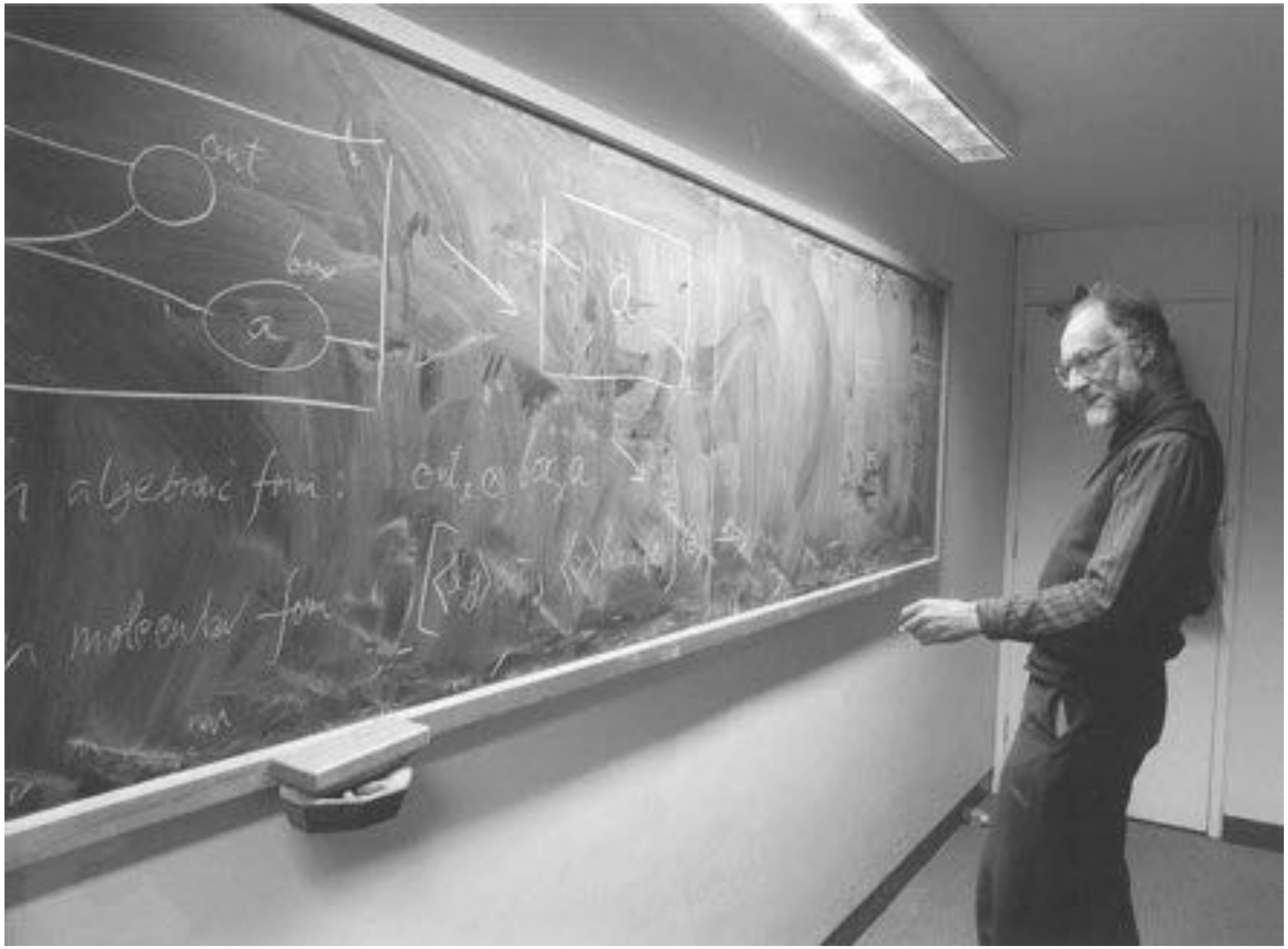
Programming Language Foundations in Agda

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

Proof Assistants

Robin Milner



LCF

Gerard Huet & Thierry Coquand

Coq

Conor McBride and James McKenna

Epigram

Ulf Norell and Andreas Abel

Agda

Propositions as Types

DOI:00.0000/000

Connecting mathematical logic and computation, it ensures that some aspects of programming are absolute.

BY PHILIP WADLER

Propositions as Types

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,

Figure 1. Gerhard Gentzen (1935)—Natural Deduction.

$$\dfrac{A \qquad B}{A \mathbin{\&} B}\;\&\text{-I} \qquad \dfrac{A \mathbin{\&} B}{A}\;\&\text{-E}_1 \qquad \dfrac{A \mathbin{\&} B}{B}\;\&\text{-E}_2$$

$$\dfrac{\begin{array}{c}[A]^x \\ \vdots \\ B\end{array}}{A \supset B}\;\supset\text{-I}^x \qquad \dfrac{A \supset B \qquad A}{B}\;\supset\text{-E}$$

Figure 2. A proof.

$$\dfrac{\dfrac{\dfrac{[B \mathbin{\&} A]^z}{A}\;\&\text{-E}_2 \qquad \dfrac{[B \mathbin{\&} A]^z}{B}\;\&\text{-E}_1}{A \mathbin{\&} B}\;\&\text{-I}}{(B \mathbin{\&} A) \supset (A \mathbin{\&} B)}\;\supset\text{-I}^z$$

Figure 4. Simplifying a proof.

$$\dfrac{\dfrac{\dfrac{\dfrac{[B \,\&\, A]^z}{A}\,\&\text{-E}_2 \qquad \dfrac{[B \,\&\, A]^z}{B}\,\&\text{-E}_1}{A \,\&\, B}\,\&\text{-I}}{(B \,\&\, A) \supset (A \,\&\, B)}\,\supset\text{-I}^z \qquad \dfrac{B \qquad A}{B \,\&\, A}\,\&\text{-I}}{A \,\&\, B}\,\supset\text{-E}$$

$$\Downarrow$$

$$\dfrac{\dfrac{\dfrac{B \qquad A}{B \,\&\, A}\,\&\text{-I}}{A}\,\&\text{-E}_2 \qquad \dfrac{\dfrac{B \qquad A}{B \,\&\, A}\,\&\text{-I}}{B}\,\&\text{-E}_1}{A \,\&\, B}\,\&\text{-I}$$

$$\Downarrow$$

$$\dfrac{A \qquad B}{A \,\&\, B}\,\&\text{-I}$$

Figure 5. Alonzo Church (1935)—Lambda Calculus.

$$\frac{M : A \qquad N : B}{\langle M, N \rangle : A \times B}\times\text{-I} \qquad \frac{L : A \times B}{\pi_1 L : A}\times\text{-E}_1 \qquad \frac{L : A \times B}{\pi_2 L : B}\times\text{-E}_2$$

$$\frac{\begin{array}{c}[x : A]^x \\ \vdots \\ N : B\end{array}}{\lambda x. N : A \to B}\to\text{-I}^x \qquad \frac{L : A \to B \qquad M : A}{L\,M : B}\to\text{-E}$$

Figure 6: A program.

$$\dfrac{\dfrac{\dfrac{[z : B \times A]^z}{\pi_2\, z : A}\times\text{-E}_2 \qquad \dfrac{[z : B \times A]^z}{\pi_1\, z : B}\times\text{-E}_1}{\langle \pi_2\, z, \pi_1\, z \rangle : A \times B}\times\text{-I}}{\lambda z\,.\,\langle \pi_2\, z, \pi_1\, z \rangle : (B \times A) \to (A \times B)}\to\text{-I}^z$$

Figure 8. Evaluating a program.

$$
\dfrac{
  \dfrac{
    \dfrac{[z : B \times A]^z}{\pi_2\, z : A}\,{\times}\text{-E}_2
    \qquad
    \dfrac{[z : B \times A]^z}{\pi_1\, z : B}\,{\times}\text{-E}_1
  }{\langle \pi_2\, z, \pi_1\, z \rangle : A \times B}\,{\times}\text{-I}
}{\lambda z.\, \langle \pi_2\, z, \pi_1\, z \rangle : (B \times A) \to (A \times B)}\,{\to}\text{-I}^z
\qquad
\dfrac{y : B \qquad x : A}{\langle y, x \rangle : B \times A}\,{\times}\text{-I}
$$

$$
\dfrac{\lambda z.\, \langle \pi_2\, z, \pi_1\, z \rangle : (B \times A) \to (A \times B) \qquad \langle y, x \rangle : B \times A}{(\lambda z.\, \langle \pi_2\, z, \pi_1\, z \rangle)\ \langle y, x \rangle : A \times B}\,{\to}\text{-E}
$$

$$\Downarrow$$

$$
\dfrac{
  \dfrac{\dfrac{y : B \qquad x : A}{\langle y, x \rangle : B \times A}\,{\times}\text{-I}}{\pi_2\, \langle y, x \rangle : A}\,{\times}\text{-E}_2
  \qquad
  \dfrac{\dfrac{y : B \qquad x : A}{\langle y, x \rangle : B \times A}\,{\times}\text{-I}}{\pi_1\, \langle y, x \rangle : B}\,{\times}\text{-E}_1
}{\langle \pi_2\, \langle y, x \rangle, \pi_1\, \langle y, x \rangle \rangle : A \times B}\,{\times}\text{-I}
$$

$$\Downarrow$$

$$
\dfrac{x : A \qquad y : B}{\langle x, y \rangle : A \times B}\,{\times}\text{-I}
$$

Philip Wadler,
Propositions as Types,
Communications of the ACM
December 2015

Programming Language Foundations in Agda

(Programming Language) Foundations in Agda

Programming (Language Foundations) in Agda

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

Please send your comments!

https://github.com/plfa

And your pull requests!

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 $|- t_1 \in T_2 \rightarrow T$ 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_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
  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 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

  ⊢λ : ∀ {Γ x 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

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

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

Progress + Preservation
= Evaluation

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 $\Gamma \vdash \mathtt{t}' : \mathtt{T}$. $\square$

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

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 $F_{<:}$ terms $\mathtt{t}$ and $\mathtt{t}'$, decide whether $\mathtt{t} \longrightarrow \mathtt{t}'$.
2. Given $F_{<:}$ 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 $F_{<:}$ 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
```

Substitution:
Single vs Simultaneous

Boolean vs Decidable

Conclusions

vilem
@buggymcbugfix

Follow

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!