Agda II – Take One

Ulf Norell

May 10, 2006
What’s the point?

- of Agda II
  - Solid theoretical foundation (lacking in Agda)
    - Small well-defined core language with nice metatheory.
    - Transparent translation from the full language to the core language.

- of this talk
  - Present the (full) language from a user’s perspective.
The Logical Framework

The Basic Language

(Terms)  \( s, t \) ::= x | c | f | s t | \lambda x \to t | \lambda (x : A) \to t

(Types)  \( A, B \) ::= (x : A) \to B | A \to B | t | \alpha

(Sorts)  \( \alpha, \beta \) ::= Set_i | Set | Prop

Note: Set \( \neq \) Prop.

Example: polymorphic identity

\( id : (A : \text{Set}) \to A \to A \)

\( id = \lambda (A : \text{Set}) (x : A) \to x \)
What’s there and what’s not

- **Features**
  - Inductive datatypes
  - Functions by pattern matching
  - Implicit arguments
  - Module system

- **Not Yet Features**
  - $\Pi$ in Set
  - Signatures and structures
  - Inductive families
What does it mean?

We don’t have

\[
\Gamma \vdash A : \text{Set} \quad \Gamma, x : A \vdash B : \text{Set} \\
\Gamma \vdash (x : A) \rightarrow B : \text{Set}
\]

Consequences:

We can’t do

\[
\text{Rel } A = A \rightarrow A \rightarrow \text{Prop} \\
\text{apply} : \text{List } (\text{Nat} \rightarrow \text{Nat}) \rightarrow \text{List Nat} \rightarrow \text{List Nat}
\]
Why don’t we have it?
- Ask Thierry... (The metatheory gets tricky when you combine $\eta$-equality and $\Pi$ in $\text{Set}$.)

What to do about it:
- Get the metatheory straightened out (e.g. $\eta$-equality for datatypes).
- Abandon $\eta$-equality.
- Abandon $\Pi$ in $\text{Set}$. 
Signatures and Structures

- What does it mean?
  - In Agda you can say (something like)

\[
\text{Pair } A \ B = \text{sig } \begin{align*}
\text{fst} & : A \\
\text{snd} & : B
\end{align*}
\]

\[
p : \text{Pair } \text{Nat} \ \text{Nat}
\]

\[
p = \text{struct } \begin{align*}
\text{fst} & = 3 \\
\text{snd} & = 7
\end{align*}
\]

\[
\text{three} = p.\text{fst}
\]

- Why don’t we have it?
  - We want to start simple.
  - Signatures and structures will appear in Agda II – Take Two (but probably not in the same form as in Agda).
Inductive Families

What does it mean?
- For instance:

```
data Vec (A : Set) : Nat → Set where
  vnil    : Vec A zero
  vcons   : (n : Nat) → A → Vec A n → Vec A (suc n)
```

Why don’t we have it?
- The inductive families in Agda are very limited in terms of what you can do with them.
- We want something better, which will require some thinking.
Datatypes

- Standard, garden-variety, strictly positive datatypes:

\[
\begin{align*}
\text{data } \text{Nat} &: \text{Set} \quad \text{where} \\
\text{zero} &: \text{Nat} \\
\text{suc} &: \text{Nat} \to \text{Nat}
\end{align*}
\]

\[
\begin{align*}
\text{data } \text{Exist} \ (A : \text{Set}) \ (P : A \to \text{Prop}) &: \text{Prop} \quad \text{where} \\
\text{witness} &: \ (x : A) \to P \ x \to \text{Exist} \ A \ P
\end{align*}
\]

\[
\begin{align*}
\text{data } \text{Acc} \ (A : \text{Set}) \ (\langle \rangle : A \to A \to \text{Prop}) \ (x : A) &: \text{Prop} \quad \text{where} \\
\text{acc} &: \ (y : A) \to y \ \langle \rangle x \to \text{Acc} \ A \ \langle \rangle y \to \text{Acc} \ A \ \langle \rangle x
\end{align*}
\]

- Note that \text{data} ... is a declaration (not a term or type).
Definitions by Pattern Matching

- Functions are defined by pattern matching
  - Arbitrarily nested, exhaustive, possibly overlapping patterns.
  - No case expressions!

\[
\begin{align*}
(+) & : \text{Nat} \to \text{Nat} \to \text{Nat} \\
\text{zero} + m &= m \\
\text{suc } n + m &= \text{suc } (n + m)
\end{align*}
\]

\[
\begin{align*}
\text{eqNat} & : \text{Nat} \to \text{Nat} \to \text{Bool} \\
\text{eqNat } \text{zero } \text{zero} &= \text{true} \\
\text{eqNat } (\text{suc } n) (\text{suc } m) &= \text{eqNat } n m \\
\text{eqNat } _ _ &= \text{false}
\end{align*}
\]
You can have mutually inductive-recursive definitions:

```plaintext
mutual
  even : Nat → Bool
  even zero    = true
  even (suc n) = odd n

  odd : Nat → Bool
  odd zero     = false
  odd (suc n)  = even n
```

I’d show the standard universe construction example of induction-recursion, but you need $\Pi$ in $\text{Set}$ for that.
Local functions

- Functions (and datatypes) can be local to a definition:

```agda
reverse : (A : Set) → List A → List A
reverse A xs = rev xs nil

where
  rev : List A → List A → List A
  rev nil ys = ys
  rev (x :: xs) ys = rev xs (x :: ys)
```

We allow general recursion.
Termination checking is done separately (as in Agda).

Example:

\[
\text{qsort} : \text{List Nat} \rightarrow \text{List Nat} \\
\text{qsort} \text{ nil} = \text{nil} \\
\text{qsort} \text{ (x :: xs)} = \text{filter} (\lambda y \rightarrow y < x) \text{ xs} ++ \\
\text{ x :: filter} (\lambda y \rightarrow y \geq x) \text{ xs}
\]
There are two kinds of meta variables (only one in Agda):

- Interaction points: ? and {! ... !}
- Go figure\(^1\): _

The type checker should be able to figure out the value of a go figure without user intervention...

...whereas the value of an interaction point is supplied by the user.

We use go figures to implement implicit arguments.

\(^1\)Conorism
Implicit Arguments

- Curly braces \{\} are used to indicate implicitness:

Syntax

\[
\begin{align*}
  s, t & \ ::= \ldots \mid s \{t\} \mid \lambda x \to t \mid \lambda x : A \to t \mid _- \\
  A, B & \ ::= \ldots \mid \{x : A\} \to B \mid \{A\} \to B
\end{align*}
\]

\[
\begin{align*}
  id : \{A : Set\} \to A \to A \\
  id \{A\} x & = x \\
  zero' & = id \{Nat\} zero
\end{align*}
\]

- Implicit arguments can be omitted: \(id x\) means \(id \{\_\} x\).
- Both in left-hand-sides and right-hand-sides:

\[
\begin{align*}
  id : \{A : Set\} \to A \to A \\
  id x & = x
\end{align*}
\]
**Example**

```agda
data List (A : Set) : Set where
  nil   : List A
  (_∷_) : A → List A → List A

(++) : {A : Set} → List A → List A → List A
nil  ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

- Note that constructors are polymorphic:
  - ⊢ nil : List A, for any A
  - ⊬ nil : {A : Set} → List A.
Module System

- **Purpose:**
  - Control the scope of names.
  - (Not to model algebraic structures.)
- **Guiding principle:**
  - Scope checking should not require type checking or computation.
- **Consequence:**
  - Modules are not first class.
Submodules

Each source file contains a single module, which in turn can contain any number of submodules:

```
module Prelude where
  module Nat where
    ...
  module List where
    ...
    module Fold where
      ...
      ...
```

Ulf Norell  Agda II – Take One
To use a module from a file the module has to be *imported*

```agda
import Prelude
```

We can then use the names in the module fully qualified

```agda
one = Prelude.Nat.suc Prelude.Nat.zero
```

Or we can *open* a module

```agda
open Prelude.Nat
one = suc zero
```
We can exercise finer control over what is imported or opened.

```agda
import Prelude as P
open P.Nat, hiding (+), renaming (zero to z)
open P.List, using (replicate)
zz : P.List.List Nat
zz = replicate (suc (suc z)) z
```
Controlling what is exported

- Private things are not exported.

```agda
module BigProof where
  private minorLemma = ...
  mainTheorem : P == NP
  mainTheorem = ... minorLemma ...
```

- Abstract things export only their type.

```agda
module Stack where
  abstract
    Stack : Set → Set
    Stack = List
```

- Private things still reduce, abstract things don’t.
Parameterised Modules

- Modules can be parameterised.

```agda
module Monad (M : Set → Set)
  (return : {A : Set} → A → M A)
  ((>>=) : {A, B : Set} → M A → (A → M B) → M B)
where
  liftM : {A, B : Set} → (A → B) → M A → M B
  liftM f m = m >>= \x → return (f x)
```

- And instantiated

```agda
module MonadList = Monad List singleton (flip concatMap)
lemma : {A, B : Set} → (f : A → B) → (xs : List A) →
  map f xs == MonadList.liftM f xs
```

- You need to instantiate a parameterised module to use it.
Agda II is very much work in progress.

At this point very little is set in stone, so if you think things should be a different way now is the time to speak up.

Most of what you’ve seen will be available for use during the 4th Agda Implementors Meeting starting next week in Japan.