Programming Language Foundations in Agda

Philip Wadler
(with Wen Kokke and Jeremy Siek)
University of Edinburgh / IOHK / Rio de Janeiro
Lambda Days, Friday 19 February 2021
Bugs
Dad jokes
The Curry-Howard homeomorphism
Programming Language Foundations in Agda
(Programming Language)
Foundations in Agda
Programming (Language Foundations) in Agda
The origins of PLFA
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
The Software Foundations series is a broad introduction to the mathematical underpinnings of reliable software.

The principal novelty of the series is that every detail is one hundred percent formalized and machine-checked: the entire text of each volume, including the exercises, is literally a "proof script" for the Coq proof assistant.

The exposition is intended for a broad range of readers, from advanced undergraduates to PhD students and researchers. No specific background in logic or programming languages is assumed, though a degree of mathematical maturity is helpful. A one-semester course can expect to cover Logical Foundations plus most of Programming Language Foundations or Verified Functional Algorithms, or selections from both.

**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.

**Volume 2**

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

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
• Getting Started

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: Boolean and decision procedures
Extrinsic vs Intrinsic: Intrinsic is Golden
Extrinsic:
Named variables, separate types

Intrinsic:
de Bruijn indexes, inherently typed

Lines of code, omitting examples

\[
\frac{451}{275} = 1.6
\]

\[
\frac{275}{451} = 0.6
\]
id : Set
id = String

data Term : Set where
  _ : Id → Term
  λ__ : Id → Term → Term
  __ : Term → Term → Term

data Type : Set where
  __ : Type → Type → Type
  `N : Type

data Context : Set where
  ⊥ : Context
  _,_ : Context → Id → Type → Context

data _∈_ : Context → Id → Type → Set where

  Z : ∀ (Γ × A)
    ---------------------------------
    → Γ , x : A ∈ x : A

  S : ∀ (Γ × y A B)
    → x ≠ y
    → Γ ⊢ x : A
    ---------------------------------
    → Γ , y : B ∈ x : A

data _→_ : Context → Term → Type → Set where

  |- : ∀ (Γ × A)
    → Γ ⊢ x : A
    ---------------------------------
    → Γ ⊢ x : A

  |-λ : ∀ (Γ × N A B)
    → Γ , x : A ⊢ N : B
    ---------------------------------
    → Γ ⊢ λ x → N : A → B

  _,_ : ∀ (Γ × L M A B)
    → Γ ⊢ L : A → B
    → Γ ⊢ M : A
    ---------------------------------
    → Γ ⊢ L · M : B
data Type : Set where
  _\_\_ : Type → Type → Type
  N : Type

data Context : Set where
  ∅ : Context
  _/\_ : Context → Type → Context

data _∈\_ : Context → Type → Set where

  Z : ∀ {Γ A}
      _____________
      → Γ , A ∈ A

  S_ : ∀ {Γ A B}
      → Γ ∈ A
      _____________
      → Γ , B ∈ A

data _∈\_ : Context → Type → Set where

  _\_ : ∀ {Γ} {A}
      → Γ ∈ A
      _____________
      → Γ ⊢ A

  λ_ : ∀ {Γ} {A B}
      → Γ , A ⊢ B
      _____________
      → Γ ⊢ A → B

  _\_ : ∀ {Γ} {A B}
      → Γ ⊢ A → B
      → Γ ⊢ A
      _____________
      → Γ ⊢ B
Progress + Preservation = Animation
Functional Big-step Semantics

Scott Owens\textsuperscript{1}, Magnus O. Myreen\textsuperscript{2}, Ramana Kumar\textsuperscript{3}, and Yong Kiam Tan\textsuperscript{4}

\textsuperscript{1} School of Computing, University of Kent, UK
\textsuperscript{2} CSE Department, Chalmers University of Technology, Sweden
\textsuperscript{3} NICTA, Australia
\textsuperscript{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-
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 \( t \Rightarrow^* t' \), 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 \texttt{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.

```coq
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'' : \exists 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.
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
  - Absence of interface
  - Notation facilities
  - Choice of variable names

My Coq proof scripts do not have the conciseness and elegance of Jérôme Voillon's. Sorry, I've been using Coq for only 6 years...

- Leroy (2005)
Mechanized Metatheory for the Masses: The POPLMARK Challenge

Brian E. Aydemir¹, Aaron Bohannon¹, Matthew Fairbairn², J. Nathan Foster¹, Benjamin C. Pierce¹, Peter Sewell², Dimitrios Vytiniotis¹, Geoffrey Washburn¹, Stephanie Weirich¹, and Steve Zdancewic¹

¹ Department of Computer and Information Science, University of Pennsylvania
² Computer Laboratory, University of Cambridge
Challenge 2A: Type Safety for Pure $F_\omega$.

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 $F_\omega$.

3.3 Theorem [Preservation]: If $\Gamma \vdash t : T$ and $t \rightarrow t'$, then $\Gamma \vdash t' : T$. \hfill $\square$

3.4 Theorem [Progress]: If $t$ is a closed, well-typed $F_\omega$ term (i.e., if $\vdash t : T$ for some $T$), then either $t$ is a value or else there is some $t'$ with $t \rightarrow t'$. \hfill $\square$
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 $F_{\prec}$ terms $t$ and $t'$, decide whether $t \rightarrow t'$.
2. Given $F_{\prec}$ terms $t$ and $t'$, decide whether $t \rightarrow^* t' \not\rightarrow$, where $\rightarrow^*$ is the reflexive-transitive closure of $\rightarrow$.
3. Given an $F_{\prec}$ term $t$, find a term $t'$ such that $t \rightarrow t'$.
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.

\[
\text{eval} : \forall \{L \ A\} \rightarrow \text{Gas} \\
\rightarrow \emptyset \vdash L : A \\
\text{Steps } L \text{ eval } \{L\} \ (\text{gas } \text{zero}) \vdash L = \text{steps } (L \vdash) \text{ out-of-gas} \\
\text{eval } \{L\} \ (\text{gas } (\text{suc } m)) \vdash L \text{ with progress } \vdash L \\
... \mid \text{done } VL = \text{steps } (L \vdash) \text{ (done } VL) \\
... \mid \text{step } L \rightarrow M \text{ with eval } (\text{gas } m) \ (\text{preserve } \vdash L L \rightarrow M) \\
... \mid \text{steps } M \rightarrow N \text{ fin} = \text{steps } (L \rightarrow (L \rightarrow M \rightarrow M) \rightarrow N) \text{ fin}
\]
eval (gas 100) (≡two ⋅ ≡suc ⋅ ≡zero) =

steps
((λ "s" ⇒ (λ "z" ⇒ ` "s" ⋅ (` "s" ⋅ ` "z"))) ⋅ (λ "n" ⇒ `suc ` "n")
 ⋅ `zero
→⟨ ξ−1 (β−λ V−λ) ⟩
(λ "z" ⇒ (λ "n" ⇒ `suc ` "n") ⋅ ((λ "n" ⇒ `suc ` "n") ⋅ `zero)
 ⋅ `zero
→⟨ β−λ V−zero ⟩
(λ "n" ⇒ `suc ` "n") ⋅ ((λ "n" ⇒ `suc ` "n") ⋅ `zero)
→⟨ ξ−2 V−λ (β−λ V−zero) ⟩
(λ "n" ⇒ `suc ` "n") ⋅ `suc `zero
→⟨ β−λ (V−suc V−zero) ⟩
 `suc (`suc `zero)
)
(done (V−suc (V−suc V−zero)))

≡ refl
Agda for Fun and Profit: IOHK and Cardano
Lambda Calculus  
Alonzo Church, 1932-40

Natural Deduction  
Gerhard Gentzen, 1935
System F

Polymorphic Lambda Calculus

Jean-Yves Girard, 1972

John Reynolds, 1974
Plutus Core

**Kinds**

\[ \text{J,K} ::= \]

\[ * \]

\[ \text{J \to K} \]

**Terms**

\[ \text{L,M,N} ::= \]

\[ x \]

\[ \lambda x : A.N \]

\[ \Lambda x : K.N \]

\[ \text{wrap } M \]

\[ \text{unwrap } M \]

\[ \rho \]
Plutus Core in Agda

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

data _⊢_* : Ctx• → Kind → Set where
  ` : Φ ⊢* J
      ______
      → Φ ⊢* J

  χ : Φ ,* K ⊢* J
     __________
     → Φ ⊢* K ⇒ J

  _·_ : Φ ⊢* K ⇒ J
     _________
     → Φ ⊢* K

  Π : Φ ,* K ⊢* *
     __________
     → Φ ⊢* *

data _⊢_ : ∀ Γ → 1 Γ ⊢* J → Set where
  ` : Γ ⊃ A
      ______
      → Γ ⊢ A

  χ : Γ , A ⊢ B
     __________
     → Γ ⊢ A ⇒ B

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

  λ : Γ ,* K ⊢ B
     __________
     → Γ ⊢ Π B

  _··_ : Γ ⊢ Π B
     _________
     → (A : ∥ Γ ⊢* K)

  conv : A ≡β B
     _________
     → Γ ⊢ B [ A ]

  conv : A ≡β B
     _________
     → 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 2018 Best Paper Award
1st Place

Philip Wadler

Programming Language Foundations in Agda
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.
Propositions as Types
Philip Wadler
University of Edinburgh
Strange Loop
St Louis, 25 August 2015

"Propositions as Types" by Philip Wadler
61,321 views
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')
  rewrite ≡-comm m' (suc n')
    [ sym (+-assoc n' m' (n' * m'))]
≡-comm n' m'
≡-comm n' m'
≡-comm n' (suc m')
≡-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!
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: \texttt{subst N x M} vs \texttt{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 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 \( \emptyset \vdash M : A \) then either \( M \) is a value or there is an \( N \) such that \( M \rightarrow N \).

To formulate this property, we first introduce a relation that captures what it means for a term \( M \) to make progress.

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

  step : \( \forall \{ N \} \) → \( M \rightarrow N \) → 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 \( M \rightarrow N \), 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.

\[
\begin{align*}
\text{progress} &: \forall \{M, A\} \\
& \quad \varnothing \vdash M : A \\
& \quad \text{---------}
& \quad \text{Progress } M \\
\text{progress} &: (\vdash \lambda \vdash N) \\
& \quad \text{= done } V-\lambda \\
\text{progress} &: (\vdash L \cdot \vdash M) \text{ with progress } \vdash L \\
& \quad \text{... | step } L \rightarrow L' \\
& \quad \text{... | done } V L \text{ with progress } \vdash M \\
& \quad \text{... | step } M \rightarrow M' \\
& \quad \text{... | done } V M \text{ with canonical } \vdash L V L \\
& \quad \text{... | C-} \lambda _{ } \\
& \quad \text{progress } \vdash \text{zero} \\
& \quad \text{= done } V-\text{zero} \\
\text{progress} &: (\vdash \text{suc} \vdash M) \text{ with progress } \vdash M \\
& \quad \text{... | step } M \rightarrow M' \\
& \quad \text{... | done } V M \\
\text{progress} &: (\vdash \text{case } l \vdash M \vdash N) \text{ with progress } \vdash L \\
& \quad \text{... | step } L \rightarrow L' \\
& \quad \text{... | done } V L \text{ with canonical } \vdash L V L \\
& \quad \text{... | C-zero} \\
& \quad \text{... | C-suc CL} \\
\text{progress} &: (\vdash \mu \vdash M) \\
& \quad \text{= step } \beta-\mu
\end{align*}
\]
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 $L \cdot M$, recursively apply progress to the derivation that $L$ is well-typed.
  - If the term steps, we have evidence that $L \rightarrow L'$, which by $\xi$ means that our original term steps to $L' \cdot M$
  - 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 $M \rightarrow M'$, which by $\xi$ means that our original term steps to $L \cdot M'$. Step $\xi$ 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 $\beta$.

The remaining cases are similar. If by induction we have a step case we apply a $\xi$ rule, and if we have a done case then either we have a value or apply a $\beta$ rule. For fixpoint, no induction is required as the $\beta$ 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.
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.

\begin{verbatim}
  Theorem progress : \forall t T,
  empty |- t \in T \rightarrow
  value t \lor \exists t', t \Rightarrow t'.
\end{verbatim}

**Proof:** By induction on the derivation of \( |- t \in T \).

- The last rule of the derivation cannot be \( T_{\text{Var}} \), since a variable is never well typed in an empty context.
- The \( T_{\text{True}}, T_{\text{False}}, \) and \( T_{\text{Abs}} \) 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 \( T_{\text{App}} \), then \( t \) has the form \( t_1 \, t_2 \) for some \( t_1 \) and \( t_2 \), where \( |- t_1 \in T_2 \rightarrow T \) 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 \( ST_{\text{AppAbs}} \).
    - Otherwise, \( t_2 \) can take a step, and hence so can \( t_1 \, t_2 \) by \( ST_{\text{App2}} \).
  - If \( t_1 \) can take a step, then so can \( t_1 \, t_2 \) by \( ST_{\text{App1}} \).
- If the last rule of the derivation is \( T_{\text{If}} \), then \( t = \text{if} \, t_1 \, \text{then} \, t_2 \, \text{else} \, t_3 \), where \( t_1 \) has type \( \text{Bool} \). By the IH, \( t_1 \) either is a value or takes a step.
  - If \( t_1 \) is a value, then since it has type \( \text{Bool} \) it must be either \( \text{true} \) or \( \text{false} \). If it is \( \text{true} \), then \( t \) steps to \( t_2 \); otherwise it steps to \( t_3 \).
  - Otherwise, \( t_1 \) takes a step, and therefore so does \( t \) (by \( ST_{\text{If}} \)).
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.