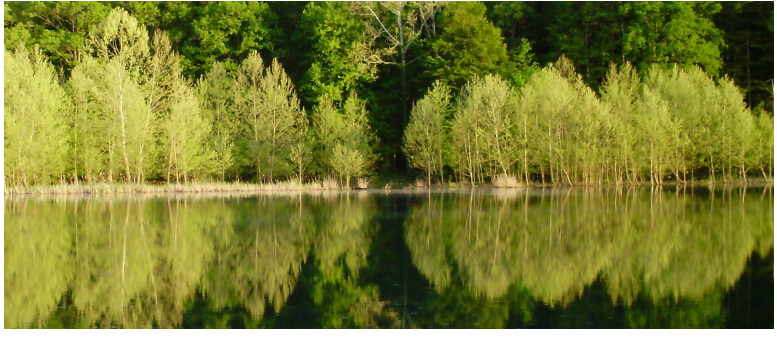
Introduction to Agda

Jeremy G. Siek



KTH Royal Institute of Technology, 2024

Who am I? Jeremy Siek

Professor at Indiana University, Bloomington.

I conduct research to improve programming languages. I invented *gradual typing* with Walid Taha.

I teach the data structures course to undergraduate students. I teach compilers to both undergraduate and graduate students. Last year I published *Essentials of Compilation* (MIT Press) with versions in Racket and Python.

I've used proof assistants in my research for almost 20 years. I initially used Isabelle, but I switched to Agda in 2019. I wrote Part 3 of *Programming Language Foundations in Agda*.

Introduction to Agda

Agda is a pure functional language with dependent types that is used to create formal models and prove properties of those models.

Data Types

Natural numbers:

```
data ℕ : Set where
  zero : ℕ
  suc : ℕ → ℕ
```

Singly-linked lists:

```
data List (A : Set) : Set where
  [] : List A
  _::_ : (x : A) (xs : List A) → List A
```

Recursive Functions

List append:

```
infixr 5 _++_

_++_ : {A : Set} → List A → List A → List A
[] ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

List reverse:

```
rev : {A : Set} → List A → List A
rev [] = []
rev (x :: xs) = rev xs ++ (x :: [])
```

Curry-Howard Correspondence

$$\textit{Types} \Longleftrightarrow \textit{Propositions}$$

$$\textit{Programs} \Longleftrightarrow \textit{Proofs}$$

$$\textit{Recursion} \Longleftrightarrow \textit{Induction}$$

Equality (Propositional)

In Agda, we represent relations using dependent data types.

```
infix 4 _≡_
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x

sym : {A : Set} {x y : A} → x ≡ y → y ≡ x
sym refl = refl

trans : {A : Set} {x y z : A} → x ≡ y → y ≡ z → x ≡ z
trans refl refl = refl

cong : {A B : Set} (f : A → B) {x y : A} → x ≡ y → f x ≡ f y
cong f refl = refl
```

Proving Properties

```
rev-rev : {A : Set} → (xs : List A) → rev (rev xs) ≡ xs
rev-rev = ?
```

Interactive proof demonstration.

Modeling Programming Languages

The untyped lambda calculus using the de Bruijn representation of variables.

Abstract syntax:

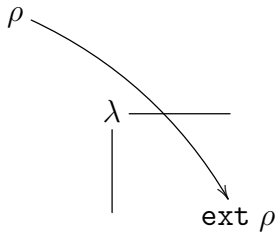
```
data Term : Set where
  Var : ℕ → Term
  Lam : Term → Term
  App : Term → Term → Term
```

de Bruijn example:

$$\lambda x.\lambda y.y \quad \Longrightarrow \quad \lambda.\lambda.0$$
$$\lambda x.\lambda y.x \quad \Longrightarrow \quad \lambda.\lambda.1$$

Renaming Variables

```
ext : (ℕ → ℕ) → (ℕ → ℕ)
ext ρ zero = zero
ext ρ (suc x) = suc (ρ x)
```

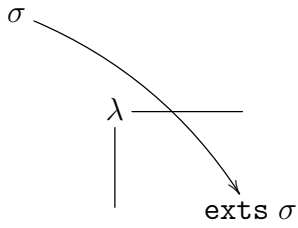



```
ren : (ℕ → ℕ) → Term → Term
ren ρ (Var x) = Var (ρ x)
ren ρ (Lam N) = Lam (ren (ext ρ) N)
ren ρ (App L M) = App (ren ρ L) (ren ρ M)
```

Type-Preserving Renaming and Substitution by Conor McBride

Simultaneous Substitution

```
exts : (ℕ → Term) → (ℕ → Term)
exts σ zero = Var zero
exts σ (suc x) = ren suc (σ x)
```




```
sub : (ℕ → Term) → Term → Term
sub σ (Var x) = σ x
sub σ (Lam N) = Lam (sub (exts σ) N)
sub σ (App L M) = App (sub σ L) (sub σ M)
```

Type-Preserving Renaming and Substitution by Conor McBride

Substitutions as Streams $[M_0, M_1, M_2, M_3, \ldots]$

```
cons : Term → (ℕ →Term) → (ℕ → Term)
cons M σ zero = M
cons M σ (suc x) = σ x

car : (ℕ → Term) → Term
car σ = σ 0

cdr : (ℕ → Term) → (ℕ → Term)
cdr σ x = σ (suc x)

sub0 : Term → Term → Term
sub0 M N = sub (cons M (λ x → Var x)) N
```

$$\texttt{cons}\, M_0\, [M_1, \ldots] = [M_0, M_1, \ldots]$$
$$\texttt{car}\, [M_0, M_1, \ldots] = M_0$$
$$\texttt{cdr}\, [M_0, M_1, \ldots] = [M_1, \ldots]$$

$$\texttt{cons}\, M\, [0, 1, \ldots] = [M, 0, 1, \ldots]$$

Lambda Calculus Reduction: “full β”

```
infix 2 _⟶_
data _⟶_ : Term → Term → Set where
  ξ₁ : {L L' M : Term}
     → L ⟶ L'
     → App L M ⟶ App L' M
  ξ₂ : {L M M' : Term}
     → M ⟶ M'
     → App L M ⟶ App L M'
  β : {N M : Term}
     → App (Lam N) M ⟶ sub0 M N
  ζ : {N N' : Term}
     → N ⟶ N'
     → Lam N ⟶ Lam N'
```

Multi-step Reduction

```
infix 2 _↠_
data _↠_ : Term → Term → Set where
  done : (M : Term)
    → M ↠ M
  step : {L M N : Term}
    → L ⟶ M
    → M ↠ N
    → L ↠ N
```

Proving Basic Properties of Reduction

Multi-step reduction is transitive

```
multi-trans : {L M N : Term} → L ↠ M → M ↠ N → L ↠ N
```

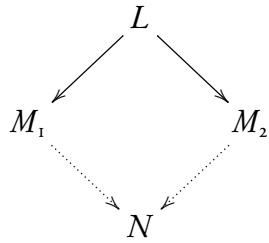
Multi-step is a congruence wrt. application

```
appL-cong : {L L' : Term} (M : Term)
  → L ↠ L'
  → App L M ↠ App L' M
```

Interactive proof demonstration.

Advanced Properties of Reduction

```
confluence : {L M₁ M₂ : Term}
  → L ↠ M₁ → L ↠ M₂
  → ∃[ N ] (M₁ ↠ N) × (M₂ ↠ N)
```



Require advanced properties of substitution:

```
sub-sub0-commute : {N M : Term}{σ : ℕ → Term}
  → sub σ (sub0 M N) ≡ sub0 (sub σ M) (sub (exts σ) N)
```

Problem: proving advanced properties of substitution is tedious to do for every new programming language.

Abstract Binding Trees

```
module ABT (Op : Set) (sig : Op → List ℕ) where

  data ABT : Set where
    Var : ℕ → ABT
    Node : (op : Op) → Args (sig op) → ABT

  data Args where
    nil : Args []
    cons : {b : ℕ} → ABT
        → {bs : List ℕ} → Args bs
        → Args (b :: bs)
```

Practical Foundations for Programming Languages by Robert Harper

Lambda Calculus as an ABT

```
data LamOps : Set where
  op-lam : LamOps
  op-app : LamOps

lam-sig : LamOps → List ℕ
lam-sig op-lam = 1 :: []
lam-sig op-app = 0 :: 0 :: []

open import ABT LamOps lam-sig

pattern Lam N = Node op-lam (cons N nil)
pattern App L M = Node op-app (cons L (cons M nil))
```

Renaming Variables in ABTs

Iterate ext multiple times:

```
extn : ℕ → (ℕ → ℕ) → (ℕ → ℕ)
extn zero ρ = ρ
extn (suc n) ρ = extn n (ext ρ)
```

Renaming ABT's:

```
ren : (ℕ → ℕ) → ABT → ABT
rens : (ℕ → ℕ) → {bs : List ℕ} → Args bs → Args bs

ren ρ (Var x) = Var (ρ x)
ren ρ (Node op args) = Node op (rens ρ args)
rens ρ nil = nil
rens ρ (cons {b} M args) = cons (ren (extn b ρ) M) (rens ρ args)
```

Simultaneous Substitution in ABTs

Iterate `exts` multiple times:

```
extsn : ℕ → (ℕ → ABT) → (ℕ → ABT)
extsn zero σ = σ
extsn (suc n) σ = extsn n (exts σ)
```

Substitution for ABT's:

```
sub : (ℕ → ABT) → ABT → ABT
subs : (ℕ → ABT) → {bs : List ℕ} → Args bs → Args bs

sub σ (Var x) = σ x
sub σ (Node op args) = Node op (subs σ args)
subs σ nil = nil
subs σ (cons {b} M args) = cons (sub (extsn b σ) M) (subs σ args)
```

Properties of Substitution, Once and For All!

```
scons : ABT → (ℕ → ABT) → (ℕ → ABT)
scons M σ zero = M
scons M σ (suc x) = σ x

sub0 : ABT → ABT → ABT
sub0 M N = sub (scons M (λ x → Var x)) N

sub-sub0-commute : {N M : ABT}{σ : ℕ → ABT}
  → sub σ (sub0 M N) ≡ sub0 (sub σ M) (sub (exts σ) N)
```

Conclusion

- Agda is a proof assistant that truly embraces the Curry-Howard correspondence: proving is programming!
- Agda is relatively easy to learn for functional programmers.
- de Bruijn indices with simultaneous substitution is a good way to formalize variables.
- Dependent types are good for more than fixed-length vectors!
- Abstract binding trees can free us from the tedium of reasoning about variables and substitution.