# Toward Agda-2
# Agda at CVS

Makoto Takeyama (just presenting)
Research Center for Verification and Semantics, AIST
(AIST/CVS)

2005-09-22

## Agda at CVS

**In collaboration with Chalmers University of Technology.**

- Agda
  - Plug-in Mechanism
  - Proof Libraries: Modal Propositional Mu, Kleene Algebra, Hoare Logic, TLAT logic, Modal Predicate Logic
  - Tutorial document
  - Agda Course
- Mendori prototype for 1st-class modules and functional-logic programming
- Agate compiler for dependently typed functional programming.
- Agda-2 Plan

## background: Comprehensive Verification Environment

- **A CVS goal:** Rigorously combine the strength of different formal verification methods and tools.
- **Problems:**
  - Different styles, formalisms, logics, $\cdots \Rightarrow$ Weak links in translations.
  - ?Emphasis on push-button solutions? $\Rightarrow$ Restrictive logics $\Rightarrow$ Low-level/contorted modeling, play-it-by-ear abstraction, $\cdots$; "yes, ok" does not help understanding.
- **Our approach:**
  - Unify various methods within a single, flexible, powerful framework.
  - Use an interactive proof assistant as a universal interface to the processes of using individual methods and tools

$\Rightarrow$ **Use Agda and Develop Agda-2**

## Agda

- **Agda Language:**
  Functional Programming Language based on Martin-Löf Type Theory.

- **Agda Interactive Proof-Assistant:**

The latest of the ALF-familiy family developed at Chalmers since '80s. AIST/CVS joined the effort since '04.

## Agda

**Agda Language:**

Functional Programming Language based on Martin-Löf Type Theory

- Dependent types, Inductive definitions, Universes, Termination checking
- Logic via Curry-Howard correspondence
- Some mod. cons. as a programming language: package, signature, $\cdots$

**vs. HOL based** (Why Agda): Computation(programs) is inherently integrated with logic (specifications / proofs). So programs and models themselves, not their logical axiomatization thereof, can be written (as well as tools).

## Agda

**Agda Interactive Proof-Assistant**

- Structured editor that is "typing-information aware"
- Refinement-style development programs and proofs
- Incomplete terms with metavariables
- Automatic synthesis of fragments by unification-based constraint solving

**vs. "Tactics-based" proof-assistant** (Why Agda):

On-screen, direct manipulation of proofs and programs.

Like **vi**, unlike **ed**. ~~Like **MS-Word**, unlike **TeX**.~~

Declarative, not procedural (in terminlogy of F. Wiedijk's)

## Agda Plugin-Mechanism

[D. Ikegami(CVS) with help from C. Coquand and U. Norell (Chalmers)]

- Generic mechanism to integrate external tools with Agda.
- Enables division of labor:

  Interactive Proving $\Leftarrow$ Problem decomposition, abstraction,
  lemma formulation, $\cdots$ with human insights and powerful logic.

  Automatic Proving $\Leftarrow$ Filling in tedious details with proof-search.
- Current instance: `santa+Gandalf` **First-Order Logic prover** plugin
- Planned:

  Abstraction tool for pointer manipulating programs **TLAT** [Takahashi &al., CVS]

  AC tree-automata tool **ACTAS** [Ohsaki &al., CVS]

  Model checkers `SMV`, `SPIN`, $\cdots$

  Random testing **QuickCheck** [Claessen & Hughes, Chalmers]

## Plugin Basic Interaction

1. Give inputs to the external tool in the form of a goal

   $\texttt{my\_proof ::}\quad P$
   $\quad\texttt{=}\ \texttt{\{ external}\ \mathit{name}\quad \mathit{opts}\quad \mathit{arg}1\quad \mathit{arg}2\quad \cdots\texttt{\}42}$

   | where | | |
   |---|---|---|
   | | $P$ | Problem description (as an Agda type) |
   | | $name$ | Plugin name (string) |
   | | $opts$ | Options for the plugin and external tool (string) |
   | | $arg_i$ | Extra arguments (Agda expressions) |

2. Invoke the "give" command.

3a. If the tool succeeds, `my_proof` is accepted as a postulated constant of type $P$ (an atomic proof of $P$).

   $\texttt{my\_proof ::}\ P$
   $\quad\texttt{=}\quad \texttt{external}\ \mathit{name}\quad \mathit{opts}\quad \mathit{arg}1\quad \mathit{arg}2\quad \cdots$

## Plugin Basic interaction

3b. If the tool fails, an error message shows the diagnosis from the tool and the goal remains unsolved.

```
my_proof :: P
  = { external name  opts  arg1  arg2  ···}42
{- Error: The problem P could not be solved.
   name said
   A counter example found: ...
-}
```

- Typing rule: $\dfrac{\text{The tool says ok}}{\texttt{external ... ::}P}$

The tool is called everytime the rule is applied.

## FOL Plugin

- First order logic prover plugin. `external ''fol''` *args* $\cdots$
- Input formulas given in the fragment of Agda type expression constructed from: (predicate) variables $X$`::  D ->...->Set`, $P$ `->` $Q$, `And` $P$ $Q$, `Or` $P$ $Q$, `Not` $P$, `(x::D)->` $P$ $(\forall x.P)$, `Exists (\(x::D)->` $P$`)` $(\exists x.P)$
- Extra *args* used to suggest useful lemmas to the prover.
- Higher-level description by computing the input formula in Agda: e.g.,
  $\forall x \, \varepsilon \, [x_1, x_2, \cdots, x_n].P\ x = \top \wedge P\ x_1 \wedge P\ x_2 \wedge \cdots \wedge P\ x_n$

```
  For_Elems_in::(xs::List D)-> (P::D -> Set)-> Set
  For_Elems_in [ ]       P = Truth
  For_Elems_in (x:xs') P = P x `And` For_Elems_in xs' P
```

- Translation to the `Santa` intermediate format[Claessen] then to `Gandalf` prover[Tammet]. Retargettable to other provers.

## Plugin To-do's

- **Re-establishing correctness**. For e.g. the FOL-plugin, Gandalf's logic is classical and over non-empty domain. We should accept "ok" only when proofs in Agda can be reconstructed. Need to
  - restrict the context in which `external ''fol''` can appear, or
  - restrict the form of input formulas (open geometric formulas; cf. [A. Abel, T. Coquand, U. Norell '05])
- More plugin instances, case studies, $\cdots$

## Proof Libraries

- Enables construction of proofs in formal systems other than the native Agda logic.
- In development:
  Modal Propositional Mu (Nagayama),
  Kleene Algebra and Hoare Logic (Kinoshita),
  TLAT logic (Tanabe),
  Modal Predicate Logic (Okamoto)
- Uses Agda as a meta-level language, “Logical Framework.”
- Embedding styles: Shallow embedding and Deep embedding

## Proof Libraries: Shallow Embedding

- Gives the semantics of, or interprets, or constructs a model of, the object-level system's formulas in Agda.
  Inference rules are proven in Agda (soundness).
- Object-level derivations are built from the rules, and/or from the model (with care).

E.g., Modal Mu with state set `S::Set` and transition `R::S-> S-> Se`

```
Prop::Set  =  S -> Set    -- modal mu props are
                          -- the Agda predicates on S.
And(P,Q::Prop)::Prop  =  \s -> P s × Q s
Box(P  ::Prop)::Prop  =  \s -> (t::S)-> R s t -> P t
                          -- (□P) s  ≡  ∀t. R s t ⇒ P t
(|-) :: Prop -> Prop -> Set    -- modal mu derivability is
P |- Q = (s::S)-> P s -> Q s  -- the Agda inclusion of predicates.
introAnd(Gamma,P,Q::Prop)::(Gamma |- P)-> (Gamma |- Q)->
                                (Gamma |- And P Q)
  = \ p q s g -> pair (p s g) (q s g)
```

## Proof Libraries: Deep Embedding

- Inductively defines the syntax of the object-level system's formulas and derivations. Inference rules are constructors.

```
data Prop :: Set = Atom(name::AtomicProps) | And(P,Q::Prop) | ...
                 | Box(P::Prop) | Mu(var::Vars)(P::Prop) | ...

idata (|-) :: Prop -> Prop -> Set where
   assumption(Gamma,P::Prop) :: (Gamma, P) |- P
 | introAnd(Gamma,P,Q::Prop) :: (Gamma |- P)-> (Gamma |- Q)->
                                     (Gamma |- And P Q)
 | introBox(P::Prop)         :: (Top |-     P) ->
                                (Top |- Box P)
 | ...
```

## Proof Libraries: Shallow vs. Deep

- Shallow:

  + consistency w.r.t. Agda,
  + Agda as the semantic common ground for different systems,
  + easier to define and use.

- Deep:

  + "completeness" (no model specific "theorems")
  + meta-theorems by induction on derivations,
  + automation of syntactic manipulation by Agda programs.

- Both are needed.

## Mendori

[Jeff Polakow (CVS)]

- A type checker for a Type Theory extension (minus inductive types) by [C. Coquand, R. Pollack, M. Takeyama '03].
  - Signatures with definitions
  - Record subtyping
- Stronger treatment of metavariables
- Basis for experiments on
  - structuring techniques
  - combining functional programming with Twelf-style logic-programming / proof-search.

## Mendori: Signature with definitions

- As organizational devices, Agda only has
  - 2nd-class parameterised module that bundles definitions.
    ```
    package MyPack(x1::A1)(x2::A2) where
      MyType :: Set =  ...
      myFunc :: MyType -> MyType = ...
    ```
  - labelled records (`struct`) that bundles values, of types given in `signature`
    ```
    MySig  :: Set   = sig    { label1 ::A1; label2 ::A2; ... }
    myVals :: MySig = struct { label1 = a1; label2 = a2; ... }
    ```

- We want to have both at once, e.g. when formalizing algebraic structures.

# Mendori: Signature with definitions

```
KA :: Type = sig                  -- Kleene Algebra signature
   K     :: Set                   -- requiring carrier set
   (==) :: K -> K -> Set          -- requiring some primitive predicates and operat
   (·)   :: K -> K -> K
   (+)   :: K -> K -> K
   star :: K -> K
   ...
   (<=) :: K -> K -> Set  =    p q ->  p + q == q
   ...                              -- Definition of derived concepts ...
   axiom14 :: (p,q,r::K)->
               q  + p  r <= r ->   -- ... used in later fields
               star p  q <= r
   ...
...
lem(X::KA) :: Transitive (X.(<=))  =  ...
                         -- the definition is available for any X::KA
```

## Mendori: Record Subtyping

- `sig` $S_1 \sqsubseteq$ `sig` $S_2$ if all fields required by $S_2$ are also required by $S_1$.
  E.g. `KAT` $\sqsubseteq$ `KA`, and any $X$ `::` `KAT` is also an $X$ `::` `KA` where

```
KAT :: Type = sig     -- Kleene Algebra with Test
 (... KA's fields ...)
  B   :: Set          -- plus some more fields required
  j   :: B -> K
  not :: B -> B
  ...
```

- $A \sqsubseteq$ `sig{ }` for any type $A$.
- Contravariantly extended to function types.

$$\frac{A_2 \sqsubseteq A_1 \qquad B_1\ x\ \sqsubseteq B_2\ x \quad (x : A_2)}{(x :: A_1) \to B_1 \sqsubseteq (x :: A_2) \to B_2}$$

## Mendori: Stronger treatmen of metavariables

- Stronger than Agda: Named meta-variables, multiple occurrences, without restrictions on positions.

- Inputs to the checker = incomplete term $M[?A, ?B, \cdots]$.
  Output =
  $$\begin{cases} \text{No, the error is ...} \\ \\ \text{Yes, where } (?A \ := ?X2 \texttt{ -> } ?X7, \quad ?B := \texttt{sig}\{f ::?X3; ?X15\}, \cdots) \\ \text{provided these constraints can be solved:} \\ \quad Nat \texttt{ -> } ?X4 \ \sqsubseteq \ ?X3, \quad ?X7 \ \sqsubseteq \ ?X7 \texttt{ -> } ?X5, \ldots \end{cases}$$

- Need to simplify inequational unification problems $\{A_i \sqsubseteq B_i\}_i$, wihch is subtle.
  E.g. $?X \ \sqsubseteq \ ?X$ `->` $?Y$ has infinitely many solutions.

## Mendori: towards incorporating Twelf-style

- Twelf - the implementation of *Edinburgh* Logical Framework.
- Embedding of object-level systems by extending the stock of primitive constants (signature) only. No inductive definitions or recursive functions within the Framework. The restriction is neede
to justify Higher Order Abstract Syntax representations.
- Proving about the object systems are done by induction on the whole of (canonical) Twelf terms, at the meta-level.
- Proving/programming is a logic programming over Twelf terms:

```
append   : list -> list -> list -> type.  % append is a type family.
appNil   : append nil K K.
appCons  : append L K M -> append (cons X L) K (cons X M)
....
twelf> ?- append (cons true nil) (cons false nil) L.
L = (cons true (cons false nil))
```

with facilities to check **`append`** really is a binary function, etc.

- Type Theory is thus used at a very different level from Martin-Löf'
LF. We plan to experiments on mixing this style with Agda's functional programming/proving style using Mendori.

## Agate Compiler

[H. Ozaki, N. Kato (CVS)]

- Goal:

  Practical dependently typed functional programming language
  – Efficiency, IO, programming methodology, library, . . .
  that can be rigorously reasoned about.
  – (superset of) Agda, Agda2

- Objectives:
  - Developing the practice of dependently typed programming,
  - Verified systems/tools for end-to-end verification (**Agda in Agda**)
  - Optimization techniques utilizing dependent types,
  - Verified components with specs,
  - . . .

## Existing Implementations of Dependently Typed languages

(why we do not go with them)

- Depedent ML (H. Xi):
  - Restricted notion of dep. types. Mostly geared for array bound checking, pre/post conditions on int. variables, sizes, etc.
  - Type-checking involves automatic theorem proving.
- Cayenne (L. Augustsson):
  - Sometimes, (good) taste overides foundational concerns. Undecidable type-checking.
  - Future continuity of development uncertain.
- Coq term language (byte-code compiled, B. Gregoir et. al.): (not meant for practical programming)

## Agate: Interpreter

[Ozaki] A preparatory development.

- Front-end transforms Agda code to Abstract Syntax Tree for untyped $\lambda$-calculus with constructors/case and struct/field-selection.
- Backend evaluates the AST. An extended and improved version of Setsoft's "Lazy Abstract Machine."
- Written in Java.

# Agate: Dependently Typed Programming

A spectrum of uses of dependent types: Logical $\longleftrightarrow$ Computational

- Logical: as predicates for e.g. pre/post conditions.

```
(>) :: Nat -> Nat -> Set
head :: (xs::List X)-> (length xs > 0)-> X
head (x:xs') p = x
head []      p = elimAbsurd p  -- assuming p :: 0 > 0, anything g
find :: (P::X -> Bool)-> (xs::List X)->
        sig {x::X; p::True(P x)} `Or` NotFound
```

+ Familiar. Meshes well with conventional Formal Methods.
– Proofs pollute programs. Boring.

```
cadr (X::Set) :: (xs::List X) -> (length xs > 1) -> X
cadr xs p = head (tail xs (lem1 p)) (lem2 p)
```

- Computational: making explicit, and computing with, the essential attributes(?) of values.

## Agate: Dependently Typed Programming

- Computational: making explicit, and computing with, the essential attributes(?) of values.

```
Vec :: (X::Set)-> (n::Nat) -> Set
-- xs :: Vec X n is a list whose length is exactly  n
Vec X 0     = Singleton
Vec X (1+n) = X × Vec X n
head(X::Set) :: (n::Nat) -> VecX (1+n) -> X
head n xs = fst xs -- no need to consider [] at all.
zip(X,Y::Set):: (n::Nat)-> (xs::Vec X  n)-> (ys::Vec Y  n)->
                Vec (X × Y)  n

AVL(X::Set) :: (height::Nat)-> Set

Exp :: (freeVars :: List Vars) -> Set
TypedExp :: (fvs::List Vars)-> (context :: Typing fvs) -> Typ ->
```

+ True Correct-by-Construction, no need even to prove it correct.

± New thinking, new techniques must be developed.

## Agate: Implementation

- Front end: reuse Agate interpreter's one.
  Backend: Glasgow Haskell Compiler (the only choice for lazy language.)

- Problem: – Dependent types do not map to Haskell-types.
  – GHC's untyped intermediate-language stage is hard to interface with.
  – Most GHC optimizations are done on typed IL.

- Our approach:
  Embed untyped $\lambda$-calculus in Haskell, at the source level.

# Agate: Translation to Haskell

- All-in-one Haskell data-type `Val` to represent Agda values, with Higher-order abstract syntax approach.

```
  -- Agda code (imagine this were not Haskell-typeable)
data Nat = Zero | Succ(n::Nat)
add ::Nat -> Nat -> Nat
add = \m -> \n -> case m of (Zero   )-> n
                            (Succ m')-> Succ(add m' n)
  -- Haskell translation
data Val  = Abs (Val -> Val) -- Agda functions
          | Zero | Succ Val  -- Agda constructors
          | ...              -- Agda struct, etc
(@$) :: Val -> Val -> Val    -- application
(Abs f) @$ v = f v
-- no other clauses are needed for well-typed Agda programs
add :: Val
add = Abs( \m -> Abs( \n ->
      case m of (Zero   ) -> n
                (Succ m') -> Succ (add @$ m' @$ n)))
```

## Agate: Translation to Haskell

Fairly efficient: GHC can optimize away much of `(VAbs...)@$...`, if the translation takes care to break up recursive definitions.

```
add :: Val
add = Abs( \m -> Abs( \n ->  addBody m n))
addBody :: Val -> Val -> Val
addBody m n = case m of (Zero   )-> n
                        (Succ m')-> Succ (add @$ m' @$ n)
        -- GHC immediately unfold this to addBody m' n
```

## Agate: Translation to Haskell

- No problem with non-haskell-typeable functions, e.g. variadic sum:

```
-- Agda code
N_AryFunc:: Nat -> Set
N_AryFunc = \n -> case n of (Zero   )-> Int
                            (Succ n')-> (Int -> N_AryFunc n')

-- sum n x0 x1 ... x_n = x0 + x1 + ... + x_n
sum :: (n::Nat) -> (x0::Int)->  N_AryFunc n
sum = \n x0 -> case n of (Zero   )->  x0
                         (Succ n')-> \x1 -> sum n' (x0+x1)
```

## Agate: IO

- IO computation can be typed in Agda as in Haskell.

```
-- Agda code
postulate IO :: Set -> Set
postulate returnIO :: (X::Set)|-> X -> IO X
postulate bindIO :: (X,Y::Set)|-> IO X -> (X -> IO Y) -> IO Y
postulate putStrLn :: String -> IO Unit
postulate getLine  :: IO(String)

-- read-print loop
main :: IO Unit
main = getLine `bindIO` putStrLn `bindIO` (\ _ -> main)
```

- Extend the Haskell type `Val` with Haskell IO-computation value.

```
-- Haskell code
data Val = ... | VIO (IO Val)
               | VString String | Vunit
```

# Agate: IO

- Haskell translations are as expected:

```
returnIO, bindIO, putStrLn’, getStrLn’, main :: Val
returnIO  = Abs( VIO . return )
bindIO    = Abs( \(VIO ma)-> Abs( \(VAbs f)->
            VIO(ma >>= \a -> let (VIO mb) = f a in mb)))
putStrLn’ = Abs(\(VString s)-> VIO(putStrLn s >> return Vunit))
getStrLn’ = VIO(VString ‘fmap‘ getStrLN)
main = bindIO @$ (bindIO @$ getLineLn’ @$ putStrLn’)
              @$ VAbs(\ _ -> main)
```

- GHC runtime system can run it.

```
runVal :: Val -> IO()
runVal (VIO m) = m >> return ()
```

- Can/should Agda typechecker run it??

## Agate: Struct

```
data Val = ... | VStruct [(Name, Val)]

(@.) :: Val -> Name -> Val -- field selection
(VStruct xvs) @. y  = let (Just v) = lookup y xvs  in v
```

## Agate: Open Agda terms

- Type checker needs to evaluate open terms.
- Extend the type `Val` for neutral Agda terms (a term whose head is a variable),

  ```
  data Val = ... | VN Neut
  data Neut = Var Name | App Neut Val | Dot Neut Label | ...
  ```

  and the definitions of `(@$)` and `(@.)` accordingly.

  ```
  (VN n) @$ v = VN (App n v)
  (Vn n) @. x = VN (Dot n x)
  ```

- The other approach for untyping Haskell using `unsafeCoerce` cannot do this.
- Efficient reduction during type checking is important for "reflective" proofs where reasoning is replaced by computations.

## Towards Agda2

- Goals: Clear theory, clear implementation, production quality, programming conveniences, ...
  Best of everything but **never sacrificing rigor and clarity**.

- Staged theorising and implementation:
  - Untyped Core programming language
  - Core Type System
  - Full language: meta variables, modules, hidden arguments, complex pattern matching, $\cdots$

  The full language is to be explained and justified in terms of the translation to the Core (cf. Haskell report).

## Agda2: Core Programming Language

[Coquand, Coquand, Norell (Chalmers), Takeyama, Polakow (CVS), for now]

- Term $M, N \;::=\; x \mid MN \mid \lambda x.M$ untyped $\lambda$-calculus

  $\mid\; c$ primitive constants

  $\mid\; f$ constants defined by pattern-matching

  $\mid\; a$ non-recursive abbreviations

  Fixed arities for constants $\mathrm{ar}(0) = 0$, $\mathrm{ar}(\mathsf{suc}) = 1$, $\mathrm{ar}(\mathsf{add}) = 2$, $\cdots$

- $f$ defined by left-linear mutually-disjoint pattern-matching clauses:

$$\left[\begin{array}{l} f\ x_1\ \ldots\ x_n\,(c\ y_1\ \ldots\ y_m\,) \;=\; M \quad (\mathrm{ar}(f) = n+1 \text{ and } \mathrm{ar}(c) = m \\ f\ x_1\ \ldots\ x_n\,(c'\ y'_1\ \ldots\ y'_{m'}) \;=\; M' \\ \qquad \vdots \end{array}\right.$$

  A term $f\ e_1\ \ldots\ e_n\ e$ that does not match is in a normal form.

## Agda2: Core Program Example

| primitive $c$ | ar($c$) | defined $f$ | ar($f$) |
|---|---|---|---|
| `0` | 0 | `add` | 2 |
| `suc` | 1 | `iter` | 3 |

```
add x  0       = x
add x (suc y) = suc (add x y)

iter x1 x2  0       = x2
iter x1 x2 (suc y) = x1 (iter x1 x2 y)
```

## Agda2: Core Operational Semantics

For closed (well-typed) terms

$$\frac{}{\lambda x.M \Downarrow \lambda x.M} \qquad \frac{}{c\,\boldsymbol{M} \Downarrow c\,\boldsymbol{M}} \qquad \frac{M \Downarrow \lambda x.M' \quad M'(x = N) \Downarrow v}{M\,N \Downarrow v}$$

$$\frac{N \Downarrow c\,\boldsymbol{N} \quad M(\boldsymbol{x} = \boldsymbol{M},\ \boldsymbol{y} = \boldsymbol{N}) \Downarrow v}{f\,\boldsymbol{M}\,N \Downarrow v} \left( \begin{array}{l} f\,\boldsymbol{x}\,(c\,\boldsymbol{y}) = M \\ \text{length}(\boldsymbol{x}) = \text{length}(\boldsymbol{M}) \\ \text{length}(\boldsymbol{y}) = \text{length}(\boldsymbol{N}) \end{array} \right)$$

## Agda2: Core Typing Overview

- Type system syntactically guarantees that well-typed terms are semantically correct.
- Semantics guarantees Strong Normalization for correct terms and justifies certain conversions.
- Modular two-stage proof:

(1) Give a domain model $\mathsf{D}$ s.t. $[\![M]\!] \neq \bot$ implies that $M$ is SN [T. Coquand & A. Spivak]

(2) Build a type system sound with respect to a PER model over $\mathsf{D}$ s.t. $\vdash M : A$ implies $[\![M]\!] \neq \bot$.

## Agda2: Core Typing Overview

- Martin-Löf's Logical Framework.
- Syntactic types are just some terms of the language, constructed with designated primitive constants.

$$\mathsf{Set}, \quad \mathsf{El}\, M, \quad \mathsf{Fun}\;\, A\; (\lambda x.B),$$
$$\mathrm{ar}(\mathsf{Set}) = 0, \quad \mathrm{ar}(\mathsf{El}) = 1, \quad \mathrm{ar}(\mathsf{Fun}) = 2$$

## Agda2: Core Type System

- Judgement forms:

$$\begin{array}{ll} \Gamma \text{ correct} & \\ \Gamma \vdash A \;\; \text{Type} & \Gamma \vdash A = B \;\; \text{Type} \\ \Gamma \vdash M : A & \Gamma \vdash M = N : A \end{array}$$

- Parameterised in a signature $\Sigma$ for constants (not shown).

## Agda2: Core Type System

$$\frac{}{()\ \text{correct}} \qquad \frac{\Gamma\ \text{correct} \quad \Gamma \vdash A \quad x \notin \Gamma}{\Gamma,\ x{:}A\ \text{correct}}$$

$$\frac{\Gamma\ \text{correct}}{\Gamma \vdash \mathsf{Set}} \qquad \frac{\Gamma \vdash A \quad \Gamma,\ x : A \vdash B}{\Gamma \vdash \mathsf{Fun}\, A\, (\lambda x.B)} \qquad \frac{\Gamma \vdash M : \mathsf{Set}}{\Gamma \vdash \mathsf{El}\, M}$$

$$\frac{\Gamma\ \text{correct} \quad x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma\ \text{correct} \quad c : A \in \Sigma}{\Gamma \vdash c : A}$$

$$\frac{\Gamma,\ x : A \vdash M : B}{\Gamma \vdash \lambda x.M : \mathsf{Fun}\, A\, (\lambda x.B)} \qquad \frac{\Gamma \vdash M : \mathsf{Fun}\, A\, (\lambda x.B) \quad \Gamma \vdash N : A}{\Gamma \vdash M\, N : B(x = N)}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash A = B}{\Gamma \vdash M : B}$$

# Agda2: Core Type System

$$\frac{}{\Gamma \vdash \mathsf{Set} = \mathsf{Set}} \qquad \frac{\Gamma \vdash A_1 = A_2 \quad \Gamma,\ x : A_1 \vdash B_1 = B_2}{\Gamma \vdash \mathsf{Fun}\, A_1\, (\lambda x.B_1) = \mathsf{Fun}\, A_2\, (\lambda x.B_2)}$$

$$\frac{\Gamma \vdash M_1 = M_2 : \mathsf{Set}}{\Gamma \vdash \mathsf{El}\, M_1 = \mathsf{El}\, M_2} \qquad \frac{\Gamma,\ x : A \vdash M_1\, x = M_2\, x : B}{\Gamma \vdash M_1 = M_2 : \mathsf{Fun}\, A\, (\lambda x.B)}\,\text{ext}$$

$$\frac{\Gamma \vdash M_1 = M_2 : \mathsf{Fun}\, A\, (\lambda x.B) \quad \Gamma \vdash N_1 = N_2 : A}{\Gamma \vdash M_1\, N_1 = M_2\, N_2 : B(x = N_1)}$$

$$\frac{\Gamma,\ x : A \vdash M : B \quad \Gamma \vdash N : A}{\Gamma \vdash (\lambda x.M)N = M(x = N) : B(x = N)}\,\beta$$

## (Telescope)

- $\boldsymbol{T} ::= () \mid (x : A)\boldsymbol{T}$

- obvious concatenations: $\Gamma\,\boldsymbol{T}$, $\boldsymbol{T}_1\,\boldsymbol{T}_2$

- obvious macros: $\mathsf{Fun}\,\boldsymbol{T}\,B$, $\lambda\boldsymbol{T}.M$, $M\,\boldsymbol{T}$, $c\boldsymbol{T}$(as a pattern). Ignore types except for the first.

$$\begin{aligned}
\mathsf{Fun}(x : A)(y : B)C &\equiv \mathsf{Fun}\,A\,(\lambda x.\,\mathsf{Fun}\,B\,(\lambda y.C)) \\
\lambda(x : A)(y : B).M &\equiv \lambda x.\lambda y.M \\
M((x : A)(y : B)) &\equiv M\,x\,y \\
(c((x : A)(y : B))) &\equiv (c\,x\,y)
\end{aligned}$$

- "$\boldsymbol{M}$ fits $\boldsymbol{T}$": $\dfrac{}{\Gamma \vdash () : ()}$ $\qquad$ $\dfrac{\Gamma \vdash M : A \quad \Gamma \vdash \boldsymbol{M} : \boldsymbol{T}(x = M)}{\Gamma \vdash (M, \boldsymbol{M}) : (x : A)\boldsymbol{T}}$

## Agda2: Core Inductive Definitions

- A declaration

$$\left[\begin{array}{lcll} d & : & \mathsf{Fun}\,\boldsymbol{T}_{\mathrm{param}}\,\boldsymbol{T}_{\mathrm{index}} & \mathsf{Set} \\ c_1 & : & \mathsf{Fun}\,\boldsymbol{T}_{\mathrm{param}}\,\boldsymbol{T}_1 & \mathsf{El}(d\,\boldsymbol{T}_{\mathrm{param}}\,\boldsymbol{M}_1) \\ c_2 & : & \mathsf{Fun}\,\boldsymbol{T}_{\mathrm{param}}\,\boldsymbol{T}_2 & \mathsf{El}(d\,\boldsymbol{T}_{\mathrm{param}}\,\boldsymbol{M}_2) \\ & \vdots & & \end{array}\right.$$

  extends the signature with primitive constants of shown types.

- It is correct if those types are correct and if, in each $\boldsymbol{T}_i$, $\mathsf{Set}$ does not ocur and $d$ appears only as $(d\,\boldsymbol{T}_{\mathrm{param}}\ldots)$. (Positivity check by termination checker.)

## Agda2: Core Pattern-Matching Function Definition

- A declaration

$$\left[\begin{array}{lllll} f & \multicolumn{4}{l}{: \mathsf{Fun}\, \boldsymbol{T}_{\text{param}} \boldsymbol{T}_{\text{index}}\ \mathsf{Fun}\,(\mathsf{El}(d\, \boldsymbol{T}_{\text{param}}\ \boldsymbol{T}_{\text{index}}))\, B} \\ f & \boldsymbol{T}_{\text{param}} \boldsymbol{T}_{\text{index}} & (c_1 \boldsymbol{T}_1) & = & N_1 \\ f & \boldsymbol{T}_{\text{param}} \boldsymbol{T}_{\text{index}} & (c_2 \boldsymbol{T}_2) & = & N_2 \\ \vdots & & & & \end{array}\right.$$

extends the signature with the defined constant $f$ and the computation rules.

- It is correct if $d$ is a data type defined as in the previous slide, if th type of $f$ is correct, if the clauses match the constructors, and if

$$\boldsymbol{T}_{\text{param}}\ \boldsymbol{T}_i \vdash N_i(\boldsymbol{T}_{\text{index}} = \boldsymbol{M}_i) : B(\boldsymbol{T}_{\text{index}} = \boldsymbol{M}_i)$$

- Other elimination patterns can be reduced to this.

## Agda2: Core Universes

- Extends MLLF with one universe to have a stratified hierarchy of $\mathsf{Set}_0$, $\mathsf{Set}_1$, $\cdots$.
- For $M : \mathsf{Set}_i$, $\mathsf{El}_i\, M$ is a type of "size" $i$
- Size-$i$ types are closed under $\mathsf{Fun}$, and also of size $i+1$.

$$\frac{\Gamma \vdash A \;\; \mathrm{Type}_i \qquad \Gamma, x{:}A \vdash B \;\; \mathrm{Type}_i}{\Gamma \vdash \mathsf{Fun}\, A\, (\lambda x.B) \;\; \mathrm{Type}_i} \qquad \frac{\Gamma \vdash A \;\; \mathrm{Type}_i}{\Gamma \vdash A \;\; \mathrm{Type}_{i+1}}$$

- It is not the case that $\mathsf{Set}_i : \mathsf{Set}_{i+1}$.
  Nor that, for $M : \mathsf{Set}_i$, $M : \mathsf{Set}_{i+1}$.
- But a one constructor inductive type can be defined to that effect:

$$\left[\begin{array}{l} \widehat{\mathsf{Set}_i} : \mathsf{Set}_{i+1} \\ \widehat{(-)} : \mathsf{Set}_i \longrightarrow \mathsf{El}_{i+1}(\widehat{\mathsf{Set}_i}) \end{array}\right.$$

## Agda2: Core Termination Checking

- Termination checking is separated from type checking.

- It will check inductive definitions are positive and loops in call-chains decrease "sizes" of arguments (Size-change termination, [D. Wahlstedt (Chalmers)]). More liberal than substructural recursion.