A Semantics for Agda’s Mixed Inductive-Coinductive Types

Andreas Abel

Department of Computer Science
Ludwig-Maximilians-University Munich

15th Agda Implementor’s Meeting (AIM XV)
Fischbachau, near Munich, Germany
22 February 2012
Core Language for Agda

- Small language which can express Agda’s features
  1. (Co)inductive families
  2. Dependent pattern matching
  3. Coverage
  4. Universe stratification
  5. Termination

- Small Trusted Code Base
- Explaining induction-recursion, induction-induction, ...
- Input for backend (compiler)
Hitting the Core

Elements of a core language for logic and computation.

With quotes from L. & A. Wachowski,
*The Matrix Reloaded*
Causality (Implication)

Merovingian: You see, there is only one constant, one universal, It is the only real truth: causality. Action. Reaction. Cause and effect.

Lambda-calculus and dependent function space.

\[(x : A) \rightarrow B\]
\[x\]
\[\lambda x \rightarrow t\]
\[t u\]
Structure (Conjunction)

**KEYMAKER:** The system is based on the rules of a building.
One system built on another.
If one fails, all fail.

Pairing and $\Sigma$-type.

\[(x : A) & B, t, u, (p, p')\]
Choice (Disjunction)

**The Oracle:** We can never see past the choices we don’t understand.

**Morpheus:** Everything begins with choice.

**Neo:** Choice. The problem is choice.

Finite enumerations.

```
Bool
true
false
```
Recursion

**Agent Jackson:** You.

**Smith:** Yes me. Me, me, me!

**Agent Jackson/Smith:** Me too!

**Smith:** Go ahead, shoot. The best thing about being me—there's so many me.

Well-founded recursion.

\[ f_i = \ldots f_j \ldots \ [j < i] \]
Information (Subtyping)

**Merovingian:** Ai-ai-ai-ai-ai-ai, woman, this is nothing, c’est rien, c’est rien du tout.

No information. The unit type.

\[ A \leq \top \]

**The Oracle:** You can save Zion if you reach *The Source.*

All information. Pure potential. The empty type.

\[ \bot \leq A \]
From MiniAgda to a Core Language

- Dependent functions $\Pi$
- Dependent pairs $\Sigma$
- Enumerations $\text{Bool} = \{ \text{true}, \text{false} \}$
- Non-dependent pattern matching
- Equality
- Well-founded recursion
- Stratified universes
- Higher-order subtyping
- Sizes and measures
- Bounded size quantification $\forall i < j, \exists i < j$
Datatypes via Bounded Quantification

- Inductive: Lists (map)
- Coinductive: Streams
- Mixed: Stream processors (run)
MiniAgda code I

-- Prelude

data Empty {}
data Unit { unit }

let bot (A : Set)(x : Empty) : A
  = x

let top (A : Set)(x : A) : Unit
  = x

-- * Booleans

data Bool { true; false }
fun if : [A : Set] -> (b : Bool) -> (t, e : A) -> A
{ if A true t e = t
; if A false t e = e
}

fun If : (b : Bool) -> ++(A, B : Set) -> Set
{ If true A B = A
; If false A B = B
}

-- * Either: disjoint sum type

let Either ++(A, B : Set) = (b : Bool) & If b B A
pattern left a = (false, a)
pattern right b = (true, b)
MiniAgda code III

-- * Maybe: option type

let Maybe ++(A : Set) = Either Unit A
pattern nothing = left unit
pattern just a  = right a

-- Examples

-- lists

{-
List A 0    = {nil}
List A n+1  = {nil} \ {cons a as | a : A, as : List A n}
MiniAgda code IV

List A i = {nil, cons a as | a : A, as : \(\forall j < i.\) List A j} -}

cofun List : ++(A : Set) +(i : Size) -> Set
{ List A i = Maybe (A & \([j < i] & \) List A j) }
pattern nil = nothing
pattern cons a as = just (a, as)

fun map : \([A, B : Set] \ (f : A -> B) \ [i : Size] \ (l : List A i) -> List B i \{ map A B f i nil = nil ; map A B f i (cons a (j < i, as)) = cons (f a) (j, map A B f j as) \}
MiniAgda code V

-- streams

{-

Stream A 0  = A & \top
Stream A n+1 = A & Stream A n
Stream A oo  = /\i. Stream A i

Stream A i  = A & /\j<i. Stream A j

-}

cofun Stream : ++(A : Set) -(i : Size) -> Set
{ Stream A i = A & ([j < i] -> Stream A j)
}
fun head : \([A : Set][i : Size](as : Stream A i) \rightarrow A
{ head A i (a, s) = a }

fun tail : \([A : Set][i : Size](as : Stream A i) \rightarrow Stream A i
{ tail A i (a, s) = s i }

fun repeat : (A : Set)(a : A) [i : Size] |i| \rightarrow Stream A i
{ repeat A a i = a , repeat A a }

-- repeat A a i = a , (\ i’ \rightarrow repeat A a i’)

-- stream processors
MiniAgda code VII

-- data SP (A B : Set) : Set where
--  get : (A -> SP A B) -> SP A B
--  put : B -> co (SP A B) -> SP A B

-- run : {A B : Set} -> SP A B -> Stream A -> Stream B
-- run (get f) (a :: as) = run (f a) (force as)
-- run (put b sp) as = b :: Delay (run (force sp) as)

cofun SP : -(A : Set) ++(B : Set) -(i : Size) +(j : Size) -> Set
  { SP A B i j = Either (A -> ([j' < j] & SP A B i j'))
    (B & ([i' < i] -> SP A B i' #))
  }

pattern get f = left f
pattern put b sp = right (b, sp)
fun run : [A, B : Set] [i, j : Size] |i,j| \[sp : SP A B i j\]
  (as : Stream A #) \(\rightarrow\) Stream B i
  \{ run A B i j (get f) as = case f (head A # as)
    \{(j', sp) \rightarrow run A B i j' sp (tail A # as)\} \}
; run A B i j (put b sp) as = b ,
  (\ i' \rightarrow run A B i' # (sp i') as) \}