A Module System for Agda

Ulf Norell

Chalmers University of Technology

December 20, 2006
I (boldly) claim: “You don’t need a fancy module system”
I (boldly) claim: “You don’t need a fancy module system”
..and you tell me why I’m wrong.
Design of the module system

- **Purpose**
  - handle the scope of names

- **Goals**
  - (reasonably) simple
  - clear separation between scope checking and type checking

- **Consequences**
  - Modules don’t have types,
  - they’re not higher order
Design of the module system

Purpose
- handle the scope of names

Goals
- (reasonably) simple
- clear separation between scope checking and type checking

Consequences
- Modules don’t have types,
- they’re not higher order
- and they don’t have a categorical semantics.
Distinguish between modules and records.

- Modules structure names
- Records structure data
- Records are first class
- and should be used for things that the module system can’t do.
Distinguish between modules and records.

- Modules structure names
- Records structure data
- Records are first class
- and should be used for things that the module system can’t do.
- ..unfortunately we don’t have records yet.
A simple example

A module contains a bunch of declarations

module A where
  id : (A : Set) -> A -> A
  id A x = x

Outside the module the contents can be accessed using qualified names

zero' = A.id Nat zero

Or we can open the module to bring the contents into scope

open A
zero' = id Nat zero
Controlling what is imported

When opening a module we can choose to only bring certain names into scope.

```agda
open Nat, using (Nat) -- only Nat
plus : Nat -> Nat -> Nat
plus = Nat.plus

open Nat, hiding (plus) -- everything but plus

-- everything, but rename zero and suc
open Nat, renaming (zero to z, suc to s)
_+_ : Nat -> Nat -> Nat
z + m = m
s n + m = s (n + m)
```
You can declare things *private*, meaning that they will not be accessible outside the module (but they can still be computed with).

```agda
module Proof where
  private boringLemma : (A : Set) -> A
  boringLemma = ..
  mainTheorem : P == NP
  mainTheorem = boringLemma (P == NP)
```
Abstract definitions

An *abstract* definition does not reduce outside the module.

```agda
module A where
  abstract z : Nat
     z = zero
  -- here z reduces to zero
  zIsZero : z == zero
  zIsZero = refl

  -- but not here
  zIsZero : A.z == zero
  zIsZero = A.zIsZero {- we can’t use refl -}
```

Care has to be taken so that the definition of z doesn’t escape.
Parameterised modules

Modules can be parameterised (similar to sections in Coq)

```agda
module Sort (A : Set)(_<_ : A -> A -> Bool) where
  sort : List A -> List A
  sort xs = ..
```

A parameterised module can be applied to create a new module

```agda
module SortNat = Sort Nat natLess
```

Design decision: Is the following valid?

```agda
Sort.sort : (A : Set) -> (A -> A -> Bool) ->
  List A -> List A
```
A program can be split over multiple files.

- Principle: keep the file system out of the source code
- Each file contains a single top level module whose name corresponds to the file name.
- Type checking a file produces an interface file, containing essentially a dump of the proof state.
- Saves a lot of re-type checking.
Overview of the syntax

Decl ::= module M Tel where Decls
   | module M Tel = M’ Exprs [Modifiers]
   | import M [ as M’ ] [Modifiers]
   | open M [, public ] [Modifiers]
   | private Decls
   | abstract Decls
   | ...

Modifier ::= , using (x, ..)
   | , hiding (x, ..)
   | , renaming (x to y, ..)
Revisiting the goals

Our goals:

- Simple
  - We like to think it is.
Our goals:

- Simple
  - We like to think it is.
- Clear separation between scope checking and type checking.
  - No type checking during scope checking
  - No scope checking during type checking
No type checking during scope checking

- Modules cannot be passed around..
- ..and they don’t have types..
- ..so we don’t need type checking to figure out what names a particular module contains.
No scope checking during type checking

- Remove the module system during scope checking.
  - Modules are about managing names, so this should be possible.
  - Except.. performing module instantiations at scope checking might generate a lot of extra work for the type checker.
The type checking will see:

```
Decl ::= section M Tel Decls
|   apply M = M Exprs
|   import M
|   ..
```

- Names are fully qualified
- Scope control has disappeared
Implementing the scope checker

data Scope = Scope { name :: Name , publicNames :: Names , privateNames :: Names }

type Names = Map ConcreteName QualifiedName
type State = Stack Scope

- Entering a module:
  - push an empty scope on the stack
  - if parameterised, output a section

- Exiting a module: pop a scope from the stack
  - discard private names
  - put public names in the current scope (but qualified)
Example

module A where
  f : T <-
module B0 where
  g : T
module B where
  private g : T A - public : f -> A.f
module C where
  h : T
module A where
  f : T

module B₀ where
  g : T <--

module B where
  B₀ - public: g -> A.B₀.g
  private g : T A - public: f -> A.f

module C where
  h : T
Example

module A where
  f : T
module B0 where
  g : T
module B where
  B - private: g  ->  A.B.g
  A - public: f   ->  A.f
module C where
  B0   ->  A.B0
  B0.g ->  A.B0.g

Current stack

Example

module A where
  f : T
module B0 where
  g : T
module B where
  private g : T
module C where
  h : T

Current stack

C - public : h -> A.B.C.h
B - private: g -> A.B.g
A - public : f -> A.f
B0 -> A.B0
B0.g -> A.B0.g
module A where
  f : T

module B0 where
  g : T

module B where
  private g : T

module C where
  h : T

Current stack

B - public : C.h -> A.B.C.h
private: g -> A.B.g
A - public : f -> A.f
B0 -> A.B0
B0.g -> A.B0.g
<--
module A where
  f : T
module B0 where
  g : T
module B where
  A - public : f -> A.f
  private g : T B0 -> A.B0
module C where
  B0.g -> A.B0.g
  h : T B.C.h -> A.B.C.h
<--
Introduction
The Module System
Implementation
Conclusions

Example

Output from scope checking

A.f : T
A.B0.g : T
A.B.g : T
A.B.C.h : T
Other operations

- open A
  - for each A.B.x → y add B.x → y to the top scope
  - no output
- module A Δ = B es
  - push a module A
  - open B, public
  - pop A
  - if Δ is non-empty, output
    section _ Δ where apply A = B es
- using, hiding, renaming just affects what is added to the scope
- name resolution - look up the concrete name (in any part of the stack)
Implementing the type checker

After type checking:

- All definitions are lambda lifted.

What does the type checker have to do?

- Collect parameters
- Lambda lift definitions (after type checking)
- Apply sections (apply \( A = B \ es \))
  - check that the arguments \( es \) match the parameters of \( B \)
  - for each definition \( B.C.f \) create a new definition \( A.C.f = B.C.f \ es \)
Conclusions and Future work

Future work

- Mutual recursion between modules
  - same file: easy
  - different files: requires more machinery (including syntax!)
- Unifiying modules and local definitions
- Add records and **try some real examples**

Conclusions

- Simple - yes!
- Sufficiently powerful
Conclusions and Future work

Future work

- Mutual recursion between modules
  - same file: easy
  - different files: requires more machinery (including syntax!)
- Unifying modules and local definitions
- Add records and try some real examples

Conclusions

- Simple - yes!
- Sufficiently powerful
  - exercise for the audience