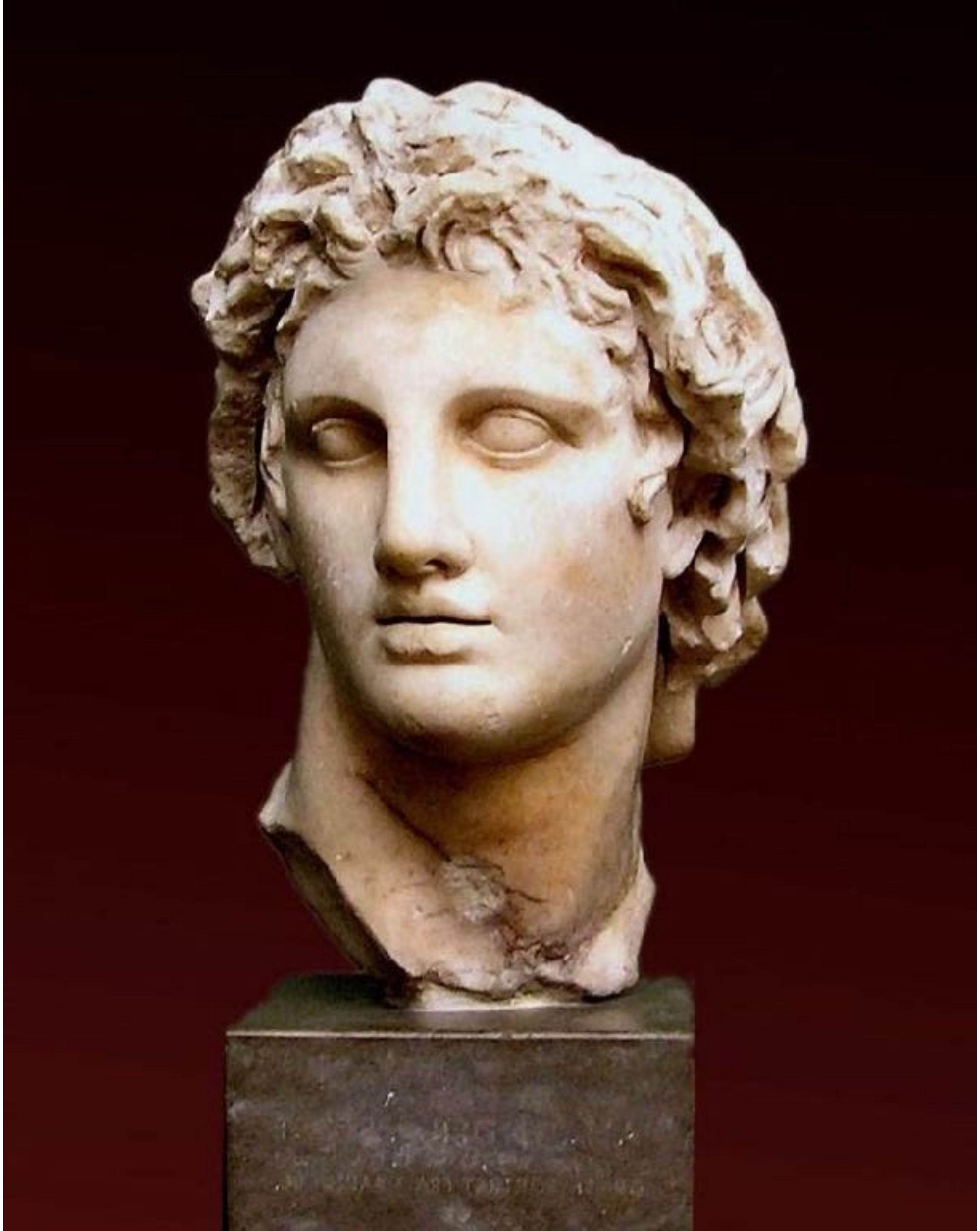
Programming Language Foundations in Agda

Philip Wadler
(with Wen Kokke)
University of Edinburgh / IOHK / Rio de Janeiro
LFCS Lab Lunch, 9 October 2018

John Hughes



Rio de Janeiro

http://plfa.inf.ed.ac.uk

Please send your comments!

https://github.com/plfa

And your pull requests!

Programming Language Foundations in Agda

(Programming Language) Foundations in Agda

Programming (Language Foundations) in Agda

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 ∈ T_2 → T and |- t_2 ∈ 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.
```

Inherently Typed is Golden

	Lines of code, omitting examples
Named variables, separate types	451
de Bruijn indexes, inherently typed	275

451 / 275 = 1.6

275 / 451 = 0.6

```
Id : Set
Id = String

data Term : Set where
  `_      :  Id → Term
  ƛ_⇒_    :  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 x A}
      ------------------
    → Γ , x ⦂ A ∋ x ⦂ A

  S : ∀ {Γ x y A B}
    → x ≢ y
    → Γ ∋ x ⦂ A

    → Γ , y ⦂ B ∋ x ⦂ A

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
```

```
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

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

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

Progress + Preservation
= Evaluation

By analogy, we will use the name *gas* for the parameter which puts a bound on the number of reduction steps. Gas is specified by a natural number.

```
data Gas : Set where
  gas : ℕ → Gas
```

When our evaluator returns a term `N`, it will either give evidence that `N` is a value or indicate that it ran out of gas.

```
data Finished (N : Term) : Set where

  done :
      Value N
      ----------
    → Finished N

  out-of-gas :
      ----------
      Finished N
```

Given a term `L` of type `A`, the evaluator will, for some `N`, return a reduction sequence from `L` to `N` and an indication of whether reduction finished.

```
data Steps (L : Term) : Set where

  steps : ∀ {N}
    → L —↠ N
    → Finished N
      ----------
    → Steps L
```

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
```

Substitution:
Single vs Simultaneous

Boolean vs Decidable

http://plfa.inf.ed.ac.uk

https://github.com/plfa

Or search for “Kokke Wadler”

Please send your comments and pull requests!