Programming with Universes, Generically

Andres Löh

Well-Typed LLP

24 January 2012

Well-Typed

An introduction to Agda

Agda

- Functional programming language
- Static types
- Dependent types
- Pure (explicit effects)
- Total (mostly)

Agda

- Functional programming language
- Static types
- Dependent types
- Pure (explicit effects)
- Total (mostly)

- Actively developed at Chalmers University
- Ulf Norell and many others
- Written in Haskell

Simple Agda

Superficially looks a bit like Haskell:

$$
\begin{array}{l}
\textbf{data}\ \mathbb{N}\ :\ \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{zero}\ :\ \mathbb{N} \\
\quad \mathsf{suc}\ :\ \mathbb{N} \to \mathbb{N} \\
\_+\_\ :\ \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
\mathsf{zero} + \mathsf{n} \quad = \ \mathsf{n} \\
\mathsf{suc}\ \mathsf{m} + \mathsf{n}\ = \ \mathsf{suc}\ (\mathsf{m} + \mathsf{n})
\end{array}
$$

Simple Agda

Superficially looks a bit like Haskell:

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

map : {A B : Set} → (A → B) → List A → List B
map f [ ]       = [ ]
map f (x :: xs) = f x :: map f xs
```

Dependent types

Types can depend on terms:

```
data Vec (A : Set) : ℕ → Set where
  []  : Vec A zero
  _::_ : {n : ℕ} → A → Vec A n → Vec A (suc n)

_++_ : {A : Set} {m n : ℕ} →
       Vec A m → Vec A n → Vec A (m + n)
[]        ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

Dependent types

Types can depend on terms:

```
data Vec (A : Set) : ℕ → Set where
  [ ]   : Vec A zero
  _::_  : {n : ℕ} → A → Vec A n → Vec A (suc n)

_++_ : {A : Set} {m n : ℕ} →
         Vec A m → Vec A n → Vec A (m + n)
[ ]       ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

- Computation during type-checking
- When are two types equal?

Type equality

Are these equal?

$\mathsf{Vec}\ \mathbb{N}\ (2 + 2)$
$\mathsf{Vec}\ \mathbb{N}\ 4$

Type equality

Are these equal?

$\text{Vec}\ \mathbb{N}\ (2 + 2)$
$\text{Vec}\ \mathbb{N}\ 4$

And these?

$\text{Vec}\ \mathbb{N}\ (n + 1)$
$\text{Vec}\ \mathbb{N}\ (1 + n)$

Type equality

Are these equal?

$\mathsf{Vec}\ \mathbb{N}\ (2 + 2)$
$\mathsf{Vec}\ \mathbb{N}\ 4$

And these?

$\mathsf{Vec}\ \mathbb{N}\ (n + 1)$
$\mathsf{Vec}\ \mathbb{N}\ (1 + n)$

Simple rule: types are reduced according to their definitions as far as possible and then checked for (alpha-)equality.

Type equality

Are these equal?

$$\mathsf{Vec}\ \mathbb{N}\ (2 + 2)$$
$$\mathsf{Vec}\ \mathbb{N}\ 4$$

And these?

$$\mathsf{Vec}\ \mathbb{N}\ (n + 1)$$
$$\mathsf{Vec}\ \mathbb{N}\ (1 + n)$$

Simple rule: types are reduced according to their definitions as far as possible and then checked for (alpha-)equality.

- $2 + 2$ reduces to 4, so the first two are equal.

Type equality

Are these equal?

$\mathsf{Vec}\ \mathbb{N}\ (2 + 2)$
$\mathsf{Vec}\ \mathbb{N}\ 4$

And these?

$\mathsf{Vec}\ \mathbb{N}\ (n + 1)$
$\mathsf{Vec}\ \mathbb{N}\ (1 + n)$

Simple rule: types are reduced according to their definitions as far as possible and then checked for (alpha-)equality.

- $2 + 2$ reduces to 4, so the first two are equal.
- $n + 1$ is stuck, because $\_+\_$ is defined by induction on the first argument. The second two are not equal.

Programs and Proofs

Totality

Function in Agda are supposed to be **total**:

- defined on all inputs,
- terminating.

Totality

Function in Agda are supposed to be **total**:

- defined on all inputs,
- terminating.

So, despite computation on the type level, type checking is decidable.

Enforcing totality

Relatively simple (but conservative) checks:

- Case distinctions have to be exhaustive.
- Recursion only on structurally smaller terms.
- Datatypes must be strictly positive.

Enforcing totality

Relatively simple (but conservative) checks:

- Case distinctions have to be exhaustive.
- Recursion only on structurally smaller terms.
- Datatypes must be strictly positive.

Consequence: Agda has **uninhabited** types!

```
data ⊥ : Set where
```

No constructors, thus no way to construct values of type `⊥`.

Whereas in Haskell:

```
loop :: forall a. a
loop x = x
```

Curry-Howard isomorphism

Agda becomes interesting as a **logic**.

property	type
proof	program

Curry-Howard isomorphism

Agda becomes interesting as a **logic**.

property	type
proof	program
truth	inhabited type
falsity	uninhabited type
conjunction	pair
disjunction	union type
implication	function
negation	function to the uninhabited type

Curry-Howard isomorphism

Agda becomes interesting as a **logic**.

property proof	type program
truth	inhabited type
falsity	uninhabited type
conjunction	pair
disjunction	union type
implication	function
negation	function to the uninhabited type
universal quantification	dependent function
existential quantification	dependent pair

A type representing equality

```
data _≡_ {A : Set} : A → A → Set where
  refl : {x : A} → x ≡ x
```

The value `refl` is a **witness** that two terms of type `A` are actually equal.

A type representing equality

```
data _≡_ {A : Set} : A → A → Set where
  refl : {x : A} → x ≡ x
```

The value `refl` is a **witness** that two terms of type `A` are actually equal.

Where Agda's built-in (definitional) equality isn't enough, we can explicitly prove (and use) equality using `_≡_`.

Programming is proving

Equality is a congruence:

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

Programming is proving

Equality is a congruence:

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

Zero is a right-unit of addition:

```
example : (n : ℕ) → (n + zero) ≡ n
example zero      = refl
example (suc n)   = cong suc (example n)
```

Programming is proving

Equality is a congruence:

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

Zero is a right-unit of addition:

```
example : (n : ℕ) → (n + zero) ≡ n
example zero    = refl
example (suc n) = cong suc (example n)
```

Proofs are easier to do **interactively**.

Dependently typed programming

- Programming with data that maintains complex invariants, verified by the type checker.
- Stating and proving properties about programs within the program, using the same language.
- Using precise types to guide the programming process.

Datatype-genericity

Reuse vs. type safety

Types make things **different**.

They sometimes seem to stand in the way of **code reuse**.

Reuse vs. type safety

Types make things **different**.

They sometimes seem to stand in the way of **code reuse**.

A lot of this tension is already addressed by **polymorphism**, which now corresponds to **universal quantification**.

But what if we want different (but related) behaviour for different types?

Datatype-generic programs

Datatype-generic programs allow you to **inspect the structure of datatypes** while defining a function.

Datatype-generic programs

Datatype-generic programs allow you to **inspect the structure of datatypes** while defining a function.

Classic examples:

- structural equality, structural ordering
- serialization, deserialization
- parsing, pretty-printing
- mapping, traversing, transforming, querying

Historical context

- Active research topic since about 15 years.
- A lot of promising approaches, many based on Haskell.
- Related to OO design patterns such as Visitor and Iterator, but also to techniques such as model-driven design.
- Related to meta-programming, but with the goal to be type-safe.
- Historically required significant language extensions or a preprocessor.
- Advances in FP type systems have made it possible to develop datatype-generic programs (nearly) directly in Haskell.
- Dependent types are even more powerful than current Haskell, so DGP in Agda should be easy …

Universes

Universe

A **universe** is a type of codes together with an interpretation function that computes types from codes:

$$
\begin{aligned}
\mathsf{Code} &: \mathsf{Set} \\
[\![\_]\!] &: \mathsf{Code} \to \mathsf{Set}
\end{aligned}
$$

We cannot inspect types directly.

But we can inspect codes!

Universe

A **universe** is a type of codes together with an interpretation function that computes types from codes:

$$\begin{aligned}
&\mathsf{Code} &&: \mathsf{Set} \\
&[\![_]\!] &&: \mathsf{Code} \to \mathsf{Set}
\end{aligned}$$

We cannot inspect types directly.

But we can inspect codes!

A **(datatype-)generic function** is a function defined by induction on the codes:

$$\mathsf{gf} : (\mathsf{C} : \mathsf{Code}) \to \ldots [\![\, \mathsf{C} \,]\!] \ldots$$

A simple example

We have **already** seen a universe.

A simple example

We have **already** seen a universe.

```
data Vec (A : Set) : ℕ → Set where
  [ ]   : Vec A zero
  _::_  : {n : ℕ} → A → Vec A n → Vec A (suc n)
```

Here, ℕ are the codes, and Vec is an A -indexed family of interpretation functions.

A simple example

We have **already** seen a universe.

```
data Vec (A : Set) : ℕ → Set where
  [ ]   : Vec A zero
  _::_  : {n : ℕ} → A → Vec A n → Vec A (suc n)
```

Here, ℕ are the codes, and Vec is an A -indexed family of interpretation functions.

Thus _++_ is a generic function on this universe.

A different way to define the vector universe

Again, we use $\mathbb{N}$ as type of codes.

```
Vec : (A : Set) → ℕ → Set
Vec A zero      = ⊤
Vec A (suc n)   = A × Vec A n
```

Another interpetation for natural numbers

With Fin, we define the family of finite types:

$$
\begin{aligned}
&\mathsf{Fin} : \mathbb{N} \to \mathsf{Set} \\
&\mathsf{Fin}\ \mathsf{zero} \quad\ \ = \bot \\
&\mathsf{Fin}\ (\mathsf{suc}\ n) = \top \uplus \mathsf{Fin}\ n
\end{aligned}
$$

Another interpetation for natural numbers

With Fin, we define the family of finite types:

```
Fin : ℕ → Set
Fin zero     = ⊥
Fin (suc n)  = ⊤ ⊎ Fin n
```

Safe lookup:

```
lookup : {A : Set} → (n : ℕ) → Fin n → Vec A n → A
lookup zero    ()        _
lookup (suc n) (inj₁ tt) (x , xs) = x
lookup (suc n) (inj₂ i ) (x , xs) = lookup n i xs
```

Another definition for finite types

Finite types are closed under union and cartesian product:

```
data Code : Set where
  c0    : Code
  c1    : Code
  _⊕_  : Code → Code → Code
  _⊗_  : Code → Code → Code
```

```
⟦_⟧ : Code → Set
⟦ c0     ⟧ = ⊥
⟦ c1     ⟧ = ⊤
⟦ C ⊕ D ⟧ = ⟦ C ⟧ ⊎ ⟦ D ⟧
⟦ C ⊗ D ⟧ = ⟦ C ⟧ × ⟦ D ⟧
```

Generic equality on finite types

```
_==_ : (C : Code) → ⟦ C ⟧ → ⟦ C ⟧ → Bool
_==_ c0      () ()
_==_ c1      tt tt = true
_==_ (C ⊕ D) (inj₁ x₁) (inj₁ x₂) = _==_ C x₁ x₂
_==_ (C ⊕ D) (inj₂ y₁) (inj₂ y₂) = _==_ D y₁ y₂
_==_ (C ⊕ D) _         _         = false
_==_ (C ⊗ D) (x₁ , y₁) (x₂ , y₂) = _==_ C x₁ x₂ ∧ _==_ D y₁ y₂
```

Relations between universes

$$
\begin{aligned}
&\mathsf{size} : \mathsf{Code} \to \mathbb{N} \\
&\mathsf{size}\ \mathsf{c0} &&= 0 \\
&\mathsf{size}\ \mathsf{c1} &&= 1 \\
&\mathsf{size}\ (\mathsf{C} \oplus \mathsf{D}) &&= \mathsf{size}\ \mathsf{C} + \mathsf{size}\ \mathsf{D} \\
&\mathsf{size}\ (\mathsf{C} \otimes \mathsf{D}) &&= \mathsf{size}\ \mathsf{C} * \mathsf{size}\ \mathsf{D}
\end{aligned}
$$

We can prove (in Agda) that the two definitions of finite types are related:

$$
\begin{aligned}
&\mathsf{fromFin} &&: (\mathsf{n} : \mathbb{N}) \to \mathsf{Fin}\ \mathsf{n} \to [\![\ \mathsf{natCode}\ \mathsf{n}\ ]\!] \\
&\mathsf{toFin} &&: (\mathsf{C} : \mathsf{Code}) \to [\![\ \mathsf{C}\ ]\!] \to \mathsf{Fin}\ (\mathsf{size}\ \mathsf{C}) \\
&\mathsf{toFromId} &&: \{\mathsf{n} : \mathbb{N}\}\ (\mathsf{i} : \mathsf{Fin}\ \mathsf{n}) \to \\
& && \quad \mathsf{i} \equiv \mathsf{toFin}\ (\mathsf{natCode}\ \mathsf{n})\ (\mathsf{fromFin}\ \mathsf{i})
\end{aligned}
$$

Adding recursion

A universe for polynomial functors:

```
data Code : Set where
  c0    : Code
  c1    : Code
  _⊕_   : Code → Code → Code
  _⊗_   : Code → Code → Code
  rec   : Code
```

We interpret codes as **type constructors** now:

$$
\begin{array}{lcl}
[\![\_]\!] & : & \mathsf{Code} \to \mathsf{Set} \to \mathsf{Set} \\
[\![\ c0\ ]\!]\ X & = & \bot \\
[\![\ c1\ ]\!]\ X & = & \top \\
[\![\ C \oplus D\ ]\!]\ X & = & [\![\ C\ ]\!]\ X \uplus [\![\ D\ ]\!]\ X \\
[\![\ C \otimes D\ ]\!]\ X & = & [\![\ C\ ]\!]\ X \times [\![\ D\ ]\!]\ X \\
[\![\ rec\ ]\!]\ X & = & X
\end{array}
$$

Mapping over functors

We traverse the structure, only modifying parameter positions:

$$
\begin{array}{llll}
\mathsf{map} & : \{X\ Y : \mathsf{Set}\} \to (C : \mathsf{Code}) \to & & \\
 & \quad (X \to Y) \to [\![\, C \,]\!]\ X \to [\![\, C \,]\!]\ Y & & \\
\mathsf{map}\ \mathsf{c0} & \mathsf{f}\ () & & \\
\mathsf{map}\ \mathsf{c1} & \mathsf{f}\ \mathsf{tt} & = & \mathsf{tt} \\
\mathsf{map}\ (C \oplus D) & \mathsf{f}\ (\mathsf{inj}_1\ x) & = & \mathsf{inj}_1\ (\mathsf{map}\ C\ \mathsf{f}\ x) \\
\mathsf{map}\ (C \oplus D) & \mathsf{f}\ (\mathsf{inj}_2\ y) & = & \mathsf{inj}_2\ (\mathsf{map}\ D\ \mathsf{f}\ y) \\
\mathsf{map}\ (C \otimes D) & \mathsf{f}\ (x\ ,\ y) & = & \mathsf{map}\ C\ \mathsf{f}\ x\ ,\ \mathsf{map}\ D\ \mathsf{f}\ y \\
\mathsf{map}\ \mathsf{rec} & \mathsf{f}\ x & = & \mathsf{f}\ x
\end{array}
$$

Taking fixed points

We plug in the data structure itself for the parameter position:

```
data μ (C : Code) : Set where
  ⟨_⟩ : ⟦ C ⟧ (μ C) → μ C
```

Taking fixed points

We plug in the data structure itself for the parameter position:

```
data μ (C : Code) : Set where
  ⟨_⟩ : ⟦ C ⟧ (μ C) → μ C
```

Example: Binary trees.

```
BinTree : Set
BinTree = μ (c1 ⊕ (rec ⊗ rec))

leaf : BinTree
leaf = ⟨ inj₁ tt ⟩

true : BinTree → BinTree → BinTree
true l r = ⟨ inj₂ (l , r) ⟩
```

Generically traversing a recursive structure

$$\mathsf{cata} : \{\mathsf{C} : \mathsf{Code}\}\,\{\mathsf{X} : \mathsf{Set}\} \to (\llbracket\, \mathsf{C} \,\rrbracket\, \mathsf{X} \to \mathsf{X}) \to \mu\, \mathsf{C} \to \mathsf{X}$$
$$\mathsf{cata}\,\{\mathsf{C}\}\,\phi\,\langle\, \mathsf{x}\,\rangle = \phi\,(\mathsf{map}\;\mathsf{C}\;(\mathsf{cata}\;\phi)\;\mathsf{x})$$

$$\mathsf{height} : \mathsf{BinTree} \to \mathbb{N}$$
$$\mathsf{height} = \mathsf{cata}\,[\,\mathsf{const}\;0\,,\,\lambda\,(\mathsf{x}\,,\,\mathsf{y}) \to 1 + \mathsf{max}\;\mathsf{x}\;\mathsf{y}\,]$$

What's next?

Many approaches that have been tried in Haskell over the years are similar to the one we have just seen:

- **regular** library
- **PolyP** adds a parameter slot (so we can model lists, labelled trees, etc.)
- **multirec** library adds an index to the `rec` constructor, so that we can define fixed points of mutually recursive types

What's next?

Many approaches that have been tried in Haskell over the years are similar to the one we have just seen:

- **regular** library
- **PolyP** adds a parameter slot (so we can model lists, labelled trees, etc.)
- **multirec** library adds an index to the `rec` constructor, so that we can define fixed points of mutually recursive types

Agda helps us:

- to relate and understand all these approaches,
- to generalize even further,
- to prove properties of the resulting generic functions.

How many universes do we need?

Without dependent types (Haskell):

- Defining one universe is ok, but mapping between universes is infeasible.
- We have to decide which representation we want to use.
- But the choice is difficult.
- Simple universes represent less types but allow more functions to be defined.

With Agda, we do not have to decide:

- We can define functions generically over a “suitable” universe.
- We can change representations as needed.
- Ideally, we’d model the complete **data** construct as a universe (**levitation**).

Programming generically within Agda

Dependent types encourage us to make more distinctions than we are used to make:

- lists,
- vectors,
- sorted lists,
- lists with an even number of elements,
- lists containing only even numbers.

All of these become different **types**, yet we still want to perform similar **operations**.

More generic tools

We need more generic tools and special-purpose universes. Examples:

- Many list-like structures can be represented as **reflexive transitive closures** of suitable **binary relations**.
- We can relate unconstrained data structures such as lists to constrained data structures such as vectors by a generic process called **algebraic ornamentation**.

Conclusions

- Datatype-generic programming allows code to be **reused** more often.
- Generic functions are very abstract, but the **types help** you to write them.
- The stronger the type system, the more **important** (but also the **easier**) generic programming becomes.
- With dependent types, generic programming is just (ordinary) **programming**.
- Developing dependently typed generic programs is **fun**.

Thanks for listening – Questions?