Agda II – Take One

Ulf Norell

May 10, 2006

1. Introduction
   - Motivation
   - The Basics
   - Features and Not
2. Not Yet Features
   - Pi in Set
   - Signatures and Structures
   - Inductive Families
3. Features
   - Datatypes
   - Definitions by Pattern Matching
   - Implicit Arguments
   - Module System
4. Conclusions

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

$$
\begin{array}{llll}
\text{(Terms)} & s, t & ::= & x \mid c \mid f \mid s\,t \mid \lambda x \to t \mid \lambda(x : A) \to t \\
\text{(Types)} & A, B & ::= & (x : A) \to B \mid A \to B \mid t \mid \alpha \\
\text{(Sorts)} & \alpha, \beta & ::= & Set_i \mid Set \mid Prop
\end{array}
$$

- Note: $Set \neq Prop$.

Example: polymorphic identity

$$id : (A : Set) \to A \to A$$
$$id = \lambda(A : Set)(x : A) \to x$$

What’s there and what’s not

- Features
  - Inductive datatypes
  - Functions by pattern matching
  - Implicit arguments
  - Module system
- Not Yet Features
  - Π in Set
  - Signatures and structures
  - Inductive families

Π in Set

- What does it mean?

We don't have

$$\frac{\Gamma \vdash A : Set \quad \Gamma, x : A \vdash B : Set}{\Gamma \vdash (x : A) \to B : Set}$$

- Consequences:

We can't do

$$Rel\ A = A \to A \to Prop$$
$$apply : List\ (Nat \to Nat) \to List\ Nat \to List\ Nat$$

Π in Set

- Why don't we have it?
  - Ask Thierry... (The metatheory gets tricky when you combine η-equality and Π in *Set*.)
- What to do about it:
  - Get the metatheory straightened out (e.g. η-equality for datatypes).
  - Abandon η-equality.
  - Abandon Π in *Set*.

Signatures and Structures

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

$$
\begin{array}{l}
\mathit{Pair}\ A\ B = \textbf{sig}\ \begin{array}{lcl} \mathit{fst} & : & A \\ \mathit{snd} & : & B \end{array} \\
p : \mathit{Pair}\ \mathit{Nat}\ \mathit{Nat} \\
p = \textbf{struct}\ \begin{array}{lcl} \mathit{fst} & = & 3 \\ \mathit{snd} & = & 7 \end{array} \\
\mathit{three} = p.\mathit{fst}
\end{array}
$$

- 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:

$$
\begin{array}{lll}
\textbf{data}\ \mathit{Vec}\ (A : \mathit{Set}) : \mathit{Nat} \to \mathit{Set}\ \textbf{where} & & \\
\quad \mathit{vnil} & : & \mathit{Vec}\ A\ \mathit{zero} \\
\quad \mathit{vcons} & : & (n : \mathit{Nat}) \to A \to \mathit{Vec}\ A\ n \to \mathit{Vec}\ A\ (\mathit{suc}\ n)
\end{array}
$$

- 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:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Nat} : \mathit{Set}\ \textbf{where} \\
\quad \begin{array}{lll} \mathit{zero} & : & \mathit{Nat} \\ \mathit{suc} & : & \mathit{Nat} \to \mathit{Nat} \end{array} \\
\\
\textbf{data}\ \mathit{Exist}\ (A : \mathit{Set})\ (P : A \to \mathit{Prop}) : \mathit{Prop}\ \textbf{where} \\
\quad \begin{array}{lll} \mathit{witness} & : & (x : A) \to P\ x \to \mathit{Exist}\ A\ P \end{array} \\
\\
\textbf{data}\ \mathit{Acc}\ (A : \mathit{Set})\ ((<) : A \to A \to \mathit{Prop})\ (x : A) : \mathit{Prop}\ \textbf{where} \\
\quad \begin{array}{lll} \mathit{acc} & : & ((y : A) \to y < x \to \mathit{Acc}\ A\ (<)\ y) \to \mathit{Acc}\ A\ (<)\ x \end{array}
\end{array}
$$

- Note that **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!

$$
\begin{array}{lllll}
\multicolumn{5}{l}{(+) : \mathit{Nat} \rightarrow \mathit{Nat} \rightarrow \mathit{Nat}} \\
\mathit{zero} & + & m & = & m \\
\mathit{suc}\ n & + & m & = & \mathit{suc}\ (n + m)
\end{array}
$$

$$
\begin{array}{lllll}
\mathit{eqNat} : & \mathit{Nat} \rightarrow & \mathit{Nat} \rightarrow & & \mathit{Bool} \\
\mathit{eqNat} & \mathit{zero} & \mathit{zero} & = & \mathit{true} \\
\mathit{eqNat} & (\mathit{suc}\ n) & (\mathit{suc}\ m) & = & \mathit{eqNat}\ n\ m \\
\mathit{eqNat} & \_ & \_ & = & \mathit{false}
\end{array}
$$

Mutual induction-recursion

- You can have mutually inductive-recursive definitions:

mutual

$$
\begin{array}{llcl}
\mathit{even} : & \mathit{Nat} \to & & \mathit{Bool} \\
\mathit{even} & \mathit{zero} & = & \mathit{true} \\
\mathit{even} & (\mathit{suc}\ n) & = & \mathit{odd}\ n \\
\\
\mathit{odd} : & \mathit{Nat} \to & & \mathit{Bool} \\
\mathit{odd} & \mathit{zero} & = & \mathit{false} \\
\mathit{odd} & (\mathit{suc}\ n) & = & \mathit{even}\ n
\end{array}
$$

- I'd show the standard universe construction example of induction-recursion, but you need Π in *Set* for that.

Local functions

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

$$
\begin{array}{l}
\mathit{reverse} : (A : \mathit{Set}) \to \mathit{List}\ A \to \mathit{List}\ A \\
\mathit{reverse}\ A\ \mathit{xs} = \mathit{rev}\ \mathit{xs}\ \mathit{nil} \\
\quad \textbf{where} \\
\qquad \begin{array}{lllll}
\mathit{rev} : & \mathit{List}\ A \to & \mathit{List}\ A \to & & \mathit{List}\ A \\
\mathit{rev} & \mathit{nil} & \mathit{ys} & = & \mathit{ys} \\
\mathit{rev} & (x :: \mathit{xs}) & \mathit{ys} & = & \mathit{rev}\ \mathit{xs}\ (x :: \mathit{ys})
\end{array}
\end{array}
$$

Termination

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

$$
\begin{array}{llcl}
\mathit{qsort} : & \mathit{List\ Nat} \to & & \mathit{List\ Nat} \\
\mathit{qsort} & \mathit{nil} & = & \mathit{nil} \\
\mathit{qsort} & (x :: \mathit{xs}) & = & \mathit{filter}\ (\lambda y \to y < x)\ \mathit{xs} \mathbin{++} \\
 & & & x :: \mathit{filter}\ (\lambda y \to y \geq x)\ \mathit{xs}
\end{array}
$$

Meta Variables

- 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

$$\begin{array}{lll} s,t & ::= & \ldots \mid s\ \{t\} \mid \lambda\{x\} \to t \mid \lambda\{x : A\} \to t \mid \_ \\ A,B & ::= & \ldots \mid \{x : A\} \to B \mid \{A\} \to B \end{array}$$

$$\begin{array}{l} id\ :\ \{A : Set\} \to A \to A \\ id\ \{A\}\ x = x \\ zero' = id\ \{Nat\}\ zero \end{array}$$

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

$$\begin{array}{l} id\ :\ \{A : Set\} \to A \to A \\ id\ x = x \end{array}$$

Example

$$
\begin{array}{l}
\textbf{data}\ \mathit{List}\ (A : \mathit{Set}) : \mathit{Set}\ \textbf{where} \\
\quad \begin{array}{lcl}
\mathit{nil} & : & \mathit{List}\ A \\
(::) & : & A \to \mathit{List}\ A \to \mathit{List}\ A
\end{array}
\end{array}
$$

$$
\begin{array}{l}
(++) : \{A : \mathit{Set}\} \to \mathit{List}\ A \to \mathit{List}\ A \to \mathit{List}\ A \\
\begin{array}{llcll}
\mathit{nil} & ++ & \mathit{ys} & = & \mathit{ys} \\
(x :: \mathit{xs}) & ++ & \mathit{ys} & = & x :: (\mathit{xs} ++ \mathit{ys})
\end{array}
\end{array}
$$

- Note that constructors are polymorphic:
  - $\vdash \mathit{nil} : \mathit{List}\ A$, for any A
  - $\nvdash \mathit{nil} : \{A : \mathit{Set}\} \to \mathit{List}\ A$.

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
      ...
    ...
```

Accessing the Module Contents

- To use a module from a file the module has to be *imported*

import *Prelude*

- We can then use the names in the module fully qualified

$one = Prelude.Nat.suc\ Prelude.Nat.zero$

- Or we can *open* a module

open *Prelude.Nat*
$one = suc\ zero$

Controlling what is imported

- We can exercise finer control over what is imported or opened.

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.

$$
\begin{array}{l}
\textbf{module } \mathit{BigProof} \textbf{ where} \\
\quad \textbf{private } \mathit{minorLemma} = \ldots \\
\quad \mathit{mainTheorem} : P == NP \\
\quad \mathit{mainTheorem} = \ldots \mathit{minorLemma} \ldots
\end{array}
$$

- Abstract things export only their type.

$$
\begin{array}{l}
\textbf{module } \mathit{Stack} \textbf{ where} \\
\quad \textbf{abstract} \\
\quad\quad \mathit{Stack} : \mathit{Set} \to \mathit{Set} \\
\quad\quad \mathit{Stack} = \mathit{List}
\end{array}
$$

- Private things still reduce, abstract things don't.

Parameterised Modules

- Modules can be parameterised.

$$
\begin{array}{l}
\textbf{module}\ \mathit{Monad}\ \ (M : \mathit{Set} \to \mathit{Set}) \\
\qquad\qquad\qquad\quad (\mathit{return} : \{A : \mathit{Set}\} \to A \to M\ A) \\
\qquad\qquad\qquad\quad ((>>=) : \{A, B : \mathit{Set}\} \to M\ A \to (A \to M\ B) \to M\ B) \\
\quad \textbf{where} \\
\qquad \mathit{liftM} : \{A, B : \mathit{Set}\} \to (A \to B) \to M\ A \to M\ B \\
\qquad \mathit{liftM}\ f\ \ m = m >>= \lambda x \to \mathit{return}\ (f\ \ x)
\end{array}
$$

- And instantiated

$$
\begin{array}{l}
\textbf{module}\ \mathit{MonadList} = \mathit{Monad}\ \mathit{List}\ \mathit{singleton}\ (\mathit{flip}\ \mathit{concatMap}) \\
\mathit{lemma} :\ \ \{A, B : \mathit{Set}\} \to (f : A \to B) \to (\mathit{xs} : \mathit{List}\ A) \to \\
\qquad\qquad \mathit{map}\ f\ \mathit{xs} == \mathit{MonadList.liftM}\ f\ \mathit{xs}
\end{array}
$$

- You need to instantiate a parameterised module to use it.

That's it folks

- 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.