Hierarchical Modules in Agda – Scoping Rules

Andreas Abel

22 August 2019, revised 14 December 2019

1 Preliminaries

A programming language is both a mathematical object and a means of communication involving human beings. Thus, the design of programming languages cannot only focus on logical, semantic and efficiency aspects, but needs to take psychological dimensions into account, as well as human culture and experience, in particular in the field of software engineering.

In the design of scoping rules we have to follow the intuitive approach of humans to names and aim for the prevention of human misunderstandings and errors.

2 Pure Module Calculus

The Agda language has hierarchical modules which are referred to by non-empty lists of names, the last of those being the name of the module itself and the rest the names of its ancestors, i.e., enclosing modules. By just studying a fragment of the module language we can already formulate most of the principles of scope, ambiguity, and shadowing of names.

The grammar of the pure module calculus is given by the following BNF-style rules.

```
d ∈ Decl                        -- declaration (statement)
  ::= a 'module' x 'where' d*   -- module definition
    | 'open' q a                -- import from q

x ∈ Name                        -- simple name
q ∈ QName                       -- qualified name (non-empty list)
  ::= x⁺                        -- separator: '.'

a ∈ Access                      -- exported by 'open'?
  ::= 'private'                 -- not exported
    | 'public'                  -- exported
```

The access modifier a defaults to `public` in module definitions, and to `private` in imports. In Agda's concrete syntax, access modifiers are only present when diverging from the default.

Module references q such as in the `open` statement are always *relative* to the current point of view.

A module is populated by two actions: the definition of a new module x with content m of private and public definitions, or the private or public import of names x with meanings u from a module q. Definitions and imports may overlap, thus, x may become ambiguous. The value m of a module can thus be defined as a finite map from simple names x to a set containing all denotations u of x with access status a and origin o. The set of triples *(o,a,u)* shall be called a `NameSet`.

The value of a module is relative to a signature, which is a finite map from absolute names u to their contents m. For our purposes, one global such signature Σ is sufficient which is extended whenever a new module is fully defined.

```
u ∈ AName                                -- absolute name (unique)
o ∈ Origin                               -- origin of a binding
  ::= 'def'                              -- defined in this module
    | 'imp'                              -- imported from a module
NameSet = Set (Origin × Access × AName)  -- (ambiguous) denotation
m ∈ ModuleContent = Name ↦ NameSet       -- module value (content)
Σ ∈ Sig = AName ↦ ModuleContent          -- global signature
```

An import `open` q a adds the public content of q with accessibility a to the current module.

2.1 Ambiguity and clashes

It is useful to allow ambiguous names. While such names cannot be referenced, they may exist due to imports introducing overlap. For example:

```
module Top where
  module M where
    module O where
    module P where
  module N where
    module O where
    module Q where
  open M
  open N
  open P
  open Q
```

Both `M` and `N` define `O`, thus opening both `M` and `N` introduces an ambiguous name `O` with denotations `Top.M.O` and `Top.N.O`. However, this should not eagerly raise an error message; we may not care about `O`. We can still reference `P` imported from `M` and `Q` imported from `N` without ambiguity.

An attempt to reference an ambiguous name will raise an error which we call here `AmbiguousName`.

However, *wild ambiguity*, i.e., only denying the reference of ambiguous names, is not good enough for a robust software development process. In the following, we investigate some principles of *sane ambiguity*.

2.1.1 Names exported by a module may not be ambiguous

Ambiguity is a convenience for imports, allowing the omission of explicit import lists. *Exporting* an ambiguous name is pointless, as it can never be referenced. Thus, ambiguous exports should be ruled out by the scope checker.

Principle 1. A name can only have one **public** denotation.

Thus, a wellformed `NameSet` contains at most a single triple of the form *(o,*`public`*,u)*.

When checking the definition `module x where ds` of a new public module, we raise exception `PublicConflict` if `x` already has a public denotation in the current module. Likewise, this error is raised during `open q public` if any of the exported names of `q` already has a public denotation in the current module.

Private module definitions and imports are unaffected by this principle.

2.1.2 Names cannot be defined twice in a module

However, ambiguity should not be introduced in the absence of any imports, even not for private identifiers.

Principle 2. A name can only have a single **definition** in a module.

Thus, we may not define the same name `x` twice via `module x where...`, regardless of its accessibility. It may be a bit surprising that even a private definition cannot be shadowed by a public definition. But to the human eye, having two definitions of the same name in the same context is confusing. Also, since they have the same absolute name, those names cannot be refered to sensibly in a unique way, unless we let the accessiblity modifier be part of the name — complications we spare ourselves.

Formally, a wellformed `NameSet` contains at most a single triple of the form *(*`def`*,a,u)*.

When checking a definition `[private] module x where...` we raise the error `DuplicateDefinition` if `x` already has a defined denotation in the current module.

2.1.3 Reasonable permutability

The two principles we have seen so far do not contradict the following guarantee:

> Principle 3. In a well-formed module that has only *external* imports, shuffling the statements never introduces a `PublicConflict` or `DuplicateDefinition` error.

Restricting to the import from *external* modules is essential, internal references may change with permutation of the statements.

Principle 3 allows us to rearrange statements within sensible restrictions, e.g, we can always swap the following two statments

```
open M
private module N where ds
```

if the content of `ds` is independent from the content of `M`.

Slightly debatable is the permission to shadow a public import via a private definition:

```
module M where
  module L where
    module N where
      module O where
  open L public
  private module N where
```

Module `M` still exports `N` with content `O` even though inside `M`, name `N` has become ambiguous. The permission is in the spirit of Principle 3 to guarantee a certain order independence of wellformedness. The above code is allowed since there is no good reason to reject the code below:

```
module M where
  private module N where
  module L where
    module N where
      module O where
  open L public
```

2.1.4 Shadowing of definitions in parent modules

Current Agda (2.6.0) does not allow defining names that are already in scope. In contrast, our principle 2 only rules out shadowing definitions of the current module.

Principle 4. Definitions in parent modules may be shadowed.

This is the topic of Agda enhancement request #3801. Principle 4 is utilized by many of the examples presented so far.

2.1.5 Remark: accessibility in relation to export lists

In Agda, accessiblity information is attached to names *introduced* into a module. This is similar to accessibility modifiers in classes in Java-like languages or in ML signatures. Other languages, like Haskell, use export lists instead. Export lists have the advantage to gather all exports in one place, whereas in Agda without tool support, it is not always easy to see what a module exports. One has to traverse the DAG given by the `open _ public` statements. However, with respect to ambiguity, the Agda approach is more permissive. There, we can export ambiguous names als long they have only one public denotation. The disambiguation happens at introduction time. Export lists can only contain non-ambiguous references, of course. Haskell does have a remedy for this, though: qualified exports.

2.2 Formal specification

Let us specify the evaluation rules for declarations d via pseudo-code which operates on a state consisting of the following data:

1. the global signature Σ, a heap mapping unique names u to their content m, and
2. the context Γ, a stack of unfinished modules represented by pairs (x,m) of names x and current module content m.

(The state can be managed via a state monad which we call `ScopeM`.)

Service functions concerning the signature Σ are:

- `m ← getModule u` retrieves the contents `m` of the module designated by pointer `u`.
- `u ← allocModule q m` allocates a new module with absolute name `q` and content `m` and returns its uid `u` (pointer into the heap).

Service functions of the stack Γ are, beyond

- `push (x,m)` and `(x,m) ← pop`

two functions to unzip the stack into a list of names and a list of contents:

- `q ← getCurrentModuleName` extracts the sequence of module names `x` from the stack to get a hierachical module name `q`.
- `Γ ← cxt` returns the module contents of the whole stack, as a list with the top of the stack first. This is the context in which we resolve names.

More complex services will be defined from these primitive services below.

The main procedure of the scope checker is `checkDecl d` which checks `d` for well-formedness in the current context and modifies the context according to `d`.

```
checkDecl (a 'module' x 'where' ds) = do
  newModule x
  for d ∈ ds
    checkDecl d
  u ← closeModule
  addContent { x ↦ ('def',a,u) }

checkDecl ('open' q a) = do
  m ← lookupModuleContent q
  addContent { x ↦ ('imp',a,u) | x ↦ (_,'public',u) ∈ m }
```

The bracketing `newModule` / `closeModule` has a straightforward definition:

```
newModule x = push (x, ∅)

closeModule = do
  q      ← getCurrentModuleName
  (_, m) ← pop
  u      ← allocModule q m
  return u
```

The function `addContent` introduces new denotations into the current module, which is the top of the stack. The addition may introduce ambiguity violating principles 1 and 2; thus, we check for such conflicts.

```
addContent m₂ = do
  (x, m₁) ← pop
  let m = m₁ ∪ m₂
  if ∃x,u₁≠u₂. { x ↦ (_,'public',u₁), x ↦ (_,'public',u₂) } ⊆ m
     raise PublicConflict
  if ∃x,u₁≠u₂. { x ↦ ('def',_,u₁), x ↦ ('def',_,u₂) } ⊆ m
     raise DuplicateDefinition
  push (x, m)
```

Finally, `m ← lookupModuleContent q` resolves reference `q` and returns its value `m`. The reference might be undefined or ambiguous; then we raise `NotInScope` or `AmbiguousName`, respectively.

The name `q` might resolve in the current module or any of its parents. Thus we need to work through the whole context (stack) `Γ`. A naive procedure would first look for `q` in the current module (top of the stack), and when catching `NotInScope` continue recursively with the remaining modules in the stack. However, this would succeed for the following example:

```
module Top where
```

```
module M where
  module N where
    module P where
module O where
  module M where
  open M.N
```

While `M.N` is not defined in the current module `O`, it *is* defined in its parent module `Top`, thus, the `open` statement succeeds. However, this procedure suggest a different semantics of modules: maps from *qualified* names `q` to module contents, rather than maps from simple names `x`. In essence, this blends the contents of the current module and its parents together, where child content shadows parent content. The `open` statement would lose its compositionality; the following example does not succeed:

```
module Top where
  module M where
    module N where
      module P where
  module O where
    module M where
    open M
    open N
```

That `open M.N` should succeed but not `open M` followed by `open N` feels problematic. One would naturally expect that `M` has a submodule `N` which one can bring into scope by opening `M`.

The correct resolution of `q` should go through the stack, but commit to one stack element `m` as soon as the head of `q` can be resolved in `m`. The most direct implementation uses a failure continuation which is first set to look in the remaining stack and then reset to throw a `NotInScope` error:

```
lookupModuleContent q = do
  Γ ← cxt
  loop Γ
where
  err = raise NotInScope

  loop []     = err
  loop (m:ms) = lookFor q m (loop ms)

  lookFor (x:xs) m continuation = do
    case { u | x ↦ (_,_,u) ∈ m } of
      ∅  → continuation
      {u} → if null xs then return u else do
              m' ← getModule u
              lookFor xs m' err  -- discards continuation!
```

```
      else → raise AmbiguousName
```

The third argument of `lookFor` is the `continuation` that is only invoked should already the head `x` of `q` be unbound in `m`. Initially, `continuation` is set to `loop ms` that will search through the remaining modules `ms`, but after successful location of `x` in `m`, the `continuation` is reset to throw a `NotInScope` error.

(The skilled functional programmer will spot that `loop` is nothing but `foldr (lookFor q) err`.)

A Haskell implementation of the scope checker presented here is available on github.

2.3 Related Work

Ulf Norell [1] has described the design of module system for Agda, and its informal semantics, in 2006. At that time, Agda 2 was in the prototyping phase. In contrast, the present work is a reconstruction of scope checking given the current implementation in Agda 2.6.0 and some of the envisioned modifications, e.g., issue #3801.

There are two essential differences to Norell's conception [1]:

1. Norell requires names to be unambiguous always. The move towards ambiguous names happened later to improve the user experience when dealing with imports. Further, Agda was extended by ambiguous constructors (and later projections), that can be resolved by the type checker.
2. Norell does not use a signature (heap) to store fully defined modules. Rather, closing a module merges its contents into the parent module. As a consequence, module contents give meaning to qualified names q, rather than simple names x as in our semantics.

Concerning 2., we discussed that this semantics makes `open` less compositional in our setting. In present Agda 2.6.0 (and, presumably, Norell's setting), this is however not a problem, since shadowing definitions of parent modules, as proposed in issue #3801, is ruled out there. It is not clear yet whether the current semantics of modules can nicely integrate shadowing of parent definitions.

3 References

[1] Ulf Norell, A Module System for Agda, slides for talk at CHIT-CHAT 2006, Nijmegen, NL, 20 December 2006.