An Agda scope checker implemented in Agda

Master's thesis in Computer science and engineering

FRANCESCO GAZZETTA

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

MASTER'S THESIS 2023

An Agda scope checker implemented in Agda

Francesco Gazzetta

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Division of Computing Science
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

An Agda scope checker implemented in Agda

Francesco Gazzetta

Supervisor: Andreas Abel, Department of Computer Science and Engineering
Examiner: Patrik Jansson, Department of Computer Science and Engineering

Master's Thesis 2023
Department of Computer Science and Engineering
Division of Computing Science
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2023

An Agda scope checker implemented in Agda

Francesco Gazzetta
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Agda [1] is a Haskell-style, purely functional, dependently typed programming language and theorem prover. Scope checking is the process of analyzing the *Abstract Syntax Tree* (AST) of a program and resolving all references to symbols by connecting them to the corresponding declarations of said symbols. The Agda scope checker – as well as the rest of the compiler – is written in Haskell, and does not include any proof about the reachability of declarations. In this thesis, we present a scope checker for the Agda language, written in Agda itself, and prove the correctness of its name resolution algorithm with reference to the reachability properties of the Agda language. The result of this scope checking pass is a correct by construction AST that contains a proof that the syntax is well-scoped represented as paths from references of names (eg. `x`) to declarations (eg. `x = ...`).

Keywords: Computer science, Agda, compilers, scoping, well-scoped syntax, dependent types, name resolution, formal verification.

Acknowledgments

I would like to thank my supervisor Andreas Abel, my examiner Patrik Jansson, Beatrice, and my family.

Francesco Gazzetta, Gothenburg, October 2023

Contents

List of Figures

1

Introduction

Agda [1] is a Haskell-style, purely functional, dependently typed programming language and theorem prover. Its only compiler is currently written in unverified Haskell [2]. One important milestone of any programming language is the implementation of a *self-hosting compiler*, that is a compiler for the language written in the language itself. This demonstrates its maturity, eliminates dependencies on other languages, and enables a cycle of positive feedback.

A compiler is usually composed of sequential phases that produce a series of intermediate languages, the last being machine code. For example, the phases could be:

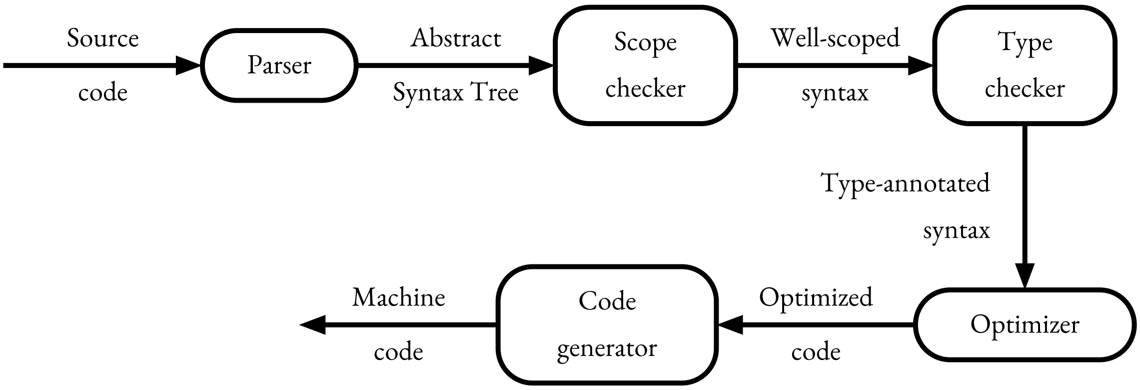
Parsing: a stream of characters is read and transformed into a hierarchy of syntactical constructs, the *Abstract Syntax Tree*.

Scope checking: References to names, such as variable use, are checked to ensure that the corresponding definition is in scope. This will be explained in more detail in 1.3.2.

Type checking: types of expressions are checked for consistency, type annotations are added to the syntax tree.

Optimization: the program is transformed into a semantically equivalent but more efficient version of itself, often through multiple passes.

Code generation: code in the target language, usually machine code, is generated.



Conventionally, *checking* phases in compiler pipelines do not retain evidence of the successful check[1]. For instance, type checkers do not store typing derivations. Likewise, scope checkers, the programs that resolve symbol references, do not retain evidence that every symbol could be resolved. Successive phases operate on *abstract syntax trees* (AST) without those guarantees, and have to perform unsafe lookups when operating on symbols. The guarantee that the lookup will succeed is thus only informal, and could fail if the compiler is changed.

In a more type-safe compiler, the proof that a reference is satisfied would be stored in the syntax tree. Lookups would then leverage the proof to find the referenced symbol in a safe way.

In the case of Agda, a self-hosting compiler would exploit the program verification facilities of the language to prove many useful properties about the various compiler phases.

- The scope checking phase would ensure that symbols are in scope, as explained above.
- The type checking phase would offer guarantees about the types of expressions.
- The optimization phase would have invariants over those types.
- The code generation phase would use those guarantees to avoid unsafe casts with no possibility of error.

Such a compiler would possess a number of correctness and safety properties by construction: the proofs generated in each phase would support the next phases, avoiding many unsafe operations.

A verified compiler would also result in a better understanding of Agda semantics, and could potentially uncover bugs or inconsistencies in the Haskell implementation or in the specification.

[1]Notable exceptions are *CompCert* [3] and *CakeML* [4].

In this project, we implemented the scope checking phase, that is, the phase that checks whether all referenced symbols are in scope, returning a well-scoped AST.

1.1 Contributions

The contributions we bring with this work are as follows:

- A well-scoped abstract syntax definition for the Agda programming language.
- A scope checker for Agda that produces the above syntax starting from the concrete one, if the input program is correctly scoped.
- A series of proofs about said abstract syntax that ensure that the result of the scope checker can only be correctly scoped.
- A more rigorous definition of some of Agda scoping rules.

1.2 Structure

In the rest of Chapter 1, we give an introduction to the features of the Agda language that are of interest for this project, explain the concept of scope checking, and cover some previous works on the matter. In Chapter 2, we analyze small subsets of the Agda language and provide a well-scoped syntax for each one. Finally, in Chapter 3 we apply the techniques used in the previous chapter to build the actual well-scoped syntax for Agda. We also explore its limitation and alternatives.

1.3 Background

1.3.1 Agda

Agda [1] is a Haskell-style, purely functional, dependently typed programming language and theorem prover.

1.3.1.1 Agda programs

An Agda program can be composed of multiple files. Agda files begin by defining the name of the module and its imports of other files through the `import` keyword, and continue with a series of declarations.

```
module M where

open import Agda.Builtin.Nat

four : Nat
four = 2 + 2
```

```
id : {a : Set} → a → a
id x = x
```

Note: in this thesis report, Agda syntax examples have a line to their left to distinguish them from code implementing the scope checker and well-scoped syntax.

Declaring the name of the module is optional and we will mostly omit it in examples. Scope checking multiple files rather than multiple modules in a single files is mostly a matter of reading the files in the right order, which is not relevant to the project, so we will focus on scope checking a single file and ignore `import` statements too.

Declarations have to be defined in order: a reference to a declaration cannot precede the declaration itself.

There are many types of declarations. The most common are type signatures (`four : Nat` and `id : a → a` in the example above) and function definitions that bind expressions to names (`four = 2 + 2` and `id x = x`). Usually type signatures and function definitions are coupled together. Their syntax is similar to Haskell. Other types of declarations are datatype definition, module definition and module use, and other miscellaneous constructs. Modules especially are of particular interest in this work, since they play a big part in name resolution: they are the main way a user of the language can create named scopes and they provide a good number of features. Comments start with `--`. Blocks of code are delimited by indentation.

1.3.1.2 Module system

The purpose of the module system is to structure the program in a hierarchical manner; each module can contain a number of declarations, including other modules. The contents of a module are determined by the indentation; at each nesting level, the amount of indentation increases.

```
module M where
  x : Nat
  x = 4
  module N where
    y : Nat
    y = 2
  y' = N.y
```

In the above example, `module M` contains `x`, `y'`, and `module N`, that in turn contains `y`.

There are two ways to access a declaration from outside the module it is declared in.

The first is to *qualify* its name by prepending the names of the modules containing it. For example, the fully qualified name of `y` is `M.N.y`, and to refer to `y` from within `M`, simply `N.y` can be used.

The second is to *open* the module containing the definition we want to use. Opening a module exposes all names defined in that module as if they were defined where the **open** declaration is.

```
module M where
  x = 1
open M
y = x + 1
```

In the above example, we can use `x` unqualified even if it is defined inside a module because we opened the module first.

Unlike many programming languages where module definitions and imports have to be declared in a predetermined section, usually at the top of the file before anything else, Agda modules can be defined or opened at any point in the program where a declaration can be. For example, this code is valid:

```
x : Nat
x = 0
module M where
  y : Nat
  y = 1
y : Nat
y = 2
-- here y is defined as 2
open M
-- here referring to y is ambiguous
```

Open declarations can also specify an *import directive* that can expose only a subset of definitions from the opened module, or change their names.

- `using (x; y; module M)`
- `hiding (x; y; module M)`
- `renaming (x to y; module M to N)`

Choosing whether to expose a definition can also be done from inside the opened module. All definitions inside a *private block* will not be reachable by qualified name syntax nor will be exposed by open declarations from outside the module they are defined in.

```
module M where
  private
    x = 0

-- Using M.x here is an error
```

```
open M

-- Using x here is an error
```

Modules can also be arbitrarily parametrized, and definitions inside a parametrized module will have access to the parameters. In the following example, `n` is a parameter of type `Nat` of `module M`, and we can see it used in expressions inside the module.

```
module M (n : Nat) where
  add : Nat → Nat
  add x = x + n

  mult : Nat → Nat
  mult x = x * n
```

Such parametrized definitions can be used from outside their module by either by providing the parameter directly, or by applying the entire module at once through *module application*.

For example, we can pass two arguments to `M.add`, the first being the module argument and the second being the regular argument.

```
four = M.add 2 2
```

On the other hand we can use module application to apply `M` to `3` and name the applied module `MThree`.

```
module MThree = M 3
```

Functions from `MThree` are already applied to the module argument, so they can be applied directly to their regular arguments.

```
five = MThree.add 2
six = MThree.mult 2
```

1.3.2 Scope checking

Scope checking is a compilation (or interpretation) step that happens after parsing and that, acting on an abstract syntax tree, checks whether each of the name references points to a name definition that is *in scope*, meaning it is usable in that particular place by writing its name or a variation of it. Conditions for a name to be in scope can vary wildly between languages. In *statically scoped*[2] languages like Agda, the definition of a name has to be reachable from its reference through a path in the syntax tree following the *scoping rules* imposed by the language, such as "previous declarations in the same module are in scope".

[2] Also known as *lexically scoped*

For example, in the program below, the declaration `y = x + 1` references `x`, which is found to be in scope because in Agda, referring to previous definitions in an upper module is allowed. In the last line of the program, however, the reference to `y` is invalid, because in Agda references to definitions within other modules have to be qualified with the module name (`N.y`). In that line, `y` alone is not in scope.

```
module M where
  x : Nat
  x = 0

  module N where
    y : Nat
    y = x + 1 -- x is in scope

  z : Nat
  z = y + 1 -- error: y is out of scope
```

If the definition is found to be in scope, the scope checker will then link it to the reference, for example by means of inserting it in a lookup table. This way, this information can easily be used by successive compilation phases, without having to find the definition each time, and without having to worry about handling scope errors. For example, when a type checker encounters a function call, it needs to check that all of this arguments are of the type specified in the function signature. To do that, it needs to retrieve the definition of the function, that includes its signature. If information about definitions of names was stored in the scope-checked syntax tree, it is enough to perform a lookup for the function name.

However, table lookup is an inherently unsafe operation, that can fail if the entry happens to be missing.

1.3.3 Limitations of unverified scope checkers

If the scope checker is not formally verified there is a risk that, due to errors in the implementation, references are not actually linked to a definition, or are linked to the incorrect one. When changes to the scope checker are made, there is a risk of introducing errors. Name lookups in successive phases would then fail.

Moreover, even assuming a perfect implementation, there is a loss of information. In the scope-checking phase, information about the presence of in-scope variables in the name lookup table exists and is usable. After this phase, without some kind of formal verification there would be no way to restore this information, and unsafe lookup operations would still be required. To act on the lookup result, the implementer would have to manually assert the safety of the operation.

1.3.4 Well-scoped by definition

By implementing a scope checker in Agda, it is possible to leverage dependent types to build an abstract syntax tree that is well-scoped by construction. A *well-scoped abstract syntax tree* is a type of syntax tree whose terms can only be constructed if a proof[3] of reachability of definitions from references is also provided. The proof can also enforce other constraints, such as matching the kinds of references and definitions. For example, a module import can only refer to a module definition, not to a function definition.

The proofs are produced directly by the scope checker when name resolution is performed, thus preserving the information gained during the operation.

Given a well-scoped syntax tree, it is possible to look up definitions from references with absolute certainty of success, enforced by the types.

1.3.5 Previous works

1.3.5.1 Scopes as Types

In *Scopes as Types* [5], A. Rouvoet et al. introduce a language to model name binding and resolution in a generic way, similar to how there exist parser generators that produce parsing code starting from a grammar written in a language-agnostic syntax. In the framework proposed by the authors, scope graphs are used to represent the binding structure of a program, where scopes are represented as vertices and reachability relations are represented by edges. To define how name resolution should actually be performed, a resolution calculus is introduced, capable of expressing reachability constraints and priority and uniqueness properties on bindings. They go on demonstrating how they modeled some programming language constructs in this language, and propose a limited version of a resolution algorithm.

The scope checker we developed is defined in a similar way. While in the paper a generic graph structure is used, we took advantage of the knowledge of Agda syntax and we used distinct datatype declarations to represent different parts of the graph, with types being the vertices and constructors the edges. Constraints on the constructors of the well-scoped tree serve the same function as the resolution calculus. This will be seen in more detail in Section 2.3.

1.3.5.2 Knowing When to Ask

In *Knowing When to Ask* [6], the authors expand on the previous work by developing an algorithm to perform the actual name lookup in a sound way. They especially pay attention to the order of lookups, since looking up a name may depend on a successful resolution of its dependencies. For example, resolving an imported name first requires resolving the import.

[3] A proof in Agda is a term belonging to a type corresponding to the proposition that is being proven. This is known as the *Curry-Howard correspondence* and will be shown in more detail in Section 2.2.

In this project, the difficulties to overcome were similar; in Agda there are many different and often indirect ways to refer to a binding, and the module system is especially complex. On the other hand, we had language-specific knowledge, so we manually took all dependencies into account when developing the resolution algorithm.

2

Scope checking fragments of the Agda language

Because the Agda language has a sizable and complex syntax, we will proceed by introducing some of its features one at a time. In each section of this chapter, a simple formal language that supports one of these features will be presented, together with the syntax its scope checker would yield.

2.1 Names

The most fundamental piece of data every scope checker has to deal with, is the unqualified name, that is a name without any contextual information attached. For example, in the λ-calculus expression $\lambda x.y$, variables x and y are both unqualified names[1]. A name can be defined as a string, but should be thought of as an opaque[2] type, supporting only equality checks. To ensure that we will not rely on implementation details, we declare it as a postulate of type `Set`, the type of types, omitting a definition entirely. In the actual scope checker code, this can be replaced by whatever is returned by the parser. The `Name` we will define represents a name in the source program, thus we declare it under `module C`, standing for *concrete syntax*.

```
module C where
  -- An atomic name opaque to the scope checker
  postulate Name : Set
```

We will often use names as arguments of other types. For example, this is a proof that a name is present in a two-element list that contains that name in its head:

```
prop1 : (x : C.Name) → (y : C.Name) → x ∈ x :: [ y ]
prop1 x y = here refl
```

Note: In the Agda code we present, we use definitions from the Agda standard library [7], such as $\in$.

[1] *Qualified* names will be presented in Section 2.3.
[2] Opaque types cannot be inspected by external code. No assumption should be made about their internal structure, and they should only be interacted with through the provided interface. In this case, for example, we can't assume that names are sequences of characters.

We give the ∈ and :: type constructors names `x` and `y` as arguments. Calling this function requires to explicitly supply `x` and `y`, but this kind of argument can almost always be derived from the context.

In these cases, Agda allows to mark arguments as *implicit*[3] by surrounding them with braces. Implicit arguments can be omitted both from function definitions and calls. We can change the previous definition to one that uses implicit arguments:

```
prop2 : {x : C.Name} → {y : C.Name} → x ∈ x :: [ y ]
prop2 = here refl
```

This modified `prop2` can be called without passing `x` and `y`, which are inferred from context.

Implicit arguments are often enough to remove clutter in an Agda program, but are insufficient in our case. When a specific set of names is always used to refer to implicit arguments of a specific type, it is desirable to omit their declaration entirely. In Agda this can be done with *generalized variables*[4], a way to declare generalization of variables in types in a single place in the code. When a generalized variable `v : T` is used in a type and `v` is not already bound, Agda will automatically insert an implicit argument `{v : T}`. This avoids visual noise when the binding of the variable is obvious from the context. For example, we can now rewrite our property avoiding all the clutter from the previous examples.

```
prop3 : x ∈ x :: [ y ]
prop3 = here refl
```

To this end, we define two *generalized variables* named `x` and `y` of type `C.Name`. We define them outside of the `C` module, so they will not need to be qualified on use.

```
variable x y : C.Name
```

2.2 λ-calculus

As a first fragment of the Agda language, let us consider the untyped λ-calculus. λ-calculus expressions are composed by abstractions, applications, and variables (Equation (2.1)).

$$e \quad ::= \quad \lambda v.e \quad | \quad e_1\, e_2 \quad | \quad v \tag{2.1}$$

λ-calculus expressions can be represented in Agda with the following datatype:

[3] More information about implicit arguments is available in the Agda documentation [8] at implicit-arguments.html

[4] More information about generalized variables is available in the Agda documentation [8] at generalization-of-declared-variables.html

```
data Expr : Set where
  -- λv.e
  abs : C.Name → Expr → Expr
  -- e₁ e₂
  app : Expr → Expr → Expr
  -- v
  var : C.Name → Expr
```

The above code is an Agda type definition. It is composed of a signature for the type constructor, in this case `Expr`, and arbitrarily many type signatures for the data constructors, in this case `abs`, `app`, and `var`. In Agda, types form an infinite hierarchy of *universes*[5], and `Set`, also called $\texttt{Set}_0$ for *universe 0*, is the lowest level, containing only *small types*, such as the one we are defining, hence `Expr` being of type `Set`.

Taking the above `Expr` definition, the untyped λ-expression $\lambda x.\lambda y.x$ would translate to `abs x (abs y (var x))`, where `x y : C.Name`.

As explained in Section 1.3.4, when such an expression is scope checked, a second syntax tree is produced, where all names are either bindings such as definitions and λ-abstractions, or valid references to existing bindings. Even when dealing with a language as simple as λ-calculus, there are many ways to represent a well-scoped syntax tree. One way is by expressing the well-scoped syntax as a relation between a scope and a concrete syntax. In Agda, a relation is usually expressed as a type constructor taking the relation sets members as parameters. For example, assuming the existence of a `Scope` type representing the set of names in scope, a well-scoped `ExprRelation` could be typed as (definition omitted):

```
data ExprRelation : Scope → Expr → Set
```

Since this type constructor takes two parameters, its type signature is a function type taking those and returning, again, `Set`.

Since this relation is directly over the concrete syntax, it also encodes the correspondence between unscoped and well-scoped syntaxes, but starts to become exceedingly complex as the syntax grows. For more information about this approach, see Section 3.5.

Another way to achieve this is by encoding references as De Brujin indices in an entirely separate syntax tree. The idea is to eliminate names from abstractions and replace names in references with an index representing the distance to the abstraction where the variable was bound. For example, $\lambda x.\lambda y.x$ would translate to $\lambda.\lambda.2$, because x refers to the second abstraction going outwards[6]. If the well-scoped syntax is written in a dependently typed language such as Agda, it can be made so

[5] More information about universes is available in the Agda documentation [8] at universe-levels.html and sort-system.html

[6] We assume indices start at 1. Existing literature is split between 0 and 1.

that De Brujin indices never exceed the number of abstractions enclosing them, by construction.

First, we re-define `Scope` to be isomorphic to $\mathbb{N}$ (in Peano representation), representing the number of abstractions in scope at any given point in the program, that is also the maximum number indices can assume.

```
-- Isomorphic to ℕ⁰
data Scope : Set where
  -- Empty scope, isomorphic to 0
  ϵ : Scope
  -- Expansion of scope, isomorphic to succ
  _▷ : Scope → Scope
variable sc : Scope
```

For example the scope within three bindings would be `ϵ ▷ ▷ ▷`. We chose `ϵ` and `_▷` as constructor names to show how the scope is similar to a stack. This will become more useful starting from later in this section as scopes increase in complexity.

Then, we define `SName`, a *well-scoped name*, also equivalent to a natural number but with the added restriction of being less than or equal to its `Scope` type parameter.

```
-- Isomorphic to {n ∈ ℕ⁺ . n ≤ sc}
data SName : Scope → Set where
  here : SName (sc ▷)
  there : SName sc → SName (sc ▷)
```

This restriction, as in all the types we will define, is enforced through the data constructors. The constructor `here`, meaning the name is defined in the immediately enclosing abstraction, ensures at least one abstraction is actually present by matching on `_▷` in its `Scope` parameter. The constructor `there`, meaning the name is defined in an outer abstraction, is the inductive step, taking another `SName` as parameter. This is a recurring pattern, as well as the key of the *Curry-Howard correspondence*: types define a well-scoped syntax (propositions), data constructors define the rules of the syntax (introduction rules), and well-scoped structures are evidence of the well-scoped syntax (proofs). The well-typed term `there here : SName (sc ▷ ▷)` is evidence that a De Brujin index of 2 (`there here`) is well scoped when inside at least two abstractions (`sc ▷ ▷`).

Finally, we define the actual syntax, `Expr`.

```
data Expr (sc : Scope) : Set where
  abs : Expr (sc ▷) → Expr sc
  app : Expr sc → Expr sc → Expr sc
  var : SName sc → Expr sc
```

The new `Expr` is similar to the previous one, but it adds a new type parameter `sc` of type `Scope`. The constructors were also adapted to make use of `Scope` and `SName`:

abs is defined so that its argument is parametrized by a `Scope` extended by 1 with respect to the scope of the `abs` expression. There is no `C.Name` argument.

app simply passes the `Scope` to its two arguments, unaffected.

var has a `SName sc` argument instead of `C.Name`. The `SName` is parametrized by the scope of the `var` expression. This way, the De Brujin index can only point to an existing binding.

In short, `C.Name`s get transformed into `SName`s, and the well-scoped syntax elements (just `Expr` for now) are indexed by the `Scope`.

With these changes, given an `Expr` ϵ, any `var` in it must by definition point to an `abs` in its scope.

In this syntax, $\lambda x.\lambda y.x$ would translate to this more complex expression:

```
abs (abs (var (there here)))
```

Notably, an expression such as

```
abs (var (there here))
```

which is equivalent to $\lambda.2$, would fail to typecheck as `Expr` ϵ:

```
(_sc_1 ▷) != ϵ of type Scope
when checking that the expression here has type SName ϵ
```

It would instead correctly typecheck as `Expr (`ϵ `▷)`, letting us explicitly acknowledge free variables by giving the type of the expression an expanded scope.

The syntax we just defined fulfills well the purpose of permitting only well-scoped programs to exist, but falls short in keeping track of names of abstractions and references, which is a fundamental feature used by practical compilers to produce user-readable messages, for example in case of a typechecking failure.

In the following iteration, we preserve names, both at definition and use site. Moreover, we introduce a proof that the name of a reference is always the same as the one of the definition it points to. This proof allows successive compilation phases to use these references without resorting to partial lookup functions, and can be the basis for more complex proofs.

We are going to alter all three types from the previous listings.

```
data Scope : Set where
  -- Empty scope
  ϵ   : Scope
  -- "Snoc" (opposite of cons)
```

```
    _▷_ : Scope → C.Name → Scope
  variable sc : Scope
```

The type `Scope` is no longer isomorphic to ℕ. Instead, it is now a *snoc-list* (a linked list where elements are added to the right end, opposite of a traditional *cons-list*) of `C.Name`s. We are using this type of list because it mirrors the natural direction of extension of a scope in a program: new definitions are added at the end of the program, and affect successive definitions. Its first constructor, ϵ, is the empty list, or empty scope, and its second constructor, `_▷_` is the *snoc operation* (opposite of *cons* in cons-lists) that takes a new argument of type `C.Name` and appends it to the scope. This is the name that was bound in that position of the scope.

```
  data SName : Scope → C.Name → Set where
    here : SName (sc ▷ x) x
    there : SName sc x → SName (sc ▷ y) x
```

The type constructor of the new `SName` has a second type parameter of type `C.Name` that is the same as the one of the scope it points to. This is ensured by the `here` constructor, where the `x : C.Name` type parameter is constrained to be the same as the top of the `Scope`. As before, the `there` constructor propagates the constraint to the upper scope by taking a `SName` parametrized on it.

```
  data Expr (sc : Scope) : Set where
    abs : (x : C.Name) → Expr (sc ▷ x) → Expr sc
    app : Expr sc → Expr sc → Expr sc
    var : (x : C.Name) → SName sc x → Expr sc
```

Finally, constructors of `Expr` are also augmented with names. Constructor `var` takes the name of the reference and ensures that it is the same as the one in the new `SName` type parameter, and `abs` links the name on top of the new scope to the name of the newly bound variable. Constructor `app` is unchanged.

2.2.1 Name shadowing

When a name is defined more than one time in a scope, *name shadowing* occurs. The later (or inner) definition *shadows* the earlier (outer) one, rendering it unreachable. Name shadowing is handled correctly only if references to names are always resolved to the latest (innermost) definition.

In the following example, with proper name resolution, x should refer to the second λx, ie. `here`.

$$\lambda x.\lambda x.x$$

The data structure we defined only ensures that variables refer to reachable abstractions of the same name. It does not enforce proper handling of shadowed bindings. For example, the following expression typechecks successfully, but is clearly ill-formed: `there here` refers to the outermost abstraction, and while it does have the same name, there is another binding for `x` that is closer to the reference.

```
shadowing : C.Name → Expr ϵ
--                 +-----------------+
--                 V                 |
shadowing x = abs x (abs x (var x (there here)))
```

The same will be true for other parts of the well-scoped syntax. We will check for shadowing in the scope checking phase, but we will not attach that information to the well-scoped syntax. In Section 3.4.2 we write more about this limitation.

2.2.2 Top-level declarations

Well-scoped top-level declarations can be implemented in a very similar way, since their structure is so similar to the λ-calculus we introduced. Bindings act like λ-abstractions, where a variable is bound for inner expressions. In a list of top-level declarations, the "inner expression" where the new variable is bound would be the tail of the list, or in more complex syntaxes, the branches of the syntax tree.

We briefly show a simple example of well-scoped syntax of a hypothetical language with only two declarations: variable definition, written as `define x` where `x` is the variable name, and variable reference, written as `reference x`, that can use previously defined variables. For example:

```
define x
reference x
define y
reference x
reference y
```

A well-scoped syntax for this language is remarkably similar to the one for λ-calculus. We omit `Scope` and `SName` definitions, as they are identical to the ones in the previous section.

```
data Decls (sc : Scope) : Set where
  ϵ   : Decls sc
  def : (x : C.Name) → Decls (sc ▷ x) → Decls sc
  ref : (x : C.Name) → SName sc x → Decls sc → Decls sc

-- Syntax of this program:
--   define x
--   reference x
example : C.Name → Decls ϵ
example x = def x (ref x here ϵ)
```

While this principle will be applied to many syntactic elements the next sections, in Section 2.3 we will instead implement declarations in a way that is more suited to features that depend on lists of declarations, such as mutual recursion, and that makes lists of declarations easier to extend.

2.3 Module calculus

Almost all practical programming languages include among their features a *module system*, a way to separate a program hierarchically into sections called *modules*, that can reference each other by name. Definitions are scoped by module, and mechanisms to manipulate modules and their interfaces are provided. A more detailed description of the Agda module system can be found in Section 1.3.1.2. In this section we will just use plain modules and the simplest of those manipulation features: module assignment.

Syntactically, Agda modules are introduced by the **module** keyword and are based on indentation. References to names defined inside a module from outside of that same module have to be prefixed with the name of the module followed by a dot. This happens recursively for nested modules. For example `M.N.B` refers to a `B` defined inside a module `N` itself defined within a module `M`.

```
module Module where
  module InsideModule where

-- References to InsideModule must be
-- qualified with Module
module OutsideModule = Module.InsideModule
```

We call these extended names prefixed by the names of their enclosing modules *qualified names*. We define qualified names as non-empty cons-lists of regular unqualified names, again inside the `C` module.

```
-- Continuing module C
  data QName : Set where
    -- Construction from unqualified name
    qName : Name → QName
    -- Name qualification
    qual : Name → QName → QName
```

Variables for `C.Name` were defined as `x` and `y`, so we choose `xs` and `ys` as `C.QName` variables to remind us that they are simply (non empty) lists of names.

```
variable xs ys : C.QName
```

The constructor `qName` builds a qualified name from a simple name, while `qual` further qualifies a name with the name of the module it is in.

Modules can be given additional names with a statement such as `module A = B`. This is a subset of the module application feature introduced in Section 1.3.1.2 that we will call *module assignment*.

```
module M where
  module A where
```

```
  -- Modules can nest arbitrarily
  module N where
    module B where

-- Example assignments of nested modules
module A' = M.A
module B' = M.N.B
module N' = M.N
module B'' = N'.B
```

A representation of the above syntax as an Agda datatype might look like this:

```
data Decl : Set where
  modl
    : C.Name -- Name of the module
    → List Decl -- Inner declarations
    → Decl
  modlAssignment
    : C.Name -- Left hand side
    → C.QName -- Right hand side
    → Decl
```

The constructor `modl` represents a module declaration, with its name and a list of inner declarations, and `modlAssignment` represents a declaration of a module assignment, with new and old name.

The well-scoped syntax follows the same pattern as the example in Section 2.2, but this time it employs mutually recursive definitions. This means that we first have to declare the types of our definitions, and only after that the definition bodies. We start back from `Scope`.

```
data Scope : Set
variable sc : Scope

data Decl (sc : Scope) : Set
variable d : Decl sc
data Decls (sc : Scope) : Set
variable ds : Decls sc

data DName : (sc : Scope) → Decl sc → C.QName → Set
data DsName : (sc : Scope) → Decls sc → C.QName → Set
data SName : Scope → C.QName → Set
```

As seen in the above listing, we still have a `Scope` of type `Set`. `Decl` is a declaration, and `Decls` is a list of declarations in a module, as we will show in the constructor definitions. Like `Expr` in the previous section, syntactic elements such as `Decl` and `Decls` are parametrized by their scope. This is a pattern that will repeat throughout.

Finally, we now have **three** types of well-scoped names: two of them for referencing a definition that is inside of the two syntactic elements (`DName` for `Decl` and `DsName` for `Decls`), and one for the scope itself (`SName`).

We can now define the bodies of the data definitions. We start with `Scope`. While in the λ-calculus example we only had to keep track of the list of names used in enclosing abstractions, with a module system we also need to keep track of the previous declarations in the same module and in all the enclosing modules. Moreover, we have to **point** to those declarations instead of just keeping track of names, so that they are easily accessible for further processing. Therefore, in this module calculus `Scope` changes from a simple list to a tree that connects all the in-scope syntax.

The actual definition of `Scope` though is still quite simple: a snoc-list of `Decls`. This is because the only connections that `Scope` adds to the syntax tree are **up through the module hierarchy**, while all other in-scope locations in the syntax can be reached by traversing the declarations referenced by `Scope`.

```
data Scope where
  -- Empty scope
  ϵ : Scope
  -- Scope expansion
  _▷_
    : (sc : Scope) -- Upper scope
    → Decls sc
    → Scope
```

Implementing a well-scoped syntax for module assignments is not straightforward. Instead of doing that directly, we first implement a further subset that we will call *module references*. A module reference is a module assignment without a left-hand-side. A module reference does not define any name. We write it as `module _ = A`, using _ to signify the absence of a name. This is not actually valid Agda syntax, but we use it nonetheless as an intermediate step to get to the full assignment syntax.

Accordingly, a declaration can be:

- A module, that has a name and contains more declarations inheriting the parent scope.
- A module reference, that points to an existing module through a `SName sc ys`, a well-scoped name `ys` over the scope `sc` of the module reference.

```
data Decl sc where
  modl : C.Name → Decls sc → Decl sc
  modlReference : SName sc ys → Decl sc
```

`Decls` is slightly more complex than a plain list of declarations. Declarations must be able to refer to previous declarations in the same module, so when a new declaration is appended to the list, its scope is extended with the tail of declarations: `sc ▷ ds`. Like `Scope`, `Decls` is also a *snoc-list*.

```
data Decls sc where
  -- Empty list
  ϵ : Decls sc
  -- "Snoc"
  _▷_
    : (ds : Decls sc) -- Previous declarations
    -- Last declaration. Previous declarations are in scope
    → Decl (sc ▷ ds)
    → Decls sc
```

Finally, we define well-scoped names. If a scope is the tree of reachable (in-scope) syntax, a well-scoped name is **a path through it**, pointing to a specific declaration in the tree and asserting that it defines a specific name. It is similar in meaning and implementation to the ∈ proposition over lists, that asserts that a specific element is present in a list and points to its location.

`DName sc d xs` asserts that a qualified name `xs` is defined in declaration `d`. A name can be defined in a declaration in two ways.

- The qualified name is composed of one segment, `qName x`, and the declaration is a module named `x`.
- The qualified name is composed of multiple segments, `qual x xs`. The first segment, `x`, matches the name of the module, and the rest, `xs`, is defined recursively in the contents of the module.

```
data DName where
  -- It is this module
  thisModule : {ds : Decls sc} → DName sc (modl x ds) (C.qName x)
  -- It is inside this module
  inside
    : {ds : Decls sc} -- Declarations within the module
    -- The name is defined in one of the declarations
    → DsName sc ds xs
    → DName sc (modl x ds) (C.qual x xs)
  -- There is no constructor for modlReference
  -- because it does not define names
```

Similarly, `DsName sc ds xs` asserts that `xs` is defined in one of the declarations in `ds`. Due to the similarity with ∈, we use the same constructor names: `here` and `there`. The constructor `here` takes a `DName` of the declaration at the head of the list, while `there` takes another `DsName` of the declarations at the tail. Note that the `here` constructor takes care of extending the scope of the last declaration with all the declarations that came before it (`sc ▷ ds`).

```
data DsName where
  -- It is in this last declaration (d)
  here : DName (sc ▷ ds) d xs → DsName sc (ds ▷ d) xs
  -- It is in one of the previous declarations (ds)
  there : DsName sc ds xs → DsName sc (ds ▷ d) xs
```

Finally, `SName sc xs` ties it all together by asserting that `xs` is defined in scope `sc`, that is, either in the same module (`here`) or in one of the upper modules (`there`).

```
data SName where
  -- It is in this module
  here : DsName sc ds xs → SName (sc ▷ ds) xs
  -- It is in one of the upper modules
  there : SName sc xs → SName (sc ▷ ds) xs
```

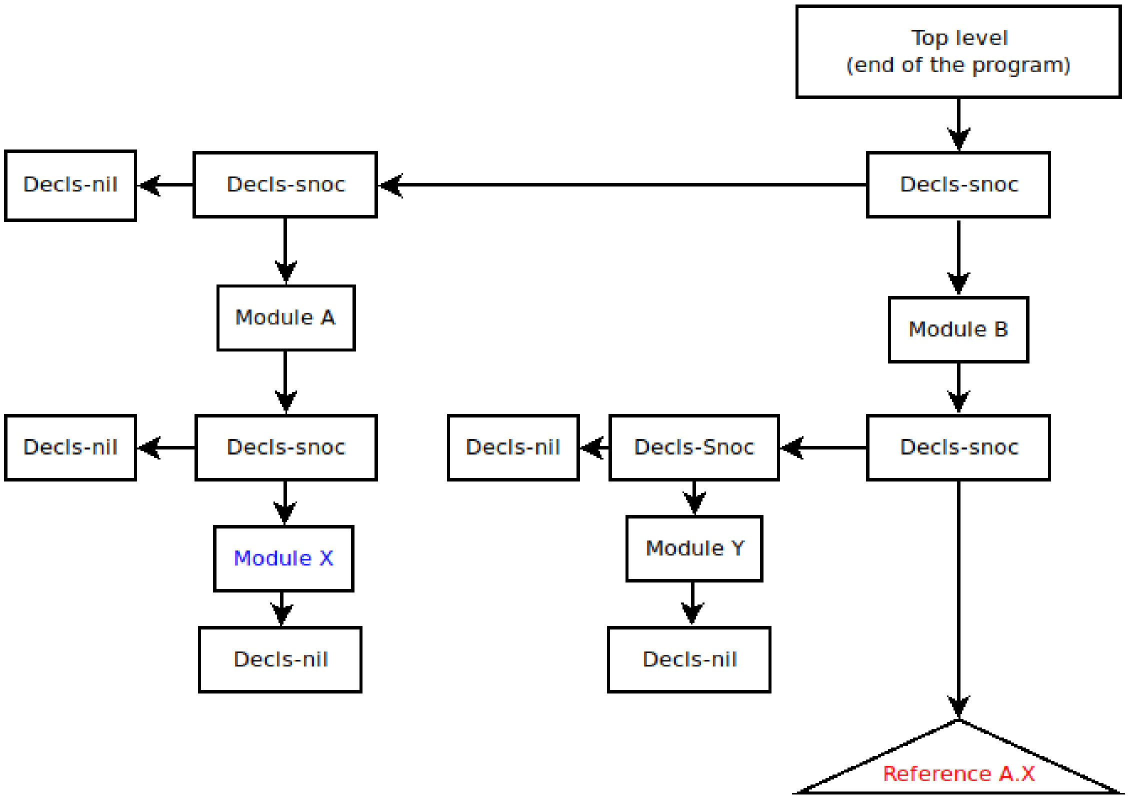
The structure of the scope and of well-scoped names can be better explained with a graphical representation of a program. In Figure 2.1 the structure of the following program is represented as a graph.

```
module A where
  module X where
module B where
  module Y where
  module _ = A.X
```

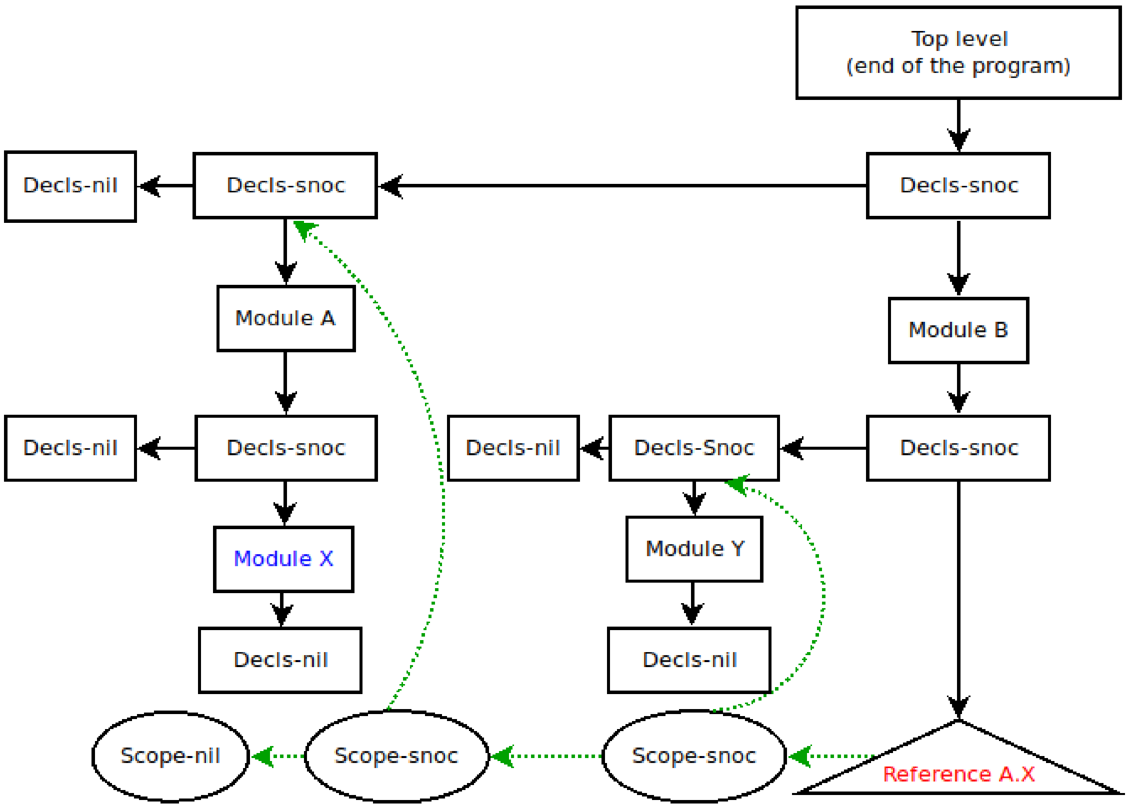
Nodes are constructors of the well-scoped syntax, and directed edges are pointers between them, ie. the constructors arguments. The module reference is highlighted in red, and the module it is referring to, `A`, is highlighted in blue. We can see that there is no path from the reference to `A` through the directed graph even though according to the semantics of the language it is accessible. The role of `Scope` is to produce such a path.

In the bottom half of the figure, the `Scope` that parametrized the reference is displayed. We can see that it provides edges up through the syntax tree, producing a second tree that covers all in-scope syntax. Module `Y` is covered, being in a previous declaration inside the same module as the reference, and so is `A`, being defined in the upper scope, and `X`, through `A`. `B`, instead, is not covered, as that would create cyclic and forward references to modules defined after the reference. Other top-level modules defined after `A` also are not covered by the tree.

The `Scope` connects all reachable syntax, but does not guarantee that the referenced name is reached. That is the role of `SName`: a path through the `Scope` tree that proves the reachability of a name. In Figure 2.2, the path encoded by a possible `SName` from the scope of the reference to module `X` is highlighted, with the names of the well-scoped name constructors written on the side.



(a) Syntax of a module expression



(b) The same syntax, with the scope of the reference added (green dotted arrows)

Figure 2.1: Representation of a scope as a graph through the syntax

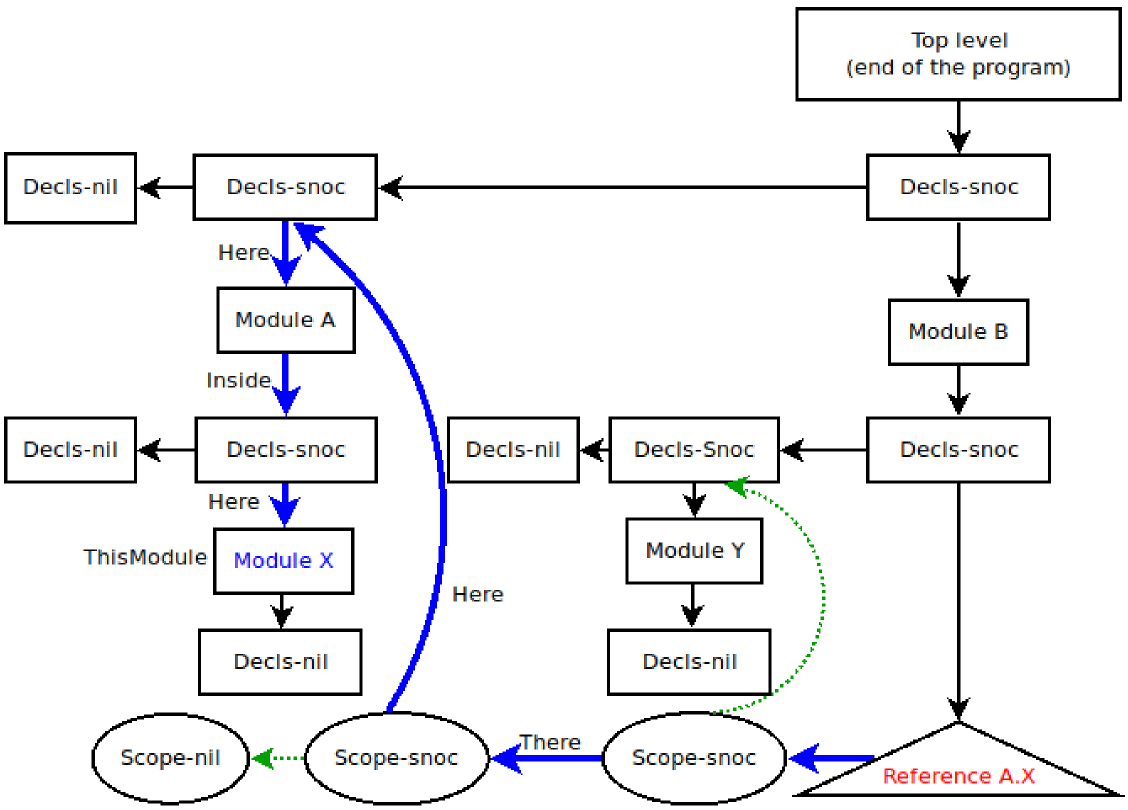


Figure 2.2: The same syntax and scope as Figure 2.1, but part of the scope is highlighted (thick blue arrows), representing the `SName` proof of reachability

2.3.1 Module assignment

We now extend module references to module assignments. The main challenge is being able to represent a well-scoped name that goes through a module assignment. When a well-scoped name reaches a module assignment, not only does it have to continue its path to the module referenced in the right-hand-side of the assignment (for that it would be enough to reference the well-scoped name of the assignment), it may also have to continue its path further inside the module pointed to by the right-hand side of the assignment.

```
module A where
  module B where
module A' = A
module B' = A'.B
```

For example, in the above program, to produce a well-scoped `B'` definition we must go through a few steps. We have to first build a path to `A'`, which is handled without problems by the previously defined well-scoped names. Then, the path must continue to `A`, which is also already possible through the well-scoped name in the right hand side of `module A' = A`. Finally, we must provide a path from `A` to `B`. This is problematic because the well-scoped name pointing to `A` does not provide direct access to its contents (it only proves its existence and reachability), so reusing that path is difficult. In other words, the portion of the path inside `A` will be completely detached from the rest of the path, because it is based on a portion of the syntax (the declarations inside `A`) that is not proven to be reached by the right hand side of `module A' = A`.

The problem is solved by augmenting well-scoped names with the contents of the pointed module. To do so, we first change the types of well-scoped names to include a `Decls` parameter at the end. We also introduce more generalized variables of type `Decls` called `pointedDs` and `pointedDs'` that will be used later to refer to the definitions inside the module pointed by well-scoped names.

```
data Scope : Set
variable sc sc' sc'' : Scope

data Decl (sc : Scope) : Set
variable d : Decl sc
data Decls (sc : Scope) : Set
variable ds : Decls sc
variable pointedDs pointedDs' : Decls sc

data DName : (sc : Scope)
           → Decl sc
           → C.QName
           → Decls sc' -- Body of the the pointed module
           → Set
```

```
data DsName : (sc : Scope)
            → Decls sc
            → C.QName
            → Decls sc'
            → Set
data SName : Scope
           → C.QName
           → Decls sc'
           → Set
variable sn : SName sc xs pointedDs
```

Then, we add the left-hand-side name `x` to the definition of the `modlAssignment` constructor (former `modlReference`).

```
data Decl sc where
  modl : (x : C.Name) (ds : Decls sc) → Decl sc
  modlAssignment : (x : C.Name) (sn : SName sc ys pointedDs)
                 → Decl sc
```

The most important change is in the `DName` definition (see below).

- In `thisModule`, the pointed declarations (`pointedDs`) are taken from the contents of the module (`modl x pointedDs`).
- In `inside`, the pointed declarations (`pointedDs`) are simply propagated.
- In `thisAssignment`, which covers the case where the last segment of the qualified name was reached, the pointed declarations (`pointedDs`) of the right-hand-side well-scoped name (`sn`) are used as pointed declarations of the whole `DName`.
- In `insideAssignment`, which covers the case where there are other name segments to follow, the pointed declarations (`pointedDs`) of the right-hand-side well-scoped name (`sn`) are used as a basis for continuing the path through a `DsName` parametrized on them. The declarations (`pointedDs'`) pointed by *that* `DsName` are used as pointed declarations of the whole constructor, as they are what is actually pointed by the well-scoped name.

```
data DName where
  -- The pointed decls are built
  thisModule : {ds : Decls sc}
    → DName sc (modl x pointedDs) (C.qName x) pointedDs
  -- The pointed decls are propagated
  inside : {ds : Decls sc}
    → DsName sc ds xs pointedDs
    → DName sc (modl x ds) (C.qual x xs) pointedDs
  -- The pointed decls are connected
  thisAssignment : {sn : SName sc ys pointedDs}
```

```
    → DName sc (modlAssignment x sn) (C.qName x) pointedDs
  insideAssignment : {sn : SName sc ys pointedDs}
    → DsName sc' pointedDs xs pointedDs'
    → DName sc (modlAssignment x sn) (C.qual x xs) pointedDs'
```

Finally, we propagate `pointedDs` through `DsName` and `SName`.

```
data DsName where
  here : DName (sc ▷ ds) d xs pointedDs
       → DsName sc (ds ▷ d) xs pointedDs
  there : DsName sc ds xs pointedDs
        → DsName sc (ds ▷ d) xs pointedDs

data SName where
  here : DsName sc ds xs pointedDs
       → SName (sc ▷ ds) xs pointedDs
  there : SName sc xs pointedDs
        → SName (sc ▷ ds) xs pointedDs
```

These modifications combined make it possible to build a proof of indirect reachability of a definition without having to perform unnecessary work.

2.4 Module calculus, with open statements

Another similar construct present in the Agda language is the open statement. As explained in Section 1.3.1.2, **`open`** `M` exposes the contents of module `M` as if they were defined in the same scope as the open statement, and this can be further controlled with **`using`**, **`hiding`**, and **`renaming`** directives.

The well-scoped syntax of open statements differs from the one of module assignments in two major ways: first, when building a scoped name through an open statement the first component of the qualified name is not consumed, since the contents of the module are exposed directly; and second, all exported names are collected in advance in what we will call an *interface*.

An interface is defined as a list of entries that map exported names to scoped names internal to the opened module, on which the interface is parametrized. Though a map would be more appropriate, a list was chosen for simplicity. It is built by the scope checker by processing the contents of the opened module together with the import directives. In a sense, it is comparable to defining an assignment for every name exported by the opened module.

The reason why we chose to collect all exported names in advance in an interface is that it makes building scoped names through it much simpler. Building scoped names through a raw open statement would require considering the directives. The proof of reachability would have to include a proof of membership in **`using`**, or a negative proof of existence in **`hiding`**, or a change of name through **`renaming`**, or almost any combination of those. This way instead it is enough to prove membership to the interface.

We implement an interface as follows.

```
-- An interface entry defines a mapping between the exported
-- name, and the Name of the internal name
record InterfaceEntry (ds : Decls sc) : Set where
  inductive
  constructor interfaceEntry
  field
    exportedName : C.QName
    innerQName : C.QName
    innerPointedSc : Scope
    innerPointedDs : Decls innerPointedSc
    innerDsName : DsName sc ds innerQName innerPointedDs

-- A module interface is a list of well-scoped exported names
-- (entries)
Interface : Decls sc → Set
Interface ds = List (InterfaceEntry ds)
```

For this definition we use the **record** syntax, which is a generalization of dependent products that allows us to define a type with multiple interdependent parameters in a clear manner over multiple lines. The keyword **inductive** defines this as an *recursive inductive record*. Recursive records are required to specify **inductive** or **coinductive** depending on the type of recursion (in short, finite or infinite). It is recursive because it contains names and declaration that as we will show can in turn contain interfaces. The **constructor** and **field** keywords define, respectively, the name of the constructor of the record and the names and types of the constructor parameters.

- `exportedName` is the name with which the definition attached to this entry is accessible from the outside of a module, taking into account **renaming**.
- `innerQName` is the name of that definition as declared inside the module.
- `innerPointedDs` are the declarations inside that definition (see Section 2.3.1).
- `innerDsName`, finally, is the well-scoped name pointing to the inner definition. That is, the portion of the path from reference to definition that goes through the opened module.

The `Decl` type is augmented with a new `opn` constructor, taking a scoped name pointing to the opened module and an interface parametrized over the declarations pointed by the scoped name.

```
data Decl (sc : Scope) : Set where
  opn : (m : SName sc xs pointedDs)
      → (iface : Interface pointedDs)
      → Decl sc
```

Finally, an `imp` constructor is added to `DName`, taking a proof that the name is a member of the interface.

```
data DName (sc : Scope) : Decl sc → C.QName → Decls sc' → Set
  where
  -- It is imported from another module
  imp : ∀{mName} → {m : SName _ mName ds}
      → {iface : Interface ds}
      → {dsn : DsName sc' ds ys pointedDs}
      → interfaceEntry xs ys sc'' pointedDs dsn ∈ iface
      → DName sc (opn m iface) xs pointedDs
```

2.5 Module calculus, with private blocks

In Agda, private blocks can prevent definitions from getting referenced outside of the module they are defined in. For example, in the following listing, module `A` can be referenced by the module application inside `M`, but not by the one outside.

```
module M where
  private
    module A where
  module A' = A

module A'' = M.A -- Error
```

We will add private blocks to the module references syntax used in Section 2.3. Instead of having a separate syntax element for them, we propagate the information about the kind of access we have over a declaration to the declarations themselves. To do this, we first define an `Access` type with constructors `publ` and `priv` that will be used to record whether a declaration was defined in a private block or not.

```
-- Whether a declaration is accessible from outside its scope
data Access : Set where
  -- It was not defined in a private block,
  -- and is publicly accessible
  publ : Access
  -- It was defined in a private block,
  -- and is not publicly accessible
  priv : Access
variable acc : Access
```

We also define a related but distinct type, `Clearance`, that will be used to express whether, at a certain point of the path defined by a scoped name, we are allowed to reference private definitions.

The types `Access` and `Clearance` are isomorphic and even have the same constructors, but are not to be confused.

```
-- The level of access to private declarations we have
-- at a certain point
data Clearance : Set where
  -- We can access only public declarations
  publ : Clearance
  -- We can access public and private declarations
  priv : Clearance
variable cl : Clearance
```

We define a relation named `canSee` between `Clearance` and `Access`. The relation is inhabited when we are allowed to reference a declaration tagged with `Access` when having `Clearance`. A `Clearance` of `priv` means we are allowed to reference everything, while a clearance of `publ` means we are only able to access definitions marked as `publ`.

```
-- Can we see a declaration marked with Access
-- when having a Clearance?
_canSee_ : Clearance → Access → Set
-- A priv Clearance can see everything
priv canSee _ = ⊤
-- A publ Clearance can only see publ Access
publ canSee publ = ⊤
publ canSee priv = ⊥
```

Specifically, we want to be able to access `priv` definitions only when referencing them from inside the same module. Hence the definition of a module constitutes a boundary in the syntax tree where well-scoped names drop their `Clearance` to `publ`.

We proceed by modifying the types we defined for module references.

The types of `Scope`, `Decl`, and `Decls` do not change. A `Clearance` type parameter is added to all the scoped names, meaning that at that point of the path through the scope, there is a certain clearance level.

```
data Scope : Set
variable sc : Scope

data Decl (sc : Scope) : Set
variable d : Decl sc
data Decls (sc : Scope) : Set
```

```
variable ds : Decls sc

-- All scoped names are parametrised by Clearance
data DName : (sc : Scope) → Clearance → Decl sc → C.QName
           → Set
data DsName : (sc : Scope) → Clearance → Decls sc → C.QName
            → Set
data SName : Scope → Clearance → C.QName → Set
```

The bodies of `Scope` and `Decls` do not change.

In the constructor `modl` of the type `Decl`, we add a parameter of type `Access` to represent whether the module was defined in a private block or not. The constructor `modlReference` has no `Access` parameter because it does not define any name.

In the constructor `modlReference`, in the scoped name defining the reference, we use a `priv` clearance, since the scoped name represents a part of the reference path that is in the same module as the reference itself. In other words, when we look for the definition we first look inside the module we are in, where we are allowed to access private definitions.

```
data Decl sc where
  modl : Access → C.Name → Decls sc → Decl sc
  -- We start with Clearance set to priv
  modlReference : SName sc priv ys → Decl sc
```

In every constructor of `DName` we use `canSee` to ensure that the current `Clearance` is sufficient to continue building the path. When descending inside a module with the constructor `inside`, the `Clearance` parameter is switched to `publ`, since we are no longer accessing declarations from the same module.

```
data DName where
  thisModule
    : {ds : Decls sc}
    -- We ensure the declaration is accessible
    → cl canSee acc
    → DName sc cl (modl acc x ds) (C.qName x)
  inside
    : {ds : Decls sc}
    -- We ensure the declaration is accessible
    → cl canSee acc
    -- When descending inside a module, Clearance switches
    -- to publ
    → DsName sc publ ds xs
    → DName sc cl (modl acc x ds) (C.qual x xs)
```

Finally, `DsName` and `SName` simply propagate the clearance without modifying it.

```
data DsName where
  here : DName (sc ▷ ds) cl d xs → DsName sc cl (ds ▷ d) xs
  there : DsName sc cl ds xs → DsName sc cl (ds ▷ d) xs

data SName where
  here : DsName sc cl ds xs → SName (sc ▷ ds) cl xs
  there : SName sc cl xs → SName (sc ▷ ds) cl xs
```

2.6 Module calculus, with expressions

Next, we combine the simplest form of the module calculus (without references, assignment, application, nor open statements) with expressions. We add a new type of declaration that assigns an expression to a name. For example, the following example is supported.

```
-- We define three identity functions, each based on the previous

f = λ x → x

module M where
  g = λ x → f x

h = M.g
```

To express this syntax, we need to introduce two new concepts.

The first is what we will call the *kind* of names, not to be confused with the concept of *type* as used in a type checker. A name kind defines the context in which a name can be used. It will let us distinguish names used for entirely different purposes, such as module names or names that can be used inside of expressions. In the full well-scoped syntax, it will allow us to attach kind-specific information to names. It is best understood with an example (bringing temporarily back open statements for the sake of clarity).

```
module M
  a = λ x → x

-- Correct usage of names
open M
f = λ x → a

-- Name kinds are mismatched
open a
f = λ x → M
```

In the above code, we first show correct usage of names according to their kinds. Name `M` is the name of a module, and it is used in an appropriate context, an `open` statement. Name `a` is also used correctly as an expression in an abstraction. In the last two lines instead, name kinds are mismatched: `M` is used as an expression and `a` as a module. As such, they make the definition nonsensical, and highlight the need to distinguish them.

We define `NameKind` as either the kind of names that can be used as modules, or the kind of names that can be used in expressions.

```
-- The kind of definition a scoped name refers to
data NameKind : Set where
  -- It must be a module
  moduleName : NameKind
  -- It must be a name that can be used in an expression
  exprName : NameKind
variable nk : NameKind
```

The second concept is the *block*. Until now, the scope was always built from homogeneous layers: either λ-abstractions in the λ-calculus or lists of declarations in the module calculus. When mixing modules and expressions, we can have both, so we add a new `Block` type to capture this. A `Block` is a layer of a `Scope` that can either be of λ-abstractions or declarations. In the full well-scoped syntax, blocks will be extended to support all the other syntactic elements.

We modify the usual types to include both blocks and name kinds. In particular, every scoped name is parametrized by a name kind.

```
data Scope : Set
variable sc : Scope

-- A block is a layer of a scope
data Block (sc : Scope) : Set
variable block : Block sc

data Expr (sc : Scope) : Set
data Decl (sc : Scope) : Set
variable d : Decl sc
data Decls (sc : Scope) : Set
variable ds : Decls sc

-- Every scoped name is parametrized by a specific name kind
data DName
  : (sc : Scope) → Decl sc → C.QName → NameKind → Set
data DsName
  : (sc : Scope) → Decls sc → C.QName → NameKind → Set
-- New
```

```
data BName
  : (sc : Scope) → Block sc → C.QName → NameKind → Set
data SName : Scope → C.QName → NameKind → Set
```

Now we can define the body of `Block`. It is either a list of declarations, or an abstraction.

```
data Block sc where
  -- It is a layer of declarations
  bDecls : Decls sc → Block sc
  -- It is a lambda abstraction
  bLam : C.Name → Block sc
```

We also modify `Scope` to use `Block` instead of `Decls`.

```
data Scope where
  ϵ : Scope
  _▷_
    : (sc : Scope)
    -- A scope is now made of blocks
    → Block sc
    → Scope
```

Expressions are similar to how they were defined in Section 2.2. The difference is in how the scope is extended and in the scoped name parametrized by kind.

```
data Expr sc where
  -- On abstractions, a bLam block is added
  abs : (x : C.Name) → Expr (sc ▷ bLam x) → Expr sc
  app : Expr sc → Expr sc → Expr sc
  -- We use exprName to avoid referring to a module
  var : (xs : C.QName) → SName sc xs exprName → Expr sc
```

The `Decl` type has a new `expr` constructor, representing the assignment of an expression to a name.

```
data Decl sc where
  modl : C.Name → Decls sc → Decl sc
  modlReference : SName sc ys moduleName → Decl sc
  expr : C.Name → Expr sc → Decl sc -- New
```

The `Decls` type is also adapted to use the new `Block` type. Since it extends it with declarations, it uses a `bDecls` block.

```
data Decls sc where
  ϵ : Decls sc
  _▷_
    : (ds : Decls sc)
```

```
    -- To add declarations to the scope, we use bDecls
    → Decl (sc ▷ bDecls ds)
    → Decls sc
```

In `DName`, we make sure to set the name kind to the appropriate value depending on whether the name refers to a module or an expression.

```
data DName where
  -- It is a module, so we set the name kind to moduleName
  thisModule : {ds : Decls sc}
             → DName sc (modl x ds) (C.qName x) moduleName
  -- The name kind is propagated
  inside
    : {ds : Decls sc}
    → DsName sc ds xs nk
    → DName sc (modl x ds) (C.qual x xs) nk
  -- It is this expression, so we set the name kind to exprName
  thisExpr : {e : Expr sc}
           → DName sc (expr x e) (C.qName x) exprName
```

Lastly, we propagate the name kind through `DsName`, `BName`, and `SName`, and we terminate the path if a lambda abstraction with appropriate kind and name is found.

```
-- The name kind is propagated
data DsName where
  here : DName (sc ▷ bDecls ds) d xs nk
       → DsName sc (ds ▷ d) xs nk
  there : DsName sc ds xs nk
        → DsName sc (ds ▷ d) xs nk

data BName where
  inDecls : DsName sc ds xs nk
          → BName sc (bDecls ds) xs nk
  -- The path terminates at the abstraction
  inLam : BName sc (bLam x) (C.qName x) exprName

-- The name kind is propagated
data SName where
  here : BName sc block xs nk
       → SName (sc ▷ block) xs nk
  there : SName sc xs nk
        → SName (sc ▷ block) xs nk
```

2.7 Mutual interleaved bindings

When designing a well-scoped syntax for mutual declarations, we face an interesting challenge: declarations need to be able to refer to each other in arbitrary ways, and we also need a way to reach all clauses of a function or data definition by starting from its signature, so a compiler can process the function in its entirety.

The first issue is solved by forward declarations and interleaved declarations[7]. These allow the user to freely interleave declarations and definitions of different functions, so that all references happen after the corresponding declarations. These features align with the `Decls` definition, where successive declarations are allowed to refer to previous ones.

To solve the second problem, we have to collect all the clauses of a function under its type signature. We will do this by means of a wrapper that decorates type signatures with clauses.

We start by adding a block for function left-hand-sides, as described in Section 2.6.

```
data Block (sc : Scope) : Set where
  bDecls : Decls sc → Block sc
  bLhs : C.Name → Block sc
```

Then we forward-declare `Mutual`, a wrapper for the `Decls` type that will be used when mutual declarations are needed.

```
Mutual : (sc : Scope) → Decls sc → Set
```

We add constructors for type signatures (`typeSig`) and function clauses (`funClause`) to `Decl`. Type signatures are composed of the name of the binder and the expression representing its type. Function clauses have a name, a left-hand-side, and a right-hand-side expression. For simplicity, we define the left-hand-side as a single binding. The `mutual'` constructor represents mutual blocks of declarations.

```
data Decl sc where
  typeSig : (x : C.Name) → (ty : Expr sc) → Decl sc
  funClause : (x : C.Name) → (lhs : C.Name)
            → Expr (sc ▷ bLhs lhs) → Decl sc
  mutual' : Mutual sc ds → Decl sc
```

We add a way to attach a function clause to a matching[8] type signature, and call it `Clause`. We use `SName` to ensure the function clause is reachable, but we base it on a *top scope*. The top scope is the scope containing the entirety of the mutual block. Its origin will be shown later.

[7]More information about forward and interleaved declarations is available in the Agda documentation [8] at mutual-recursion.html

[8]Meaning they share the `Name`

```
data Clause (topScope : Scope) : Decl sc → Set where
  funClause : SName topScope (C.qName x)
            → Clause topScope (typeSig x ty)
```

By building a list of `Clause`s, we can attach all clauses to their matching type signature, so that they are all reachable from it. Note that `d`, the type signature declaration, is fixed.

```
Clauses : (topScope : Scope) → Decl sc → Set
Clauses topScope d = List (Clause topScope d)
```

Now we define `Deco`, a wrapper over declarations, specifically type signatures, that decorates them with function clause information from the top scope `topScope`. This decorator is defined such that its structure matches the one of `Decls`, with matching empty and snoc constructors. The key component of the definition is the snoc constructor, `ds ▷[ d , cs ]`. It appends to previous decorated declarations (`ds`) a new declaration (`d`) and its clauses (`cs`), matching the undecorated list of declarations (`ds ▷ d`) in the second argument of `Deco`.

```
data Deco (topScope : Scope) : Decls sc → Set where
  ϵ : Deco topScope {sc} ϵ
  _▷[_,_] : Deco topScope {sc} ds
          → (d : Decl (sc ▷ bDecls ds)) → Clauses topScope d
          → Deco topScope (ds ▷ d)
variable deco : Deco sc ds
```

We complete the definition of the `Mutual` wrapper. It is a decorator where the first argument (the top scope) is the parent scope augmented with all the mutual declarations, and the second argument (the declarations) are the undecorated mutual declarations.

```
Mutual sc ds = Deco (sc ▷ bDecls ds) ds
```

Afterwards we add a well-scoped name for definitions in mutual declarations. Its structure is identical to the one of `DsName` in Section 2.3.

```
data MName (sc : Scope) : Deco (sc ▷ bDecls ds') ds → C.QName
                        → Set where
  here  : ∀{clauses} → DName _ d xs
        → MName sc (deco ▷[ d , clauses ]) xs
  there : ∀{clauses} → MName sc {ds = ds} deco xs
        → MName sc (deco ▷[ d , clauses ]) xs
```

Finally we can insert a new constructor in `DName`, where we propagate the decorator from the `MName` to the `mutual'` declaration.

```
data DName sc where
  thisMutual : MName sc deco xs
    → DName sc (mutual' deco) xs
```

3

The complete well-scoped syntax and scope checker

Having explored how various fragments of Agda can be expressed in a well-scoped manner in Chapter 2, we are now equipped to present the entirety of this project.

The implementation consists of two main parts: the well-scoped abstract syntax, and the scope checker.

The well-scoped syntax captures, in addition to the program itself, the properties we want to prove about it, making it impossible to construct an ill-scoped program due to the use of well-scoped names in all references. This is the most important part, as it allows us to formalize what it means for an Agda program to be well-scoped.

The scope checker starts from the concrete syntax of an Agda program and produces a well-scoped AST.

3.1 The concrete syntax

As explained in Chapter 1, the scope checker's input is the language's syntax tree directly after parsing. The syntax tree's representation is close to the actual syntax of the Agda language as it is written. This is opposed to the abstract well-scoped syntax the scope checker will output. For this reason, we will simply call the scope checker's input *concrete syntax*.

Since parsing is not the focus of this project, we obtain the concrete syntax by calling the parser of the Agda compiler [2]. The Agda compiler is written in Haskell, and Agda conveniently provides a Haskell *foreign function interface*, which is a way for a programming language to interact with libraries written in another programming language. Therefore, if appropriate types, signatures, and annotations are provided, it is possible to call functions and get data from the Haskell implementation of Agda. To use the existing Agda parser, we wrote Agda types corresponding to the Haskell syntax types and Agda signatures corresponding to the Haskell parsing functions. This was kept under a separate module from the rest of the scope checker. The rest of the project can simply call the main parser function as if everything was written in Agda. Eventually, this could be replaced by a pure Agda implementation.

3.2 The well-scoped abstract syntax

In Chapter 2 we showed well-scoped syntax definitions for many fragments of the Agda language. In this section, we will apply the techniques we described all at once to form the complete well-scoped abstract syntax. We will go through the syntax definitions and explain how this was done.

This abstract syntax does not cover every feature of Agda, but it does cover a wide enough variety of them so that implementing the rest should be a matter of applying the same methods to the remaining features. For example, *pattern synonyms* and *named where blocks* are not covered, but the first are similar to function and data declarations, while the second are similar to module declarations.

3.2.1 Basic names and types

First we define the part of the syntax that is common between concrete and well-scoped syntaxes, that is, basic names and literals.

Basic names, without proofs attached, are identical to the ones defined in Section 2.1. We add a new `Literal` type that defines various types of Agda literals such as numbers and strings. These basic constructs are all also part of the concrete syntax definition, so we define everything under `module C`.

```
module C where
  Name : Set
  Name = String

  data QName : Set where
    qual : Name → QName → QName
    qName : Name → QName

  data Literal : Set where
    litNat : ℕ → Literal
    litWord64 : Word64 → Literal
    litFloat : Float → Literal
    litString : String → Literal
    litChar : Char → Literal

variable x y : C.Name
variable xs ys xs' : C.QName
```

Before anything else, since many definitions are going to depend on each other, we also need to forward-declare some basic types: `Scope`, `Decls`, and `Decl`, as previously seen in Section 2.3.

```
data Scope : Set
variable
  sc sc' : Scope

-- Declarations in a scope.
data Decls (sc : Scope) : Set
variable
  ds ds' : Decls sc

data Decl (sc : Scope) : Set
```

3.2.2 Private blocks

Types and relations used for private blocks are as seen in Section 2.5, with a few more generalized variables for convenience.

```
-- Whether a declaration is accessible from outside its scope
data Access : Set where
  publ : Access
  priv : Access
variable acc : Access

-- The level of access to private declarations we have
-- at a certain point
data Clearance : Set where
  publ : Clearance
  priv : Clearance
variable cl cl' : Clearance

-- Can we see a declaration marked with Access
-- when having a Clearance?
_canSee_ : Clearance → Access → Set
priv canSee _ = ⊤
publ canSee publ = ⊤
publ canSee priv = ⊥
```

3.2.3 Name kinds

Until now, all definitions were identical to the ones in Chapter 2. With the `NameKind` definition, we start to diverge. The full name kind is more complex than the one seen in Section 2.6; there is a hierarchy of four different name kinds that we need to distinguish:

- Module names, carrying the contents of the module and replacing `pointedDs` as used in Section 2.3.1.
- Constructor names, the only kind of name supporting overloading, as we will explain in Section 3.2.4.
- Function clause names, used in mutual declarations to ensure that only function clauses are tied together, as we show in Section 3.2.6.2.
- Other names, that we do not need to treat specially.

Furthermore, a name kind can carry a kind-specific piece of information about the symbol that is of interest to the scope checker. For a module, this is its contents, since they are necessary for accessing other names within that module.

Since new declarations can only refer to previously declared names, we start with the bottom of the hierarchy, that is, the more specific name kinds. These are `funClauseName` for function clauses and `otherName`, a catch all for everything not covered by other name kinds.

```
data NotConNameKind : Set where
  funClauseName : NotConNameKind
  otherName : NotConNameKind -- Everything else
variable ncnk : NotConNameKind
```

We proceed with another layer of the hierarchy, that includes `conName` for constructors, and `notConName` that encompasses the previous definition (`NotConName`).

```
-- We need to distinguish the two because of overloading
data NotModuleNameKind : Set where
  notConName : NotConNameKind → NotModuleNameKind
  conName : NotModuleNameKind
variable nmnk : NotModuleNameKind
```

Finally we define the top level of the hierarchy, `NameKind`, that distinguishes between modules and everything else. The constructor `moduleName` is the only one where we attach a piece of data, namely the declarations inside the pointed module. We do not attach any data to the other constructors, but we could for example attach type information if it turned out to be useful in other compilation phases.

```
data NameKind : Set where
  moduleName : Decls sc → NameKind
  notModuleName : NotModuleNameKind → NameKind
variable nk : NameKind
```

We also add a few *pattern synonyms* to avoid repeating the entire hierarchy of constructors when referring to a specific kind.

```
-- We define shorthands for specific name kinds
pattern ..conName = notModuleName conName
pattern ..notConName n = notModuleName (notConName n)
pattern ..funClauseName = notModuleName (notConName funClauseName)
pattern ..otherName = notModuleName (notConName otherName)
```

Pattern synonyms[1] are aliases that can be used as patterns and therefore can appear in parameters. For example, having defined the patterns above, the following function left-hand-sides are equivalent:

```
f ..conName =
f (notModuleName conName) =
```

3.2.4 Overloading

Next, we declare the higher level scoped names. This includes the usual `SName`, augmented with both `Clearance` and `NameKind`, and a new type of name that we will call simply `Name`. It represents a possibly ambiguous name, and allows us to express the constructor overloading feature of Agda. Agda allows ambiguous names to pass the scope checking phase for further disambiguation by the type checker, but only for constructor names. The constructors `moduleName` and `notConName` do not point to constructors, so they only have one scoped name parameter, while `conNames` points to constructors, so it takes a non-empty list of scoped names to support overloading.

```
-- A well-scoped name in a scope.
-- The name exists in scope, is accessible by a Clearance,
-- and carries a NameKind.
data SName : Scope → Clearance → C.QName → NameKind → Set

data Name : Scope → Clearance → C.QName → NameKind → Set
  where
  moduleName : SName sc cl xs (moduleName ds)
             → Name sc cl xs (moduleName ds)
  notConName : SName sc cl xs (..notConName ncnk)
             → Name sc cl xs (..notConName ncnk)
  conNames : List⁺ (SName sc cl xs ..conName)
           → Name sc cl xs ..conName
```

Note: `List⁺` is a *non-empty list*, as implemented in the Agda standard library [7].

3.2.5 Scopes

Now we define blocks, preceded by a few forward declarations of binders present in the Agda language (we will complete their definition in Section 3.2.6.1).

[1] More information about pattern synonyms is available in the Agda documentation [8] at language/pattern-synonyms.html

- `TypedBinding` is a binder of the form `(a : A)` or `{a : A}`, where `a` is the bound variable and `A` is a type, as used for example in Π types.
- `LamBinding` is a superset of `TypedBinding` used in lambdas where the type can be omitted.
- `Telescope` is a list of `TypedBinding`s such as `(a : A) (b : B)`, again as used in Π types.
- `LamBindings` are the parameters of *parametrized datatypes*[2].

```
data TypedBinding (sc : Scope) : Set
variable tb : TypedBinding sc
data LamBinding (sc : Scope) : Set
variable lb : LamBinding sc
data Telescope (sc : Scope) : Set
variable tel : Telescope sc
data LamBindings (sc : Scope) : Set
variable lbs : LamBindings sc
```

The first two constructors of `Block` can be recognized from Section 2.6, except for the parameter of `bLam` that is now a `LamBinding`. Lambda expressions in Agda are actually more complex than a simple binding, as we will show in Section 3.2.6.1. The new constructors take just their corresponding binder, which can be as little as a single `Name` representing a datatype being defined in the case of `bThisData`.

```
-- A block is a layer of a scope
data Block (sc : Scope) : Set where
  bDecls : (ds : Decls sc) → Block sc
  bLam : (arg : LamBinding sc) → Block sc
  bLbs : (args : LamBindings sc) → Block sc
  bTel : (tel : Telescope sc) → Block sc
  bTb : (tb : TypedBinding sc) → Block sc
  bThisData : C.Name → Block sc
  bLhs : C.Name → Block sc
```

The body of `Scope`, though, remains a snoc-list of blocks.

```
-- A scope is a snoc list of blocks
data Scope where
  ϵ     : Scope
  _▷_ : (sc : Scope) → Block sc → Scope
infixl 20 _▷_
```

[2]More information about parametrized datatypes is available in the Agda documentation [8] at data-types.html#parametrized-datatypes

3.2.6 Syntactic elements

In this section we define syntactic elements such as modules and functions. They can roughly be divided into expressions and declarations.

3.2.6.1 Expressions

The expression definitions are highly recursive. For example, an expression can contain a typed λ-abstraction, whose type is an expression itself. Consequently, we need to forward-declare Expr.

```
data Expr (sc : Scope) : Set
variable ty : Expr sc
```

Then, we define the bodies of various bindings. It is noteworthy that in telescopes and lambda bindings, which are both chains of multiple bindings, the scope is extended with every item, so that they can be referenced in successive items.

```
-- Typed bindings, for example (x y : T)
data TypedBinding sc where
  tBind : List C.Name -- Bound names (x and y)
        → Expr sc -- Type (T)
        → TypedBinding sc

-- Lambda bindings
data LamBinding sc where
  domainFree : C.Name -- Just a name
             → LamBinding sc
  domainFull : TypedBinding sc -- Name(s) with type
             → LamBinding sc

-- In a telescope, each element binds a name
-- that can be used in the rest
data Telescope sc where
  ϵ   : Telescope sc
  _▷_ : (tel : Telescope sc) -- Previous items
      → TypedBinding (sc ▷ bTel tel)
      → Telescope sc

-- The same applies for LamBindings
data LamBindings sc where
  ϵ   : LamBindings sc
  _▷_ : (lbs : LamBindings sc) -- Previous items
      → LamBinding (sc ▷ bLbs lbs)
      → LamBindings sc
```

In Expr itself, we apply Section 2.2 and Section 2.6. Constructor var uses Name to allow constructor overloading, it sets the clearance to priv and the name kind to `notModule`, meaning anything but a module. Constructors `lam`, `pi`, and `let'` (respectively λ-abstractions, Π-types, and let clauses) extend the scope of their inner expression with the respective block.

```
data Expr sc where
  universe : ℕ → Expr sc -- Universes (Set n)
  var : (xs : C.QName)
      → Name sc priv xs (notModuleName nmnk)
      → Expr sc
  app : (e₁ : Expr sc)
      → (e₂ : Expr sc)
      → Expr sc
  lam : (arg : LamBinding sc)
      → (e : Expr (sc ▷ bLam arg))
      → Expr sc
  lit : C.Literal
      → Expr sc
  fun : (e₁ : Expr sc)
      → (e₂ : Expr sc)
      → Expr sc
  underscore : Expr sc
  pi : (arg : TypedBinding sc)
     → (ty : Expr (sc ▷ bTb arg))
     → Expr sc
  let' : (ds : Decls sc)
       → Expr (sc ▷ bDecls ds)
       → Expr sc
variable expr : Expr sc
```

3.2.6.2 Declarations

We define `Interface`, as seen in Section 2.4. There are two changes in `InterfaceEntry`. First, we substitute the pointed declarations with a name kind, that as we have shown in Section 2.6 contains the pointed declarations in case the entry refers to a module. Second, the inner name is now a fully-fledged `Name`.

```
-- An interface entry defines a mapping between
-- the exported name, and the Name of the internal name
record InterfaceEntry (ds : Decls sc) : Set where
  inductive
  constructor interfaceEntry
  field
    exportedName : C.QName
    innerQName : C.QName
    innerNameKind : NameKind
    innerName : Name (sc ▷ bDecls ds) publ innerQName innerNameKind
```

```
-- A module interface is a list of well-scoped exported names
-- (entries)
Interface : Decls sc → Set
Interface ds = List (InterfaceEntry ds)
```

Before defining declarations, we forward-declare `Mutual` and `FunClause`, used for mutual declarations (Section 2.7). We also define data constructors, composed of a name and a type.

```
-- Forward declarations for mutual
Mutual : (sc : Scope) → Decls sc → Set
data FunClause (sc : Scope) : Set

data Constructor (sc : Scope) : Set where
  constructor' : (x : C.Name) → (ty : Expr sc) → Constructor sc
```

We proceed with the definition of `Decl`. Among its constructors, we can find familiar ones such as `modl` and `opn`. They are all augmented with access and name kind information.

Among the new constructors, `data'` is interesting: it is composed of two kinds of bindings and a list of constructors, and each of those elements is parametrized by a scope extended with all the preceding ones.

```
-- A well-scoped declaration is one of
--
--   * A module definition.
--   * Importing the declarations of another module via `open`.
data Decl sc where
  modl : (acc : Access) (x : C.Name) (modArgs : Telescope sc)
       → (ds : Decls (sc ▷ bTel modArgs)) → Decl sc
  opn : ∀{mScope}
      → (mName : C.QName) → (mDecls : Decls mScope)
      → (m : SName sc priv mName (moduleName mDecls))
      → (acc : Access)
      → (iface : Interface mDecls)
      → Decl sc
  axiom : (x : C.Name) → (ty : Expr sc) → Decl sc
  data' : (x : C.Name)
        → (args : LamBindings sc)
        → (ty : Expr (sc ▷ bLbs args))
        → (constructors
             : List (Constructor
                 (sc ▷ bLbs args ▷ bThisData x)))
        → Decl sc
  typeSig : (x : C.Name) → (ty : Expr sc) → Decl sc
  funClause : (x : C.Name) → FunClause sc → Decl sc
```

```
    mutual' : Mutual sc ds → Decl sc
  variable
    d d' : Decl sc
```

The Decls type is the same as defined in Section 2.6.

```
  data Decls sc where
    ϵ    : Decls sc
    _▷_ : (ds : Decls sc) → Decl (sc ▷ bDecls ds) → Decls sc
```

Mutual declarations are as defined in Section 2.7.

```
  data RHS (sc : Scope) : Set where
    absurdRhs : RHS sc
    rHS : (e : Expr sc) → RHS sc

  data FunClause sc where
    funClause : (lhs : C.Name)
              → (whereClause : Decls (sc ▷ bLhs lhs))
              → RHS (sc ▷ bLhs lhs ▷ bDecls whereClause)
              → FunClause sc

  data Clause (sc : Scope) : Decl sc' → Set where
    funClause : SName sc priv (C.qName x) ..funClauseName
              → Clause sc (typeSig x ty)

  Clauses : (sc : Scope) → Decl sc' → Set
  Clauses sc d = List (Clause sc d)

  data Deco (topScope : Scope) : Decls sc → Set where
    ϵ : Deco topScope {sc} ϵ
    _▷[_,_] : Deco topScope {sc} ds
            → (d : Decl (sc ▷ bDecls ds)) → Clauses topScope d
            → Deco topScope (ds ▷ d)
  variable deco : Deco sc ds

  Mutual sc ds = Deco (sc ▷ bDecls ds) ds
```

3.2.7 Well-scoped names

Finally we define well-scoped names. We begin with a forward declaration of DName. We can see it has both a Clearance and a NameKind.

The DsName type is again very similar as its previous definitions: it propagates everything and adds the declarations to the scope.

```
-- Forward declaration for DName
-- A well-scoped name (of a module) in a declaration.
-- The same considerations as for SName apply.
data DName (sc : Scope)
  : Clearance → Decl sc → C.QName → NameKind → Set

-- A well-scoped name (of a module) in a list of declarations.
-- The same considerations as for SName apply.
data DsName (sc : Scope)
  : Clearance → Decls sc → C.QName → NameKind → Set where
  here : DName (sc ▷ bDecls ds) cl d xs nk
       → DsName sc cl (ds ▷ d) xs nk
  there : DsName sc cl ds xs nk
        → DsName sc cl (ds ▷ d) xs nk
```

The well-scoped name in mutual definitions traverses the decorator, again like in Section 2.7.

```
-- A well-scoped name in a mutual block
data MName (sc : Scope)
  : Deco (sc ▷ bDecls ds') ds → C.QName → NameKind → Set where
  here  : ∀{clauses} → DName _ publ d xs nk
        → MName sc (deco ▷[ d , clauses ]) xs nk
  there : ∀{clauses} → MName sc {ds = ds} deco xs nk
        → MName sc (deco ▷[ d , clauses ]) xs nk
```

In DName we can note the use of scope blocks, canSee relation between clearance and access, name kinds, interface entries.

```
data DName sc where
  content : {ds : Decls (sc ▷ bTel tel)}
    → cl canSee acc
    → DName sc cl (modl acc x tel ds) (C.qName x) (moduleName ds)
  inside : {ds : Decls (sc ▷ bTel tel)}
    → cl canSee acc
    → DsName (sc ▷ bTel tel) publ ds xs nk
    → DName sc cl (modl acc x tel ds) (C.qual x xs) nk
  imp : ∀{iface m sn}
    → cl canSee acc
    → interfaceEntry xs ys nk sn ∈ iface
    → DName sc cl (opn xs' ds m acc iface) xs nk
  thisAxiom : DName sc cl (axiom x ty) (C.qName x) ..otherName
  thisTypeSig : DName sc cl (typeSig x ty) (C.qName x) ..otherName
  thisFunClause : ∀{clause}
    → DName sc cl (funClause x clause) (C.qName x) ..funClauseName
  thisData : ∀{constructors}
    → DName sc cl (data' x lbs ty constructors) (C.qName x)
```

```
        ..otherName
    thisDataCon : ∀{constructors dataTy}
      → constructor' x ty ∈ constructors
      → DName sc cl (data' y lbs dataTy constructors) (C.qName x)
        ..conName
    thisMutual : MName sc deco xs nk
      → DName sc cl (mutual' deco) xs nk
```

We define other well-scoped names for various parts of the syntax. All of them follow the same pattern: they are relations between a scope, a syntactic element, and a name, asserting that the name is defined in the syntactic element parametrized by the scope. For example, `TelName sc tel x` asserts that the name `x` is defined somewhere in telescope `tel`.

```
  -- Well-scoped name in a typed binding
  data TBName (sc : Scope) : TypedBinding sc → C.Name → Set
    where
    tbName : ∀{x ns ty}
           → x ∈ ns -- The name is present in the list of bindings
           → TBName sc (tBind ns ty) x

  -- Well-scoped name in a lambda binding
  data LBName (sc : Scope) : LamBinding sc → C.Name → Set where
    -- The name is defined directly
    domainFree : ∀{x} → LBName sc (domainFree x) x
    -- The name is defined in a typed binding
    domainFull : TBName sc tb x → LBName sc (domainFull tb) x

  -- Well-scoped name in a telescope
  data TelName (sc : Scope) : Telescope sc → C.Name → Set where
    -- The name is defined in the last element of the telescope
    here : TBName (sc ▷ bTel tel) tb x → TelName sc (tel ▷ tb) x
    -- The name is defined in the rest of the telescope
    there : TelName sc tel x → TelName sc (tel ▷ tb) x

  -- Well-scoped name in lambda bindings
  data LBsName (sc : Scope) : LamBindings sc → C.Name → Set where
    -- The name is defined in the last lambda binding
    here : LBName (sc ▷ bLbs lbs) lb x → LBsName sc (lbs ▷ lb) x
    -- The name is defined in the rest of the lambda bindings
    there : LBsName sc lbs x → LBsName sc (lbs ▷ lb) x
```

In the end, we define the well-scoped name in a scope, including `BName` like in Section 2.6.

```
-- Well-scoped name in a scope block
data BName (sc : Scope) (cl : Clearance)
  : (block : Block sc) (xs : C.QName) (nk : NameKind) → Set where
  inDecls
    : DsName sc cl ds xs nk
    → BName sc cl (bDecls ds) xs nk
  inLam
    : LBName sc lb x
    → BName sc cl (bLam lb) (C.qName x) ..otherName
  inTb
    : TBName sc tb x
    → BName sc cl (bTb tb) (C.qName x) ..otherName
  inTel
    : TelName sc tel x
    → BName sc cl (bTel tel) (C.qName x) ..otherName
  inLBs
    : LBsName sc lbs x
    → BName sc cl (bLbs lbs) (C.qName x) ..otherName
  inThisData
    : BName sc cl (bThisData x) (C.qName x) ..otherName

-- Well-scoped name in a scope
data SName where
  site : ∀{block} → BName sc cl block xs nk
       → SName (sc ▷ block) cl xs nk
  parent : ∀{block} → SName sc cl xs nk
         → SName (sc ▷ block) cl xs nk
```

3.3 The scope checker

We have developed a scope checker that takes the concrete syntax as input and returns the well-scoped abstract syntax on success. This scope checker works much like a regular scope checker would, except that when names are looked up we are careful to preserve the evidence that names are in-scope under the form of the well-scoped names shown in Chapter 2. We will not explain the scope checker further, since we consider the description of its implementation to be out of the scope of this thesis for a few reasons:

- The thesis focuses on the well-scoped syntax.
- A large part of the scope checker consists in dealing with the details of the concrete syntax, which we did not show either.
- Apart from preserving evidence, no novel techniques are employed.

Nonetheless, the source code of the scope checker is available (Appendix B).

3.3.1 Golden testing

We used *golden testing* to test the scope checker. Golden testing is a testing technique in which a program is run against a set of inputs and its output is recorded in a series of files called *golden files*. The output is manually checked for correctness once, then it is saved for comparison with future test runs. This kind of test is useful during development as it catches any unwanted change in behavior. We used the *goldplate* [9] golden testing tool to automate the tests, since it is well adapted to our use case.

We wrote a number of test cases based on the subset of syntax we covered and were able to check that, of those, the programs accepted by the scope checker and transformed into well-scoped syntax are the same that are accepted by the Agda compiler and vice versa.

We recorded the scope errors generated by the scope checker for ill-scoped inputs or simply an “OK” string for well-scoped inputs in the golden files.

3.4 Limitations

Because the Agda syntax is so large, this project has a few limitations.

3.4.1 Partial coverage of the Agda syntax

As mentioned, we only covered a subset of Agda. It is large enough to cover all unique classes of constructs, but it is nonetheless incomplete. Therefore, using this scope checker on the wider corpus of Agda code is as of writing infeasible. We believe, though, that we have laid the foundation to accomplish this goal.

3.4.2 No proof of uniqueness of resolved names

While the well-scoped syntax contains guarantees that all names correctly resolve to a declaration, there is no guarantee that this resolution is unique. Absence of shadowing can still be checked (and it is in our scope checker implementation), but the proof of its completion is not retained since it would not be as useful as a proof of reachability. For example, a double declaration of a module followed by an `open` statement referring to either the first or the second declaration is ill-scoped Agda code, since shadowing of modules is not allowed. While the scope checker itself would reject it, the well-scoped syntax can still express that program. In Section 2.2.1 we explained how this can happen. This is a minor problem; the main goal of the well-scoped syntax is to provide a basis for further scope-safe compilation passes, and that objective is attained.

3.5 Alternative approaches

The way this scope checker was built is not the only possibility.

An alternative approach we tried before settling on the well-scoped abstract syntax explained above, was to base the well-scoped syntax on the concrete one, only adding proofs to the existing structure. We briefly mentioned this in Section 2.2.

While this technique would have come with its advantages, it clashes with the complexity of the concrete syntax of Agda. Import lists are exemplary of that. In the well-scoped syntax we chose to represent them as mappings between internal and external names. In the concrete syntax they are represented as three lists: the *using* list, the *hiding* list, and the *renaming* list. A predicate stating that a name is defined in this last kind of syntax would have to state:

- That the name is not present in the hiding list.
- That the name is present in the using list and defined in the opened module, or
- That the name is present in the renaming list, and the original name is defined in the opened module.

Furthermore, some combinations of these lists are invalid. For example a name cannot be both present in the using list and in the renaming list.

In conclusion, we consider our choice of separating the well-scoped syntax from the concrete syntax to be more suited to our objective.

4

Conclusion

With this project, we demonstrated it is possible to write a well-scoped by construction syntax for a practical and complex language such as Agda.

We briefly introduced Agda and the concept of scope checking. We built well-scoped syntaxes for fragments of Agda, each demonstrating a different feature or technique. We integrated the aforementioned syntax fragments in a single well-scoped syntax, covering a sufficient portion of the Agda syntax. Finally, we built a scope checker that transforms concrete syntax into well-scoped abstract syntax.

4.1 Future work

There are many ways in which this work could be expanded and built upon.

The Agda to Haskell bindings to the Agda parser can be extracted into a separate library, which can be useful for working in Agda on other aspects of Agda.

As mentioned in Section 3.4.1, the well-scoped syntax can be expanded until the entire Agda language is covered. At that point, complete comparisons with the Agda scope checker can be performed, potentially uncovering bugs and inconsistencies.

The testsuite (Section 3.3.1) could be expanded, both by adding more test cases and by improving the detail of existing test cases. Instead of simply checking that the code is scope checked successfully, we can implement pretty-printing of the well-scoped syntax, and include the pretty-printed syntax in the golden files. An open question then is how to represent well-scoped names so that they can be accurately captured in the golden files. Alternatively, the tests could be written directly in Agda, where equality checks can be performed directly on the syntax trees instead of their serialization.

More proofs can be built upon the well-scoped syntax. For example, it could be proved that turning it back into concrete syntax causes no loss of information. Proving the correct handling of shadowing and ambiguous names as explained in Section 2.2.1 and Section 3.4.2 is another worthwhile goal.

Finally, since this project is a first small step towards a fully verified Agda compiler, the most ambitious and important effort would be to work on other compilation steps – such as type checking and code generation – while using this scope checker as basis, taking advantage of the proofs embedded in the well-scoped syntax.

Bibliography

[1] U. Norell, “Towards a practical programming language based on dependent type theory,” *PhD thesis. Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden*, 2007, ISBN 978-91-7291-996-9. [Online]. Available: https://www.cse.chalmers.se/~ulfn/papers/thesis.pdf

[2] “The Agda website,” accessed: 2022-08-20. [Online]. Available: https://wiki.portal.chalmers.se/agda

[3] X. Leroy, “Formal verification of a realistic compiler,” *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[4] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens, “CakeML: A verified implementation of ML,” in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’14. New York, NY, USA: Association for Computing Machinery, 2014, p. 179–191, ISBN 978-14-5032-544-8. [Online]. Available: https://doi.org/10.1145/2535838.2535841

[5] H. van Antwerpen, C. Bach Poulsen, A. Rouvoet, and E. Visser, “Scopes as types,” *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–30, 2018.

[6] A. Rouvoet, H. Van Antwerpen, C. Bach Poulsen, R. Krebbers, and E. Visser, “Knowing when to ask: sound scheduling of name resolution in type checkers derived from declarative specifications,” *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–28, 2020.

[7] “The Agda standard library,” accessed: 2022-08-20. [Online]. Available: https://github.com/agda/agda-stdlib

[8] “The Agda documentation,” accessed: 2022-08-20. [Online]. Available: https://agda.readthedocs.io/en/v2.6.2.2

[9] “Goldplate: A lightweight golden test runner,” accessed: 2023-05-16. [Online]. Available: https://github.com/fugue/goldplate

[10] "Introduction to the EUPL licence," accessed: 2023-05-26. [Online]. Available: https://joinup.ec.europa.eu/collection/eupl/introduction-eupl-licence

A

Appendix 1 – Full code listings

This appendix contains the full code listings from which snippets present thorough this document are cut. The code is also available at `https://git.sr.ht/~fgaz/master-thesis`, under the *code* directory.

Concrete.agda

```
module Concrete where

open import Data.List using (_::_; [_])
open import Data.List.Membership.Propositional using (_∈_)
open import Data.List.Relation.Unary.Any using (here)
open import Relation.Binary.PropositionalEquality using (refl)

module C where
  -- An atomic name opaque to the scope checker
  postulate Name : Set
-- Continuing module C
  data QName : Set where
    -- Construction from unqualified name
    qName : Name → QName
    -- Name qualification
    qual : Name → QName → QName

variable x y : C.Name
variable xs ys : C.QName

prop1 : (x : C.Name) → (y : C.Name) → x ∈ x :: [ y ]
prop1 x y = here refl

prop2 : {x : C.Name} → {y : C.Name} → x ∈ x :: [ y ]
prop2 = here refl

prop3 : x ∈ x :: [ y ]
prop3 = here refl
```

LambdaCalculusBase.agda

```
module LambdaCalculusBase where

open import Concrete

data Expr : Set where
  -- λv.e
  abs : C.Name → Expr → Expr
  -- e₁ e₂
  app : Expr → Expr → Expr
  -- v
  var : C.Name → Expr
```

LambdaCalculusWellScoped.agda

```
module LambdaCalculusWellScoped where

-- Isomorphic to ℕ⁰
data Scope : Set where
  -- Empty scope, isomorphic to 0
  ϵ : Scope
  -- Expansion of scope, isomorphic to succ
  _▷ : Scope → Scope
variable sc : Scope

-- Isomorphic to {n ∈ ℕ⁺ . n ≤ sc}
data SName : Scope → Set where
  here : SName (sc ▷)
  there : SName sc → SName (sc ▷)

data Expr (sc : Scope) : Set where
  abs : Expr (sc ▷) → Expr sc
  app : Expr sc → Expr sc → Expr sc
  var : SName sc → Expr sc
```

LambdaCalculusWithNames.agda

```
module LambdaCalculusWithNames where

open import Concrete

data Scope : Set where
  -- Empty scope
  ϵ   : Scope
  -- "Snoc" (opposite of cons)
  _▷_ : Scope → C.Name → Scope
variable sc : Scope

data SName : Scope → C.Name → Set where
  here : SName (sc ▷ x) x
  there : SName sc x → SName (sc ▷ y) x

data Expr (sc : Scope) : Set where
  abs : (x : C.Name) → Expr (sc ▷ x) → Expr sc
  app : Expr sc → Expr sc → Expr sc
  var : (x : C.Name) → SName sc x → Expr sc

shadowing : C.Name → Expr ϵ
--                  +-----------------+
--                  V                 |
shadowing x = abs x (abs x (var x (there here)))
```

DefRefCalculus.agda

```
module DefRefCalculus where

open import Concrete

data Scope : Set where
  ϵ   : Scope
  _▷_ : Scope → C.Name → Scope
variable sc : Scope

data SName : Scope → C.Name → Set where
  here : SName (sc ▷ x) x
  there : SName sc x → SName (sc ▷ y) x

data Decls (sc : Scope) : Set where
  ϵ   : Decls sc
  def : (x : C.Name) → Decls (sc ▷ x) → Decls sc
  ref : (x : C.Name) → SName sc x → Decls sc → Decls sc

-- Syntax of this program:
--   define x
--   reference x
example : C.Name → Decls ϵ
example x = def x (ref x here ϵ)
```

ModulesExample.agda

```
module ModulesExample where

module Module where
  module InsideModule where

-- References to InsideModule must be
-- qualified with Module
module OutsideModule = Module.InsideModule

module M where
  module A where
  -- Modules can nest arbitrarily
  module N where
    module B where

-- Example assignments of nested modules
module A' = M.A
module B' = M.N.B
module N' = M.N
module B'' = N'.B
```

ModulesConcrete.agda

```
module ModulesConcrete where

open import Concrete

open import Data.List using (List)

data Decl : Set where
  modl
    : C.Name -- Name of the module
    → List Decl -- Inner declarations
    → Decl
  modlAssignment
    : C.Name -- Left hand side
    → C.QName -- Right hand side
    → Decl
```

ModulesWellScoped.agda

```
module ModulesWellScoped where

open import Concrete

data Scope : Set
variable sc : Scope

data Decl (sc : Scope) : Set
variable d : Decl sc
data Decls (sc : Scope) : Set
variable ds : Decls sc

data DName : (sc : Scope) → Decl sc → C.QName → Set
data DsName : (sc : Scope) → Decls sc → C.QName → Set
data SName : Scope → C.QName → Set

data Scope where
  -- Empty scope
  ϵ : Scope
  -- Scope expansion
  _▷_
    : (sc : Scope) -- Upper scope
    → Decls sc
    → Scope

data Decl sc where
  modl : C.Name → Decls sc → Decl sc
  modlReference : SName sc ys → Decl sc

data Decls sc where
  -- Empty list
  ϵ : Decls sc
  -- "Snoc"
  _▷_
    : (ds : Decls sc) -- Previous declarations
    -- Last declaration. Previous declarations are in scope
    → Decl (sc ▷ ds)
    → Decls sc

data DName where
  -- It is this module
  thisModule : {ds : Decls sc} → DName sc (modl x ds) (C.qName x)
  -- It is inside this module
  inside
    : {ds : Decls sc} -- Declarations within the module
```

```
    -- The name is defined in one of the declarations
    → DsName sc ds xs
    → DName sc (modl x ds) (C.qual x xs)
  -- There is no constructor for modlReference
  -- because it does not define names

data DsName where
  -- It is in this last declaration (d)
  here : DName (sc ▷ ds) d xs → DsName sc (ds ▷ d) xs
  -- It is in one of the previous declarations (ds)
  there : DsName sc ds xs → DsName sc (ds ▷ d) xs

data SName where
  -- It is in this module
  here : DsName sc ds xs → SName (sc ▷ ds) xs
  -- It is in one of the upper modules
  there : SName sc xs → SName (sc ▷ ds) xs
```

ModuleAssignment.agda

```
module ModuleAssignment where

open import Concrete

data Scope : Set
variable sc sc' sc'' : Scope

data Decl (sc : Scope) : Set
variable d : Decl sc
data Decls (sc : Scope) : Set
variable ds : Decls sc
variable pointedDs pointedDs' : Decls sc

data DName : (sc : Scope)
           → Decl sc
           → C.QName
           → Decls sc' -- Body of the the pointed module
           → Set
data DsName : (sc : Scope)
            → Decls sc
            → C.QName
            → Decls sc'
            → Set
data SName : Scope
           → C.QName
           → Decls sc'
           → Set
variable sn : SName sc xs pointedDs

data Scope where
  ϵ   : Scope
  _▷_ : (sc : Scope) → Decls sc → Scope

data Decl sc where
  modl : (x : C.Name) (ds : Decls sc) → Decl sc
  modlAssignment : (x : C.Name) (sn : SName sc ys pointedDs)
                 → Decl sc

data Decls sc where
  ϵ   : Decls sc
  _▷_ : (ds : Decls sc) → Decl (sc ▷ ds) → Decls sc

data DName where
  -- The pointed decls are built
  thisModule : {ds : Decls sc}
```

```
    → DName sc (modl x pointedDs) (C.qName x) pointedDs
  -- The pointed decls are propagated
  inside : {ds : Decls sc}
    → DsName sc ds xs pointedDs
    → DName sc (modl x ds) (C.qual x xs) pointedDs
  -- The pointed decls are connected
  thisAssignment : {sn : SName sc ys pointedDs}
    → DName sc (modlAssignment x sn) (C.qName x) pointedDs
  insideAssignment : {sn : SName sc ys pointedDs}
    → DsName sc' pointedDs xs pointedDs'
    → DName sc (modlAssignment x sn) (C.qual x xs) pointedDs'

data DsName where
  here : DName (sc ▷ ds) d xs pointedDs
       → DsName sc (ds ▷ d) xs pointedDs
  there : DsName sc ds xs pointedDs
        → DsName sc (ds ▷ d) xs pointedDs

data SName where
  here : DsName sc ds xs pointedDs
       → SName (sc ▷ ds) xs pointedDs
  there : SName sc xs pointedDs
        → SName (sc ▷ ds) xs pointedDs
```

Open.agda

```
module Open where

open import Concrete
open import ModuleAssignment hiding (Decl; DName)
open import Data.List using (List)
open import Data.List.Membership.Propositional using (_∈_)

-- An interface entry defines a mapping between the exported
-- name, and the Name of the internal name
record InterfaceEntry (ds : Decls sc) : Set where
  inductive
  constructor interfaceEntry
  field
    exportedName : C.QName
    innerQName : C.QName
    innerPointedSc : Scope
    innerPointedDs : Decls innerPointedSc
    innerDsName : DsName sc ds innerQName innerPointedDs

-- A module interface is a list of well-scoped exported names
-- (entries)
Interface : Decls sc → Set
Interface ds = List (InterfaceEntry ds)

data Decl (sc : Scope) : Set where
  opn : (m : SName sc xs pointedDs)
      → (iface : Interface pointedDs)
      → Decl sc

data DName (sc : Scope) : Decl sc → C.QName → Decls sc' → Set
  where
  -- It is imported from another module
  imp : ∀{mName} → {m : SName _ mName ds}
      → {iface : Interface ds}
      → {dsn : DsName sc' ds ys pointedDs}
      → interfaceEntry xs ys sc'' pointedDs dsn ∈ iface
      → DName sc (opn m iface) xs pointedDs
```

Access.agda

```
module Access where
open import Data.Unit using (⊤)
open import Data.Empty using (⊥)

open import Concrete


-- Whether a declaration is accessible from outside its scope
data Access : Set where
  -- It was not defined in a private block,
  -- and is publicly accessible
  publ : Access
  -- It was defined in a private block,
  -- and is not publicly accessible
  priv : Access
variable acc : Access

-- The level of access to private declarations we have
-- at a certain point
data Clearance : Set where
  -- We can access only public declarations
  publ : Clearance
  -- We can access public and private declarations
  priv : Clearance
variable cl : Clearance

-- Can we see a declaration marked with Access
-- when having a Clearance?
_canSee_ : Clearance → Access → Set
-- A priv Clearance can see everything
priv canSee _ = ⊤
-- A publ Clearance can only see publ Access
publ canSee publ = ⊤
publ canSee priv = ⊥

data Scope : Set
variable sc : Scope

data Decl (sc : Scope) : Set
variable d : Decl sc
data Decls (sc : Scope) : Set
variable ds : Decls sc

-- All scoped names are parametrised by Clearance
data DName : (sc : Scope) → Clearance → Decl sc → C.QName
```

```
           → Set
data DsName : (sc : Scope) → Clearance → Decls sc → C.QName
            → Set
data SName : Scope → Clearance → C.QName → Set

data Scope where
  ϵ : Scope
  _▷_
    : (sc : Scope)
    → Decls sc
    → Scope

data Decl sc where
  modl : Access → C.Name → Decls sc → Decl sc
  -- We start with Clearance set to priv
  modlReference : SName sc priv ys → Decl sc

data Decls sc where
  ϵ : Decls sc
  _▷_
    : (ds : Decls sc)
    → Decl (sc ▷ ds)
    → Decls sc

data DName where
  thisModule
    : {ds : Decls sc}
    -- We ensure the declaration is accessible
    → cl canSee acc
    → DName sc cl (modl acc x ds) (C.qName x)
  inside
    : {ds : Decls sc}
    -- We ensure the declaration is accessible
    → cl canSee acc
    -- When descending inside a module, Clearance switches
    -- to publ
    → DsName sc publ ds xs
    → DName sc cl (modl acc x ds) (C.qual x xs)

data DsName where
  here : DName (sc ▷ ds) cl d xs → DsName sc cl (ds ▷ d) xs
  there : DsName sc cl ds xs → DsName sc cl (ds ▷ d) xs

data SName where
  here : DsName sc cl ds xs → SName (sc ▷ ds) cl xs
  there : SName sc cl xs → SName (sc ▷ ds) cl xs
```

ModulesAndExpressions.agda

```
module ModulesAndExpressions where

open import Concrete

-- The kind of definition a scoped name refers to
data NameKind : Set where
  -- It must be a module
  moduleName : NameKind
  -- It must be a name that can be used in an expression
  exprName : NameKind
variable nk : NameKind

data Scope : Set
variable sc : Scope

-- A block is a layer of a scope
data Block (sc : Scope) : Set
variable block : Block sc

data Expr (sc : Scope) : Set
data Decl (sc : Scope) : Set
variable d : Decl sc
data Decls (sc : Scope) : Set
variable ds : Decls sc

-- Every scoped name is parametrized by a specific name kind
data DName
  : (sc : Scope) → Decl sc → C.QName → NameKind → Set
data DsName
  : (sc : Scope) → Decls sc → C.QName → NameKind → Set
-- New
data BName
  : (sc : Scope) → Block sc → C.QName → NameKind → Set
data SName : Scope → C.QName → NameKind → Set

data Block sc where
  -- It is a layer of declarations
  bDecls : Decls sc → Block sc
  -- It is a lambda abstraction
  bLam : C.Name → Block sc

data Scope where
  ϵ : Scope
  _▷_
    : (sc : Scope)
```

```
    -- A scope is now made of blocks
    → Block sc
    → Scope

data Expr sc where
  -- On abstractions, a bLam block is added
  abs : (x : C.Name) → Expr (sc ▷ bLam x) → Expr sc
  app : Expr sc → Expr sc → Expr sc
  -- We use exprName to avoid referring to a module
  var : (xs : C.QName) → SName sc xs exprName → Expr sc

data Decl sc where
  modl : C.Name → Decls sc → Decl sc
  modlReference : SName sc ys moduleName → Decl sc
  expr : C.Name → Expr sc → Decl sc -- New

data Decls sc where
  ε : Decls sc
  _▷_
    : (ds : Decls sc)
    -- To add declarations to the scope, we use bDecls
    → Decl (sc ▷ bDecls ds)
    → Decls sc

data DName where
  -- It is a module, so we set the name kind to moduleName
  thisModule : {ds : Decls sc}
             → DName sc (modl x ds) (C.qName x) moduleName
  -- The name kind is propagated
  inside
    : {ds : Decls sc}
    → DsName sc ds xs nk
    → DName sc (modl x ds) (C.qual x xs) nk
  -- It is this expression, so we set the name kind to exprName
  thisExpr : {e : Expr sc}
           → DName sc (expr x e) (C.qName x) exprName

-- The name kind is propagated
data DsName where
  here : DName (sc ▷ bDecls ds) d xs nk
       → DsName sc (ds ▷ d) xs nk
  there : DsName sc ds xs nk
        → DsName sc (ds ▷ d) xs nk

data BName where
  inDecls : DsName sc ds xs nk
```

```
          → BName sc (bDecls ds) xs nk
  -- The path terminates at the abstraction
  inLam : BName sc (bLam x) (C.qName x) exprName

-- The name kind is propagated
data SName where
  here : BName sc block xs nk
       → SName (sc ▷ block) xs nk
  there : SName sc xs nk
        → SName (sc ▷ block) xs nk
```

InterleavedMutual.agda

```
module InterleavedMutual where

open import Data.List using (List)
open import Concrete

data Scope : Set
variable sc sc' : Scope
data Decls (sc : Scope) : Set
variable
  ds ds' : Decls sc
data Decl (sc : Scope) : Set
variable d : Decl sc
postulate Expr : Scope → Set
variable ty : Expr sc
postulate SName : Scope → C.QName → Set
data DName (sc : Scope) : Decl sc → C.QName → Set

data Block (sc : Scope) : Set where
  bDecls : Decls sc → Block sc
  bLhs : C.Name → Block sc

data Scope where
  ϵ   : Scope
  _▷_ : (sc : Scope) → Block sc → Scope

data Decls sc where
  ϵ   : Decls sc
  _▷_ : (ds : Decls sc) (d : Decl (sc ▷ bDecls ds)) → Decls sc

Mutual : (sc : Scope) → Decls sc → Set

data Decl sc where
  typeSig : (x : C.Name) → (ty : Expr sc) → Decl sc
  funClause : (x : C.Name) → (lhs : C.Name)
            → Expr (sc ▷ bLhs lhs) → Decl sc
  mutual' : Mutual sc ds → Decl sc

data Clause (topScope : Scope) : Decl sc → Set where
  funClause : SName topScope (C.qName x)
            → Clause topScope (typeSig x ty)

Clauses : (topScope : Scope) → Decl sc → Set
Clauses topScope d = List (Clause topScope d)

data Deco (topScope : Scope) : Decls sc → Set where
```

```
  ϵ : Deco topScope {sc} ϵ
  _▷[_,_] : Deco topScope {sc} ds
          → (d : Decl (sc ▷ bDecls ds)) → Clauses topScope d
          → Deco topScope (ds ▷ d)
variable deco : Deco sc ds

Mutual sc ds = Deco (sc ▷ bDecls ds) ds

data MName (sc : Scope) : Deco (sc ▷ bDecls ds') ds → C.QName
                        → Set where
  here  : ∀{clauses} → DName _ d xs
        → MName sc (deco ▷[ d , clauses ]) xs
  there : ∀{clauses} → MName sc {ds = ds} deco xs
        → MName sc (deco ▷[ d , clauses ]) xs

data DName sc where
  thisMutual : MName sc deco xs
    → DName sc (mutual' deco) xs
```

Complete.agda

```
module Complete where

open import Data.Unit using (⊤)
open import Data.Empty using (⊥)
open import Data.List using (List)
open import Data.List.Membership.Propositional using (_∈_)
open import Data.List.NonEmpty using (List⁺)
open import Data.Nat using (ℕ)
open import Data.String using (String)
open import Data.Char using (Char)
open import Data.Word using (Word64)
open import Data.Float using (Float)


module C where
  Name : Set
  Name = String

  data QName : Set where
    qual : Name → QName → QName
    qName : Name → QName

  data Literal : Set where
    litNat : ℕ → Literal
    litWord64 : Word64 → Literal
    litFloat : Float → Literal
    litString : String → Literal
    litChar : Char → Literal

variable x y : C.Name
variable xs ys xs' : C.QName

data Scope : Set
variable
  sc sc' : Scope

-- Declarations in a scope.
data Decls (sc : Scope) : Set
variable
  ds ds' : Decls sc

data Decl (sc : Scope) : Set

-- Whether a declaration is accessible from outside its scope
data Access : Set where
```

```
  publ : Access
  priv : Access
variable acc : Access

-- The level of access to private declarations we have
-- at a certain point
data Clearance : Set where
  publ : Clearance
  priv : Clearance
variable cl cl' : Clearance

-- Can we see a declaration marked with Access
-- when having a Clearance?
_canSee_ : Clearance → Access → Set
priv canSee _ = ⊤
publ canSee publ = ⊤
publ canSee priv = ⊥

data NotConNameKind : Set where
  funClauseName : NotConNameKind
  otherName : NotConNameKind -- Everything else
variable ncnk : NotConNameKind

-- We need to distinguish the two because of overloading
data NotModuleNameKind : Set where
  notConName : NotConNameKind → NotModuleNameKind
  conName : NotModuleNameKind
variable nmnk : NotModuleNameKind

data NameKind : Set where
  moduleName : Decls sc → NameKind
  notModuleName : NotModuleNameKind → NameKind
variable nk : NameKind

-- We define shorthands for specific name kinds
pattern ..conName = notModuleName conName
pattern ..notConName n = notModuleName (notConName n)
pattern ..funClauseName = notModuleName (notConName funClauseName)
pattern ..otherName = notModuleName (notConName otherName)

-- A well-scoped name in a scope.
-- The name exists in scope, is accessible by a Clearance,
-- and carries a NameKind.
data SName : Scope → Clearance → C.QName → NameKind → Set

data Name : Scope → Clearance → C.QName → NameKind → Set
```

```
  where
  moduleName : SName sc cl xs (moduleName ds)
             → Name sc cl xs (moduleName ds)
  notConName : SName sc cl xs (..notConName ncnk)
             → Name sc cl xs (..notConName ncnk)
  conNames : List⁺ (SName sc cl xs ..conName)
           → Name sc cl xs ..conName

data TypedBinding (sc : Scope) : Set
variable tb : TypedBinding sc
data LamBinding (sc : Scope) : Set
variable lb : LamBinding sc
data Telescope (sc : Scope) : Set
variable tel : Telescope sc
data LamBindings (sc : Scope) : Set
variable lbs : LamBindings sc

-- A block is a layer of a scope
data Block (sc : Scope) : Set where
  bDecls : (ds : Decls sc) → Block sc
  bLam : (arg : LamBinding sc) → Block sc
  bLbs : (args : LamBindings sc) → Block sc
  bTel : (tel : Telescope sc) → Block sc
  bTb : (tb : TypedBinding sc) → Block sc
  bThisData : C.Name → Block sc
  bLhs : C.Name → Block sc

-- A scope is a snoc list of blocks
data Scope where
  ϵ     : Scope
  _▷_ : (sc : Scope) → Block sc → Scope
infixl 20 _▷_

data Expr (sc : Scope) : Set
variable ty : Expr sc

-- Typed bindings, for example (x y : T)
data TypedBinding sc where
  tBind : List C.Name -- Bound names (x and y)
        → Expr sc -- Type (T)
        → TypedBinding sc

-- Lambda bindings
data LamBinding sc where
  domainFree : C.Name -- Just a name
             → LamBinding sc
```

```
  domainFull : TypedBinding sc -- Name(s) with type
             → LamBinding sc

-- In a telescope, each element binds a name
-- that can be used in the rest
data Telescope sc where
  ϵ   : Telescope sc
  _▷_ : (tel : Telescope sc) -- Previous items
      → TypedBinding (sc ▷ bTel tel)
      → Telescope sc

-- The same applies for LamBindings
data LamBindings sc where
  ϵ   : LamBindings sc
  _▷_ : (lbs : LamBindings sc) -- Previous items
      → LamBinding (sc ▷ bLbs lbs)
      → LamBindings sc

data Expr sc where
  universe : ℕ → Expr sc -- Universes (Set n)
  var : (xs : C.QName)
      → Name sc priv xs (notModuleName nmnk)
      → Expr sc
  app : (e₁ : Expr sc)
      → (e₂ : Expr sc)
      → Expr sc
  lam : (arg : LamBinding sc)
      → (e : Expr (sc ▷ bLam arg))
      → Expr sc
  lit : C.Literal
      → Expr sc
  fun : (e₁ : Expr sc)
      → (e₂ : Expr sc)
      → Expr sc
  underscore : Expr sc
  pi : (arg : TypedBinding sc)
     → (ty : Expr (sc ▷ bTb arg))
     → Expr sc
  let' : (ds : Decls sc)
       → Expr (sc ▷ bDecls ds)
       → Expr sc
variable expr : Expr sc

-- An interface entry defines a mapping between
-- the exported name, and the Name of the internal name
record InterfaceEntry (ds : Decls sc) : Set where
```

```
  inductive
  constructor interfaceEntry
  field
    exportedName : C.QName
    innerQName : C.QName
    innerNameKind : NameKind
    innerName : Name (sc ▷ bDecls ds) publ innerQName innerNameKind

-- A module interface is a list of well-scoped exported names
-- (entries)
Interface : Decls sc → Set
Interface ds = List (InterfaceEntry ds)

-- Forward declarations for mutual
Mutual : (sc : Scope) → Decls sc → Set
data FunClause (sc : Scope) : Set

data Constructor (sc : Scope) : Set where
  constructor' : (x : C.Name) → (ty : Expr sc) → Constructor sc

-- A well-scoped declaration is one of
--
--    * A module definition.
--    * Importing the declarations of another module via `open`.
data Decl sc where
  modl : (acc : Access) (x : C.Name) (modArgs : Telescope sc)
       → (ds : Decls (sc ▷ bTel modArgs)) → Decl sc
  opn : ∀{mScope}
      → (mName : C.QName) → (mDecls : Decls mScope)
      → (m : SName sc priv mName (moduleName mDecls))
      → (acc : Access)
      → (iface : Interface mDecls)
      → Decl sc
  axiom : (x : C.Name) → (ty : Expr sc) → Decl sc
  data' : (x : C.Name)
        → (args : LamBindings sc)
        → (ty : Expr (sc ▷ bLbs args))
        → (constructors
             : List (Constructor
                 (sc ▷ bLbs args ▷ bThisData x)))
        → Decl sc
  typeSig : (x : C.Name) → (ty : Expr sc) → Decl sc
  funClause : (x : C.Name) → FunClause sc → Decl sc
  mutual' : Mutual sc ds → Decl sc
variable
  d d' : Decl sc
```

```
data Decls sc where
  ϵ    : Decls sc
  _▷_ : (ds : Decls sc) → Decl (sc ▷ bDecls ds) → Decls sc

data RHS (sc : Scope) : Set where
  absurdRhs : RHS sc
  rHS : (e : Expr sc) → RHS sc

data FunClause sc where
  funClause : (lhs : C.Name)
            → (whereClause : Decls (sc ▷ bLhs lhs))
            → RHS (sc ▷ bLhs lhs ▷ bDecls whereClause)
            → FunClause sc

data Clause (sc : Scope) : Decl sc' → Set where
  funClause : SName sc priv (C.qName x) ..funClauseName
            → Clause sc (typeSig x ty)

Clauses : (sc : Scope) → Decl sc' → Set
Clauses sc d = List (Clause sc d)

data Deco (topScope : Scope) : Decls sc → Set where
  ϵ : Deco topScope {sc} ϵ
  _▷[_,_] : Deco topScope {sc} ds
          → (d : Decl (sc ▷ bDecls ds)) → Clauses topScope d
          → Deco topScope (ds ▷ d)
variable deco : Deco sc ds

Mutual sc ds = Deco (sc ▷ bDecls ds) ds

-- Forward declaration for DName
-- A well-scoped name (of a module) in a declaration.
-- The same considerations as for SName apply.
data DName (sc : Scope)
  : Clearance → Decl sc → C.QName → NameKind → Set

-- A well-scoped name (of a module) in a list of declarations.
-- The same considerations as for SName apply.
data DsName (sc : Scope)
  : Clearance → Decls sc → C.QName → NameKind → Set where
  here : DName (sc ▷ bDecls ds) cl d xs nk
       → DsName sc cl (ds ▷ d) xs nk
  there : DsName sc cl ds xs nk
        → DsName sc cl (ds ▷ d) xs nk
```

```
-- A well-scoped name in a mutual block
data MName (sc : Scope)
  : Deco (sc ▷ bDecls ds') ds → C.QName → NameKind → Set where
  here  : ∀{clauses} → DName _ publ d xs nk
        → MName sc (deco ▷[ d , clauses ]) xs nk
  there : ∀{clauses} → MName sc {ds = ds} deco xs nk
        → MName sc (deco ▷[ d , clauses ]) xs nk

data DName sc where
  content : {ds : Decls (sc ▷ bTel tel)}
    → cl canSee acc
    → DName sc cl (modl acc x tel ds) (C.qName x) (moduleName ds)
  inside : {ds : Decls (sc ▷ bTel tel)}
    → cl canSee acc
    → DsName (sc ▷ bTel tel) publ ds xs nk
    → DName sc cl (modl acc x tel ds) (C.qual x xs) nk
  imp : ∀{iface m sn}
    → cl canSee acc
    → interfaceEntry xs ys nk sn ∈ iface
    → DName sc cl (opn xs' ds m acc iface) xs nk
  thisAxiom : DName sc cl (axiom x ty) (C.qName x) ..otherName
  thisTypeSig : DName sc cl (typeSig x ty) (C.qName x) ..otherName
  thisFunClause : ∀{clause}
    → DName sc cl (funClause x clause) (C.qName x) ..funClauseName
  thisData : ∀{constructors}
    → DName sc cl (data' x lbs ty constructors) (C.qName x)
        ..otherName
  thisDataCon : ∀{constructors dataTy}
    → constructor' x ty ∈ constructors
    → DName sc cl (data' y lbs dataTy constructors) (C.qName x)
        ..conName
  thisMutual : MName sc deco xs nk
    → DName sc cl (mutual' deco) xs nk

-- Well-scoped name in a typed binding
data TBName (sc : Scope) : TypedBinding sc → C.Name → Set
  where
  tbName : ∀{x ns ty}
          → x ∈ ns -- The name is present in the list of bindings
          → TBName sc (tBind ns ty) x

-- Well-scoped name in a lambda binding
data LBName (sc : Scope) : LamBinding sc → C.Name → Set where
  -- The name is defined directly
  domainFree : ∀{x} → LBName sc (domainFree x) x
  -- The name is defined in a typed binding
```

```
  domainFull : TBName sc tb x → LBName sc (domainFull tb) x

-- Well-scoped name in a telescope
data TelName (sc : Scope) : Telescope sc → C.Name → Set where
  -- The name is defined in the last element of the telescope
  here : TBName (sc ▷ bTel tel) tb x → TelName sc (tel ▷ tb) x
  -- The name is defined in the rest of the telescope
  there : TelName sc tel x → TelName sc (tel ▷ tb) x

-- Well-scoped name in lambda bindings
data LBsName (sc : Scope) : LamBindings sc → C.Name → Set where
  -- The name is defined in the last lambda binding
  here : LBName (sc ▷ bLbs lbs) lb x → LBsName sc (lbs ▷ lb) x
  -- The name is defined in the rest of the lambda bindings
  there : LBsName sc lbs x → LBsName sc (lbs ▷ lb) x

-- Well-scoped name in a scope block
data BName (sc : Scope) (cl : Clearance)
  : (block : Block sc) (xs : C.QName) (nk : NameKind) → Set where
  inDecls
    : DsName sc cl ds xs nk
    → BName sc cl (bDecls ds) xs nk
  inLam
    : LBName sc lb x
    → BName sc cl (bLam lb) (C.qName x) ..otherName
  inTb
    : TBName sc tb x
    → BName sc cl (bTb tb) (C.qName x) ..otherName
  inTel
    : TelName sc tel x
    → BName sc cl (bTel tel) (C.qName x) ..otherName
  inLBs
    : LBsName sc lbs x
    → BName sc cl (bLbs lbs) (C.qName x) ..otherName
  inThisData
    : BName sc cl (bThisData x) (C.qName x) ..otherName

-- Well-scoped name in a scope
data SName where
  site : ∀{block} → BName sc cl block xs nk
       → SName (sc ▷ block) cl xs nk
  parent : ∀{block} → SName sc cl xs nk
         → SName (sc ▷ block) cl xs nk
```

B

Appendix 2 – Source code of the well-scoped syntax and scope checker

The complete Agda source code of the well-scoped syntax and scope checker is available at `https://git.sr.ht/~fgaz/agda-scope` and it is released under the EUPL license [10].

Additionally, the LaTeX source code of this thesis is available at `https://git.sr.ht/~fgaz/master-thesis`.