Dependently Typed Programming in Agda

Ulf Norell

TLDI'09

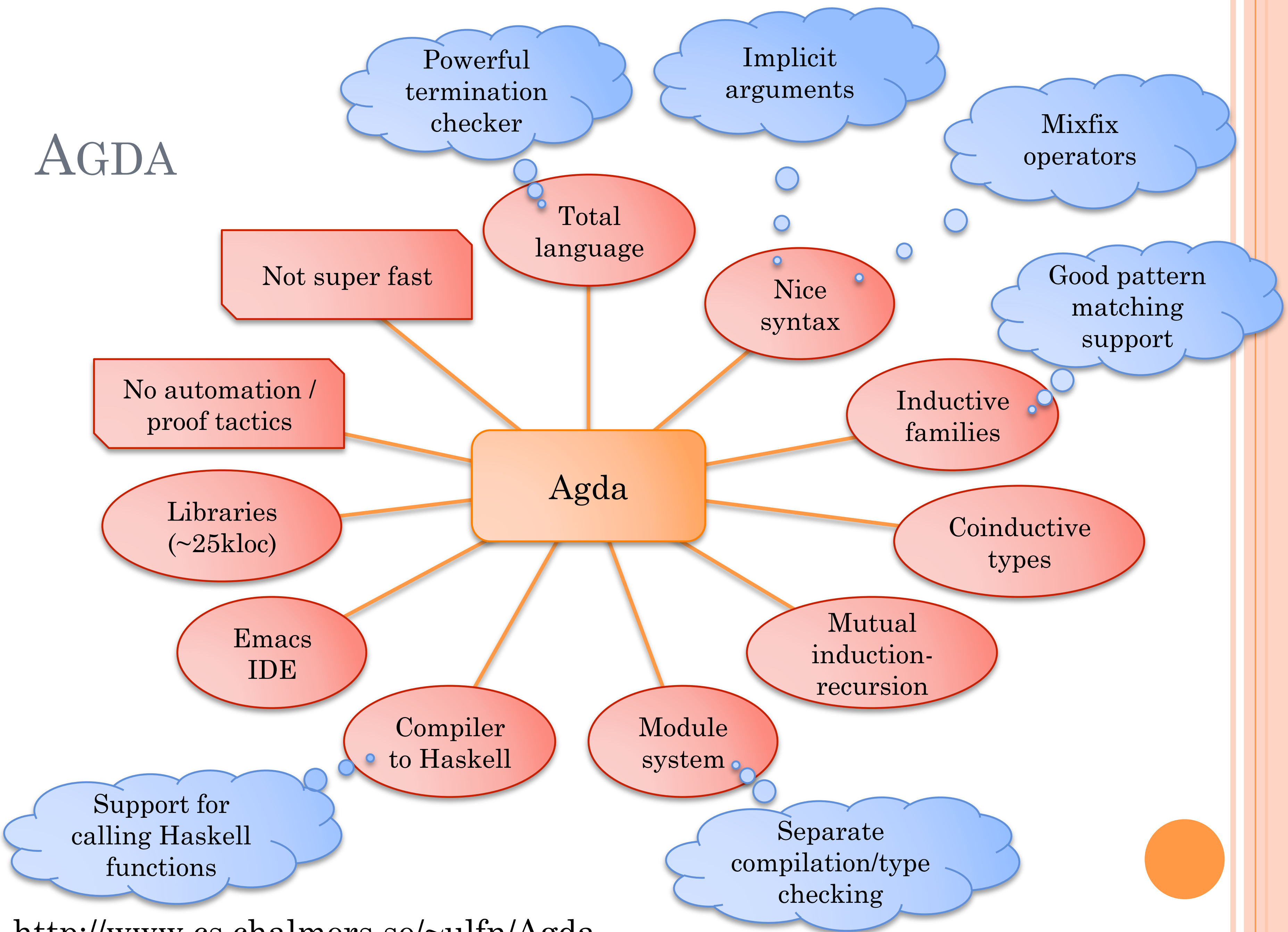
Savannah, Georgia

January 24, 2009

Dependently Typed Programming

- Dependently typed programs
  - as opposed to simply typed programs with dependently typed proofs
  - dependent types = more precise types
- Trade-off: precision vs. extra work
  - Often, more precise types does not mean more complicated programs
  - The type checker can do a lot of work for us
- Key tools
  - Indexed inductive definitions
  - Pattern matching

AGDA



http://www.cs.chalmers.se/~ulfn/Agda

Example - List Lookup

- Here's a familiar function

```
lookup : {A : Set} → List A → ℕ → Maybe A
lookup []       n       = nothing
lookup (x :: xs) zero    = just x
lookup (x :: xs) (suc n) = lookup xs n
```

- We could proceed to prove this function correct, but…
  - Proving properties of programs is tedious
  - Anytime you need to know that lookup does the right thing you have to invoke the correctness lemmas
  - Better: write the correct function to start with!

List lookup - Specification

- What does it mean to be an element in a list?

```
data _∈_ {A : Set}(x : A) : List A → Set where
  hd : ∀ {xs}    → x ∈ x :: xs
  tl : ∀ {xs y} → x ∈ xs → x ∈ y :: xs
```

- We can recover the index of x in xs from a proof of $x \in xs$.

```
index : ∀ {A x}{xs : List A} → x ∈ xs → ℕ
index hd       = zero
index (tl x) = suc (index x)
```

Correct List lookup

- A precise type for the result of lookup

```
data Lookup {A}(xs : List A) : ℕ → Set where
  inside  : ∀ x (p : x ∈ xs) → Lookup xs (index p)
  outside : ∀ m → Lookup xs (length xs + m)
```

- The *correct by construction* lookup function

```
lookup : ∀ {A}(xs : List A)(n : ℕ) → Lookup xs n
lookup []        n        = outside n
lookup (x :: xs) zero     = inside x hd
lookup (x :: xs) (suc n) with lookup xs n
lookup (x :: xs) (suc .(index p))         | inside y p = inside y (tl p)
lookup (x :: xs) (suc .(length xs + m))   | outside m  = outside m
```

What's the pattern here?

- Define the result type of a function so that it tells you something about the arguments
  - If $lookup\ xs\ n = outside$ we learn that $n \geq length\ xs$
  - If $lookup\ xs\ n = inside\ x\ p$ we learn that n is the index encoded by a proof p that $x \in xs$
- In the terminology of McBride and McKinna
  - *Lookup* $xs\ n$ is a *view* on natural numbers n describing how n can be seen as an index into xs.

Example – Type checking λ-calculus

- Let's start with the punch line

```
data Infer (Γ : Cxt) : Raw → Set where
  good : ∀ {τ}(u : Term Γ τ) → Infer Γ (erase u)
  bad  : ∀ {e} → Infer Γ e

infer : ∀ Γ (e : Raw) → Infer Γ e
```

Raw and Typed Terms

```
data Type : Set where
  ι   : Type
  _⇒_ : Type → Type → Type
```

```
Cxt = List Type
Var Γ τ = τ ∈ Γ
```

```
data Raw : Set where
  var : ℕ → Raw
  app : Raw → Raw → Raw
  lam : Type → Raw → Raw
```

```
data Term (Γ : Cxt) : Type → Set where
  var : ∀ {τ} → Var Γ τ → Term Γ τ
  app : ∀ {σ τ} → Term Γ (σ ⇒ τ) → Term Γ σ → Term Γ τ
  lam : ∀ σ {τ} → Term (σ :: Γ) τ → Term Γ (σ ⇒ τ)
```

Comparing Types

```
data TypeCmp : Type → Type → Set where
  eq     : ∀ {τ} → TypeCmp τ τ
  not-eq : ∀ {σ τ} → TypeCmp σ τ


_=?=_ : (σ τ : Type) → TypeCmp σ τ
ι =?= ι           = eq
ι =?= (τ ⇒ τ′) = not-eq
(σ ⇒ σ′) =?= ι = not-eq
(σ ⇒ σ′) =?= (τ ⇒ τ′) with σ =?= τ | σ′ =?= τ′
(σ ⇒ σ′) =?= (.σ ⇒ .σ′) | eq | eq = eq
(σ ⇒ σ′) =?= ( τ ⇒  τ′) | _  | _  = not-eq
```

ERASURE

```
data Raw : Set where
  var : ℕ → Raw
  app : Raw → Raw → Raw
  lam : Type → Raw → Raw

data Term (Γ : Cxt) : Type → Set where
  var : ∀ {τ} → Var Γ τ → Term Γ τ
  app : ∀ {σ τ} → Term Γ (σ ⇒ τ) → Term Γ σ → Term Γ τ
  lam : ∀ σ {τ} → Term (σ :: Γ) τ → Term Γ (σ ⇒ τ)

erase : ∀ {Γ τ} → Term Γ τ → Raw
erase (var x)   = var (index x)
erase (app u v) = app (erase u) (erase v)
erase (lam σ v) = lam σ (erase v)
```

The Type Checker

```
data Infer (Γ : Cxt) : Raw → Set where
  good : ∀ {τ}(u : Term Γ τ) → Infer Γ (erase u)
  bad  : ∀ {e} → Infer Γ e

infer : ∀ Γ (e : Raw) → Infer Γ e

infer Γ (var x) with lookup Γ x
infer Γ (var .(index x))        | inside τ x = good (var x)
infer Γ (var .(length Γ + m)) | outside m  = bad

infer Γ (app e₁ e₂) with infer Γ e₁ | infer Γ e₂
infer Γ (app ._ ._) | good {σ ⇒ τ} u | good {σ′} v with σ =?= σ′
infer Γ (app ._ ._) | good {σ ⇒ τ} u | good {.σ} v | eq = good (app u v)
infer Γ (app ._ ._) | good {σ ⇒ τ} u | good {σ′} v | not-eq = bad
infer Γ (app e₁ e₂) | _ | _ = bad

infer Γ (lam σ e) with infer (σ :: Γ) e
infer Γ (lam σ ._) | good u = good (lam σ u)
infer Γ (lam σ e)  | bad     = bad
```

Example – Compiling Expressions

- A minimal expression language

```
data Expr : Set where
  lit  : ℕ → Expr
  plus : Expr → Expr → Expr

eval : Expr → ℕ
eval (lit n)        = n
eval (plus e₁ e₂) = eval e₁ + eval e₂
```

Take 1 – No Guarantees

```
data Prog : Set where
  PUSH : ℕ → Prog
  ADD  : Prog
  _,_  : Prog → Prog → Prog

compile : Expr → Prog
compile (lit n)       = PUSH n
compile (plus e₁ e₂) = compile e₂ , compile e₁ , ADD

Stack = List ℕ

exec : Prog → Stack → Stack
exec (PUSH n) S             = n :: S
exec ADD      (a :: b :: S) = a + b :: S
exec ADD      _             = []   -- not nice!
exec (p , q)  S             = exec q (exec p S)
```

Take 2 – Stack safety

```
data Prog : ℕ → ℕ → Set where
  PUSH : ∀ {n} → ℕ → Prog n (1 + n)
  ADD  : ∀ {n} → Prog (2 + n) (1 + n)
  _,_  : ∀ {n m l} → Prog n m → Prog m l → Prog n l

compile : ∀ {n} → Expr → Prog n (1 + n)
compile (lit n)        = PUSH n
compile (plus e₁ e₂) = compile e₂ , compile e₁ , ADD

Stack : ℕ → Set
Stack n = Vec ℕ n

exec : ∀ {n m} → Prog n m → Stack n → Stack m
exec (PUSH n) S            = n ∷ S
exec ADD      (a ∷ b ∷ S) = a + b ∷ S
exec (p , q)  S            = exec q (exec p S)
```

TAKE 3 – CORRECT BY CONSTRUCTION

```
Sem : ℕ → ℕ → Set
Sem n m = Stack n → Stack m

push : ∀ {n} → ℕ → Sem n (1 + n)
push a S = a ∷ S

add : ∀ {n} → Sem (2 + n) (1 + n)
add (a ∷ b ∷ S) = a + b ∷ S

data Prog : ∀ {n m} → Sem n m → Set where
  PUSH : ∀ {n}(a : ℕ) → Prog {n}       (push a)
  ADD  : ∀ {n}        → Prog {2 + n} add
  _,_  : ∀ {n m l}{φ : Sem n m}{ψ : Sem m l} →
         Prog φ → Prog ψ → Prog (ψ ∘ φ)
```

Take 3 – Correct by construction

```
data Prog : ∀ {n m} → Sem n m → Set where
  PUSH : ∀ {n}(a : ℕ) → Prog {n}      (push a)
  ADD  : ∀ {n}        → Prog {2 + n} add
  _,_  : ∀ {n m l}{φ : Sem n m}{ψ : Sem m l} →
         Prog φ → Prog ψ → Prog (ψ ∘ φ)

compile : ∀ {n}(e : Expr) → Prog {n} (λ s → eval e ∷ s)
compile (lit n)       = PUSH n
compile (plus e₁ e₂) = compile e₂ , compile e₁ , ADD
```

Conclusions

- Dependently Typed Programming
  - Write programs that don't need any proofs
  - Using *views* capturing the relation between inputs and output
  - Encode program invariants in the types
- To make this work:
  - Inductive families
  - Pattern matching