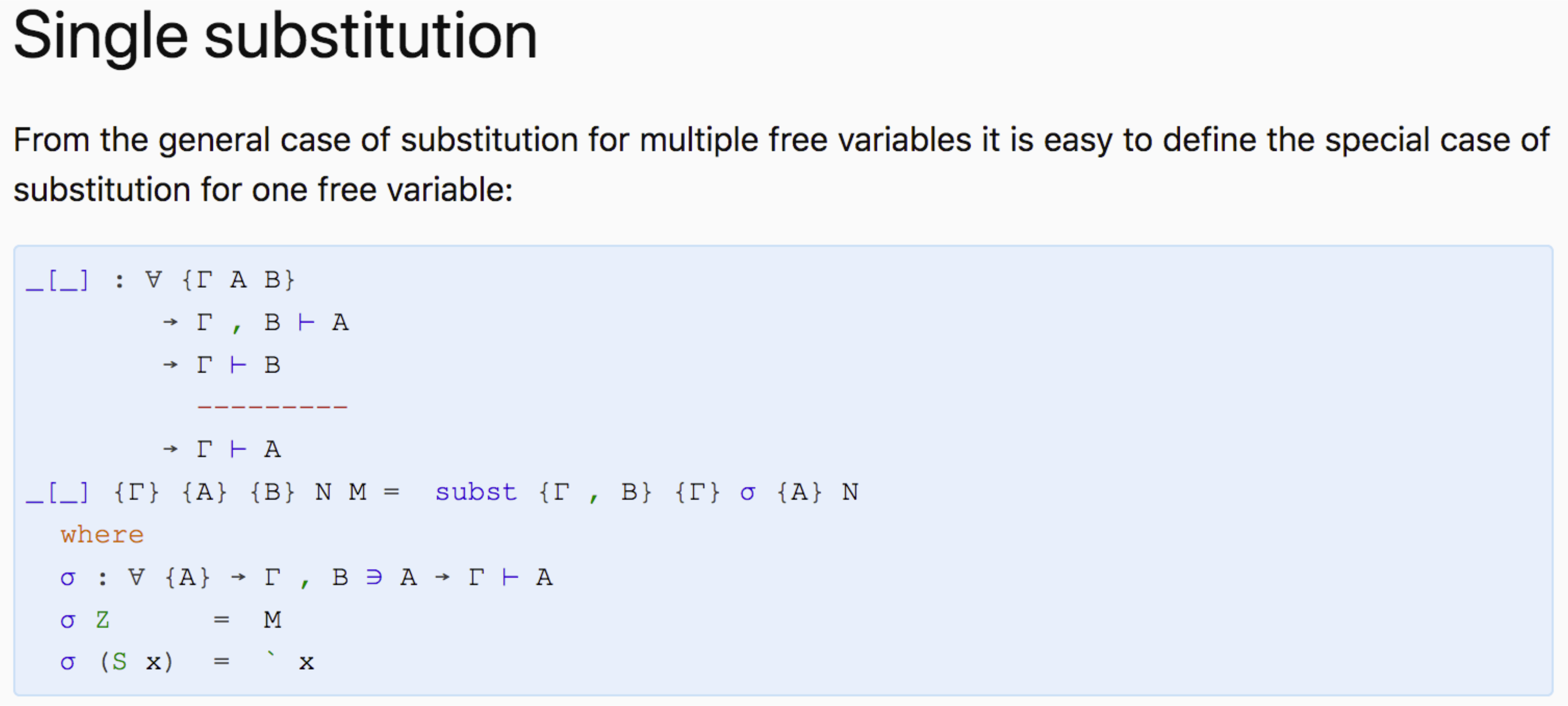
Why would this be bad? Because it violates the principle that the names of bound variables do not matter. For example, if we rename the bound variable in `t`, e.g., let `t' = \w:Bool. z`, then `[x:=s]t'` is `\w:Bool. \x:Bool. r`, which does not behave the same as `[x:=s]t = \r:Bool. \x:Bool. r`. That is, renaming a bound variable changes how `t` behaves under substitution.

```
Lemma substitution_preserves_typing : ∀Gamma x U t v T,
  (x ⊢> U ; Gamma) ⊢ t ∈ T →
  empty ⊢ v ∈ U →
  Gamma ⊢ [x:=v]t ∈ T.
```

One technical subtlety in the statement of the lemma is that we assume `v` has type `U` in the *empty* context — in other words, we assume `v` is closed. This assumption considerably simplifies the `T_Abs` case of the proof (compared to assuming `Gamma ⊢ v ∈ U`, which would be the other reasonable assumption at this point) because the context invariance lemma then tells us that `v` has type `U` in any context at all — we don't have to worry about free variables in `v` clashing with the variable being introduced into the context by `T_Abs`.

Here is the formal statement and proof that substitution preserves types:

```
subst : ∀ {Γ x N V A B}
  → ∅ ⊢ V ⦂ A
  → Γ , x ⦂ A ⊢ N ⦂ B
    --------------------
  → Γ ⊢ N [ x := V ] ⦂ B

subst {x = y} ⊢V (⊢` {x = x} Z) with x ≟ y
... | yes refl        =  weaken ⊢V
... | no  x≢y         =  ⊥-elim (x≢y refl)
subst {x = y} ⊢V (⊢` {x = x} (S x≢y ∋x)) with x ≟ y
... | yes refl        =  ⊥-elim (x≢y refl)
... | no  _           =  ⊢` ∋x
subst {x = y} ⊢V (⊢ƛ {x = x} ⊢N) with x ≟ y
... | yes refl        =  ⊢ƛ (drop ⊢N)
... | no  x≢y         =  ⊢ƛ (subst ⊢V (swap x≢y ⊢N))
subst ⊢V (⊢L · ⊢M)    =  (subst ⊢V ⊢L) · (subst ⊢V ⊢M)
subst ⊢V ⊢zero        =  ⊢zero
subst ⊢V (⊢suc ⊢M)    =  ⊢suc (subst ⊢V ⊢M)
subst {x = y} ⊢V (⊢case {x = x} ⊢L ⊢M ⊢N) with x ≟ y
... | yes refl        =  ⊢case (subst ⊢V ⊢L) (subst ⊢V ⊢M) (drop ⊢N)
... | no  x≢y         =  ⊢case (subst ⊢V ⊢L) (subst ⊢V ⊢M) (subst ⊢V (swap x≢y ⊢N))
subst {x = y} ⊢V (⊢μ {x = x} ⊢M) with x ≟ y
... | yes refl        =  ⊢μ (drop ⊢M)
... | no  x≢y         =  ⊢μ (subst ⊢V (swap x≢y ⊢M))
```



Single substitution

From the general case of substitution for multiple free variables it is easy to define the special case of substitution for one free variable:

```
_[_] : ∀ {Γ A B}
        → Γ , B ⊢ A
        → Γ ⊢ B
          ---------
        → Γ ⊢ A
_[_] {Γ} {A} {B} N M =  subst {Γ , B} {Γ} σ {A} N
  where
  σ : ∀ {A} → Γ , B ∋ A → Γ ⊢ A
  σ Z      =  M
  σ (S x)  =  ` x
```

Formalising languages following ACMM

```
ext    : (∀ {A}   →     γ ∋ A →     δ ∋ A)   -- extend
         ---------------------------------   -- simultaneous
       → (∀ {A B} → γ , B ∋ A → δ , B ∋ A)   -- renaming
                                             -- ↓
rename : (∀ {A}   →     γ ∋ A →     δ ∋ A)   -- apply
         ---------------------------------   -- simultaneous
       → (∀ {A}   →     γ ⊢ A →     δ ⊢ A)   -- renaming
                                             -- ↓
exts   : (∀ {A}   →     γ ∋ A →     δ ⊢ A)   -- extend
         ---------------------------------   -- simultaneous
       → (∀ {A B} → γ , B ∋ A → δ , B ⊢ A)   -- substitution
                                             -- ↓
subst  : (∀ {A}   →     γ ∋ A →     δ ⊢ A)   -- apply
         ---------------------------------   -- simultaneous
       → (∀ {A}   →     γ ⊢ A →     δ ⊢ A)   -- substitution
```

A Cheap Solution

- Observation: If we only ever substitute closed terms, then capture-incurring and capture-avoiding substitution behave the same.
- Second observation [Tolmach]: Replacing the standard weakening+permutation with a “context invariance” lemma makes this presentation *very* clean.
- Downside: Doesn’t work for System F

System F in Agda, for fun and profit

James Chapman[1], Roman Kireev[1], Chad Nester[2], and Philip Wadler[2]

[1] Input Output HK Ltd, Hong Kong {james.chapman,roman.kireev}@iohk.io
[2] University of Edinburgh, UK {cnester,wadler}@inf.ed.ac.uk

Abstract. System F, also known as the polymorphic λ-calculus, is a typed λ-calculus independently discovered by the logician Jean-Yves Girard and the computer scientist John Reynolds. We consider $F_{\omega\mu}$, which adds higher-order kinds and iso-recursive types. We present the first complete, intrinsically typed, executable, formalisation of System $F_{\omega\mu}$ that we are aware of. The work is motivated by verifying the core language of a smart contract system based on System $F_{\omega\mu}$. The paper is a literate Agda script [15]

You: "Did you make the right choice?"

Me: "What do you mean?"

You: "Well, you know, Agda, no tactics?"

Me: "Uh... we should talk..."

Quantitative Type Theory

Linear Types by Counting

Formalising languages following QTT

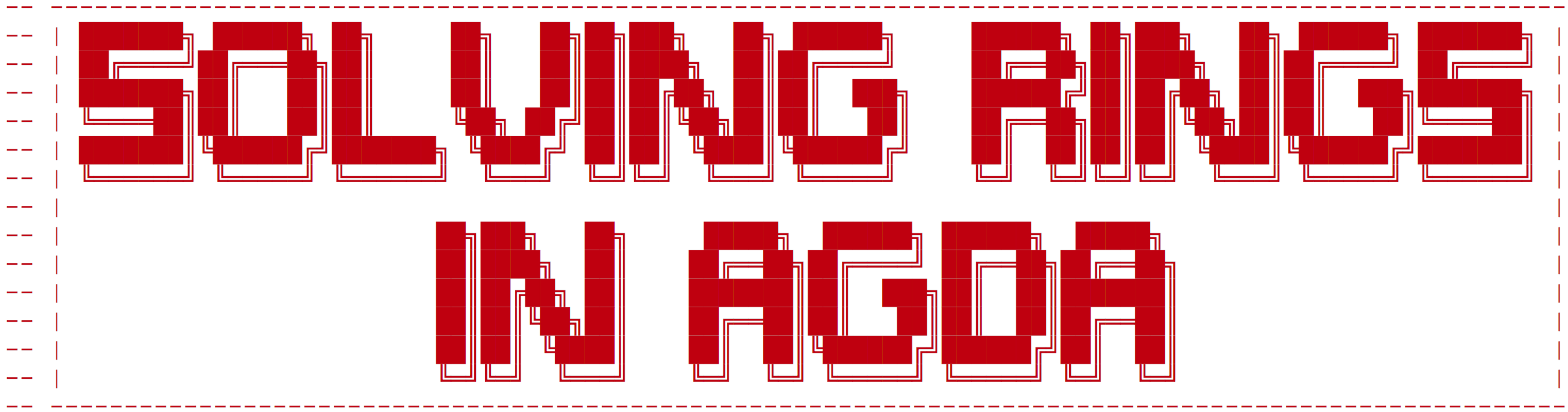
- contexts w/ resource annotations
- count resource usage with $\{0, 1, \omega\}$
- contexts parameterised over precontexts on the type level

```
_ : Ctxt (∅ , A , B , C)
_ = ∅ , 1 · A , 0 · B , 0 · C
```

Formalising languages following QTT

```
_ : ∅ , 1 • A , 0 • A ⊸ A ⊢ A
_ = ` S Z

_ : ∅ , 1 • A , 1 • A ⊸ A ⊢ A
_ = (` Z) · (` S Z)

_ : ∅ , ω • A , 1 • A ⊸ A ⊸ A ⊢ A
_ = (` Z) · (` S Z) · (` S Z)
```

```
lem-ƛ : ∀ {γ δ} (Γ : Context γ) {A} {π} {Ξ : Matrix γ δ} → _
lem-ƛ {γ} {δ} Γ {A} {π} {Ξ} =
  begin
    (π ** 0s , π * 1# · A) ⋈ (Γ ⊛ (λ x → Ξ x , 0# · A))
  ≡⟨ ⊛-zeroʳ Γ Ξ |> cong ((π ** 0s , π * 1# · A) ⋈_) ⟩
    (π ** 0s , π * 1# · A) ⋈ (Γ ⊛ Ξ , 0# · A)
  ≡⟨⟩
    π ** 0s ⋈ Γ ⊛ Ξ , (π * 1#) + 0# · A
  ≡⟨ **-zeroʳ π |> cong ((_, (π * 1#) + 0# · A) ∘ (_⋈ Γ ⊛ Ξ)) ∘ sym ⟩
    0s ⋈ Γ ⊛ Ξ , (π * 1#) + 0# · A
  ≡⟨ ⋈-identityˡ (Γ ⊛ Ξ) |> cong (_, (π * 1#) + 0# · A) ⟩
    Γ ⊛ Ξ , (π * 1#) + 0# · A
  ≡⟨ +-identityʳ (π * 1#) |> cong (Γ ⊛ Ξ ,_· A) ⟩
    Γ ⊛ Ξ , π * 1# · A
  ≡⟨ *-identityʳ π |> cong (Γ ⊛ Ξ ,_· A) ⟩
    Γ ⊛ Ξ , π · A
  ∎
```



```
--
--                                           Donnacha Oisín Kidney
--                                               17 April 2019
--
-- Compiled with Agda version 2.6.0, and standard library commit 09ebff3a4724891d8805eced1d48ecda7908d914

-- This contains the worked-through source code for:
--
--     "Automatically And Efficiently Illustrating Polynomial Equalities in Agda"
--
--     We present a new library which automates the construction of equivalence
--     proofs between polynomials over commutative rings and semirings in the
--     programming language Agda. It is asymptotically faster than Agda's existing
--     solver. We use Agda's reflection machinery to provide a simple interface to
--     the solver, and demonstrate an interesting use of the constructed relations:
--     step-by-step solutions.
```

```
--------------------------------------------------------------------------------
-- Don't understand why something works? Wanna get it explained to you? Now   --
-- you can! The solver can generate step-by-step, human-readable solutions    --
-- for learning purposes.                                                     --
--                                                                            --
module TracedExamples where                                                   --
  import Data.Nat.Show                                                        --
  open import EqBool                                                          --
  open import Relation.Traced Nat.ring  Data.Nat.Show.show public             --
  open AlmostCommutativeRing tracedRing                                       --
                                                                              --
  lemma : ∀ x y → x + y * 1 + 3 ≈ 2 + 1 + y + x                               --
  lemma = solve tracedRing                                                    --
                                                                              --
  explained                                                                   --
    : showProof (lemma "x" "y") ≡ "x + y + 3"                                 --
                                ∷ "    ={ +-comm(x, y + 3) }"                 --
                                ∷ "y + 3 + x"                                 --
                                ∷ "    ={ +-comm(y, 3) }"                     --
                                ∷ "3 + y + x"                                 --
                                ∷ []                                          --
  explained = ≡.refl                                                          --
--------------------------------------------------------------------------------
```

Programming Language Foundations in Agda

- it's there for you to use!
- it's free!
- it covers:
  - logical foundations
  - functional programming
  - simply-typed lambda calculus

Programming Language Foundations in Agda

- it's there for you to use!
- it's free!
- soon it will cover:
  - system F
  - denotational semantics
  - whatever you'd like to contribute!

Conor McBride, James McKinna, Ulf Norell, Andreas Abel, David Darais, Marko Dimjašević, Zbigniew Stanasiuk, Yasu Watanabe, Jeremy G. Siek, Chad Nester, Juhana Laurinharju, phi16, Jonathan Prieto, Alexandru Brisan, Michel Steuwer, caryoscelus, Reza Gharibi, Lorenzo Martinico, Sebastian Miele, Murilo Giacometti Rocha, Spencer Whitt, Isaac Elliott, Ingo Blechschmidt, Fangyi Zhou, Torsten Grust, Nicolas Wu, Kartik Singhal, Phil de Joux, Stephan Boyer, Zack Grannan, Léo Gillot-Lamure, Kenneth MacKenzie, Chike Abuah, Alexandre Moreno, James Wood, Stefan Kranich, kenichi-asai, Rodrigo Bernardo, Orestis Melkonian, Deniz Alp, Nathaniel Carroll, Guillaume Allais, Nils Anders Danielsson, Miëtek Bak, Gergő Érdi, Ádam Sandberg Eriksson, David Janin, András Kovács, Liam O'Connor, N. Raghavendra, Roman Kireev, Amr Sabry