Agda User Manual

Release 2.9.0

The Agda Team

Sep 10, 2026

CONTENTS


Agda

CHAPTER

ONE

OVERVIEW

Note

The Agda User Manual is a work-in-progress and is still incomplete. Contributions, additions and corrections to the Agda manual are greatly appreciated. To do so, please open a pull request or issue on the GitHub Agda page.

This is the manual for the Agda programming language, its type checking, compilation and editing system and related resources/tools. The latest PDF version of this manual can be downloaded from GitHub Actions page (instruction on how to find them).

You can find a lot of useful resources on Agda Wiki site, like tutorials, introductions, publications and books. If you're new to Agda, you should make use of the resources on Agda Wiki and chapter *Getting Started* instead of chapter *Language Reference*.

A description of the Agda language is given in chapter *Language Reference*. Guidance on how the Agda editing and compilation system can be used can be found in chapter *Tools*.

CHAPTER

TWO

GETTING STARTED

2.1 What is Agda?

Agda is a dependently typed programming language. It is an extension of Martin-Löf's type theory and is the latest in the tradition of languages developed in the programming logic group at Chalmers. Previous languages in this tradition have been Alf, Alfa, Agda 1, Cayenne. Some related languages of note are Rocq (formerly known as Coq), Epigram, Idris, and Lean.

Because of strong typing and dependent types, Agda can be used as a proof assistant, allowing one to prove mathematical theorems (in a constructive setting) and to run such proofs as algorithms.

2.1.1 Dependent types

Typing for programmers

Type theory is concerned both with programming and logic. We see the type system as a way to express syntactic correctness. A type correct program has a meaning. Lisp is a totally untyped programming language, and so are its derivatives like Scheme. In such languages, if `f` is a function, one can apply it to anything, including itself. This makes it easy to write programs (almost all programs are well formed), but it also makes it easy to write erroneous programs. Programs will raise exceptions or loop forever. And it is very difficult to analyze where the problems are.

Haskell or ML and its derivatives like Standard ML and Caml are typed languages, where functions come with a type expressing what type of arguments the program expects and what the result type is.

Between these two families of languages come languages which may or may not have a typing discipline. Most imperative languages do not come with a rich type system. For example, C is typed, but very loosely (almost everything is an integer or a variant thereof). Moreover, the typing system does not allow the definition of trees or graphs without using pointers.

All these languages are examples of **partial languages**, i.e., the result of computing the value of an expression `e` of type `T` is one of the following:

- the program terminates with a value in the type `T`

- the program `e` does not terminate
- the program raises an exception which has been caused by an incomplete definition – for instance, a function is only defined for positive integers but is applied to a negative integer.

Agda and other languages based on type theory are **total languages** in the sense that a program `e` of type `T` will always terminate with a value in `T`. No runtime error can occur, and no nonterminating programs can be written (unless explicitly requested by the programmer).

Dependent types

Dependent types are introduced by having families of types indexed by objects in another type. For instance, we can define the type `Vec n` of vectors of length `n`. This is a family of types indexed by objects in $\mathbb{N}$ (a type parameterized by natural numbers).

Having dependent types, we must generalize the type of functions and the type of pairs.

The **dependent function space** `(a : A) → B a` is the type of the functions taking an argument `a` in a type `A` and returning a result in `B a`. Here, `A` is a type, and `B` is a family of types indexed by elements in `A`.

For example, we could define the type of `n x m` matrices as a type indexed by two natural numbers. Call this type `Mat n m`. The function `identity`, which takes a natural number `n` as an argument and produces the `n x n` identity matrix, is then a function of type `identity : (n : ℕ) → Mat n n`.

Remark: We could, of course, just specify the `identity` function with the type `ℕ → Mat`, where `Mat` is the type of matrices, but this is not as precise as the dependent version.

The advantage of using dependent types is that it makes it possible to express properties of programs in the typing system. We saw above that it is possible to express the type of square matrices of length `n`. It is also possible to define the type of operations on matrices so that the lengths are correct. For instance, the type of matrix multiplication is

```
∀ {i j k} → Mat i j → Mat j k → Mat i k
```

and the type system can check that a program for matrix multiplication really produces matrices of the correct size. It can also check that matrix multiplication is only applied to matrices, where the number of columns of the first argument is the same as the number of rows in the second argument.

Dependent types and logic

Thanks to the Curry-Howard correspondence, one can express a logical specification using dependent types. For example, using only typing it is possible to define:

- equality on natural numbers,
- properties of arithmetical operations, and
- the type `(n : ℕ) → PrimeFactor n` of functions returning a certified prime factor of a given natural number.

Of course, a program of the above type will be more difficult to write than the corresponding program of type $\mathbb{N} \to \mathbb{N}$ that just computes a prime factor without proof of its properties, namely that it is a factor and prime. However, the extra effort in constructing a `PrimeFactor n` is compensated by the fact that the program is guaranteed to work correctly: it cannot produce something which is not a prime factor.

On a more mathematical level, we can express formulas and prove them using an algorithm. For example, a function of type `(n : ℕ) → PrimeFactor n` is also a proof that every natural number has a prime factor (which is trivial for non-composite numbers).

2.2 Installation

Hint

If you want a sneak peek of Agda without installing it, try the Agda Pad or the Agda Web.

Hint

This documentation is for first-time Agda users. More advanced "do it yourself" instructions can be found in the Agda repo's HACKING.md file.

2.2.1 Step 1: Install Agda

You can get the Agda type-checker in at least 4 ways:

Option 1: From GitHub releases

Pre-built `agda` binaries are available for the following platforms:

- Windows (x86-64)
- Linux (x86-64, statically linked)
- macOS (x86-64)
- macOS (arm64, i.e. Apple M-series)

They can be downloaded from the Agda GitHub releases page (under "Assets").

To install, download the appropriate archive, extract the binary, and place it in a directory listed in your `PATH` environment variable.

Option 2: From source

If you want to work on the Agda compiler itself, or you want to work with the very latest version of Agda, then you can compile it from source from the Github repository.

Both `cabal install`-based and `nix build`-based developer builds are maintained.

Miscellaneous developer information is available in the repository's `HACKING.md`.

Option 3: From package manager

`agda` binaries are distributed by various package managers, but are often out of date. Repology.org maintains a list:

As well as the repositories tracked by repology, an OS-independent binary installation of Agda is provided by the python installer.

Option 4: From text editor

Some text editor extensions (e.g. `banacorn.agda-mode` for VSCode) can install Agda on their own.

2.2.2 Step 2: Configure a text editor

Your choice of text editor matters more in Agda than it does in most other programming languages. This is because Agda code typically uses a lot of unicode symbols, and because you will typically *interact* with Agda through the text editor while writing your program.

Editors with interactive support for Agda include:

- Emacs (*agda-mode*) (most tested)
- Visual Studio Code (banacorn.agda-mode)
- Neovim (Cornelis)
- Vim (agda-vim)

2.2.3 Step 3: Install `agda-stdlib` (optional)

You may want to install Agda's standard library, although Agda will work without it.

You can install this like any other Agda library (see *Library Management*). See the agda-stdlib project's installation instructions for the steps to take to install the latest version.

2.2.4 Step 4: Install `ghc` (optional)

To compile your Agda code to an executable, `agda` calls the Haskell compiler `ghc`, which is not bundled with most Agda distributions. You can install `ghc` spearately with e.g. GHCUp.

No separate programs are called by Agda's *JavaScript backend*.

2.3 Troubleshooting

This section collects tips for fixing common installation errors. Skip it if Agda installed fine.

This list is not comprehensive, see the GitHub issue tracker for more known bugs.

2.3.1 Windows `invalid byte sequence`

If you are installing Agda using Cabal on Windows, depending on your system locale setting, `cabal install Agda` may fail with an error message:

```
hGetContents: invalid argument (invalid byte sequence)
```

If this happens, you can try changing the console code page to UTF-8 using the command:

```
CHCP 65001
```

2.3.2 macOS notarization

`agda` binaries from *GitHub releases* are not notarised by Apple, so will not run on macOS unless the quarantine attribute is removed, e.g. with `xattr -c agda`

2.3.3 Cabal issues

If you chose to install Agda with a Haskell build tool like `cabal` or `stack`, see the "Troubleshooting Cabal" section in `HACKING.md`.

2.4 'Hello world' in Agda

This section contains two minimal Agda programs that can be used to test if you have installed Agda correctly: one for using Agda interactively as a proof assistant, and one for compiling Agda programs to an executable binary. For a more in-depth introduction to using Agda, see *A taste of Agda* or the *list of tutorials*.

2.4.1 Hello, Agda!

Below is is a small 'hello world' program in Agda (defined in a file `hello.agda`).

```
data Greeting : Set where
  hello : Greeting

greet : Greeting
greet = hello
```

This program defines a *data type* called `Greeting` with one constructor `hello`, and a *function definition* `greet` of type `Greeting` that returns `hello`.

To load the Agda file, open it in Emacs and load it by pressing `C-c C-l` (`Ctrl+c` followed by `Ctrl+l`). You should now see that the code is highlighted and there should be a message `*All done*`. If this is the case, congratulations! You have correctly installed Agda and the Agda mode for Emacs. If you also want to compile your Agda programs, continue with the next section.

2.4.2 Hello, World!

Below is a complete executable 'hello world' program in Agda (defined in a file `hello-world.agda`)

```
module hello-world where

open import Agda.Builtin.IO using (IO)
open import Agda.Builtin.Unit using (⊤)
open import Agda.Builtin.String using (String)

postulate putStrLn : String → IO ⊤
{-# FOREIGN GHC import qualified Data.Text as T #-}
{-# COMPILE GHC putStrLn = putStrLn . T.unpack #-}

main : IO ⊤
main = putStrLn "Hello world!"
```

This code is self-contained and has several declarations:

1. Imports of the `IO`, `⊤` and `String` types from the Agda Builtin library.
2. A postulate of the function type `putStrLn`.
3. Two *pragmas* that tell Agda how to compile the function `putStrLn`.
4. A definition of the function `main`.

To compile the Agda file, either open it in Emacs and press `C-c C-x C-c` or run `agda --compile hello-world.agda` from the command line. This will create a binary `hello-world` in the current directory that prints `Hello world!`. To find out more about the `agda` command, use `agda --help`.

Note

As you can see from this example, by default Agda includes only minimal library support through the `Builtin` modules. The Agda Standard Library provides bindings for most commonly used Haskell functions, including `putStrLn`. For a version of this 'hello world' program that uses the standard library, see *Building an Executable Agda Program*.

2.5 A Taste of Agda

The objective of this section is to provide a first glimpse of Agda with some small examples. The first one is a demonstration of dependently typed programming, and the second shows how to use Agda as a proof assistant. Finally, we build a complete program and compile it to an executable program with the GHC and Javascript backends.

2.5.1 Preliminaries

This section assumes you have have installed *installed Agda*, a compatible version of the *standard library*, and *GHC*.

Agda programs are typically developed *interactively*, which means that one can type check code which is not yet complete but contain "holes" which can be filled in later. Editors with support for interactive development of Agda programs include Emacs via the *Emacs mode*, Atom via the agda mode for Atom, Visual Studio Code via the agda mode for VSCode, and Vim via agda-vim.

Hint

If you want a sneak peek of Agda without installing it, try the Agda Pad

Note

In this introduction we use several of Agda's interactive commands to get information from the typechecker and manipulate code with holes. Here is a list of the commands that will be used in this tutorial:

- `C-c C-l`: Load the file and type-check it.
- `C-c C-d`: Deduce the type of a given expression.
- `C-c C-n`: Normalise a given expression.
- `C-c C-,`: Shows the type expected in the current hole, along with the types of any local variables.
- `C-c C-c`: Case split on a given variable.
- `C-c C-SPC`: Replace the hole with a given expression, if it has the correct type.
- `C-c C-r`: Refine the hole by replacing it with a given expression applied to an appropriate number of new holes.
- `C-c C-x C-c` (`C-x C-c` in VS Code): Compile an Agda program.

See *Notation for key combinations* for a full list of interactive commands (keybindings).

2.5.2 Programming With Dependent Types: Vectors

In the code below, we model the notion of *vectors* (in the sense of computer science, not in the mathematical sense) in Agda. Roughly speaking, a vector is a list of objects with a determined length.

```
module hello-world-dep where

open import Data.Nat using (ℕ; zero; suc)

data Vec (A : Set) : ℕ → Set where
  []  : Vec A zero
  _::_ : ∀ {n} (x : A) (xs : Vec A n) → Vec A (suc n)

infixr 5 _::_
```

Paste or type the code above in a new file with name `hello-world-dep.agda`. Load the file (in Emacs `C-c C-l`). This also saves the file. If the agda source code was loaded correctly, you should see that the code is highlighted and see a message ***All done*** .

Note

If a file does not type check Agda will complain. Often the cursor will jump to the position of the error, and the error will (by default) be underlined. Some errors are treated a bit differently, though. If Agda cannot see that a definition is terminating/productive it will highlight it in *light salmon*, and if some meta-variable other than the goals cannot be solved the code will be highlighted in *yellow* (the highlighting may not appear until after you have reloaded the file). In case of the latter kinds of errors you can still work with the file, but Agda will (by default) refuse to import it into another module, and if your functions are not terminating Agda may hang. See *Background highlighting* for a full list of the different background colors used by Agda.

> **Tip**
>
> If you do not like the way Agda syntax or errors are highlighted (if you are colour-blind, for instance), then you can tweak the settings by typing `M-x customize-group RET agda2-highlight RET` in Emacs (after loading an Agda file) and following the instructions.

Agda programs are structured into *modules*. Each Agda file has one *top-level module* whose name must match the name of the file, and zero or more nested modules. Each module contains a list of *declarations*. This example has a single top-level module called `hello-world-dep`, which has three declarations:

1. An `open import` statement that imports the datatype ℕ and its constructors `zero` and `suc` from the module `Data.Nat` of the standard library and brings them into scope,
2. A `data` declaration defining the datatype `Vec` with two constructors: the empty vector constructor `[]` and the *cons* constructor _::_,
3. And finally an `infixr` declaration specifying the *precedence* for the *cons* operation.

> **Tip**
>
> Agda uses Unicode characters in source files (more specifically: the UTF-8 character encoding), such as ℕ, →, and ∷ in this example. Many mathematical symbols can be typed using the corresponding LaTeX command names. To learn how to enter a unicode character, move the cursor over it and enter `M-x describe-char` or `C-u C-x =`. This displays all information on the character, including how to input it with the Agda input method. For example, to input ℕ you can type either `\Bbb{N}` or `\bN`. See *Unicode input* for more details on entering unicode characters.

The datatype `Vec`

Let us start by looking at the first line of the definition of `Vec`:

```
data Vec (A : Set) : ℕ → Set where
```

This line declares a new *datatype* and names it `Vec`. The words `data` and `where` are keywords, while the part `Vec (A : Set) : ℕ → Set` determines the type of `Vec`.

`Vec` is not a single type but rather a *family of types*. This family of types has one *parameter* `A` of type `Set` (which is the *sort* of *small types*, such as ℕ, `Bool`, ...) and one *index* of type ℕ (the type of natural numbers). The parameter `A` represents the type of the objects of the vector. Meanwhile, the index represents the length of the vector, i.e. the number of objects it contains.

Together, this line tells us that, for any concrete type `B : Set` and any natural number `m : ℕ`, we are declaring a new type `Vec B m`, which also belongs to `Set`.

The constructors `[]` and _∷_

Each constructors of a datatype is declared on a separate line and indented with a strictly positive number of spaces (in this case two).

We chose the name `[]` for the first constructor. It represents the empty vector, and its type is `Vec A 0`, i.e. it is a vector of length `0`.

The second constructor is a *mixfix operator* named _∷_ (pronounced *cons*). For any number `n : ℕ`, it takes as input an object of `A` and a vector of length `n`. As output, it produces a vector with length `suc n`, the successor of `n`. The number `n` itself is an *implicit argument* to the constructor _∷_.

The final declaration with keyword `infixr` does not belong to the datatype declaration itself; therefore it is not indented. It establishes the *precedence* of the operator _∷_.

Tip

You can let Agda infer the type of an expression using the 'Deduce type' command (`C-c C-d`). First press `C-c C-d` to open a prompt, enter a term, for instance `3 :: 2 :: 1 :: []`, and press return. Agda infers its type and return the type `Vec ℕ 3`, meaning that the given term is a vector with 3 objects of type ℕ.

Note

Almost any character can be used in an identifier (like α, $\wedge$, or ♠, for example). It is therefore necessary to have spaces between most lexical units. For example `3::2::1::[]` is a valid identifier, so we need to write `3 :: 2 :: 1 :: []` instead to make Agda parse it successfully.

The total function `lookup`

Now that `Vec` is defined, we continue by defining the `lookup` function that given a vector and a position, returns the object of the vector at the given position. In contrast to the `lookup` function we could define in most (non-dependently typed) programming languages, this version of the function is *total*: all calls to it are guaranteed to return a value in finite time, with no possibility for errors.

To define this function, we use the `Fin` datatype from the standard library. `Fin n` is a type with `n` objects: the numbers `0` to `n-1` (in unary notation `zero`, `suc zero`, ...), which we use to model the `n` possible positions in a vector of length `n`.

Now create a new file called `hello-world-dep-lookup.agda` file and type or paste:

```
module hello-world-dep-lookup where

open import Data.Nat using (ℕ)
open import Data.Vec using (Vec; _::_)
open import Data.Fin using (Fin; zero; suc)

variable
  A : Set
  n : ℕ

lookup : Vec A n → Fin n → A
lookup (a :: as) zero = a
lookup (a :: as) (suc i) = lookup as i
```

The `Vec` type that we saw before is actually already in the module `Data.Vec` of the standard library, so we import it instead of copying the previous definition.

We have declared `A` and `n` as *generalizable variables* to avoid the declaration of implicit arguments. This allows us to use `A` and `n` in the type of `lookup` without binding the names explicitly. More explicitly, the full type of `lookup` (which we can get by using `C-c C-d`) is:

```
lookup : {A : Set} {n : ℕ} → Vec A n → Fin n → A
```

Warning

`zero` and `suc` are **not** the constructors of ℕ that we saw before, but rather the constructors of `Fin`. Agda allows overloading of constructor names, and disambiguates between them based on the expected type where they are

used.

The definition of the `lookup` function specifies two cases:

- Either the vector is `a :: as` and the position is `zero`, so we return the first object `a` of the vector.
- Or the vector is `a :: as` and the position is `suc i`, so we recursively look up the object at position `i` in the tail `as` of the vector.

There are no cases for the empty vector `[]`. This is no mistake: Agda can determine from the type of `lookup` that it is impossible to look up an object in the empty vector, since there is no possible index of type `Fin 0`. For more details, see the section on *coverage checking*.

2.5.3 Agda as a Proof Assistant: Proving Associativity of Addition

In this section we state and prove the associativity of addition on the natural numbers in Agda. In contrast to the previous section, we build the code line by line. To follow along with this example in Emacs, reload the file after adding each step by pressing `C-c C-l`.

Statement of associativity

We start by creating a new file named `hello-world-proof.agda`. Paste or type the following code:

```
module hello-world-proof where
```

Now we import the datatype ℕ and the addition operation `_+_`, both defined in the Agda Builtin library.

```
open import Data.Nat using (ℕ; _+_)
```

Next, we import the *propositional equality type* `_≡_` from the module `Relation.Binary.PropositionalEquality`.

```
open import Relation.Binary.PropositionalEquality using (_≡_)
```

Under the Curry-Howard correspondence, the type `x ≡ y` corresponds to the proposition stating that `x` and `y` are equal objects. By writing a function that returns an object of type `x ≡ y`, we are *proving* that the two terms are equal.

Now we can state associativity: given three (possibly different) natural numbers, adding the first to the addition of the second and the third computes to the same value as adding the addition of the first and the second to the third. We name this statement `+-assoc`.

```
+-assoc : Set
+-assoc = ∀ (x y z : ℕ) → x + (y + z) ≡ (x + y) + z
```

This is not yet a proof, we have merely written down the statement (or *enunciation*) of associativity.

Proof of associativity

The statement `+-assoc` is a member of `Set`, i.e. it is a type. Now that we have stated the property in a way that Agda understands, our objective is to prove it. To do so, we have to construct a function of type `+-assoc`.

First, we need to import the constructors `zero` and `suc` of the already imported datatype ℕ and the constructor `refl` (short for *reflexivity*) and function `cong` (short for *congruence*) from the standard library.

```
open import Data.Nat using (zero; suc)
open import Relation.Binary.PropositionalEquality using (refl; cong)
```

To prove `+-assoc` we need to find an object of that type. Here, we name this object `+-assoc-proof`.

```
+-assoc-proof : ∀ (x y z : ℕ) → x + (y + z) ≡ (x + y) + z
```

If we load now the file, Agda gives an error: "The following names are declared but not accompanied by a definition: `+-assoc-proof`". Indeed, we have only declared the type of `+-assoc-proof` but not yet given a definition. To build the definition, we need to know more about holes and case splitting.

Holes and case splitting

We can let Agda help us to write the proof by using its interactive mode. To start, we first write a simple clause so the file can be loaded even if we still do not know the proof. The clause consists of the name of the property, the input variables, the equals symbol = and the question mark `?`.

```
+-assoc-proof x y z = ?
```

When we reload the file, Agda no longer throws an error, but instead shows the message ***All Goals**** with a list of goals. We have now entered the interactive proving mode. Agda turns our question mark into what is called a *hole* `{ }0` with a label `0`. Each hole stands as a placeholder for a part of the program that is still incomplete and can be refined or resolved interactively.

Note

You are not supposed to enter a hole such as `{ }0` manually, Agda takes care of the numbering when you load the file. To insert a hole, write either `?` or `{! !}` and load the file to make Agda assign a unique number to it.

To get detailed information about a specific hole, put the cursor in it and press `C-c C-,`. This displays the type of the hole, as well as the types of all the variables in scope. In this example we get the information that the goal type is `x + (y + z) ≡ x + y + z`, and there are three variables `x`, `y`, and `z` in scope, all of type ℕ.

Note

You might wonder why Agda displays the term `(x + y) + z` as `x + y + z` (without parenthesis). This is done because of the infix statement `infixl 6 _+_` that was declared in the imported `Agda.Builtin.Nat` module. This declaration means that the _+_ operation is left-associative. More information about *mixfix operator* like the arithmetic operations. You can also check *this associativity example*.

To continue writing our proof, we now pick a variable and perform a case split on it. To do so, put the cursor inside the hole and press `C-c C-c`. Agda asks for the name of the pattern variable to case on. Let's write `x` and press return. This replaces the previous clause with two new clauses, one where `x` has been replaced by `zero` and another where it has been replaced by `suc x`:

```
+-assoc-proof zero y z = {  }0
+-assoc-proof (suc x) y z = {  }1
```

Important

The `x` in the type signature of `+-assoc-proof` is **not** the same as the `x` pattern variable in the last clause where `suc x` is written. The following would also work: `+-assoc-proof (suc x₁) y z = { }1`. The scope of a variable declared in a signature is restricted to the signature itself.

Instead of one hole, we now have two. The first hole has type `y + z ≡ y + z`, which is easy to resolve. To do so, put the cursor inside the first hole labeled `0` and press `C-c C-r`. This replaces the hole by the term `refl`, which stands for

reflexivity and can be used any time we want to construct a term of type `w ≡ w` for some term `w`.

```
+-assoc-proof zero y z = refl
+-assoc-proof (suc x) y z = {  }1
```

Now we have one hole left to resolve. By putting the cursor in it and pressing `C-c C-,` again, we get the type of the hole: `suc x + (y + z) ≡ suc x + y + z`. Agda has already applied the definition of `_+_` to replace the left-hand side `(suc x + y) + z` of the equation by `suc (x + y + z)`, and similarly replaced the right-hand side `suc x + (y + z)` by `suc (x + (y + z))`.

Tip

You can use the `go-to-definition` command by selecting the definition that you want to check eg. `_+_` and pressing `M-.` in Emacs or `C-M-\` in Atom. This takes you to the definition of `_+_`, which is originally defined in the builtin module `Agda.Builtin.Nat`.

Tip

You can ask Agda to compute the normal form of a term. To do so, place the cursor in the remaining hole (which should not contain any text at this point) and press `C-c C-n`. This prompts you for an expression to normalize. For example, if we enter `(suc x + y) + z` we get back `suc (x + y + z)` as a result.

Proof by induction

If we now look at the type of the remaining hole, we see that both the left-hand side and the right-hand side start with an application of the constructor `suc`. In this kind of situation it suffices to prove that the two arguments to `suc` are equal. This principle is called *congruence* of equality `_≡_`, and it is expressed by the Agda function `cong`.

To use `cong` we need to apply it to a function or constructor, in this case `suc`. If we ask Agda to infer the type of `cong suc` by pressing `C-c C-d` and entering the term, we get back the type `{x y : ℕ} → x ≡ y → suc x ≡ suc y`. In other words, `cong suc` takes as input a proof of an equality between `x` and `y` and produces a new proof of equality between `suc x` and `suc y`. We write `cong suc` in the hole and again press `C-c C-r` to refine the hole. This results in the new line

```
+-assoc-proof (suc x) y z = cong suc {  }2
```

where the new hole with number 2 is of type `x + (y + z) ≡ x + y + z`.

To finish the proof, we now make a recursive call `+-assoc-proof x y z`. Note that this has type `x + (y + z) ≡ (x + y) + z`, which is exactly what we need. To complete the proof, we type `+-assoc-proof x y z` into the hole and solve it with `C-c C-space`. This replaces the hole with the given term and completes the proof.

Note

When we define a recursive function like this, Agda performs *termination checking* on it. This is important to ensure the recursion is well-founded, and hence will not result in an invalid (circular) proof. In this case, the first argument `x` is structurally smaller than the first argument `suc x` on the left-hand side of the clause, hence Agda allows us to make the recursive call. Because termination is an undecidable property, Agda will not accept all terminating functions, but only the ones that are mechanically proved to terminate.

The final proof `+-assoc-proof` is defined as follows:

```
+-assoc-proof zero y z = refl
+-assoc-proof (suc x) y z = cong suc (+-assoc-proof x y z)
```

When we reload the file, we see ***All Done***. This means that `+-assoc-proof` is indeed a proof of the statement `+-assoc`.

Here is the final code of the 'Hello world' proof example, with all imports together at the top of the file:

```
module hello-world-proof where

open import Data.Nat using (ℕ; zero; suc; _+_)
open import Relation.Binary.PropositionalEquality using (_≡_; refl; cong)

+-assoc : Set
+-assoc = ∀ (x y z : ℕ) → x + (y + z) ≡ (x + y) + z

+-assoc-proof : ∀ (x y z : ℕ) → x + (y + z) ≡ (x + y) + z
+-assoc-proof zero y z = refl
+-assoc-proof (suc x) y z = cong suc (+-assoc-proof x y z)
```

Tip

You can learn more details about proving in the chapter Proof by Induction of the online book Programming Language Foundations in Agda.

2.5.4 Building an Executable Agda Program

Agda is a dependently typed functional programming language. This means that we can write programs in Agda that interact with the world. In this section, we write a small 'Hello world' program in Agda, compile it, and execute it. In contrast to the standalone example on the *Hello World page*, here we make use of the standard library to write a shorter version of the same program.

Agda Source Code

First, we create a new file named `hello-world-prog.agda` with Emacs or Atom in a folder that we refer to as our top-level folder.

```
{-# OPTIONS --guardedness #-}

module hello-world-prog where

open import IO

main : Main
main = run (putStrLn "Hello, World!")
```

A quick line-by-line explanation:

- The first line is a *pragma* (a special comment) that specifies some options at the top of the file.
- The second line declares the top-level module, named `hello-world-prog`.
- The third line imports the `IO` module from the standard library and brings its contents into scope.

- A module exporting a function `main` of type `Main` (defined in the `IO` module of the standard library) can be compiled to a standalone executable. For example: `main = run (putStrLn "Hello, World!")` runs the `IO` command `putStrLn "Hello, World!"` and then quits the program.

Compilation with GHC Backend

Once we have loaded the program in Emacs or Atom, we can compile it directly by pressing `C-c C-x C-c` and entering `GHC`. Alternatively, we can open a terminal session, navigate to the top-level folder and run:

```
agda --compile hello-world-prog.agda
```

The `--compile` flag here creates via the *GHC backend* a binary file in the top-level folder that the computer can execute.

Finally, we can then run the executable (`./hello-world-prog` on Unix systems, `hello-world-prog.exe` on Windows) from the command line:

```
$ cd <your top-level folder>
$ ./hello-world-prog
Hello, World!
```

Compilation with JavaScript Backend

The *JavaScript backend* translates the Agda source code of the `hello-world-prog.agda` file to JavaScript code.

From Emacs or Atom, press `C-c C-x C-c` and enter `JS` to compile the module to JavaScript. Alternatively, open a terminal session, navigate to the top-level folder and run:

```
agda --js hello-world-prog.agda
```

This creates several `.js` files in the top-level folder. The file corresponding to our source code has the name `jAgda.hello-world-prog.js`.

Hint

The additional `--js-optimize` flag can be used to make the generated JavaScript code faster but less readable. Moreover, the `--js-minify` flag makes the generated JavaScript code smaller and even less readable.

2.5.5 Where to go from here?

There are many books and tutorials on Agda. We recommend this *list of tutorials*.

Join the Agda Community!

Get in touch and join the Agda community, or join the conversation on the Agda Zulip.

2.6 A List of Tutorials

Note

Some of the materials linked on this page have been created for older versions of Agda and might no longer apply directly to the latest release.

2.6.1 Books on Agda

- Sandy Maguire (2023). Certainty by Construction
- Phil Wadler, Wen Kokke, and Jeremy G. Siek (2019). Programming Languages Foundations in Agda
- Aaron Stump (2016). Verified Functional Programming in Agda

2.6.2 Tutorials and lecture notes

- Brent Yorgey (2026). Proving the Fundamental Theorem of Arithmetic in Agda
- Ingo Blechschmidt (2025). Let's Play Agda
- Martín Escardó and Dan Licata (2022). Lecture notes for the HoTTEST Summer School.
- Danel Ahman and Andrej Bauer (2021). Logic in Computer Science.
- Jesper Cockx (2021). Programming and Proving in Agda. An introduction to Agda for a general audience of functional programmers. It starts from basic knowledge of Haskell and builds up to using equational reasoning to formally prove correctness of functional programs.
- effectfully (2020). Inference in Agda.
- Martín Hötzel Escardó. Introduction to Univalent Foundations of Mathematics with Agda.
- Musa Al-hassy (2019). A slow-paced introduction to reflection in Agda.
- Jesper Cockx (2019). Formalize all the things (in Agda).
- Peter Dybjer (2017). An Introduction to Programming and Proving in Agda (incomplete draft).
- Conor McBride (2015). Datatypes of Datatypes.
- Jan Malakhovski (2013). Brutal [Meta]Introduction to Dependent Types in Agda.
- Diviánszky Péter (2012). Agda Tutorial.
- Ana Bove, Peter Dybjer, and Ulf Norell (2009). A Brief Overview of Agda - A Functional Language with Dependent Types (in TPHOLs 2009) with an example of reflection. Code.
- Andreas Abel (2009). An Introduction to Dependent Types and Agda. Lecture notes used in teaching functional programming: basic introduction to Agda, Curry-Howard, equality, and verification of optimizations like fusion.
- Ulf Norell and James Chapman (2008). Dependently Typed Programming in Agda. This is aimed at functional programmers.
- Ana Bove and Peter Dybjer (2008). Dependent Types at Work. A gentle introduction including logic and proofs of programs.

2.6.3 Videos on Agda

- Jesper Cockx (2024). Programming and Proving in Agda. (Lecture at ZuriHac 2024).
- André Muricy (2023). Super Haskell: an introduction to Agda.
- Andrej Bauer (2022). Logic in Computer Science.
- HoTTEST Summer School 2022.

- Jacques Carette (2022). What I learned from formalizing Category Theory in Agda.
- Fredrik Nordvall Forsberg (2022). Advanced Functional Programming (videos of lectures).
- Peter Selinger (2021). Lectures on Agda.
- Anders Mörtberg (2021). Cubical Agda.
- Philip Wadler (2019). (Programming Languages) in Agda = Programming (Languages in Agda) (Lecture at YOW! 2019).
- Conor McBride (2014). Introduction to Dependently Typed Programming using Agda. (videos of lectures). Associated source files, with exercises.
- Daniel Licata (2013). Dependently Typed Programming in Agda (at OPLSS 2013).
- Daniel Peebles (2011). Introduction to Agda. Video of talk from the January 2011 Boston Haskell session at MIT.

2.6.4 Courses using Agda

- Computer Aided Reasoning Material for a 3rd / 4th year course (g53cfr, g54 cfr) at the university of Nottingham 2010 by Thorsten Altenkirch
- Type Theory in Rosario Material for an Agda course in Rosario, Argentina in 2011 by Thorsten Altenkirch
- Software System Design and Implementation, undergrad(?) course at the University of New South Wales by Manuel Chakravarty.
- Tüübiteooria / Type Theory, graduate course at the University of Tartu by Varmo Vene and James Chapman.
- Advanced Topics in Programming Languages: Dependent Type Systems, course at the University of Pennsylvania by Stephanie Weirich.
- Categorical Logic, course at the University of Cambridge by Samuel Staton.
- Dependently typed functional languages, master level course at EAFIT University by Andrés Sicard-Ramírez.
- Introduction to Dependently Typed Programming using Agda, research level course at the University of Edinburgh by Conor McBride.
- Agda, introductory course for master students at ELTE Eötvös Collegium in Budapest by Péter Diviánszky and Ambrus Kaposi.
- Types for Programs and Proofs, course at Chalmers University of Technology.
- Dependently typed metaprogramming (in Agda), Summer (2013) course at the University of Cambridge by Conor McBride.
- Computer-Checked Programs and Proofs (COMP 360-1), Dan Licata, Wesleyan, Fall 2013.
- Advanced Functional Programming Fall 2013 (CS410), Conor McBride, Strathclyde, notes from 2015, videos from 2017.
- Inductive and inductive-recursive definitions in Intuitionistic Type Theory, lectures by Peter Dybjer at the Oregon Programming Languages Summer School 2015.
- Introduction to Univalent Foundations of Mathematics with Agda , MGS 2019 Martín Hötzel Escardó
- Higher-Dimensional Type Theory (CSCI 8980), courses on homotopy type theory and cubical type theory, Favonia, the University of Minnesota, Spring 2020
- Correct-by-construction Programming in Agda, a course at the EUTYPES Summer School '19 in Ohrid.
- Lectures on Agda, a course by Peter Selinger at Dalhousie University, Winter 2021.
- HoTTEST Summer School 2022, online lectures by assorted instructors.

- Advanced Programming Paradigms, Postgraduate course jointly offered by the Universities of Applied Sciences in Switzerland, by Daniel Kröni and Farhad Mehta.
- Advanced Functional Programming, master level course at Utrecht University by Wouter Swierstra and Lawrence Chonavel.

2.6.5 Miscellaneous

- Agda has a Wikipedia page

CHAPTER

THREE

LANGUAGE REFERENCE

3.1 Abstract definitions

Definitions can be marked as abstract, for the purpose of hiding implementation details, or to speed up type-checking of other parts. In essence, abstract definitions behave like postulates, thus, do not reduce/compute. For instance, proofs whose content does not matter could be marked abstract, to prevent Agda from unfolding them (which might slow down type-checking).

As a guiding principle, all the rules concerning `abstract` are designed to prevent the leaking of implementation details of abstract definitions. Similar concepts of other programming language include (non-representative sample): UCSD Pascal's and Java's interfaces and ML's signatures. (Especially when abstract definitions are used in combination with modules.)

3.1.1 Synopsis

- Declarations can be marked as abstract using the block keyword `abstract`.
- Outside of abstract blocks, abstract definitions do not reduce, they are treated as postulates, in particular:
  - Abstract functions never match, thus, do not reduce.
  - Abstract data types do not expose their constructors.
  - Abstract record types do not expose their fields nor constructor.
  - Other declarations cannot be abstract.
- Inside abstract blocks, abstract definitions reduce while type checking definitions, but not while checking their type signatures. Otherwise, due to dependent types, one could leak implementation details (e.g. expose reduction behavior by using propositional equality).

  Consequently information from checking the body of a definition cannot leak into its type signature, effectively disabling type inference for abstract definitions. This means that all abstract definitions need a complete type signature.
- The reach of the `abstract` keyword block extends recursively to the `where`-blocks of a function and the declarations inside of a `record` declaration, but not inside modules declared in an abstract block.

3.1.2 Examples

Integers can be implemented in various ways, e.g. as difference of two natural numbers:

```
module Integer where

  abstract

```

(continues on next page)

(continued from previous page)

```
  ℤ : Set
  ℤ = Nat × Nat

  0ℤ : ℤ
  0ℤ = 0 , 0

  1ℤ : ℤ
  1ℤ = 1 , 0

  _+ℤ_ : (x y : ℤ) → ℤ
  (p , n) +ℤ (p' , n') = (p + p') , (n + n')

  _*ℤ_ : (x y : ℤ) → ℤ
  (a , b) *ℤ (c , d) = ((a * c) + (b * d)) , ((a * d) + (b * c))

  infixl 20 _+ℤ_
  infixl 30 _*ℤ_

  -ℤ_ : ℤ → ℤ
  -ℤ (p , n) = (n , p)

  _≡ℤ_ : (x y : ℤ) → Set
  (p , n) ≡ℤ (p' , n') = (p + n') ≡ (p' + n)

  infix 10 _≡ℤ_

  private
    postulate
      +comm : ∀ n m → (n + m) ≡ (m + n)

  invℤ : ∀ x → (x +ℤ (-ℤ x)) ≡ℤ 0ℤ
  invℤ (p , n) rewrite +comm (p + n) 0 | +comm p n = refl
```

Using `abstract` we do not give away the actual representation of integers, nor the implementation of the operations. We can construct them from `0ℤ`, `1ℤ`, `_+ℤ_`, and `-ℤ`, but only reason about equality `≡ℤ` with the provided lemma `invℤ`.

The following property `shape-of-0ℤ` of the integer zero exposes the representation of integers as pairs. As such, it is rejected by Agda: when checking its type signature, $\texttt{proj}_1$ `x` fails to type check since `x` is of abstract type `ℤ`. Remember that the abstract definition of `ℤ` does not unfold in type signatures, even when in an abstract block! To work around this we have to define aliases for the projections functions:

```
-- A property about the representation of zero integers:

  abstract
    private
      posZ : ℤ → Nat
      posZ = proj₁

      negZ : ℤ → Nat
      negZ = proj₂

      shape-of-0ℤ : ∀ (x : ℤ) (is0ℤ : x ≡ℤ 0ℤ) → posZ x ≡ negZ x
```

(continues on next page)

(continued from previous page)

```
    shape-of-0ℤ (p , n) refl rewrite +comm p 0 = refl
```

By requiring `shape-of-0ℤ` to be private to type-check, leaking of representation details is prevented.

3.1.3 Scope of abstraction

In child modules, when checking an abstract definition, the abstract definitions of the parent module are transparent:

```
module M1 where
  abstract
    x : Nat
    x = 0

  module M2 where
    abstract
      x-is-0 : x ≡ 0
      x-is-0 = refl
```

Thus, child modules can see into the representation choices of their parent modules. However, parent modules cannot see like this into child modules, nor can sibling modules see through each others abstract definitions. An exception to this is anonymous modules, which share abstract scope with their parent module, allowing parent or sibling modules to see inside their abstract definitions.

The reach of the `abstract` keyword does not extend into modules:

```
module Parent where
  abstract
    module Child where
      y : Nat
      y = 0
    x : Nat
    x = 0  -- to avoid "useless abstract" error

  y-is-0 : Child.y ≡ 0
  y-is-0 = refl
```

The declarations in module `Child` are not abstract!

3.1.4 Abstract definitions with where-blocks

Definitions in a `where` block of an abstract definition are abstract as well. This means, they can see through the abstractions of their uncles:

```
module Where where
  abstract
    x : Nat
    x = 0
    y : Nat
    y = x
      where
      x≡y : x ≡ 0
      x≡y = refl
```

3.2 Built-ins

The Agda type checker knows about, and has special treatment for, a number of different concepts. The most prominent is natural numbers, which has a special representation as Haskell integers and support for fast arithmetic. The surface syntax of these concepts are not fixed, however, so in order to use the special treatment of natural numbers (say) you define an appropriate data type and then bind that type to the natural number concept using a `BUILTIN` pragma.

Some built-in types support primitive functions that have no corresponding Agda definition. These functions are declared using the `primitive` keyword by giving their type signature.

3.2.1 Using the built-in types

While it is possible to define your own versions of the built-in types and bind them using `BUILTIN` pragmas, it is recommended to use the definitions in the `Agda.Builtin` modules. These modules are installed when you install Agda and so are always available. For instance, built-in natural numbers are defined in `Agda.Builtin.Nat`. The standard library and the agda-prelude reexport the definitions from these modules.

3.2.2 The unit type

```
module Agda.Builtin.Unit
```

The unit type is bound to the built-in `UNIT` as follows:

```
record ⊤ : Set where
{-# BUILTIN UNIT ⊤ #-}
```

Agda needs to know about the unit type since some of the primitive operations in the *reflected type checking monad* return values in the unit type.

3.2.3 The Σ-type

```
module Agda.Builtin.Sigma
```

The built-in Σ-type of dependent pairs is defined as follows:

```
record Σ {a b} (A : Set a) (B : A → Set b) : Set (a ⊔ b) where
  constructor _,_
  field
    fst : A
    snd : B fst

open Σ public

infixr 4 _,_

{-# BUILTIN SIGMA Σ #-}
```

3.2.4 Lists

```
module Agda.Builtin.List
```

Built-in lists are bound using the `LIST` built-in:

```
data List {a} (A : Set a) : Set a where
  []  : List A
  _∷_ : (x : A) (xs : List A) → List A
{-# BUILTIN LIST List #-}
infixr 5 _∷_
```

The constructors are bound automatically when binding the type. Lists are not required to be level polymorphic; `List : Set → Set` is also accepted.

As with booleans, the effect of binding the `LIST` built-in is to let you use primitive functions working with lists, such as `primStringToList` and `primStringFromList`, and letting the *GHC backend* know to compile the List type to Haskell lists.

3.2.5 Maybe

```
module Agda.Builtin.Maybe
```

Built-in maybe type is bound using the `MAYBE` built-in:

```
data Maybe {a} (A : Set a) : Set a where
  nothing : Maybe A
  just    : A → Maybe A
{-# BUILTIN MAYBE Maybe #-}
```

The constructors are bound automatically when binding the type. Maybe is not required to be level polymorphic; `Maybe : Set → Set` is also accepted.

As with list, the effect of binding the `MAYBE` built-in is to let you use primitive functions working with maybes, such as `primStringUncons` that returns the head and tail of a string (if it is non empty), and letting the *GHC backend* know to compile the Maybe type to Haskell maybes.

3.2.6 Booleans

```
module Agda.Builtin.Bool where
```

Built-in booleans are bound using the `BOOL`, `TRUE` and `FALSE` built-ins:

```
data Bool : Set where
  false true : Bool
{-# BUILTIN BOOL  Bool  #-}
{-# BUILTIN TRUE  true  #-}
{-# BUILTIN FALSE false #-}
```

Note that unlike for natural numbers, you need to bind the constructors separately. The reason for this is that Agda cannot tell which constructor should correspond to true and which to false, since you are free to name them whatever you like.

The effect of binding the boolean type is that you can then use primitive functions returning booleans, such as built-in `NATEQUALS`, and letting the *GHC backend* know to compile the type to Haskell *Bool*.

3.2.7 Natural numbers

```
module Agda.Builtin.Nat
```

Built-in natural numbers are bound using the `NATURAL` built-in as follows:

```
data Nat : Set where
  zero : Nat
  suc  : Nat → Nat
{-# BUILTIN NATURAL Nat #-}
```

The names of the data type and the constructors can be chosen freely, but the shape of the datatype needs to match the one given above (modulo the order of the constructors). Note that the constructors need not be bound explicitly.

Binding the built-in natural numbers as above has the following effects:

- The use of *natural number literals* is enabled. By default the type of a natural number literal will be `Nat`, but it can be *overloaded* to include other types as well.
- Closed natural numbers are represented as Haskell integers at compile-time.
- The compiler backends *compile natural numbers* to the appropriate number type in the target language.
- Enabled binding the built-in natural number functions described below.

Functions on natural numbers

There are a number of built-in functions on natural numbers. These are special in that they have both an Agda definition and a primitive implementation. The primitive implementation is used to evaluate applications to closed terms, and the Agda definition is used otherwise. This lets you prove things about the functions while still enjoying good performance of compile-time evaluation. The built-in functions are the following:

```
_+_ : Nat → Nat → Nat
zero  + m = m
suc n + m = suc (n + m)
{-# BUILTIN NATPLUS _+_ #-}

_-_ : Nat → Nat → Nat
n     - zero  = n
zero  - suc m = zero
suc n - suc m = n - m
{-# BUILTIN NATMINUS _-_ #-}

_*_ : Nat → Nat → Nat
zero  * m = zero
suc n * m = (n * m) + m
{-# BUILTIN NATTIMES _*_ #-}

infixl 30 _*_
infixl 20 _+_

_==_ : Nat → Nat → Bool
zero  == zero  = true
suc n == suc m = n == m
_     == _     = false
{-# BUILTIN NATEQUALS _==_ #-}

_<_ : Nat → Nat → Bool
_     < zero  = false
zero  < suc _ = true
suc n < suc m = n < m
{-# BUILTIN NATLESS _<_ #-}

div-helper : Nat → Nat → Nat → Nat → Nat
div-helper k m  zero    j      = k
div-helper k m (suc n)  zero   = div-helper (suc k) m n m
div-helper k m (suc n) (suc j) = div-helper k m n j
{-# BUILTIN NATDIVSUCAUX div-helper #-}

mod-helper : Nat → Nat → Nat → Nat → Nat
mod-helper k m  zero    j      = k
mod-helper k m (suc n)  zero   = mod-helper 0 m n m
mod-helper k m (suc n) (suc j) = mod-helper (suc k) m n j
{-# BUILTIN NATMODSUCAUX mod-helper #-}
```

The Agda definitions are checked to make sure that they really define the corresponding built-in function. The definitions are not required to be exactly those given above, for instance, addition and multiplication can be defined by recursion on either argument, and you can swap the arguments to the addition in the recursive case of multiplication.

The `NATDIVSUCAUX` and `NATMODSUCAUX` are built-ins bind helper functions for defining natural number division and

modulo operations, and satisfy the properties

```
div n (suc m) ≡ div-helper 0 m n m
mod n (suc m) ≡ mod-helper 0 m n m
```

3.2.8 Machine words

```
module Agda.Builtin.Word
module Agda.Builtin.Word.Properties
```

Agda supports built-in 64-bit machine words, bound with the `WORD64` built-in:

```
postulate Word64 : Set
{-# BUILTIN WORD64 Word64 #-}
```

Machine words can be converted to and from natural numbers using the following primitives:

```
primitive
  primWord64ToNat   : Word64 → Nat
  primWord64FromNat : Nat → Word64
```

Converting to a natural number is the trivial embedding, and converting from a natural number gives you the remainder modulo 2^{64}. The proof of the former theorem:

```
primitive
  primWord64ToNatInjective : ∀ a b → primWord64ToNat a ≡ primWord64ToNat b → a ≡ b
```

is in the `Properties` module. The proof of the latter theorem is not primitive, but can be defined in a library using *primTrustMe*.

Basic arithmetic operations can be defined on `Word64` by converting to natural numbers, performing the corresponding operation, and then converting back. The compiler will optimise these to use 64-bit arithmetic. For instance:

```
addWord : Word64 → Word64 → Word64
addWord a b = primWord64FromNat (primWord64ToNat a + primWord64ToNat b)

subWord : Word64 → Word64 → Word64
subWord a b = primWord64FromNat ((primWord64ToNat a + 18446744073709551616) -␣
↪primWord64ToNat b)
```

These compile to primitive addition and subtraction on 64-bit words, which in the *GHC backend* map to operations on Haskell 64-bit words (`Data.Word.Word64`).

3.2.9 Integers

```
module Agda.Builtin.Int
```

Built-in integers are bound with the `INTEGER` built-in to a data type with two constructors: one for positive and one for negative numbers. The built-ins for the constructors are `INTEGERPOS` and `INTEGERNEGSUC`.

```
data Int : Set where
  pos    : Nat → Int
  negsuc : Nat → Int
{-# BUILTIN INTEGER       Int    #-}
```

(continues on next page)

(continued from previous page)

```
{-# BUILTIN INTEGERPOS    pos    #-}
{-# BUILTIN INTEGERNEGSUC negsuc #-}
```

Here `negsuc n` represents the integer `-n - 1`. Unlike for natural numbers, there is no special representation of integers at compile-time since the overhead of using the data type compared to Haskell integers is not that big.

Built-in integers support the following primitive operation (given a suitable binding for *String*):

```
primitive
  primShowInteger : Int → String
```

3.2.10 Floats

```
module Agda.Builtin.Float
module Agda.Builtin.Float.Properties
```

Floating point numbers are bound with the `FLOAT` built-in:

```
postulate Float : Set
{-# BUILTIN FLOAT Float #-}
```

This lets you use *floating point literals*. Floats are represented by the type checker as IEEE 754 binary64 double precision floats, with the restriction that there is exactly one NaN value. The following primitive functions are available (with suitable bindings for *Nat*, *Bool*, *String*, *Int*, *Maybe_*):

```
primitive
  -- Relations
  primFloatIsInfinite        : Float → Bool
  primFloatIsNaN             : Float → Bool
  primFloatIsNegativeZero    : Float → Bool

  -- Conversions
  primNatToFloat             : Nat → Float
  primIntToFloat             : Int → Float
  primFloatToRatio           : Float → (Σ Int λ _ → Int)
  primRatioToFloat           : Int → Int → Float
  primShowFloat              : Float → String

  -- Operations
  primFloatPlus              : Float → Float → Float
  primFloatMinus             : Float → Float → Float
  primFloatTimes             : Float → Float → Float
  primFloatDiv               : Float → Float → Float
  primFloatPow               : Float → Float → Float
  primFloatNegate            : Float → Float
  primFloatSqrt              : Float → Float
  primFloatExp               : Float → Float
  primFloatLog               : Float → Float
  primFloatSin               : Float → Float
  primFloatCos               : Float → Float
  primFloatTan               : Float → Float
  primFloatASin              : Float → Float
```

(continues on next page)

(continued from previous page)

```
  primFloatACos              : Float → Float
  primFloatATan              : Float → Float
  primFloatATan2             : Float → Float → Float
  primFloatSinh              : Float → Float
  primFloatCosh              : Float → Float
  primFloatTanh              : Float → Float
  primFloatASinh             : Float → Float
  primFloatACosh             : Float → Float
  primFloatATanh             : Float → Float
```

The primitive binary relations implement their IEEE 754 equivalents, which means that `primFloatEquality` is not reflexive, and `primFloatInequality` and `primFloatLess` are not total. (Specifically, NaN is not related to anything, including itself.)

The `primFloatIsSafeInteger` function determines whether the value is a number that is a safe integer, i.e., is within the range where the arithmetic operations do not lose precision.

Floating point numbers can be converted to their raw representation using the primitive:

```
primitive
  primFloatToWord64          : Float → Maybe Word64
```

which returns `nothing` for `NaN` and satisfies:

```
primFloatToWord64Injective : ∀ a b → primFloatToWord64 a ≡ primFloatToWord64 b → a ≡ b
```

in the `Properties` module. These primitives can be used to define a safe decidable propositional equality with the *`--safe`* option. The function `primFloatToWord64` cannot be guaranteed to be consistent across backends, therefore relying on the specific result may result in inconsistencies.

The rounding operations (`primFloatRound`, `primFloatFloor`, and `primFloatCeiling`) return a value of type `Maybe Int`, and return `nothing` when applied to NaN or the infinities:

```
primitive
  primFloatRound             : Float → Maybe Int
  primFloatFloor             : Float → Maybe Int
  primFloatCeiling           : Float → Maybe Int
```

The `primFloatDecode` function decodes a floating-point number to its mantissa and exponent, normalised such that the mantissa is the smallest possible integer. It fails when applied to NaN or the infinities, returning `nothing`. The `primFloatEncode` function encodes a pair of a mantissa and exponent to a floating-point number. It fails when the resulting number cannot be represented as a float. Note that `primFloatEncode` may result in a loss of precision.

primitive
primFloatDecode : Float → Maybe (Σ Int λ _ → Int) primFloatEncode : Int → Int → Maybe Float

3.2.11 Characters

```
module Agda.Builtin.Char
module Agda.Builtin.Char.Properties
```

The character type is bound with the `CHARACTER` built-in:

```
postulate Char : Set
{-# BUILTIN CHAR Char #-}
```

Binding the character type lets you use *character literals*. The following primitive functions are available on characters (given suitable bindings for *Bool*, *Nat* and *String*):

```
primitive
  primIsLower    : Char → Bool
  primIsDigit    : Char → Bool
  primIsAlpha    : Char → Bool
  primIsSpace    : Char → Bool
  primIsAscii    : Char → Bool
  primIsLatin1   : Char → Bool
  primIsPrint    : Char → Bool
  primIsHexDigit : Char → Bool
  primToUpper    : Char → Char
  primToLower    : Char → Char
  primCharToNat  : Char → Nat
  primNatToChar  : Nat → Char
  primShowChar   : Char → String
```

These functions are implemented by the corresponding Haskell functions from Data.Char (`ord` and `chr` for `primCharToNat` and `primNatToChar`). To make `primNatToChar` total `chr` is applied to the natural number modulo `0x110000`. Furthermore, to match the behaviour of strings, surrogate code points are mapped to the replacement character `U+FFFD`.

Converting to a natural number is the obvious embedding, and its proof:

```
primitive
  primCharToNatInjective : ∀ a b → primCharToNat a ≡ primCharToNat b → a ≡ b
```

can be found in the `Properties` module.

3.2.12 Strings

```
module Agda.Builtin.String
module Agda.Builtin.String.Properties
```

The string type is bound with the `STRING` built-in:

```
postulate String : Set
{-# BUILTIN STRING String #-}
```

Binding the string type lets you use *string literals*. The following primitive functions are available on strings (given suitable bindings for *Bool*, *Char* and *List*):

```
primitive
  primStringUncons   : String → Maybe (Σ Char (λ _ → String))
  primStringToList   : String → List Char
  primStringFromList : List Char → String
  primStringAppend   : String → String → String
  primStringEquality : String → String → Bool
  primShowString     : String → String
```

String literals can be *overloaded*.

Converting to and from a list is injective, and their proofs:

```
primitive
  primStringToListInjective : ∀ a b → primStringToList a ≡ primStringToList b → a ≡ b
  primStringFromListInjective : ∀ a b → primStringFromList a ≡ primStringFromList b →␣
↪a ≡ b
```

can found in the `Properties` module.

Strings cannot represent unicode surrogate code points (characters in the range `U+D800` to `U+DFFF`). These are replaced by the unicode replacement character `U+FFFD` if they appear in string literals.

3.2.13 Equality

```
module Agda.Builtin.Equality
```

The identity type can be bound to the built-in `EQUALITY` as follows

```
infix 4 _≡_
data _≡_ {a} {A : Set a} (x : A) : A → Set a where
  refl : x ≡ x
{-# BUILTIN EQUALITY _≡_  #-}
```

This lets you use proofs of type `lhs ≡ rhs` in the *rewrite construction*.

Other variants of the identity type are also accepted as built-in:

```
data _≡_ {A : Set} : (x y : A) → Set where
  refl : (x : A) → x ≡ x
```

The type of `primEraseEquality` has to match the flavor of identity type.

```
module Agda.Builtin.Equality.Erase
```

Binding the built-in equality type also enables the `primEraseEquality` primitive:

```
primitive
  primEraseEquality : ∀ {a} {A : Set a} {x y : A} → x ≡ y → x ≡ y
```

The function takes a proof of an equality between two values `x` and `y` and stays stuck on it until `x` and `y` actually become definitionally equal. Whenever that is the case, `primEraseEquality e` reduces to `refl`.

One use of `primEraseEquality` is to replace an equality proof computed using an expensive function (e.g. a proof by reflection) by one which is trivially `refl` on the diagonal.

primTrustMe

```
module Agda.Builtin.TrustMe
```

From the `primEraseEquality` primitive, we can derive a notion of `primTrustMe`:

```
primTrustMe : ∀ {a} {A : Set a} {x y : A} → x ≡ y
primTrustMe {x = x} {y} = primEraseEquality unsafePrimTrustMe
  where postulate unsafePrimTrustMe : x ≡ y
```

As can be seen from the type, `primTrustMe` must be used with the utmost care to avoid inconsistencies. What makes it different from a postulate is that if `x` and `y` are actually definitionally equal, `primTrustMe` reduces to `refl`. One use

of `primTrustMe` is to lift the primitive boolean equality on built-in types like *String* to something that returns a proof object:

```
eqString : (a b : String) → Maybe (a ≡ b)
eqString a b = if primStringEquality a b
               then just primTrustMe
               else nothing
```

With this definition `eqString "foo" "foo"` computes to `just refl`.

3.2.14 Sorts

The primitive sorts used in Agda's type system are declared using `BUILTIN` pragmas in the `Agda.Primitive` module. These pragmas should not be used directly in other modules, but it is possible to rename these builtin sorts when importing `Agda.Primitive`.

```
{-# BUILTIN PROP           Prop      #-}
{-# BUILTIN TYPE           Set       #-}
{-# BUILTIN STRICTSET      SSet      #-}

{-# BUILTIN PROPOMEGA      Propω     #-}
{-# BUILTIN SETOMEGA       Setω      #-}
{-# BUILTIN STRICTSETOMEGA SSetω     #-}

{-# BUILTIN LEVELUNIV      LevelUniv #-}
```

The primitive sort *Set* is automatically imported at the top of every top-level Agda module, unless the *--no-import-sorts* flag is enabled.

3.2.15 Universe levels

```
module Agda.Primitive
```

Universe levels are also declared using `BUILTIN` pragmas. In contrast to the `Agda.Builtin` modules, the `Agda.Primitive` module is auto-imported and thus it is not possible to change the level built-ins. For reference these are the bindings:

```
postulate
  Level : LevelUniv
  lzero : Level
  lsuc  : Level → Level
  _⊔_   : Level → Level → Level

{-# BUILTIN LEVEL     Level #-}
{-# BUILTIN LEVELZERO lzero #-}
{-# BUILTIN LEVELSUC  lsuc  #-}
{-# BUILTIN LEVELMAX  _⊔_   #-}
```

Note that if the flag *--level-universe* is not set, then `LevelUniv` will be `Set`.

3.2.16 Sized types

```
module Agda.Builtin.Size
```

The built-ins for *sized types* are different from other built-ins in that the names are defined by the `BUILTIN` pragma. Hence, to bind the size primitives it is enough to write:

```
{-# BUILTIN SIZEUNIV SizeUniv #-}  --  SizeUniv : SizeUniv
{-# BUILTIN SIZE     Size     #-}  --  Size     : SizeUniv
{-# BUILTIN SIZELT   Size<_   #-}  --  Size<_   : ..Size → SizeUniv
{-# BUILTIN SIZESUC  ↑_       #-}  --  ↑_       : Size → Size
{-# BUILTIN SIZEINF  ∞         #-}  --  ∞         : Size
{-# BUILTIN SIZEMAX  _⊔ˢ_     #-}  --  _⊔ˢ_     : Size → Size → Size
```

3.2.17 Coinduction

```
module Agda.Builtin.Coinduction
```

The following built-ins are used for coinductive definitions:

```
postulate
  ∞  : ∀ {a} (A : Set a) → Set a
  ♯_ : ∀ {a} {A : Set a} → A → ∞ A
  ♭  : ∀ {a} {A : Set a} → ∞ A → A
{-# BUILTIN INFINITY ∞  #-}
{-# BUILTIN SHARP    ♯_ #-}
{-# BUILTIN FLAT     ♭  #-}
```

See *Coinduction* for more information.

3.2.18 IO

```
module Agda.Builtin.IO
```

The sole purpose of binding the built-in `IO` type is to let Agda check that the `main` function has the right type (see *Compilers*).

```
postulate IO : Set → Set
{-# BUILTIN IO IO #-}
```

3.2.19 Literal overloading

```
module Agda.Builtin.FromNat
module Agda.Builtin.FromNeg
module Agda.Builtin.FromString
```

The machinery for *overloading literals* uses built-ins for the conversion functions.

3.2.20 Reflection

```
module Agda.Builtin.Reflection
```

The reflection machinery has built-in types for representing Agda programs. See *Reflection* for a detailed description.

3.2.21 Rewriting

The experimental and totally unsafe *rewriting machinery* (not to be confused with the *rewrite construct*) has a built-in `REWRITE` for the rewriting relation:

```
postulate _↦_ : ∀ {a} {A : Set a} → A → A → Set a
{-# BUILTIN REWRITE _↦_ #-}
```

This builtin is bound to the *builtin equality type* from `Agda.Builtin.Equality` in `Agda.Builtin.Equality.Rewrite`.

3.2.22 Static values

The `STATIC` pragma can be used to mark definitions which should be normalised before compilation. The typical use case for this is to mark the interpreter of an embedded language as `STATIC`:

```
{-# STATIC <Name> #-}
```

3.2.23 Strictness

```
module Agda.Builtin.Strict
```

There are two primitives for controlling evaluation order:

```
primitive
  primForce      : ∀ {a b} {A : Set a} {B : A → Set b} (x : A) → (∀ x → B x) → B x
  primForceLemma : ∀ {a b} {A : Set a} {B : A → Set b} (x : A) (f : ∀ x → B x) →␣
↪primForce x f ≡ f x
```

where `_≡_` is the *built-in equality*. At compile-time `primForce x f` evaluates to `f x` when `x` is in weak head normal form (whnf), i.e. one of the following:

- a constructor application
- a literal
- a lambda abstraction
- a type constructor application (data or record type)
- a function type
- a universe (`Set _`)

Similarly `primForceLemma x f`, which lets you reason about programs using `primForce`, evaluates to `refl` when `x` is in whnf. At run-time, `primForce e f` is compiled (by the GHC *backend*) to `let x = e in seq x (f x)`.

For example, consider the following function:

```
-- pow' n a = a 2^n
pow' : Nat → Nat → Nat
pow' zero    a = a
pow' (suc n) a = pow' n (a + a)
```

There is a space leak here (both for compile-time and run-time evaluation), caused by unevaluated `a + a` thunks. This problem can be fixed with `primForce`:

```
infixr 0 _$!_
_$!_ : ∀ {a b} {A : Set a} {B : A → Set b} → (∀ x → B x) → ∀ x → B x
f $! x = primForce x f

-- pow n a = a 2^n
pow : Nat → Nat → Nat
pow zero    a = a
pow (suc n) a =  pow n $! a + a
```

3.3 Coinduction

The corecursive definitions below are accepted if the option `--guardedness` is active:

```
{-# OPTIONS --guardedness #-}
```

(An alternative approach is to use *Sized Types*.)

3.3.1 Coinductive Records

It is possible to define the type of infinite lists (or streams) of elements of some type `A` as follows:

```
record Stream (A : Set) : Set where
  coinductive
  field
    hd : A
    tl : Stream A
```

As opposed to *inductive record types*, we have to introduce the keyword `coinductive` before defining the fields that constitute the record.

It is interesting to note that it is not necessary to give an *explicit constructor* to the record type `Stream`.

Now we can use *copatterns* to create `Streams`, like one that repeats a given element `a` infinitely many times:

```
repeat : {A : Set} (a : A) -> Stream A
hd (repeat a) = a
tl (repeat a) = repeat a
```

We can also define pointwise equality (a bisimulation and an equivalence) of a pair of `Streams` as a coinductive record:

```
record _≈_ {A} (xs : Stream A) (ys : Stream A) : Set where
  coinductive
  field
    hd-≡ : hd xs ≡ hd ys
    tl-≈ : tl xs ≈ tl ys
```

Using *copatterns* we can define a pair of functions on `Streams` such that one returns the elements in the even positions and the other the elements in the odd positions:

```
even : ∀ {A} → Stream A → Stream A
hd (even xs) = hd xs
tl (even xs) = even (tl (tl xs))

odd : ∀ {A} → Stream A → Stream A
```

(continues on next page)

(continued from previous page)

```
odd xs = even (tl xs)

split : ∀ {A} → Stream A → Stream A × Stream A
split xs = even xs , odd xs
```

as well as a function that merges a pair of `Streams` by interleaving their elements:

```
merge : ∀ {A} → Stream A × Stream A → Stream A
hd (merge (xs , ys)) = hd xs
tl (merge (xs , ys)) = merge (ys , tl xs)
```

Finally, we can prove that `merge` is a left inverse for `split`:

```
merge-split-id : ∀ {A} (xs : Stream A) → merge (split xs) ≈ xs
hd-≡ (merge-split-id _)  = refl
tl-≈ (merge-split-id xs) = merge-split-id (tl xs)
```

Coinductive Record Constructors

It is possible to give an explicit constructor to coinductive record types like `Stream`:

```
record Stream' (A : Set) : Set where
  coinductive
  constructor cons
  field
    hd : A
    tl : Stream' A
```

However, this constructor cannot be pattern-matched:

```
-- Get the third element of a stream
third : ∀{A} → Stream' A → A

-- Not allowed:
-- third (cons _ (cons _ (cons x _))) = x
```

Instead, you can use the record fields as projections:

```
third str = str .tl .tl .hd
```

The constructor can be used as usual in the right-hand side of definitions:

```
-- Prepend a list to a stream
prepend : ∀{A} → List A → Stream' A → Stream' A
prepend [] str = str
prepend (a ∷ as) str = cons a (prepend as str)
```

However, it doesn't count as 'guarding' for the productivity checker:

```
-- Make a stream with one element repeated forever
cycle : ∀{A} → A → Stream' A

-- Does not termination-check:
-- cycle a = cons a (cycle a)
```

Instead, you can use copattern matching:

```
cycle a .hd = a
cycle a .tl = cycle a
```

It is also possible to use copatterns in a *Pattern lambda*:

```
cycle' : ∀{A} → A → Stream' A
cycle' a = λ where
  .hd → a
  .tl → cycle' a
```

For more information on these restrictions, see this pull request, and this commit.

The `ETA_EQUALITY` pragma

Agda does not permit the `eta-equality` directive in `coinductive record` declarations, since η for coinductive types is unsafe in general and can make the type checker loop. For instance, the following code would lead to infinite η expansion when checking `test`:

```
record R : Set where
 coinductive; eta-equality
 field force : R
open R

foo : R
foo .force .force = foo

test : foo .force ≡ foo
test = refl
```

If you know what you are doing, you can override Agda and force a coinductive record to support η via the `ETA_EQUALITY` pragma.

```
{-# ETA_EQUALITY #-}
record R : Set where
  ...
```

3.3.2 Old Coinduction

> **Note**
>
> This is the old way of coinduction support in Agda. You are advised to use *Coinductive Records* instead.

To use coinduction it is recommended that you import the module Coinduction from the standard library. Coinductive types can then be defined by labelling coinductive occurrences using the delay operator ∞:

```
data Coℕ : Set where
   zero : Coℕ
   suc  : ∞ Coℕ → Coℕ
```

The type ∞ `A` can be seen as a suspended computation of type `A`. It comes with delay and force functions:

```
♯_ : ∀ {a} {A : Set a} → A → ∞ A
♭  : ∀ {a} {A : Set a} → ∞ A → A
```

Values of coinductive types can be constructed using corecursion, which does not need to terminate, but has to be productive. As an approximation to productivity the termination checker requires that corecursive definitions are guarded by coinductive constructors. As an example the infinite "natural number" can be defined as follows:

```
inf : CoNat
inf = suc (♯ inf)
```

The check for guarded corecursion is integrated with the check for size-change termination, thus allowing interesting combinations of inductive and coinductive types. We can for instance define the type of stream processors, along with some functions:

```
-- Infinite streams.

data Stream (A : Set) : Set where
  _::_ : (x : A) (xs : ∞ (Stream A)) → Stream A

-- A stream processor SP A B consumes elements of A and produces
-- elements of B. It can only consume a finite number of A's before
-- producing a B.

data SP (A B : Set) : Set where
  get : (f : A → SP A B) → SP A B
  put : (b : B) (sp : ∞ (SP A B)) → SP A B

-- The function eat is defined by an outer corecursion into Stream B
-- and an inner recursion on SP A B.

eat : ∀ {A B} → SP A B → Stream A → Stream B
eat (get f)    (a :: as) = eat (f a) (♭ as)
eat (put b sp) as        = b :: ♯ eat (♭ sp) as

-- Composition of stream processors.

_∘_ : ∀ {A B C} → SP B C → SP A B → SP A C
get f₁    ∘ put x sp₂ = f₁ x ∘ ♭ sp₂
put x sp₁ ∘ sp₂       = put x (♯ (♭ sp₁ ∘ sp₂))
sp₁       ∘ get f₂    = get (λ x → sp₁ ∘ f₂ x)
```

It is also possible to define "coinductive families". It is recommended not to use the delay constructor (♯_) in a constructor's index expressions. The following definition of equality between coinductive "natural numbers" is discouraged:

```
data _≈'_ : CoNat → CoNat → Set where
  zero : zero ≈' zero
  suc  : ∀ {m n} → ∞ (m ≈' n) → suc (♯ m) ≈' suc (♯ n)
```

The recommended definition is the following one:

```
data _≈_ : CoNat → CoNat → Set where
  zero : zero ≈ zero
  suc  : ∀ {m n} → ∞ (♭ m ≈ ♭ n) → suc m ≈ suc n
```

3.4 Copatterns

> **Note**
>
> If you are looking for information on how to use copatterns with coinductive records, please visit the section on *coinduction*.

Consider the following record:

```
record Enumeration (A : Set) : Set where
  constructor enumeration
  field
    start    : A
    forward  : A → A
    backward : A → A
```

This gives an interface that allows us to move along the elements of a data type `A`.

For example, we can get the "third" element of a type `A`:

```
open Enumeration

3rd : {A : Set} → Enumeration A → A
3rd e = forward e (forward e (forward e (start e)))
```

Or we can go back 2 positions starting from a given `a`:

```
backward-2 : {A : Set} → Enumeration A → A → A
backward-2 e a = backward (backward a)
  where
    open Enumeration e
```

Now, we want to use these methods on natural numbers. For this, we need a record of type `Enumeration Nat`. Without copatterns, we would specify all the fields in a single expression:

```
open Enumeration

enum-Nat : Enumeration Nat
enum-Nat = record
  { start    = 0
  ; forward  = suc
  ; backward = pred
  }
  where
    pred : Nat → Nat
    pred zero    = zero
    pred (suc x) = x

test₁ : 3rd enum-Nat ≡ 3
test₁ = refl

test₂ : backward-2 enum-Nat 5 ≡ 3
test₂ = refl
```

Note that if we want to use automated case-splitting and pattern matching to implement one of the fields, we need to do so in a separate definition.

With *copatterns*, we can define the fields of a record as separate declarations, in the same way that we would give different cases for a function:

```
open Enumeration

enum-Nat : Enumeration Nat
start    enum-Nat = 0
forward  enum-Nat n = suc n
backward enum-Nat zero    = zero
backward enum-Nat (suc n) = n
```

The resulting behaviour is the same in both cases:

```
test₁ : 3rd enum-Nat ≡ 3
test₁ = refl

test₂ : backward-2 enum-Nat 5 ≡ 3
test₂ = refl
```

3.4.1 Copatterns in function definitions

In fact, we do not need to start at `0`. We can allow the user to specify the starting element.

Without copatterns, we just add the extra argument to the function declaration:

```
open Enumeration

enum-Nat : Nat → Enumeration Nat
enum-Nat initial = record
  { start    = initial
  ; forward  = suc
  ; backward = pred
  }
  where
    pred : Nat → Nat
    pred zero    = zero
    pred (suc x) = x

test₁ : 3rd (enum-Nat 10) ≡ 13
test₁ = refl
```

With copatterns, the function argument must be repeated once for each field in the record:

```
open Enumeration

enum-Nat : Nat → Enumeration Nat
start    (enum-Nat initial) = initial
forward  (enum-Nat _) n = suc n
backward (enum-Nat _) zero    = zero
backward (enum-Nat _) (suc n) = n
```

3.4.2 Mixing patterns and copatterns

Instead of allowing an arbitrary value, we want to limit the user to two choices: `0` or `42`.

Without copatterns, we would need an auxiliary definition to choose which value to start with based on the user-provided flag:

```
open Enumeration

if_then_else_ : {A : Set} → Bool → A → A → A
if true  then x else _ = x
if false then _ else y = y

enum-Nat : Bool → Enumeration Nat
enum-Nat ahead = record
  { start    = if ahead then 42 else 0
  ; forward  = suc
  ; backward = pred
  }
  where
    pred : Nat → Nat
    pred zero    = zero
    pred (suc x) = x
```

With copatterns, we can do the case analysis directly by pattern matching:

```
open Enumeration

enum-Nat : Bool → Enumeration Nat
start    (enum-Nat true)  = 42
start    (enum-Nat false) = 0
forward  (enum-Nat _) n = suc n
backward (enum-Nat _) zero    = zero
backward (enum-Nat _) (suc n) = n
```

Tip

When using copatterns to define an element of a record type, the fields of the record must be in scope. In the examples above, we use `open Enumeration` to bring the fields of the record into scope.

Consider the first example:

```
enum-Nat : Enumeration Nat
start    enum-Nat = 0
forward  enum-Nat n = suc n
backward enum-Nat zero    = zero
backward enum-Nat (suc n) = n
```

If the fields of the `Enumeration` record are not in scope (in particular, the `start` field), then Agda will not be able to figure out what the first copattern means:

```
Could not parse the left-hand side start enum-Nat
Operators used in the grammar:
None
when scope checking the left-hand side start enum-Nat in the
definition of enum-Nat
```

The solution is to open the record before using its fields:

```
open Enumeration

enum-Nat : Enumeration Nat
start    enum-Nat = 0
forward  enum-Nat n = suc n
backward enum-Nat zero    = zero
backward enum-Nat (suc n) = n
```

3.5 Core language

A program in Agda consists of a number of declarations written in an `*.agda` file. A declaration introduces a new identifier and gives its type and definition. It is possible to declare:

- *datatypes*
- *record types* (including *coinductive records*)
- *function definitions* (including *mixfix operators*, *abstract definitions*, and *opaque definitions*)
- *modules*
- local definitions *let* and *where*
- *postulates*
- *variables*
- *pattern-synonyms*
- *precedence* (fixity)
- *pragmas*, and
- *program options*

Declarations have a signature part and a definition part. These can appear separately in the program. Names must be declared before they are used, but by separating the signature from the definition it is possible to define things in *mutual recursion*.

3.5.1 Grammar

At its core, Agda is a dependently typed lambda calculus. The grammar of terms where `a` represents a generic term is:

```
a ::= x                        -- variable
    | λ x → a                   -- lambda abstraction
    | f                        -- defined function
    | (x : a) → a               -- function space
    | F                        -- data/record type
    | c a                      -- data/record constructor
    | s                        -- sort Seti, Setω+i
```

3.5.2 Syntax overview

The syntax of an Agda program is defined in terms of three key components:

- **Expressions** write function bodies and types.
- **Declarations** declare types, data-types, postulates, records, functions etc.
- **Pragmas** define program options.

There are also three main levels of syntax, corresponding to different levels of interpretation:

- **Concrete** is the high-level sugared syntax, it representing exactly what the user wrote (Agda.Syntax.Concrete).
- **Abstract**, before typechecking (Agda.Syntax.Abstract)
- **Internal**, the full-intepreted core Agda terms, typechecked; roughly corresponding to (Agda.Syntax.Internal).

The process of translating an `*.agda` file into an executable has several stages:

```
*.agda file
   ==[ parser (Lexer.x + Parser.y) ]==>
Concrete syntax
   ==[ nicifier (Syntax.Concrete.Definitions) ]==>
'Nice' concrete syntax
   ==[ scope checking (Syntax.Translation.ConcreteToAbstract) ]==>
Abstract syntax
   ==[ type checking (TypeChecking.Rules.*) ]==>
Internal syntax
   ==[ Agda.Compiler.ToTreeless ]==>
Treeless syntax
   ==[ different backends (Compiler.MAlonzo.*, Compiler.JS.*, ...) ]==>
Source code
   ==[ different compilers (GHC compiler, ...) ]==>
Executable
```

The following sections describe these stages in more detail:

3.5.3 Lexer

Lexical analysis (aka tokenization) is the process of converting a sequence of characters (the raw `*.agda` file) into a sequence of tokens (strings with a meaning).

The lexer in Agda is generated by Alex, and is an adaptation of GHC's lexer. The main lexing function `lexer` is called by the `Agda.Syntax.Parser.Parser` to get the next token from the input.

3.5.4 Parser

The parser is the component that takes the output of the lexer and builds a data structure that we will call Concrete Syntax, while checking for correct syntax.

The parser is generated by Happy.

Example: when a name is a sequence of parts, the lexer just sees it as a string, the parser does the translation in this step.

3.5.5 Concrete Syntax

The concrete syntax is a raw representation of the program text without any desugaring at all. This is what the parser produces. The idea is that if we figure out how to keep the concrete syntax around, it can be printed exactly as the user wrote it.

3.5.6 Nice Concrete Syntax

The `Nice Concrete Syntax` is a slightly reorganized version of the `Concrete Syntax` that is easier to deal with internally. Among other things, it:

- detects mutual blocks
- assembles *definitions* from their isolated parts
- collects fixity information of *mixfix operators* and attaches it to definitions
- emits warnings for possibly unintended but still valid declarations, which essentially is dead code such as empty `instance` blocks and misplaced *pragmas*

3.5.7 Abstract Syntax

The translation from `Agda.Syntax.Concrete` to `Agda.Syntax.Abstract` involves scope analysis, figuring out infix operator precedences and tidying up definitions.

The abstract syntax `Agda.Syntax.Abstract` is the result after desugaring and scope analysis of the concrete syntax. The type checker works on abstract syntax, producing internal syntax.

3.5.8 Internal Syntax

This is the final stage of syntax before being handed off to one of the backends. Terms are well-scoped and well-typed.

While producing the `Internal Syntax`, terms are checked for safety. This safety check means *termination check* and coverage check for functions, and *positivity check* for datatypes.

Type-directed operations such as *instance resolution* and disambiguation of overloaded constructors (different constructors with the same name) also happen here.

The internal syntax `Agda.Syntax.Internal` uses the following haskell datatype to represent the grammar of a `Term` presented above.

```
data Term = Var {-# UNPACK #-} !Int Elims -- ^ @x es@ neutral
          | Lam ArgInfo (Abs Term)        -- ^ Terms are beta normal. Relevance is ignored
          | Lit Literal
          | Def QName Elims               -- ^ @f es@, possibly a delta/iota-redex
          | Con ConHead ConInfo Elims
          -- ^ @c es@ or @record { fs = es }@
          --   @es@ allows only Apply and IApply eliminations,
          --   and IApply only for data constructors.
          | Pi (Dom Type) (Abs Type)      -- ^ dependent or non-dependent function space
          | Sort Sort
          | Level Level
          | MetaV {-# UNPACK #-} !MetaId Elims
```

3.5.9 Treeless Syntax

The treeless syntax is intended to be used as input for the *compiler backends*. It is more low-level than the internal syntax and is not used for type checking. Some of the features of the treeless syntax are:

- case expressions instead of case trees
- no instantiated datatypes / constructors

For instance, the *Glasgow Haskell Compiler (GHC) backend* translates the treeless syntax into a proper GHC Haskell program.

Another backend that may be used is the *JavaScript backend*, which translates the treeless syntax to JavaScript code.

The treeless representation of the program has A-normal form (ANF). That means that all the case expressions are targeting a *single* variable, and all alternatives may only peel off one constructor.

The backends can handle an ANF syntax easier than a syntax of a language where one may case arbitrary expressions and use *deep patterns*.

3.6 Coverage Checking

To ensure completeness of definitions by pattern matching, Agda performs a coverage check on each definition by pattern matching. This page explains how this coverage check works by starting from simple examples and building up to the general case.

3.6.1 Single match on a non-indexed datatype

When a *function definition* pattern matches on a single argument of a simple (i.e. non-indexed) *datatype*, there should be a clause for each constructor. For example:

```
data TrafficLight : Set where
  red yellow green : TrafficLight

go : TrafficLight → Bool
go red    = false
go yellow = false
go green  = true
```

Alternatively, one or more cases may be replaced by a *catchall clause* that uses a variable pattern or a wildcard pattern _. In this case, the catchall clause should be last.

```
go' : TrafficLight → Bool
go' green = true
go' _     = false
```

> **Note**
>
> When the *–exact-split* flag is enabled, catchall clauses should be marked explicitly by a *catchall pragma* (`{-# CATCHALL #-}`).

The coverage check can be turned off for an individual definition by putting a `{-# NON_COVERING #-}` pragma immediately in front of the type signature.

```
{-# NON_COVERING #-}
go'' : TrafficLight → Bool
go'' red   = false
go'' green = true
```

In the special case of a datatype with no constructors (i.e. an empty type), there should be a single *absurd clause* with an *absurd pattern* `()` and no right-hand side.

```
data ⊥ : Set where
  -- no constructors

magic : {A : Set} → ⊥ → A
magic ()
```

3.6.2 Matching on multiple arguments

If a function matches on several arguments, there should be a case for each possible combinations of constructors.

```
sameColor : TrafficLight → TrafficLight → Bool
sameColor red    red    = true
sameColor red    yellow = false
sameColor red    green  = false
sameColor yellow red    = false
sameColor yellow yellow = true
sameColor yellow green  = false
sameColor green  red    = false
sameColor green  yellow = false
sameColor green  green  = true
```

Again, one or more cases may be replaced by a catchall clause.

```
sameColor' : TrafficLight → TrafficLight → Bool
sameColor' red    red    = true
sameColor' yellow yellow = true
sameColor' green  green  = true
sameColor' _      _      = false
```

3.6.3 Copattern matching

Functions that return an element of a *record type* can use *copatterns* to give the individual fields. The coverage check will ensure that there is a single case for each field of the record type. For example:

```
record Person : Set where
  field
    name : String
    age  : Nat
open Person

bob : Person
name bob = "Bob"
age  bob = 25
```

Absurd copatterns or wildcard copatterns are not supported.

3.6.4 Matching on indexed datatypes

When a function definition matches on an argument of an indexed datatype, the following conditions should be satisfied:

- For each clause that matches on a constructor pattern `c u1 ... un`, the indices of the type of the pattern should be unifiable with the indices of the datatype being matched on.
- For each constructor `c` that does not appear in a clause, unification of the indices of the type of the constructor with the indices of the datatype should end in a conflict.

For example, consider the definition of the `head` function on vectors:

```
data Vec (A : Set) : Nat → Set where
  []  : Vec A 0
  _::_ : ∀ {n} → A → Vec A n → Vec A (suc n)

head : ∀ {A m} → Vec A (suc m) → A
head (x :: xs) = x
```

The type of the pattern `x :: xs` is `Vec A (suc n)`, which is unifiable with the type `Vec A (suc m)`. Meanwhile, unification of the type `Vec A 0` of the constructor `[]` with the type `Vec A (suc n)` results in a conflict between `0` and `suc n`, so there is no case for `[]`.

In case a function matches on several arguments and one or more of them are of indexed datatypes, only those combinations of arguments should be considered where the indices do not lead to a conflict. For example, consider the `zipWith` function on vectors:

```
zipWith : ∀ {A B C m} → (A → B → C) → Vec A m → Vec B m → Vec C m
zipWith f []       []       = []
zipWith f (x :: xs) (y :: ys) = f x y :: zipWith f xs ys
```

Since both input vectors have the same length `m`, there is are no cases for the combinations where one vector has length `0` and the other has length `suc n`.

In the special case where unification ends in a conflict for *all* constructors, there should be a single absurd clase (as for an empty type). For example:

```
data Fin : Nat → Set where
  zero : ∀ {n} → Fin (suc n)
  suc  : ∀ {n} → Fin n → Fin (suc n)

no-fin-zero : Fin 0 → ⊥
no-fin-zero ()
```

In many common cases, absurd clauses may be omitted as long as the remaining clauses reveal sufficient information to indicate what arguments to case split on. As an example, consider the definition of the `lookup` function for vectors:

```
lookup : ∀ {A} {n} → Vec A n → Fin n → A
lookup []       ()
lookup (x :: xs) zero    = x
lookup (x :: xs) (suc i) = lookup xs i
```

This definition pattern matches on both its (explicit) arguments in both the absurd clause and the two regular clauses. Hence it is allowed to leave out the absurd clause from the definition:

```
lookup' : ∀ {A} {n} → Vec A n → Fin n → A
lookup' (x :: xs) zero    = x
lookup' (x :: xs) (suc i) = lookup' xs i
```

Refer to the next section for a precise explanation of when an absurd clause may be omitted.

3.6.5 General case

In the general case, the coverage checker constructs a *case tree* from the definition given by the user. It then ensures that the following properties are satisfied:

- The non-absurd clauses of a definition should arise as the leaves of the case tree.
- The absurd clauses of a definition should arise as the internal nodes of the case tree that have no children.
- Absurd clauses may be omitted if removing the corresponding internal nodes from the case tree does not result in other internal nodes becoming childless.
- Non-absurd clauses may be replaced by catchall clauses if (1) the patterns of those catchall clauses are more general than the omitted clauses, (2) the added catchall clauses are not more general than any of the clauses that follow it, and (3) removing the leaves corresponding to the omitted clauses does not result in any internal nodes becoming childless.

As an example, consider the case tree for the definition of the `lookup` function defined above:

```
lookup xs i = case xs of
  []       → case i of {}
  (x ∷ xs) → case i of
    zero    → x
    (suc i) → lookup xs i
```

The absurd clause arises from the case split on `i` in the branch where `xs = []`, which leads to zero cases. The two normal clauses arise from the two leaves of the case tree. If the case `[] → case i of {}` is removed from the case tree, all the remaining internal nodes still have at least one child, hence the absurd clause may be left out of the definition.

For a full formal description of the algorithm that Agda uses to construct a case tree and check coverage of definitions by pattern matching, refer to the article Elaborating dependent (co)pattern matching: No pattern left behind.

3.7 Cubical

The Cubical mode extends Agda with a variety of features from Cubical Type Theory. In particular, it adds computational univalence and higher inductive types, hence giving computational meaning to Homotopy Type Theory and Univalent Foundations. The version of Cubical Type Theory that Agda implements is a variation of the *CCHM* Cubical Type Theory where the Kan composition operations are decomposed into homogeneous composition and generalized transport. This is what makes the general schema for higher inductive types work, following the *CHM* paper. There is also a research paper specifically about Cubical Agda at https://www.doi.org/10.1017/S0956796821000034.

To use the cubical mode Agda needs to be run with the `--cubical` command-line-option or with `{-# OPTIONS --cubical #-}` at the top of the file.

There are also two other *variants* of the cubical mode:

- `--cubical=erased`, which is described *below*, and
- `--cubical=no-glue`, which allows Cubical features without the *Glue types*, described *below*.

The cubical mode adds the following features to Agda:

1. An interval type and path types
2. Generalized transport (`transp`)
3. Partial elements

4. Homogeneous composition (`hcomp`)
5. Glue types
6. Higher inductive types
7. Cubical identity types

There are two major libraries for Cubical Agda:

- `agda/cubical`: originally intended as a standard library for Cubical Agda available at https://github.com/agda/cubical. This documentation uses the naming conventions of this library, for a detailed list of all of the built-in Cubical Agda files and primitives see *Appendix: Cubical Agda primitives*.
- `1lab`: A formalised and cross linked reference resource for cubical methods in Homotopy Type Theory which can be found at https://1lab.dev/. Much better documented than the `agda/cubical` library and hence more accessible to newcomers. The sources can be found at https://github.com/plt-amy/1lab.

In this documentation we will rely on the `agda/cubical` library and the recommended way to get access to the cubical primitives is to add the following to the top of a file (this assumes that the `agda/cubical` library is installed and visible to Agda).

```
{-# OPTIONS --cubical #-}

open import Cubical.Core.Primitives
open import Cubical.Core.Glue
```

Follow the instructions at https://github.com/agda/cubical to install the library. In order to make this library visible to Agda add `/path/to/cubical/cubical.agda-lib` to `.agda/libraries` and `cubical` to `.agda/defaults` (where `path/to` is the absolute path to where the `agda/cubical` library has been installed). For details of Agda's library management see *Library Management*.

Expert users who do not want to rely on `agda/cubical` can just add the relevant import statements at the top of their file (for details see *Appendix: Cubical Agda primitives*). However, for beginners it is recommended that one uses at least the core part of the `agda/cubical` library.

3.7.1 The interval and path types

The key idea of Cubical Type Theory is to add an interval type `I : IUniv` (the reason this is in a special sort `IUniv` is because it doesn't support the `transp` and `hcomp` operations). A variable `i : I` intuitively corresponds to a point in the real unit interval. In an empty context, there are only two values of type `I`: the two endpoints of the interval, `i0` and `i1`.

```
i0 : I
i1 : I
```

Elements of the interval form a De Morgan algebra, with minimum ($\wedge$), maximum ($\vee$) and negation ($\sim$).

```
_∧_ : I → I → I
_∨_ : I → I → I
~_ : I → I
```

All the properties of De Morgan algebras hold definitionally. The endpoints of the interval `i0` and `i1` are the bottom and top elements, respectively.

```
i0 ∨ i  = i
i1 ∨ i  = i1
i  ∨ j  = j ∨ i
```

(continues on next page)

(continued from previous page)

```
i0 ∧ i    = i0
i1 ∧ i    = i
i  ∧ j    = j ∧ i
~ (~ i)   = i
i0        = ~ i1
~ (i ∨ j) = ~ i ∧ ~ j
~ (i ∧ j) = ~ i ∨ ~ j
```

The core idea of Homotopy Type Theory and Univalent Foundations is a correspondence between paths (as in topology) and (proof-relevant) equalities (as in Martin-Löf's identity type). This correspondence is taken very literally in Cubical Agda where a path in a type `A` is represented like a function out of the interval, `I → A`. A path type is in fact a special case of the more general built-in heterogeneous path types:

```
-- PathP : ∀ {ℓ} (A : I → Set ℓ) → A i0 → A i1 → Set ℓ

-- Non dependent path types
Path : ∀ {ℓ} (A : Set ℓ) → A → A → Set ℓ
Path A a b = PathP (λ _ → A) a b
```

The central notion of equality in Cubical Agda is hence heterogeneous equality (in the sense of `PathOver` in HoTT). To define paths we use λ-abstractions and to apply them we use regular application. For example, this is the definition of the constant path (or proof of reflexivity):

```
refl : ∀ {ℓ} {A : Set ℓ} {x : A} → Path A x x
refl {x = x} = λ i → x
```

Although they use the same syntax, a path is not exactly the same as a function. For example, typed lambdas cannot be used to form paths, they are reserved for functions:

```
not-refl : ∀ {ℓ} {A : Set ℓ} {x : A} → (i : I) → A
not-refl {x = x} = λ (i : I) → x
```

Because of the intuition that paths correspond to equality `PathP (λ i → A) x y` gets printed as `x ≡ y` when `A` does not mention `i`. By iterating the path type we can define squares, cubes, and higher cubes in Agda, making the type theory cubical. For example a square in `A` is built out of 4 points and 4 lines:

```
Square : ∀ {ℓ} {A : Set ℓ} {x0 x1 y0 y1 : A} →
           x0 ≡ x1 → y0 ≡ y1 → x0 ≡ y0 → x1 ≡ y1 → Set ℓ
Square p q r s = PathP (λ i → p i ≡ q i) r s
```

Viewing equalities as functions out of the interval makes it possible to do a lot of equality reasoning in a very direct way:

```
sym : ∀ {ℓ} {A : Set ℓ} {x y : A} → x ≡ y → y ≡ x
sym p = λ i → p (~ i)

cong : ∀ {ℓ} {A : Set ℓ} {x y : A} {B : A → Set ℓ} (f : (a : A) → B a) (p : x ≡ y)
       → PathP (λ i → B (p i)) (f x) (f y)
cong f p i = f (p i)
```

Because of the way functions compute these satisfy some new definitional equalities compared to the standard Agda definitions:

```
symInv : ∀ {ℓ} {A : Set ℓ} {x y : A} (p : x ≡ y) → sym (sym p) ≡ p
symInv p = refl

congId : ∀ {ℓ} {A : Set ℓ} {x y : A} (p : x ≡ y) → cong (λ a → a) p ≡ p
congId p = refl

congComp : ∀ {ℓ} {A B C : Set ℓ} (f : A → B) (g : B → C) {x y : A} (p : x ≡ y) →
           cong (λ a → g (f a)) p ≡ cong g (cong f p)
congComp f g p = refl
```

Path types also let us prove new things that are not provable in standard Agda. For example, function extensionality, stating that pointwise equal functions are equal, has an extremely simple proof:

```
funExt : ∀ {ℓ} {A : Set ℓ} {B : A → Set ℓ} {f g : (x : A) → B x} →
         ((x : A) → f x ≡ g x) → f ≡ g
funExt p i x = p x i
```

3.7.2 Transport

While path types are great for reasoning about equality they do not let us transport along paths between types or even compose paths, which in particular means that we cannot yet prove the induction principle for paths. As a remedy, we also have a built-in (generalized) transport operation `transp` and homogeneous composition operations `hcomp`. The transport operation is generalized in the sense that it lets us specify where it is the identity function.

```
transp : ∀ {ℓ} (A : I → Set ℓ) (r : I) (a : A i0) → A i1
```

There is an additional side condition to be satisfied for a usage of `transp` to type-check: `A` should be a constant function whenever the constraint `r = i1` is satisfied. By constant here we mean that `A` is definitionally equal to `λ _ → A i0`, which in turn requires `A i0` and `A i1` to be definitionally equal as well.

When `r` is `i1`, `transp A r` will compute as the identity function.

```
transp A i1 a = a
```

This is only sound if in such a case `A` is a trivial path, as the side condition requires.

It might seems strange that the side condition expects `r` and `A` to interact, but both of them can depend on any of the interval variables in scope, so assuming a specific value for `r` can affect what `A` looks like.

Some examples of the side condition for different values of `r`:

- If `r` is some in-scope variable `i`, on which `A` may depend as well, then `A` only needs to be a constant function when substituting `i1` for `i`.
- If `r` is `i0` then there is no restrition on `A`, since the side condition is vacuously true.
- If `r` is `i1` then `A` must be a constant function.

We can use `transp` to define regular transport:

```
transport : ∀ {ℓ} {A B : Set ℓ} → A ≡ B → A → B
transport p a = transp (λ i → p i) i0 a
```

By combining the transport and min operations we can define the induction principle for paths:

```
J : ∀ {ℓ} {A : Set ℓ} {x : A} (P : ∀ y → x ≡ y → Set ℓ)
      (d : P x refl) {y : A} (p : x ≡ y)
    → P y p
J P d p = transport (λ i → P (p i) (λ j → p (i ∧ j))) d
```

One subtle difference between paths and the propositional equality type of Agda is that the computation rule for `J` does not hold definitionally. If `J` is defined using pattern matching as in the Agda standard library then this holds, however as the path types are not inductively defined this does not hold for the above definition of `J`. In particular, transport in a constant family is only the identity function up to a path which implies that the computation rule for `J` only holds up to a path:

```
transportRefl : ∀ {ℓ} {A : Set ℓ} (x : A) → transport refl x ≡ x
transportRefl {A = A} x i = transp (λ _ → A) i x

JRefl : ∀ {ℓ} {A : Set ℓ} {x : A} (P : ∀ y → x ≡ y → Set ℓ)
        (d : P x refl) → J P d refl ≡ d
JRefl P d = transportRefl d
```

Internally in Agda the `transp` operation computes by cases on the type, so for example for Σ-types it is computed elementwise. For path types it is however not yet possible to provide the computation rule as we need some way to remember the endpoints of the path after transporting it. Furthermore, this must work for arbitrary higher dimensional cubes (as we can iterate the path types). For this we introduce the "homogeneous composition operations" (`hcomp`) that generalize binary composition of paths to n-ary composition of higher dimensional cubes.

3.7.3 Partial elements

In order to describe the homogeneous composition operations we need to be able to write partially specified n-dimensional cubes (i.e. cubes where some faces are missing). Given an element of the interval `r : I` there is a predicate `IsOne` which represents the constraint `r = i1`. This comes with a proof that `i1` is in fact equal to `i1` called `1=1 : IsOne i1`. We use Greek letters like φ or ψ when such an `r` should be thought of as being in the domain of `IsOne`.

Using this we introduce a type of partial elements called `Partial φ A`. The idea is that `Partial φ A` is the type of cubes in `A` that are only defined when `IsOne φ` holds. `Partial φ A` is a special version of `IsOne φ → A` with a more extensional judgmental equality: Two elements of `Partial φ A` are considered equal if they represent the same subcube; so, the faces of the cubes can for example be given in different order yet the two elements will still be considered the same.

There is also a dependent version of `Partial φ A` called `PartialP φ A` which requires `A` only to be defined when `IsOne φ`.

```
Partial : ∀ {ℓ} → I → Set ℓ → SSet ℓ

PartialP : ∀ {ℓ} → (φ : I) → Partial φ (Set ℓ) → SSet ℓ
```

There is a new form of pattern matching that can be used to introduce partial elements:

```
partialBool : ∀ i → Partial (i ∨ ~ i) Bool
partialBool i (i = i0) = true
partialBool i (i = i1) = false
```

The term `partialBool i` should be thought of a boolean with different values when `(i = i0)` and `(i = i1)`. Terms of type `Partial φ A` can also be introduced using a *Pattern lambda*.

```
partialBool' : ∀ i → Partial (i ∨ ~ i) Bool
partialBool' i = λ where
  (i = i0) → true
  (i = i1) → false
```

When the cases overlap they must agree:

```
partialBool'' : ∀ i j → Partial (~ i ∨ i ∨ (i ∧ j)) Bool
partialBool'' i j = λ where
  (i = i1)          → true
  (i = i1) (j = i1) → true
  (i = i0)          → false
```

Note that the order of the cases does not have to match the interval formula exactly.

Furthermore, `IsOne i0` is actually absurd:

```
empty : {A : Set} → Partial i0 A
empty = λ()
```

Cubical Agda also has cubical subtypes as in the CCHM type theory:

```
_[_↦_] : ∀ {ℓ} (A : Set ℓ) (φ : I) (u : Partial φ A) → SSet ℓ
A [ φ ↦ u ] = Sub A φ u
```

A term `v : A [ φ ↦ u ]` should be thought of as a term of type `A` which is definitionally equal to `u : A` when `IsOne φ` is satisfied. Any term `u : A` can be seen as an term of `A [ φ ↦ u ]` which agrees with itself on φ:

```
inS : ∀ {ℓ} {A : Set ℓ} {φ : I} (u : A) → A [ φ ↦ (λ _ → u) ]
```

One can also forget that a partial element agrees with `u` on φ:

```
outS : ∀ {ℓ} {A : Set ℓ} {φ : I} {u : Partial φ A} → A [ φ ↦ u ] → A
```

These coercions satisfy the following equalities:

```
outS (inS a) = a

inS {φ = φ} (outS {φ = φ} a) = a

outS {φ = i1} {u} _ = u 1=1
```

Note that given `a : A [ φ ↦ u ]` and α `: IsOne φ`, it is not the case that `outS a = u α`; however, underneath the pattern binding `(φ = i1)`, one has `outS a = u 1=1`.

With all of this cubical infrastructure we can now describe the `hcomp` operations.

3.7.4 Homogeneous composition

The homogeneous composition operations generalize binary composition of paths so that we can compose multiple composable cubes.

```
hcomp : ∀ {ℓ} {A : Set ℓ} {φ : I} (u : I → Partial φ A) (u0 : A) → A
```

When calling `hcomp {φ = φ} u u0` Agda makes sure that `u0` agrees with `u i0` on φ. The idea is that `u0` is the base and `u` specifies the sides of an open box. This is hence an open (higher dimensional) cube where the side opposite of

`u0` is missing. The `hcomp` operation then gives us the missing side opposite of `u0`. For example binary composition of paths can be written as:

```
compPath : ∀ {ℓ} {A : Set ℓ} {x y z : A} → x ≡ y → y ≡ z → x ≡ z
compPath {x = x} p q i = hcomp (λ{ j (i = i0) → x
                                ; j (i = i1) → q j })
                               (p i)
```

Pictorially we are given `p : x ≡ y` and `q : y ≡ z`, and the composite of the two paths is obtained by computing the missing lid of this open square:

```
  x             z
  ^             ^
  |             |
x |             | q j
  |             |
  x ----------> y
       p i
```

In the drawing the direction `i` goes left-to-right and `j` goes bottom-to-top. As we are constructing a path from `x` to `z` along `i` we have `i : I` in the context already and we put `p i` as bottom. The direction `j` that we are doing the composition in is abstracted in the first argument to `hcomp`.

Note that the partial element `u` does not have to specify all the sides of the open box, giving more sides simply gives you more control on the result of `hcomp`. For example if we omit the `(i = i0) → x` side in the definition of `compPath` we still get a valid term of type `A`. However, that term would reduce to `hcomp (λ{ j () }) x` when `i = i0` and so that definition would not build a path that starts from `x`.

We can also define homogeneous filling of cubes as

```
hfill : ∀ {ℓ} {A : Set ℓ} {φ : I}
        (u : ∀ i → Partial φ A) (u0 : A [ φ ↦ u i0 ])
        (i : I) → A
hfill {φ = φ} u u0 i = hcomp (λ{ j (φ = i1) → u (i ∧ j) 1=1
                               ; j (i = i0) → outS u0 })
                             (outS u0)
```

When `i` is `i0` this is `u0` and when `i` is `i1` this is `hcomp u u0`. This can hence be seen as giving us the interior of an open box. In the special case of the square above `hfill` gives us a direct cubical proof that composing `p` with `refl` is `p`.

```
compPathRefl : ∀ {ℓ} {A : Set ℓ} {x y : A} (p : x ≡ y) → compPath p refl ≡ p
compPathRefl {x = x} {y = y} p j i = hfill (λ{ _ (i = i0) → x
                                             ; _ (i = i1) → y })
                                           (inS (p i))
                                           (~ j)
```

3.7.5 Glue types

In order to be able to prove the univalence theorem we also have to add "Glue" types. These lets us turn equivalences between types into paths between types. An equivalence of types `A` and `B` is defined as a map `f : A → B` such that its fibers are contractible.

```
fiber : ∀ {ℓ} {A B : Set ℓ} (f : A → B) (y : B) → Set ℓ
fiber {A = A} f y = Σ[ x ∈ A ] f x ≡ y
```

(continues on next page)

(continued from previous page)

```
isContr : ∀ {ℓ} → Set ℓ → Set ℓ
isContr A = Σ[ x ∈ A ] (∀ y → x ≡ y)

record isEquiv {ℓ} {A B : Set ℓ} (f : A → B) : Set ℓ where
  field
    equiv-proof : (y : B) → isContr (fiber f y)

_≃_ : ∀ {ℓ} (A B : Set ℓ) → Set ℓ
A ≃ B = Σ[ f ∈ (A → B) ] (isEquiv f)
```

The simplest example of an equivalence is the identity function.

```
idfun : ∀ {ℓ} → (A : Set ℓ) → A → A
idfun _ x = x

idIsEquiv : ∀ {ℓ} (A : Set ℓ) → isEquiv (idfun A)
equiv-proof (idIsEquiv A) y =
  ((y , refl) , λ z i → z .snd (~ i) , λ j → z .snd (~ i ∨ j))

idEquiv : ∀ {ℓ} (A : Set ℓ) → A ≃ A
idEquiv A = (idfun A , idIsEquiv A)
```

An important special case of equivalent types are isomorphic types (i.e. types with maps going back and forth which are mutually inverse).

As everything has to work up to higher dimensions the Glue types take a partial family of types that are equivalent to the base type `A`:

```
Glue : ∀ {ℓ ℓ'} (A : Set ℓ) {φ : I}
     → Partial φ (Σ[ T ∈ Set ℓ' ] T ≃ A) → Set ℓ'
```

These come with a constructor and eliminator:

```
glue : ∀ {ℓ ℓ'} {A : Set ℓ} {φ : I} {Te : Partial φ (Σ[ T ∈ Set ℓ' ] T ≃ A)}
     → PartialP φ T → A → Glue A Te

unglue : ∀ {ℓ ℓ'} {A : Set ℓ} (φ : I) {Te : Partial φ (Σ[ T ∈ Set ℓ' ] T ≃ A)}
       → Glue A Te → A
```

Using Glue types we can turn an equivalence of types into a path as follows:

```
ua : ∀ {ℓ} {A B : Set ℓ} → A ≃ B → A ≡ B
ua {_} {A} {B} e i = Glue B λ{ (i = i0) → (A , e)
                             ; (i = i1) → (B , idEquiv B) }
```

The idea is that we glue `A` together with `B` when `i = i0` using `e` and `B` with itself when `i = i1` using the identity equivalence. This hence gives us the key part of univalence: a function for turning equivalences into paths. The other part of univalence is that this map itself is an equivalence which follows from the computation rule for `ua`:

```
uaβ : ∀ {ℓ} {A B : Set ℓ} (e : A ≃ B) (x : A) → transport (ua e) x ≡ e .fst x
uaβ e x = transportRefl (e .fst x)
```

Transporting along the path that we get from applying `ua` to an equivalence is hence the same as applying the equiva-

lence. This is what makes it possible to use the univalence axiom computationally in Cubical Agda: we can package up our equivalences as paths, do equality reasoning using these paths, and in the end transport along the paths in order to compute with the equivalences.

We have the following equalities:

```
Glue A {i1} Te = Te 1=1 .fst

unglue φ (glue t a) = a

glue (λ{ (φ = i1) → g }) (unglue φ g) = g

unglue i1 {Te} g = Te 1=1 .snd .fst g

glue {φ = i1} t a = t 1=1
```

For more results about Glue types and univalence see the files of Glue types and univalence in the `agda/cubical` library or the `1lab`.

3.7.6 Higher inductive types

Cubical Agda also lets us directly define higher inductive types as datatypes with path constructors. For example the circle and torus can be defined as:

```
data S¹ : Set where
  base : S¹
  loop : base ≡ base

data Torus : Set where
  point : Torus
  line1 : point ≡ point
  line2 : point ≡ point
  square : PathP (λ i → line1 i ≡ line1 i) line2 line2
```

Functions out of higher inductive types can then be defined using pattern matching:

```
t2c : Torus → S¹ × S¹
t2c point        = (base   , base)
t2c (line1 i)    = (loop i , base)
t2c (line2 j)    = (base   , loop j)
t2c (square i j) = (loop i , loop j)

c2t : S¹ × S¹ → Torus
c2t (base   , base)   = point
c2t (loop i , base)   = line1 i
c2t (base   , loop j) = line2 j
c2t (loop i , loop j) = square i j
```

When giving the cases for the path and square constructors we have to make sure that the function maps the boundary to the right thing. For instance the following definition does not pass Agda's typechecker as the boundary of the last case does not match up with the expected boundary of the square constructor (as the `line1` and `line2` cases are mixed up).

```
c2t_bad : S¹ × S¹ → Torus
c2t_bad (base   , base)   = point
```

(continues on next page)

(continued from previous page)

```
c2t_bad (loop i , base)   = line2 i
c2t_bad (base   , loop j) = line1 j
c2t_bad (loop i , loop j) = square i j
```

Functions defined by pattern matching on higher inductive types compute definitionally, for all constructors.

```
c2t-t2c : ∀ (t : Torus) → c2t (t2c t) ≡ t
c2t-t2c point        = refl
c2t-t2c (line1 _)    = refl
c2t-t2c (line2 _)    = refl
c2t-t2c (square _ _) = refl

t2c-c2t : ∀ (p : S¹ × S¹) → t2c (c2t p) ≡ p
t2c-c2t (base   , base)   = refl
t2c-c2t (base   , loop _) = refl
t2c-c2t (loop _ , base)   = refl
t2c-c2t (loop _ , loop _) = refl
```

By turning this isomorphism into an equivalence we get a direct proof that the torus is equal to two circles.

```
Torus≡S¹×S¹ : Torus ≡ S¹ × S¹
Torus≡S¹×S¹ = isoToPath (iso t2c c2t t2c-c2t c2t-t2c)
```

A type is a *proposition* if all of its elements are connected by a path:

```
IsProp : ∀ {ℓ} → Set ℓ → Set ℓ
IsProp A = (x y : A) → x ≡ y
```

Cubical Agda also supports parameterized and recursive higher inductive types, for example propositional truncation (squash types) is defined as:

```
data ∥_∥ {ℓ} (A : Set ℓ) : Set ℓ where
  |_| : A → ∥ A ∥
  squash : ∀ (x y : ∥ A ∥) → x ≡ y

recPropTrunc : ∀ {ℓ} {A : Set ℓ} {P : Set ℓ} → IsProp P → (A → P) → ∥ A ∥ → P
recPropTrunc Pprop f ∣ x ∣          = f x
recPropTrunc Pprop f (squash x y i) =
  Pprop (recPropTrunc Pprop f x) (recPropTrunc Pprop f y) i
```

Erased constructors

In combination with *--erasure* it can make sense to mark constructors as erased, in particular higher constructors such as `squash`:

```
data ∥_∥ {ℓ} (A : Set ℓ) : Set ℓ where
  |_| : A → ∥ A ∥
  @0 squash : ∀ (x y : ∥ A ∥) → x ≡ y

recPropTrunc : ∀ {ℓ} {A : Set ℓ} {P : Set ℓ} → @0 IsProp P → (A → P) → ∥ A ∥ → P
recPropTrunc Pprop f ∣ x ∣          = f x
recPropTrunc Pprop f (squash x y i) =
  Pprop (recPropTrunc Pprop f x) (recPropTrunc Pprop f y) i
```

In the code above the constructor `squash` is only available at compile-time, whereas `|_|` is also available at run-time. Clauses that match on erased constructors in non-erased positions are omitted by (at least some) compiler backends, so one can use erased names in the bodies of such clauses. (There is an exception for constructors that were not originally declared as erased, but that are currently treated as erased.)

For many more examples of higher inductive types see the `agda/cubical` library or the `1lab`.

3.7.7 Indexed inductive types

Cubical Agda has experimental support for the `transp` primitive when used to substitute the indices of an indexed inductive type. A handful of definitions (satisfying a technical restriction on their pattern matching) will compute when applied to a transport along indices. As an example of what works, let us consider the following running example:

```
data Eq {a} {A : Set a} (x : A) : A → Set a where
  reflEq : Eq x x

data Vec {a} (A : Set a) : Nat → Set a where
  []  : Vec A zero
  _::_ : ∀ {n} → A → Vec A n → Vec A (suc n)
```

Functions which match on `Eq` when all of its endpoints are variables, that is, very generic lemmas like `symEq` and `transpEq` below, will compute on all cases: they will compute to the given right-hand-side definitionally when their argument is `reflEq`, and will compute to a transport in the codomain when their argument has been transported in the second variable.

```
symEq : ∀ {a} {A : Set a} {x y : A} → Eq x y → Eq y x
symEq reflEq = reflEq

transpEq : ∀ {a} {A B : Set a} → Eq A B → A → B
transpEq reflEq x = x

pathToEq : ∀ {a} {A : Set a} {x y : A} → x ≡ y → Eq x y
pathToEq {x = x} p = transp (λ i → Eq x (p i)) i0 reflEq

module _ {a} {A B : Set a} {x y : A} {f : A ≃ B} where
  _ : symEq (reflEq {x = x}) ≡ reflEq
  _ = refl

  _ : transpEq (pathToEq (ua (idEquiv Bool))) ≡ λ x → x
  _ = refl
```

Matching on indexed types in situations where types are assumed (so their transports are also open) often generates many more transports than the comparable construction with paths would. As an example, compare the proof of `ua`β`Eq` below has four pending transports, whereas `ua`β only has one!

```
uaβEq : transpEq (pathToEq (ua f)) ≡ f .fst
uaβEq = funExt λ z →
  compPath (transportRefl (f .fst _))
    (cong (f .fst) (compPath
      (transportRefl _)
      (compPath
        (transportRefl _)
        (transportRefl _))))
```

In more concrete situations, such as when the indices are constructors of some other inductive type, pattern matching

definitions will not compute when applied to transports. For specific unsupported cases, see *What works, and what doesn't*.

If the `UnsupportedIndexedMatch` warning is enabled (it is by default), Agda will print a warning for every definition whose computational behaviour could not be extended to cover transports. Internally, transports are represented by an additional constructor, and pattern matching definitions must be extended to cover these constructors. To do this, the results of pattern matching unification must be translated into an embedding (in the HoTT sense). **This is work-in-progress.**

For the day-to-day use of Cubical Agda, it is advisable to disable the `UnsupportedIndexedMatch` warnings. You can do this using the `-WnoUnsupportedIndexedMatch` option in an `OPTIONS` pragma or in your `agda-lib` file.

What works, and what doesn't

This section lists some of the common cases where pattern matching unification produces something that can not be extended to cover transports, and the cases in which it can.

The following pair of definitions relies on injectivity for data constructors (specifically of the constructor `suc`), and so will not compute on transported values.

```
sucInjEq : ∀ {n k} → Eq (suc n) (suc k) → Eq n k
sucInjEq reflEq = reflEq

head : ∀ {n} {a} {A : Set a} → Vec A (suc n) → A
head (x ∷ _) = x
```

To demonstrate the failure of computation, we can set up the following artificial example using `head`. By passing the vector `true ∷ []` through two transports, even if they would cancel out, `head`'s computation gets stuck.

```
module _ (n : Nat) (p : n ≡ 1) where private
  vec : Vec Bool n
  vec = transport (λ i → Vec Bool (p (~ i))) (true ∷ [])

  hd : Bool
  hd = head (transport (λ i → Vec Bool (p i)) vec)

-- Does not type-check:
-- _ : hd ≡ true
-- _ = refl
-- Instead, hd is some big expression involving head applied to a
-- transport
```

If a definition is stuck on a transport, often the best workaround is to avoid treating it like the reducible expression it should be, and managing the transports yourself. For example, using the proof that `transport (sym p) (transport p x) ≡ x`, we can compute with `hd` up to a path, even if it's definitionally stuck.

```
-- Continuing from above..

  _ : hd ≡ true
  _ = cong head (transport⁻Transport (λ i → Vec Bool (p (~ i))) (true ∷ []))
```

In other cases, it may be possible to rephrase the proof in ways that avoid unsupported cases in pattern matching, and so, compute. For example, returning to `sucInj`, we can define it in terms of `apEq` (which always computes), and the fact that `suc` has a partially-defined inverse:

```
apEq : ∀ {a b} {A : Set a} {B : Set b} (f : A → B) {x y : A}
     → Eq x y → Eq (f x) (f y)
apEq f reflEq = reflEq

sucInjEq′ : ∀ {n k} → Eq (suc n) (suc k) → Eq n k
sucInjEq′ = apEq λ{ (suc n) → n ; zero → zero }
```

Definitions which rely on principles incompatible with Cubical Agda (K, injectivity of type constructors) will never compute on transports. Note that enabling both Cubical and K is not compatible with *--safe*.

Absurd clauses do not need any special handling (since the transport of an absurdity is still absurd), so definitions which rely on Agda's ability to automatically separate constructors of inductive types will not generate a `UnsupportedIndexedMatch` warning.

```
zeroNotSucEq : ∀ {n} {a} {A : Set a} → Eq zero (suc n) → A
zeroNotSucEq ()
```

Definitions whose elaboration involves using an equality derived from pattern matching in a type in `Setω` can not be extended yet. The following example is very artificial because it minimises an example from the Cubical library. The point is that to extend `test` to cover transports, we would need to, given `p : ℓ′ ≡ ℓ`, produce a `PathP (λ i → Argh ℓ (p i)) _ _`, but `Setω` is not considered fibrant yet.

```
data Argh (ℓ : Level) : Level → Setω where
  argh : ∀ {ℓ′} → Argh ℓ ℓ′ → Argh ℓ ℓ′

test : ∀ {ℓ ℓ′} → Argh ℓ ℓ′ → Bool
test {ℓ} (argh _) = true
```

Modalities & indexed matching

When using indexed matching in Cubical Agda, clauses' arguments (and their right-hand-sides) need to be transported to account for indexing, meaning that the *types* of those arguments must be well-formed *terms*.

For example, the following code is forbidden in Cubical Agda, and when `--without-K` is enabled:

```
subst : (@0 P : A → Set p) → x ≡ y → P x → P y
subst _ refl p = p
```

This is because the predicate `P` is erased, but internally, we have to transport along the argument `p` along a path involving `P`, in a relevant position.

Any argument which is used in the result type, or appears after a forced (dot) pattern, must have a modality-correct type.

3.7.8 Variants

Summary of variant compatibilities:

Current \ Imported	*--cubical=no-glue*	*--cubical=erased*	*--cubical[=full]*
--cubical=no-glue	✓	×	×
--cubical=erased	✓	✓	✓[1]
--cubical[=full]	✓	✓	✓

[1] only if *--erasure* is enabled and is used in erased positions. See *below*.

Cubical Agda with erased Glue

The option *--cubical=erased* enables a variant of Cubical Agda in which Glue (and the other builtins defined in `Agda.Builtin.Cubical.Glue`) must only be used in *erased* settings.

Regular Cubical Agda code can import code that uses *--cubical=erased*. Regular Cubical Agda code can also be imported from code that uses *--cubical=erased*, but names defined using Cubical Agda can only be used if the option *--erasure* is used. In that case the names are treated as if they had been marked as erased, with an exception related to pattern matching:

- Matching on a non-erased imported constructor does not, on its own, make Agda treat the right-hand side as erased.

The reason for this exception is that it should be possible to import the code from modules that use *--cubical*, in which the non-erased constructors are not treated as erased.

Note that names that are re-exported from a Cubical Agda module using `open import M args public` are seen as defined using Cubical Agda.

Cubical Agda without Glue

The option *--cubical=no-glue* enables a variant (strict subset) of Cubical Agda, in which primitives such as `hcomp` and `transp` are still available, but Glue types (and the other builtins defined in `Agda.Builtin.Cubical.Glue`) are disabled. Therefore, it should be sound to postulate in this variant either the uniqueness of identity proofs (UIP) or univalence; but of course not both. A source of inspiration for a Cubical Type Theory compatible with UIP is *XTT*, in which UIP holds definitionally.

If the current module enables the option *--cubical=no-glue*, then:

- It cannot import from modules with the options *--cubical* or *--cubical=erased*, since they allow the use of Glue types (to different extents).
- Modules that depend on the current module must enable any of the Cubical (variant) options: *--cubical=no-glue*, *--cubical=erased*, or *--cubical*.

On the other hand, if the current module enables any of the the opions *--cubical=erased* or *--cubical*, one can always import modules with *--cubical=no-glue*.

3.7.9 References

Cyril Cohen, Thierry Coquand, Simon Huber and Anders Mörtberg; "Cubical Type Theory: a constructive interpretation of the univalence axiom".

Thierry Coquand, Simon Huber, Anders Mörtberg; "On Higher Inductive Types in Cubical Type Theory".

Jonathan Sterling, Carlo Angiuli, Daniel Gratzer; "A Cubical Language for Bishop Sets".

3.7.10 Appendix: Cubical Agda primitives

The Cubical Agda primitives and internals are exported by a series of files found in the `lib/prim/Agda/Builtin/Cubical` directory of Agda. The `agda/cubical` library exports all of these primitives with the names used throughout this document. Experts might find it useful to know what is actually exported as there are quite a few primitives available that are not really exported by `agda/cubical`, so the goal of this section is to list the contents of these files. However, for regular users and beginners the `agda/cubical` library should be sufficient and this section can safely be ignored.

Warning: Many of the built-ins whose definitions can be written in Agda are nonetheless used internally in the implementation of cubical Agda, and using different implementations can easily lead to unsoundness. Even though they are definable in user code, this is not a supported use-case.

The key file with primitives is `Agda.Primitive.Cubical`. It exports the following `BUILTIN`, primitives and postulates:

```
{-# BUILTIN CUBEINTERVALUNIV IUniv #-}  -- IUniv : SSet₁
{-# BUILTIN INTERVAL I  #-}  -- I : IUniv
{-# BUILTIN IZERO    i0   #-}
{-# BUILTIN IONE     i1   #-}

infix 30 primINeg
infixr 20 primIMin primIMax

primitive
  primIMin : I → I → I   -- _∧_
  primIMax : I → I → I   -- _∨_
  primINeg : I → I       -- ~_

{-# BUILTIN ISONE IsOne #-} -- IsOne : I → SSet

postulate
  itIsOne : IsOne i1     -- 1=1
  IsOne1  : ∀ i j → IsOne i → IsOne (primIMax i j)
  IsOne2  : ∀ i j → IsOne j → IsOne (primIMax i j)

{-# BUILTIN ITISONE      itIsOne  #-}
{-# BUILTIN ISONE1       IsOne1   #-}
{-# BUILTIN ISONE2       IsOne2   #-}
{-# BUILTIN PARTIAL      Partial  #-}
{-# BUILTIN PARTIALP     PartialP #-}

postulate
  isOneEmpty : ∀ {a} {A : Partial i0 (Set a)} → PartialP i0 A
{-# BUILTIN ISONEEMPTY isOneEmpty #-}

primitive
  primPOr : ∀ {a} (i j : I) {A : Partial (primIMax i j) (Set a)}
          → PartialP i (λ z → A (IsOne1 i j z)) → PartialP j (λ z → A (IsOne2 i j z))
          → PartialP (primIMax i j) A

  -- Computes in terms of primHComp and primTransp
  primComp : ∀ {a} (A : (i : I) → Set (a i)) {φ : I} → (∀ i → Partial φ (A i)) → (a␣
↪: A i0) → A i1

syntax primPOr p q u t = [ p ↦ u , q ↦ t ]

primitive
  primTransp : ∀ {a} (A : (i : I) → Set (a i)) (φ : I) → (a : A i0) → A i1
  primHComp : ∀ {a} {A : Set a} {φ : I} → (∀ i → Partial φ A) → A → A
```

The interval `I` belongs to its own sort, `IUniv`. Types in this sort do not support composition and transport (unlike `Set`), but function types from types in this sort to types in `Set` do (unlike *SSet*).

The Path types are exported by `Agda.Builtin.Cubical.Path`:

```
postulate
  PathP : ∀ {ℓ} (A : I → Set ℓ) → A i0 → A i1 → Set ℓ
```

(continues on next page)

(continued from previous page)

```
{-# BUILTIN PATHP        PathP     #-}

infix 4 _≡_
_≡_ : ∀ {ℓ} {A : Set ℓ} → A → A → Set ℓ
_≡_ {A = A} = PathP (λ _ → A)

{-# BUILTIN PATH         _≡_       #-}
```

The Cubical subtypes are exported by `Agda.Builtin.Cubical.Sub`:

```
{-# BUILTIN SUB Sub #-}

postulate
  inc : ∀ {ℓ} {A : Set ℓ} {φ} (x : A) → Sub A φ (λ _ → x)

{-# BUILTIN SUBIN inS #-}

primitive
  primSubOut : ∀ {ℓ} {A : Set ℓ} {φ : I} {u : Partial φ A} → Sub _ φ u → A
```

Equivalences are exported by `Agda.Builtin.Cubical.Equiv`:

```
record isEquiv {ℓ ℓ'} {A : Set ℓ} {B : Set ℓ'} (f : A → B) : Set (ℓ ⊔ ℓ') where
  field
    equiv-proof : (y : B) → isContr (fiber f y)
infix 4 _≃_

_≃_ : ∀ {ℓ ℓ'} (A : Set ℓ) (B : Set ℓ') → Set (ℓ ⊔ ℓ')
A ≃ B = Σ (A → B) λ f → isEquiv f

equivFun : ∀ {ℓ ℓ'} {A : Set ℓ} {B : Set ℓ'} → A ≃ B → A → B
equivFun e = fst e

equivProof : ∀ {la lt} (T : Set la) (A : Set lt) → (w : T ≃ A) → (a : A)
           → ∀ ψ (f : Partial ψ (fiber (w .fst) a)) → fiber (w .fst) a [ ψ ↦ f ]
equivProof A B w a ψ fb = contr' {A = fiber (w .fst) a} (w .snd .equiv-proof a) ψ fb
  where
    contr' : ∀ {ℓ} {A : Set ℓ} → isContr A → (φ : I) → (u : Partial φ A) → A
    contr' {A = A} (c , p) φ u = hcomp (λ{ i (φ = i1) → p (u 1=1) i
                                         ; i (φ = i0) → c }) c

{-# BUILTIN EQUIV      _≃_        #-}
{-# BUILTIN EQUIVFUN   equivFun   #-}
{-# BUILTIN EQUIVPROOF equivProof #-}
```

The Glue types are exported by `Agda.Builtin.Cubical.Glue`:

```
open import Agda.Builtin.Cubical.Equiv public

primitive
  primGlue    : ∀ {ℓ ℓ'} (A : Set ℓ) {φ : I}
    → (T : Partial φ (Set ℓ')) → (e : PartialP φ (λ o → T o ≃ A))
    → Set ℓ'
```

(continues on next page)

(continued from previous page)

```
  prim^glue   : ∀ {ℓ ℓ'} {A : Set ℓ} {φ : I}
    → {T : Partial φ (Set ℓ')} → {e : PartialP φ (λ o → T o ≃ A)}
    → PartialP φ T → A → primGlue A T e
  prim^unglue : ∀ {ℓ ℓ'} {A : Set ℓ} {φ : I}
    → {T : Partial φ (Set ℓ')} → {e : PartialP φ (λ o → T o ≃ A)}
    → primGlue A T e → A
  primFaceForall : (I → I) → I
```

Note that the Glue types are uncurried in `agda/cubical` to make them more pleasant to use:

```
Glue : ∀ {ℓ ℓ'} (A : Set ℓ) {φ : I}
     → (Te : Partial φ (Σ[ T ∈ Set ℓ' ] T ≃ A))
     → Set ℓ'
Glue A Te = primGlue A (λ x → Te x .fst) (λ x → Te x .snd)
```

3.8 Cubical compatible

The option *--cubical=compatible* specifies whether the module being type-checked is compatible with Cubical Agda: modules without this flag can not be imported from *--cubical* modules.

Note

Prior to Agda 2.6.3, the *--cubical=compatible* flag did not exist, and *--without-K* also implied the (internal) generation of Cubical Agda-specific code. See Agda issue #5843 for the rationale behind this change.

Compatibility with Cubical Agda consists of:

- No reasoning principles incompatible with univalent type theory may be used. This behaviour is controlled by the *Without K* flag (*--without-K*), which *--cubical=compatible* implies.
- Due to specifics of the Cubical Agda implementation, several kinds of Agda definition need internal support code to be generated during their elaboration.

Occasionally, elaborator bugs can result in errors surfacing from these internal definitions, despite the code being type-correct. To avoid showing errors mentioning cubical definitions when the user-written code is independent of Cubical Agda, these internal definitions are now gated behind *--cubical=compatible*.

Note that code that uses (only) *--without-K* can not be imported from code that uses *--cubical*. Thus library developers are encouraged to use *--cubical=compatible* instead of *--without-K*, if possible.

Note also that Agda tends to be quite a bit faster if *--without-K* is used instead of *--cubical=compatible*.

The *--cubical=compatible* option is coinfective (see *Checking options for consistency*): the generated support code for functions may depend on those of importing modules.

3.9 Cumulativity

3.9.1 Basics

Since version 2.6.1, Agda supports optional cumulativity of universes under the *--cumulativity* flag.

```
{-# OPTIONS --cumulativity #-}
```

When the *--cumulativity* flag is enabled, Agda uses the subtyping rule `Set i =< Set j` whenever `i =< j`. For example, in addition to its usual type `Set`, `Nat` also has the type $\texttt{Set}_1$ and even `Set i` for any `i : Level`.

```
_ : Set
_ = Nat

_ : Set₁
_ = Nat

_ : ∀ {i} → Set i
_ = Nat
```

With cumulativity is enabled, one can implement lifting to a higher universe as the identity function.

```
lift : ∀ {a b} → Set a → Set (a ⊔ b)
lift x = x
```

3.9.2 Example usage: N-ary functions

In Agda without cumulativity, it is tricky to define a universe-polymorphic N-ary function type `A → A → ... → A → B` because the universe level depends on whether the number of arguments is zero:

```
module Without-Cumulativity where

  N-ary-level : Level → Level → Nat → Level
  N-ary-level ℓ₁ ℓ₂ zero    = ℓ₂
  N-ary-level ℓ₁ ℓ₂ (suc n) = ℓ₁ ⊔ N-ary-level ℓ₁ ℓ₂ n

  N-ary : ∀ {ℓ₁ ℓ₂} n → Set ℓ₁ → Set ℓ₂ → Set (N-ary-level ℓ₁ ℓ₂ n)
  N-ary zero    A B = B
  N-ary (suc n) A B = A → N-ary n A B
```

In contrast, in Agda with cumulativity one can always work with the highest possible universe level. This makes it much easier to define the type of N-ary functions.

```
module With-Cumulativity where

  N-ary : Nat → Set ℓ₁ → Set ℓ₂ → Set (ℓ₁ ⊔ ℓ₂)
  N-ary zero    A B = B
  N-ary (suc n) A B = A → N-ary n A B

  curryⁿ : (Vec A n → B) → N-ary n A B
  curryⁿ {n = zero}  f = f []
  curryⁿ {n = suc n} f = λ x → curryⁿ λ xs → f (x ∷ xs)

  _$ⁿ_ : N-ary n A B → (Vec A n → B)
  f $ⁿ []       = f
  f $ⁿ (x ∷ xs) = f x $ⁿ xs

  ∀ⁿ : ∀ {A : Set ℓ₁} n → N-ary n A (Set ℓ₂) → Set (ℓ₁ ⊔ ℓ₂)
  ∀ⁿ zero    P = P
  ∀ⁿ (suc n) P = ∀ x → ∀ⁿ n (P x)
```

3.9.3 Limitations

Currently cumulativity only enables subtyping between universes, but not between any other types containing universes. For example, `List Set` is not a subtype of `List Set`$_1$. Agda also does not have cumulativity for any other types containing universe levels, so `List {lzero} Nat` is not a subtype of `List {lsuc lzero} Nat`. Such rules might be added in a future version of Agda.

3.9.4 Constraint solving

When working in Agda with cumulativity, universe level metavariables are often underconstrained. For example, the expression `List Nat` could mean `List {lzero} Nat`, but also `List {lsuc lzero} Nat`, or indeed `List {i} Nat` for any `i : Level`.

Currently Agda uses the following heuristic to instantiate universe level metavariables. At the end of each type signature, each mutual block, or declaration that is not part of a mutual block, Agda instantiates all universe level metavariables that are *unbounded from above*. A metavariable `_l : Level` is unbounded from above if all unsolved constraints that mention the metavariable are of the form $\texttt{a}_i$ `=< _l : Level`, and `_l` does not occur in the type of any other unsolved metavariables. For each metavariable that satisfies these conditions, it is instantiated to $\texttt{a}_1 \sqcup \texttt{a}_2 \sqcup \ldots \sqcup \texttt{a}_n$ where $\texttt{a}_1$ `=< _l : Level`, ..., $\texttt{a}_n$ `=< _l : Level` are all constraints that mention *_l*.

The heuristic as described above is considered experimental and is subject to change in future versions of Agda.

3.10 Data Types

3.10.1 Simple datatypes

Example datatypes

In the introduction we already showed the definition of the data type of natural numbers (in unary notation):

```
data Nat : Set where
    zero : Nat
    suc  : Nat → Nat
```

We give a few more examples. First the data type of truth values:

```
data Bool : Set where
  true  : Bool
  false : Bool
```

The `True` set represents the trivially true proposition:

```
data True : Set where
    tt : True
```

The `False` set has no constructor and hence no elements. It represents the trivially false proposition:

```
data False : Set where
```

Another example is the data type of non-empty binary trees with natural numbers in the leaves:

```
data BinTree : Set where
  leaf   : Nat → BinTree
  branch : BinTree → BinTree → BinTree
```

Finally, the data type of Brouwer ordinals:

```
data Ord : Set where
  zeroOrd : Ord
  sucOrd  : Ord → Ord
  limOrd  : (Nat → Ord) → Ord
```

General form

The general form of the definition of a simple datatype D is the following

```
data D : Set_i where
  c_1 : A_1
  ...
  c_n : A_n
```

The name D of the data type and the names $c_1, \ldots, c_n$ of the constructors must be new w.r.t. the current signature and context, and the types $A_1, \ldots, A_n$ must be function types ending in D, i.e. they must be of the form

```
(y_1 : B_1) → ... → (y_m : B_m) → D
```

3.10.2 Parametrized datatypes

Datatypes can have *parameters*. They are declared after the name of the datatype but before the colon, for example:

```
data List (A : Set) : Set where
  []  : List A
  _::_ : A → List A → List A
```

3.10.3 Indexed datatypes

In addition to parameters, datatypes can also have *indices*. In contrast to parameters which are required to be the same for all constructors, indices can vary from constructor to constructor. They are declared after the colon as function arguments to Set. For example, fixed-length vectors can be defined by indexing them over their length of type Nat:

```
data Vector (A : Set) : Nat → Set where
  []  : Vector A zero
  _::_ : {n : Nat} → A → Vector A n → Vector A (suc n)
```

Notice that the parameter A is bound once for all constructors, while the index {n : Nat} must be bound locally in the constructor _::_.

Indexed datatypes can also be used to describe predicates, for example the predicate Even : Nat → Set can be defined as follows:

```
data Even : Nat → Set where
  even-zero  : Even zero
  even-plus2 : {n : Nat} → Even n → Even (suc (suc n))
```

General form

The general form of the definition of a (parametrized, indexed) datatype D is the following

```
data D (x_1 : P_1) ... (x_k : P_k) : (y_1 : Q_1) → ... → (y_l : Q_l) → Set ℓ where
  c_1 : A_1
```

(continues on next page)

(continued from previous page)

```
  ...
  c_n : A_n
```

where the types $A_1, \ldots, A_n$ are function types of the form

```
(z_1 : B_1) → ... → (z_m : B_m) → D x_1 ... x_k t_1 ... t_l
```

3.10.4 Strict positivity

When defining a datatype `D`, Agda poses an additional requirement on the types of the constructors of `D`, namely that `D` may only occur **strictly positively** in the types of their arguments.

Concretely, for a datatype with constructors $c_1 : A_1, \ldots, c_n : A_n$, Agda checks that each A_i has the form

```
(y_1 : B_1) → ... → (y_m : B_m) → D
```

where an argument types B_i of the constructors is either

- *non-inductive* (a *side condition*) and does not mention `D` at all,
- or *inductive* and has the form

  ```
  (z_1 : C_1) → ... → (z_k : C_k) → D
  ```

  where `D` must not occur in any C_j.

The strict positivity condition rules out declarations such as

```
data Bad : Set where
    bad : (Bad → Bad) → Bad
    --      A     B      C
    -- A is in a negative position, B and C are OK
```

since there is a negative occurrence of `Bad` in the type of the argument of the constructor. (Note that the corresponding data type declaration of `Bad` is allowed in standard functional languages such as Haskell and ML.).

Non strictly-positive declarations are rejected because they admit non-terminating functions.

If the positivity check is disabled, so that a similar declaration of `Bad` is allowed, it is possible to construct a term of the empty type, even without recursion.

```
{-# OPTIONS --no-positivity-check #-}
```

```
data ⊥ : Set where

data Bad : Set where
  bad : (Bad → ⊥) → Bad

self-app : Bad → ⊥
self-app (bad f) = f (bad f)

absurd : ⊥
absurd = self-app (bad self-app)
```

For more general information on termination see *Termination Checking*.

3.11 Flat Modality

The flat/crisp attribute `@♭`/`@flat` is an idempotent comonadic modality modeled after Spatial Type Theory and Crisp Type Theory. It is similar to a necessity modality.

This attribute is enabled using the infective flag *--cohesion*.

We can define `♭ A` as a type for any `(@♭ A : Set l)` via an inductive definition:

```
data ♭ {@♭ l : Level} (@♭ A : Set l) : Set l where
  con : (@♭ x : A) → ♭ A

counit : {@♭ l : Level} {@♭ A : Set l} → ♭ A → A
counit (con x) = x
```

When trying to provide a `@♭` arguments only other `@♭` variables will be available, the others will be marked as `@⊤` in the context. For example the following will not typecheck:

```
unit : {@♭ l : Level} {@♭ A : Set l} → A → ♭ A
unit x = con x
```

3.11.1 Pattern Matching on `@♭`

By default matching on arguments marked with `@♭` is disallowed, but it can be enabled using the option *--flat-split*. When matching on a `@♭` argument the flat status gets propagated to the arguments of the constructor

```
data _⊎_ (A B : Set) : Set where
  inl : A → A ⊎ B
  inr : B → A ⊎ B

flat-sum : {@♭ A B : Set} → (@♭ x : A ⊎ B) → ♭ A ⊎ ♭ B
flat-sum (inl x) = inl (con x)
flat-sum (inr x) = inr (con x)
```

When refining `@♭` variables the equality also needs to be provided as `@♭`

```
flat-subst : {@♭ A : Set} {P : A → Set} (@♭ x y : A) (@♭ eq : x ≡ y) → P x → P y
flat-subst x .x refl p = p
```

if we simply had `(eq : x ≡ y)` the code would be rejected.

Note that in Cubical Agda functions that match on an argument marked with `@♭` trigger the `UnsupportedIndexedMatch` warning (see *Indexed inductive types*), and the code might not compute properly.

3.11.2 The Sharp Modality

The *--cohesion* flag also enables the sharp modality as presented in Cohesive Homotopy Type Theory. The annotation `@♯`/`@sharp` is an idempotent monadic modality, which is right adjoint to the `@♭` modality.

We can define `♯` as the following record type:

```
record ♯ {l} (@♯ A : Set l) : Set l where
  constructor conSharp
  field
    @♯ ϵ : A
```

(continues on next page)

(continued from previous page)

```
open ♯

_ : ∀ {@♭ l} {@♭ A : Set l} → @♭ ♯ A → A
_ = ϵ
```

When providing a `@♯` argument, every non `@⊤` variable becomes `@♭` annotated. As a record, we can define elements of `♯` by copattern matching. After copattern matching with `ϵ`, all variables in the context become crisp, for example, in the following, we can use `a` in the constuctor of `♭`.

```
unit : ∀ {@♭ A : Set} → A → ♯ (♭ A)
unit a .ϵ = con a
```

3.12 Foreign Function Interface

3.12.1 Compiler Pragmas

There are two backend-generic pragmas used for the FFI:

```
{-# COMPILE <Backend> <Name> <Text> #-}
{-# FOREIGN <Backend> <Text> #-}
```

The `COMPILE` pragma associates some information `<Text>` with a name `<Name>` defined in the same module, and the `FOREIGN` pragma associates `<Text>` with the current top-level module. This information is interpreted by the specific backend during compilation (see below). These pragmas were added in Agda 2.5.3.

3.12.2 Haskell FFI

> **Note**
>
> This section applies to the *GHC Backend*.

The `FOREIGN` pragma

The GHC backend interprets `FOREIGN` pragmas as inline Haskell code and can contain arbitrary code (including import statements) that will be added to the compiled module. For instance:

```
{-# FOREIGN GHC import Data.Maybe #-}

{-# FOREIGN GHC
  data Foo = Foo | Bar Foo

  countBars :: Foo -> Integer
  countBars Foo = 0
  countBars (Bar f) = 1 + countBars f
#-}
```

The `COMPILE` pragma

There are four forms of `COMPILE` annotations recognized by the GHC backend

```
{-# COMPILE GHC <Name> = <HaskellCode> #-}
{-# COMPILE GHC <Name> = type <HaskellType> #-}
{-# COMPILE GHC <Name> = data <HaskellData> (<HsCon1> | .. | <HsConN>) #-}
{-# COMPILE GHC <Name> as <HaskellName> #-}
```

The first three tells the compiler how to compile a given Agda definition and the last exposes an Agda definition under a particular Haskell name allowing Agda libraries to be used from Haskell.

Using Haskell Types from Agda

In order to use a Haskell function from Agda its type must be mapped to an Agda type. This mapping can be configured using the `type` and `data` forms of the `COMPILE` pragma.

Opaque types

Opaque Haskell types are exposed to Agda by postulating an Agda type and associating it to the Haskell type using the `type` form of the `COMPILE` pragma:

```
{-# FOREIGN GHC import qualified System.IO #-}

postulate FileHandle : Set
{-# COMPILE GHC FileHandle = type System.IO.Handle #-}
```

This tells the compiler that the Agda type `FileHandle` corresponds to the Haskell type `System.IO.Handle` and will enable functions using file handles to be used from Agda.

Data types

Non-opaque Haskell data types can be mapped to Agda datatypes using the `data` form of the `COMPILE` pragma:

```
data Maybe (A : Set) : Set where
  nothing : Maybe A
  just    : A → Maybe A

{-# COMPILE GHC Maybe = data Maybe (Nothing | Just) #-}
```

The compiler checks that the types of the Agda constructors match the types of the corresponding Haskell constructors and that no constructors have been left out (on either side).

Record types

The `data` form of the `COMPILE` pragma also works with Agda's record types:

```
import Agda.Builtin.List
{-# FOREIGN GHC import Data.Tree #-}

record Tree (A : Set) : Set where
  inductive
  constructor node
  field root-label : A
  field sub-forest : Agda.Builtin.List.List (Tree A)

{-# COMPILE GHC Tree = data Tree (Node) #-}
```

Built-in Types

The GHC backend compiles certain Agda *built-in types* to special Haskell types. The mapping between Agda built-in types and Haskell types is as follows:

Agda Built-in	Haskell Type
`NAT`	`Integer`
`INTEGER`	`Integer`
`STRING`	`Data.Text.Text`
`CHAR`	`Char`
`BOOL`	`Bool`
`FLOAT`	`Double`

Warning

Haskell code manipulating Agda natural numbers as integers must take care to avoid negative values.

> **Warning**
>
> Agda `FLOAT` values have only one logical `NaN` value. At runtime, there might be multiple different `NaN` representations present. All such `NaN` values must be treated equal by FFI calls.

Using Haskell functions from Agda

Once a suitable mapping between Haskell types and Agda types has been set up, Haskell functions whose types map to Agda types can be exposed to Agda code with a `COMPILE` pragma:

```
open import Agda.Builtin.IO
open import Agda.Builtin.String
open import Agda.Builtin.Unit

{-# FOREIGN GHC
  import qualified Data.Text.IO as Text
  import qualified System.IO as IO
#-}

postulate
  stdout    : FileHandle
  hPutStrLn : FileHandle → String → IO ⊤
{-# COMPILE GHC stdout    = IO.stdout #-}
{-# COMPILE GHC hPutStrLn = Text.hPutStrLn #-}
```

The compiler checks that the type of the given Haskell code matches the type of the Agda function. Note that the `COMPILE` pragma only affects the runtime behaviour–at type-checking time the functions are treated as postulates.

> **Warning**
>
> It is possible to give Haskell definitions to defined (non-postulate) Agda functions. In this case the Agda definition will be used at type-checking time and the Haskell definition at runtime. However, there are no checks to ensure that the Agda code and the Haskell code behave the same and **discrepancies may lead to undefined behaviour**.
>
> This feature can be used to let you reason about code involving calls to Haskell functions under the assumption that you have a correct Agda model of the behaviour of the Haskell code.

Using Agda functions from Haskell

Since Agda 2.3.4 Agda functions can be exposed to Haskell code using the `as` form of the `COMPILE` pragma:

```
module IdAgda where

  idAgda : ∀ {A : Set} → A → A
  idAgda x = x

  {-# COMPILE GHC idAgda as idAgdaFromHs #-}
```

This tells the compiler that the Agda function `idAgda` should be compiled to a Haskell function called `idAgdaFromHs`. Without this pragma, functions are compiled to Haskell functions with unpredictable names and, as a result, cannot be invoked from Haskell. The type of `idAgdaFromHs` will be the translated type of `idAgda`.

The compiled and exported function `idAgdaFromHs` can then be imported and invoked from Haskell like this:

```
-- file UseIdAgda.hs
module UseIdAgda where

import MAlonzo.Code.IdAgda (idAgdaFromHs)
-- idAgdaFromHs :: () -> a -> a

idAgdaApplied :: a -> a
idAgdaApplied = idAgdaFromHs ()
```

Polymorphic functions

Agda is a monomorphic language, so polymorphic functions are modeled as functions taking types as arguments. These arguments will be present in the compiled code as well, so when calling polymorphic Haskell functions they have to be discarded explicitly. For instance,

```
postulate
  ioReturn : {A : Set} → A → IO A

{-# COMPILE GHC ioReturn = \ _ x -> return x #-}
```

In this case compiled calls to `ioReturn` will still have `A` as an argument, so the compiled definition ignores its first argument and then calls the polymorphic Haskell `return` function.

Level-polymorphic types

Level-polymorphic types face a similar problem to polymorphic functions. Since Haskell does not have universe levels the Agda type will have more arguments than the corresponding Haskell type. This can be solved by defining a Haskell type synonym with the appropriate number of phantom arguments. For instance:

```
data Either {a b} (A : Set a) (B : Set b) : Set (a ⊔ b) where
  left  : A → Either A B
  right : B → Either A B

{-# FOREIGN GHC type AgdaEither a b = Either #-}
{-# COMPILE GHC Either = data AgdaEither (Left | Right) #-}
```

Handling typeclass constraints

There is (currently) no way to map a Haskell type with type class constraints to an Agda type. This means that functions with class constraints cannot be used from Agda. However, this can be worked around by wrapping class constraints in Haskell data types, and providing Haskell functions using explicit dictionary passing.

For instance, suppose we have a simple GUI library in Haskell:

```
module GUILib where
  class Widget w
  setVisible :: Widget w => w -> Bool -> IO ()

  data Window
  instance Widget Window
  newWindow :: IO Window
```

To use this library from Agda we first define a Haskell type for widget dictionaries and map this to an Agda type `Widget`:

```
{-# FOREIGN GHC import GUILib #-}
{-# FOREIGN GHC data WidgetDict w = Widget w => WidgetDict #-}

postulate
  Widget : Set → Set
{-# COMPILE GHC Widget = type WidgetDict #-}
```

We can then expose `setVisible` as an Agda function taking a Widget *instance argument*:

```
postulate
  setVisible : {w : Set} {{_ : Widget w}} → w → Bool → IO ⊤
{-# COMPILE GHC setVisible = \ _ WidgetDict -> setVisible #-}
```

Note that the Agda `Widget` argument corresponds to a `WidgetDict` argument on the Haskell side. When we match on the `WidgetDict` constructor in the Haskell code, the packed up dictionary will become available for the call to `setVisible`.

The window type and functions are mapped as expected and we also add an Agda instance packing up the `Widget Window` Haskell instance into a `WidgetDict`:

```
postulate
  Window    : Set
  newWindow : IO Window
  instance WidgetWindow : Widget Window
{-# COMPILE GHC Window       = type Window #-}
{-# COMPILE GHC newWindow    = newWindow #-}
{-# COMPILE GHC WidgetWindow = WidgetDict #-}
```

We can then write code like this:

```
openWindow : IO Window
openWindow = newWindow         >>= λ w →
             setVisible w true >>= λ _ →
             return w
```

3.12.3 JavaScript FFI

The *JavaScript backend* recognizes `COMPILE` pragmas of the following form:

```
{-# COMPILE JS <Name> = <JsCode> #-}
```

where `<Name>` is a postulate, constructor, or data type. The code for a data type is used to compile pattern matching and should be a function taking a value of the data type and a table of functions (corresponding to case branches) indexed by the constructor names. For instance, this is the compiled code for the `List` type, compiling lists to JavaScript arrays:

```
data List {a} (A : Set a) : Set a where
  []  : List A
  _::_ : (x : A) (xs : List A) → List A

{-# COMPILE JS List = function(x,v) {
    if (x.length < 1) {
      return v["[]"]();
    } else {
      return v["_::_"](x[0], x.slice(1));
```

(continues on next page)

(continued from previous page)

```
    }
  } #-}
{-# COMPILE JS []  = Array() #-}
{-# COMPILE JS _::_ = function (x) { return function(y) { return [x].concat(y); }; } #-}
```

3.13 Function Definitions

3.13.1 Introduction

A function is defined by first declaring its type followed by a number of equations called *clauses*. Each clause consists of the function being defined applied to a number of *patterns*, followed by = and a term called the *right-hand side*. For example:

```
not : Bool → Bool
not true  = false
not false = true
```

Functions are allowed to call themselves recursively, for example:

```
twice : Nat → Nat
twice zero    = zero
twice (suc n) = suc (suc (twice n))
```

3.13.2 General form

The general form for defining a function is

```
f : (x₁ : A₁) → ... → (xₙ : Aₙ) → B
f p₁ ... pₙ = d
...
f q₁ ... qₙ = e
```

where `f` is a new identifier, p_i and q_i are patterns of type A_i, and `d` and `e` are expressions.

The declaration above gives the identifier `f` the type $(x_1 : A_1) \to \ldots \to (x_n : A_n) \to B$ and `f` is defined by the defining equations. Patterns are matched from top to bottom, i.e., the first pattern that matches the actual parameters is the one that is used.

By default, Agda checks the following properties of a function definition:

- The patterns in the left-hand side of each clause should consist only of constructors and variables.
- No variable should occur more than once on the left-hand side of a single clause.
- The patterns of all clauses should together cover all possible inputs of the function, see *Coverage Checking*.
- The function should be terminating on all possible inputs, see *Termination Checking*.

3.13.3 Special patterns

In addition to constructors consisting of constructors and variables, Agda supports two special kinds of patterns: dot patterns and absurd patterns.

Dot patterns

A dot pattern (also called *inaccessible pattern*) can be used when the only type-correct value of the argument is determined by the patterns given for the other arguments. A dot pattern is not matched against to determine the result of a function call. Instead it serves as checked documentation of the only possible value at the respective position, as determined by the other patterns. The syntax for a dot pattern is `.t`.

As an example, consider the datatype `Square` defined as follows

```
data Square : Nat → Set where
  sq : (m : Nat) → Square (m * m)
```

Suppose we want to define a function `root : (n : Nat) → Square n → Nat` that takes as its arguments a number `n` and a proof that it is a square, and returns the square root of that number. We can do so as follows:

```
root : (n : Nat) → Square n → Nat
root .(m * m) (sq m) = m
```

Notice that by matching on the argument of type `Square n` with the constructor `sq : (m : Nat) → Square (m * m)`, `n` is forced to be equal to `m * m`.

In general, when matching on an argument of type `D` i_1 `...` i_n with a constructor `c : (`x_1 `:` A_1`) → ... → (`x_m `:` A_m`) → D` j_1 `...` j_n, Agda will attempt to unify i_1 `...` i_n with j_1 `...` j_n. When the unification algorithm instantiates a variable `x` with value `t`, the corresponding argument of the function can be replaced by a dot pattern `.t`.

Using a dot pattern can help readability, but is not necessary; a dot pattern can always be replaced by an underscore or a fresh pattern variable without changing the function definition. The following are also legal definitions of `root`:

Since Agda 2.4.2.4:

```
root₁ : (n : Nat) → Square n → Nat
root₁ _ (sq m) = m
```

Since Agda 2.5.2:

```
root₂ : (n : Nat) → Square n → Nat
root₂ n (sq m) = m
```

In the case of $\texttt{root}_2$, `n` evaluates to `m * m` in the body of the function and is thus equivalent to

```
root₃ : (n : Nat) → Square n → Nat
root₃ _ (sq m) = let n = m * m in m
```

A dot pattern need not be a valid ordinary pattern at all (as in the case of `m * m` above). If it happens to be a valid ordinary pattern, then sometimes the dot can be removed without changing the function definition.

Other times, removing the dot yields a valid definition but with different definitional behavior. For instance, in the following definition:

```
data Fin : Nat → Set where
  fzero : {n : Nat} → Fin (suc n)
  fsuc : {n : Nat} → Fin n → Fin (suc n)

foo : (n : Nat) (k : Fin n) → Nat
foo .(suc zero) (fzero {zero})     = zero
foo .(suc (suc n)) (fzero {suc n}) = zero
foo .(suc _) (fsuc k)              = zero
```

removing the dots in `foo` changes the case tree so that it splits on the first argument first. This results in the third equation not holding definitionally (and thus the definition being flagged under the option *–exact-split*).

Absurd patterns

Absurd patterns can be used when none of the constructors for a particular argument would be valid. The syntax for an absurd pattern is `()`.

As an example, if we have a datatype `Even` defined as follows

```
data Even : Nat → Set where
  even-zero  : Even zero
  even-plus2 : {n : Nat} → Even n → Even (suc (suc n))
```

then we can define a function `one-not-even : Even 1 → ⊥` by using an absurd pattern:

```
one-not-even : Even 1 → ⊥
one-not-even ()
```

Note that if the left-hand side of a clause contains an absurd pattern, its right-hand side must be omitted.

In general, when matching on an argument of type `D i₁ ... iₙ` with an absurd pattern, Agda will attempt for each constructor `c : (x₁ : A₁) → ... → (xₘ : Aₘ) → D j₁ ... jₙ` of the datatype `D` to unify `i₁ ... iₙ` with `j₁ ... jₙ`. The absurd pattern will only be accepted if all of these unifications end in a conflict.

As-patterns

As-patterns (or `@-patterns`) can be used to name a pattern. The name has the same scope as normal pattern variables (i.e. the right-hand side, where clause, and dot patterns). The name reduces to the value of the named pattern. For example:

```
module _ {A : Set} (_<_ : A → A → Bool) where
  merge : List A → List A → List A
  merge xs [] = xs
  merge [] ys = ys
  merge xs@(x :: xs₁) ys@(y :: ys₁) =
    if x < y then x :: merge xs₁ ys
             else y :: merge xs ys₁
```

As-patterns are properly supported since Agda 2.5.2.

3.13.4 Case trees

Internally, Agda represents function definitions as *case trees*. For example, a function definition

```
max : Nat → Nat → Nat
max zero    n       = n
max m       zero    = m
max (suc m) (suc n) = suc (max m n)
```

will be represented internally as a case tree that looks like this:

```
max m n = case m of
  zero   → n
  suc m' → case n of
    zero   → suc m'
    suc n' → suc (max m' n')
```

Note that because Agda uses this representation of the function `max`, the clause `max m zero = m` does not hold definitionally (i.e. as a reduction rule). If you would try to prove that this equation holds, you would not be able to write `refl`:

```
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x

-- Does not work!
lemma : (m : Nat) → max m zero ≡ m
lemma = refl
```

Clauses which do not hold definitionally are usually (but not always) the result of writing clauses by hand instead of using Agda's case split tactic. These clauses are *highlighted* by Emacs.

The `--exact-split` flag causes Agda to raise a warning whenever a clause in a definition by pattern matching cannot be made to hold definitionally. Specific clauses can be excluded from this check by means of the `{-# CATCHALL #-}` pragma.

For instance, the above definition of `max` will be flagged when using the `--exact-split` flag because its second clause does not to hold definitionally.

When using the `--exact-split` flag, catch-all clauses have to be marked as such, for instance:

```
eq : Nat → Nat → Bool
eq zero    zero    = true
eq (suc m) (suc n) = eq m n
{-# CATCHALL #-}
eq _       _       = false
```

The `--no-exact-split` flag can be used to override a global `--exact-split` in a file, by adding a pragma `{-# OPTIONS --no-exact-split #-}`. This option is enabled by default.

Since version 2.8.0, Agda warns about superfluous CATCHALL pragmas, flagging a *UselessPragma*.

3.14 Function Types

Function types are written `(x : A) → B`, or in the case of non-dependent functions simply `A → B`. For instance, the type of the addition function for natural numbers is:

```
Nat → Nat → Nat
```

and the type of the addition function for vectors is:

```
(A : Set) → (n : Nat) → (u : Vec A n) → (v : Vec A n) → Vec A n
```

where `Set` is the type of sets and `Vec A n` is the type of vectors with `n` elements of type `A`. Arrows between consecutive hypotheses of the form `(x : A)` may also be omitted, and `(x : A) (y : A)` may be shortened to `(x y : A)` (see also *telescopes*):

```
(A : Set) (n : Nat)(u v : Vec A n) → Vec A n
```

Functions are constructed by *lambda expressions* or *Function Definitions*.

The application of a function `f : (x : A) → B` to an argument `a : A` is written `f a` and the type of this is `B[x := a]`.

3.14.1 Notational conventions

Function types:

```
prop₁ : ((x : A) (y : B) → C) is-the-same-as   ((x : A) → (y : B) → C)
prop₂ : ((x y : A) → C)       is-the-same-as   ((x : A)(y : A) → C)
prop₃ : (forall (x : A) → C)  is-the-same-as   ((x : A) → C)
prop₄ : (forall x → C)        is-the-same-as   ((x : _) → C)
prop₅ : (forall x y → C)      is-the-same-as   (forall x → forall y → C)
```

You can also use the Unicode symbol ∀ (type "\all" in the Emacs Agda mode) instead of `forall`.

Functional abstraction:

```
(\x y → e)                    is-the-same-as   (\x → (\y → e))
```

Functional application:

```
(f a b)                      is-the-same-as    ((f a) b)
```

3.15 Generalization of Declared Variables

3.15.1 Overview

Since version 2.6.0, Agda supports implicit generalization over variables in types. Variables to be generalized over must be declared with their types in a `variable` block. For example:

```
variable
  ℓ : Level
  n m : Nat

data Vec (A : Set ℓ) : Nat → Set ℓ where
  []  : Vec A 0
  _::_ : A → Vec A n → Vec A (suc n)
```

Here the parameter `ℓ` and the `n` in the type of `_::_` are not bound explicitly, but since they are declared as generalizable variables, bindings for them are inserted automatically. The level `ℓ` is added as a parameter to the datatype and `n` is added as an argument to `_::_`. The resulting declaration is

```
data Vec {ℓ : Level} (A : Set ℓ) : Nat → Set ℓ where
  []  : Vec A 0
  _::_ : {n : Nat} → A → Vec A n → Vec A (suc n)
```

See *Placement of generalized bindings* below for more details on where bindings are inserted.

Variables are generalized in top-level type signatures, *module telescopes*, and *record* and *datatype* parameter telescopes.

Issues related to this feature are marked with generalize in the issue tracker.

3.15.2 Nested generalization

When generalizing a variable, any generalizable variables in its type are also generalized over. For instance, you can declare `A` to be a type at some level `ℓ` as

```
variable
  A : Set ℓ
```

Now if `A` is mentioned in a type, the level `ℓ` will also be generalized over:

```
-- id : {A.ℓ : Level} {A : Set ℓ} → A → A
id : A → A
id x = x
```

The nesting can be arbitrarily deep, so

```
variable
  x : A

refl´ : x ≡ x
refl´ = refl
```

expands to

```
refl´ : {x.A.ℓ : Level} {x.A : Set x.A.ℓ} {x : x.A} → x ≡ x
```

See *Naming of nested variables* below for how the names are chosen.

Nested variables are not necessarily generalized over. In this example, if the universe level of `A` is fixed there is nothing to generalize:

```
postulate
  -- pure : {A : Set} {F : Set → Set} → A → F A
  pure : {F : Set → Set} → A → F A
```

See *Generalization over unsolved metavariables* for more details.

Note

Nested generalized variables are local to each variable, so if you declare

```
variable
  B : Set ℓ
```

then `A` and `B` can still be generalized at different levels. For instance,

```
-- _$_ : {A.ℓ : Level} {A : Set A.ℓ} {B.ℓ : Level} {B : Set B.ℓ} → (A → B) → A → B
_$_ : (A → B) → A → B
f $ x = f x
```

Generalization over unsolved metavariables

Generalization over nested variables is implemented by creating a metavariable for each nested variable and generalize over any such meta that is still unsolved after type checking. This is what makes the `pure` example from the previous section work: the metavariable created for ℓ is solved to level 0 and is thus not generalized over.

A typical case where this happens is when you have dependencies between different nested variables. For instance:

```
postulate
  Con : Set

variable
  Γ Δ Θ : Con

postulate
  Sub : Con → Con → Set

  idS : Sub Γ Γ
  _∘_ : Sub Γ Δ → Sub Δ Θ → Sub Γ Θ

variable
  δ σ γ : Sub Γ Δ

postulate
  assoc : δ ∘ (σ ∘ γ) ≡ (δ ∘ σ) ∘ γ
```

In the type of `assoc` each substitution gets two nested variable metas for their contexts, but the type of `_∘_` requires the contexts of its arguments to match up, so some of these metavariables are solved. The resulting type is

```
assoc : {δ.Γ δ.Δ : Con} {δ : Sub δ.Γ δ.Δ} {σ.Δ : Con} {σ : Sub δ.Δ σ.Δ}
        {γ.Δ : Con} {γ : Sub σ.Δ γ.Δ} → (δ ∘ (σ ∘ γ)) ≡ ((δ ∘ σ) ∘ γ)
```

where we can see from the names that $\sigma.\Gamma$ was unified with $\delta.\Delta$ and $\gamma.\Gamma$ with $\sigma.\Delta$. In general, when unifying two metavariables the "youngest" one is eliminated which is why $\delta.\Delta$ and $\sigma.\Delta$ are the ones that remain in the type.

If a metavariable for a nested generalizable variable is partially solved, the left-over metas are generalized over. For instance,

```
variable
  xs : Vec A n

head : Vec A (suc n) → A
head (x :: _) = x

-- lemma : {xs.n.1 : Nat} {xs : Vec Nat (suc xs.n.1)} → head xs ≡ 1 → (0 < sum xs) ≡␣
↪true
lemma : head xs ≡ 1 → (0 < sum xs) ≡ true
```

In the type of `lemma` a metavariable is created for the length of `xs`, which the application `head xs` refines to `suc _n`, for some new metavariable `_n`. Since there are no further constraints on `_n`, it's generalized over, creating the type

given in the comment. See *Naming of nested variables* below for how the name `xs.n.1` is chosen.

Note

Only metavariables originating from nested variables are generalized over. An exception to this is in `variable` blocks where all unsolved metas are turned into nested variables. This means writing

```
variable
  A : Set _
```

is equivalent to `A : Set ℓ` up to naming of the nested variable (see below).

Naming of nested variables

The general naming scheme for nested generalized variables is `parentVar.nestedVar`. So, in the case of the identity function `id : A → A` expanding to

```
id : {A.ℓ : Level} {A : Set ℓ} → A → A
```

the name of the level variable is `A.ℓ` since the name of the nested variable is `ℓ` and its parent is the named variable `A`. For multiple levels of nesting the parent can be another nested variable as in the `refl´` case above

```
refl´ : {x.A.ℓ : Level} {x.A : Set x.A.ℓ} {x : x.A} → x ≡ x
```

If a nested generalizable variable is solved with a term containing further metas, these are generalized over as explained in the `lemma` example above. The names of the new variables are of the form `parentName.i` where `parentName` is the name of the solved variable and `i` numbers the metas, starting from 1, in the order they appear in the solution.

If a variable comes from a free unsolved metavariable in a `variable` block (see *this note*), its name is chosen as follows:

- If it is a labelled argument to a function, the label is used as the name,
- otherwise the name is its left-to-right index (starting at 1) in the list of unnamed variables in the type.

It is then given a hierarchical name based on the named variable whose type it occurs in. For example,

```
postulate
  V : (A : Set) → Nat → Set
  P : V A n → Set

variable
  v : V _ _

postulate
  thm : P v
```

Here there are two unnamed variables in the type of `v`, namely the two arguments to `V`. The first argument has the label `A` in the definition of `V`, so this variable gets the name `v.A`. The second argument has no label and thus gets the name `v.2` since it is the second unnamed variable in the type of `v`.

If the variable comes from a partially instantiated nested variable the name of the metavariable is used unqualified.

Note

Currently it is not allowed to use hierarchical names when giving parameters to functions, see Issue #3208.

3.15.3 Placement of generalized bindings

The following rules are used to place generalized variables:

- Generalized variables are placed at the front of the type signature or *telescope*.
- Type signatures appearing inside other type signatures, for instance in *let bindings* or dependent function arguments are not generalized. Instead any generalizable variables in such types are generalized over in the parent signature.
- Variables mentioned earlier are placed before variables mentioned later, where nested variables count as being mentioned together with their parent.

> **Note**
>
> This means that an implicitly quantified variable cannot depend on an explicitly quantified one. See Issue #3352 for the feature request to lift this restriction.

Indexed datatypes

When generalizing datatype parameters and indices a variable is turned into an index if it is only mentioned in indices and into a parameter otherwise. For instance,

```
data All (P : A → Set) : Vec A n → Set where
  []  : All P []
  _::_ : P x → All P xs → All P (x :: xs)
```

Here `A` is generalized as a parameter and `n` as an index. That is, the resulting signature is

```
data All {A : Set} (P : A → Set) : {n : Nat} → Vec A n → Set where
```

3.15.4 Instance and irrelevant variables

Generalized variables are introduced as implicit arguments by default, but this can be changed to *instance arguments* or *irrelevant arguments* by annotating the declaration of the variable

```
record Eq (A : Set) : Set where
  field eq : A → A → Bool

variable
  {{EqA}} : Eq A   -- generalized as an instance argument
  .ignore : A      -- generalized as an irrelevant (implicit) argument
```

Variables are never generalized as explicit arguments.

3.15.5 Importing and exporting variables

Generalizable variables are treated in the same way as other declared symbols (functions, datatypes, etc) and use the same mechanisms for importing and exporting between modules. This means that unless marked `private` they are exported from a module.

3.15.6 Interaction

When developing types interactively, generalizable variables can be used in holes if they have already been generalized, but it is not possible to introduce *new* generalizations interactively. For instance,

```
works : (A → B) → Vec A n → Vec B {!n!}
fails : (A → B) → Vec A {!n!} → Vec B {!n!}
```

In `works` you can give `n` in the hole, since a binding for `n` has been introduced by its occurrence in the argument vector. In `fails` on the other hand, there is no reference to `n` so neither hole can be filled interactively.

3.15.7 Modalities

One can give a modality when declaring a generalizable variable:

```
variable
  @0 o : Nat
```

In the generalization process generalizable variables get the modality that they are declared with, whereas other variables always get the default modality.

3.16 Guarded Type Theory

> **Note**
>
> This is a stub.

Option `--guarded` extends Agda with Nakano's later modality and guarded recursion based on Ticked (Cubical) Type Theory *[2]*. For its usage in combination with `--cubical`, see *[1]* or the example.

The implementation currently allows for something more general than in the above reference, in preparation for the ticks described in *[3]*.

Given a type `A` in the `primLockUniv` universe we can form function types annotated with `@tick` (or its synonym `@lock`): `(@tick x : A) -> B`. Lambda abstraction at such a type introduces the variable in the context with a `@tick` annotation. Application `t u` for `t : (@tick x : A) → B` is restricted so that `t` is typable in the prefix of the context that does not include any `@tick` variables in `u`. The only exception to that restriction, at the moment, are variables of interval `I`, or `IsOne _` type.

3.16.1 References

[1] Niccolò Veltri and Andrea Vezzosi. "Formalizing pi-calculus in guarded cubical Agda." In CPP'20. ACM, New York, NY, USA, 2020.

[2] Rasmus Ejlers Møgelberg and Niccolò Veltri. "Bisimulation as path type for guarded recursive types." In POPL'19, 2019.

[3] Magnus Baunsgaard Kristensen, Rasmus Ejlers Møgelberg, Andrea Vezzosi. "Greatest HITs: Higher inductive types in coinductive definitions via induction under clocks."

3.17 Implicit Arguments

It is possible to omit terms that the type checker can figure out for itself, replacing them by an underscore (_). If the type checker cannot infer the value of an _ it will report an error. For instance, for the polymorphic identity function

```
id : (A : Set) → A → A
```

the first argument can be inferred from the type of the second argument, so we might write `id _ zero` for the application of the identity function to `zero`.

We can even write this function application without the first argument. In that case we declare an implicit function space:

```
id : {A : Set} → A → A
```

and then we can use the notation `id zero`.

Another example:

```
_==_  : {A : Set} → A → A → Set
subst : {A : Set} (C : A → Set) {x y : A} → x == y → C x → C y
```

Note how the first argument to `_==_` is left implicit. Similarly, we may leave out the implicit arguments `A`, `x`, and `y` in an application of `subst`. To give an implicit argument explicitly, enclose it in curly braces. The following two expressions are equivalent:

```
x1 = subst C eq cx
x2 = subst {_} C {_} {_} eq cx
```

It is worth noting that implicit arguments are also inserted at the end of an application, if it is required by the type. For example, in the following, `y1` and `y2` are equivalent.

```
y1 : a == b → C a → C b
y1 = subst C

y2 : a == b → C a → C b
y2 = subst C {_} {_}
```

Implicit arguments are inserted eagerly in left-hand sides so `y3` and `y4` are equivalent. An exception is when no type signature is given, in which case no implicit argument insertion takes place. Thus in the definition of `y5` the only implicit is the `A` argument of `subst`.

```
y3 : {x y : A} → x == y → C x → C y
y3 = subst C

y4 : {x y : A} → x == y → C x → C y
y4 {x} {y} = subst C {_} {_}

y5 = subst C
```

It is also possible to write lambda abstractions with implicit arguments. For example, given `id : (A : Set) → A → A`, we can define the identity function with implicit type argument as

```
id' = λ {A} → id A
```

Implicit arguments can also be referred to by name, so if we want to give the expression `e` explicitly for `y` without giving a value for `x` we can write

```
subst C {y = e} eq cx
```

In rare circumstances it can be useful to separate the name used to give an argument by name from the name of the bound variable, for instance if the desired name shadows an existing name. To do this you write

```
id₂ : {A = X : Set} → X → X  -- name of bound variable is X
id₂ x = x

use-id₂ : (Y : Set) → Y → Y
use-id₂ Y = id₂ {A = Y}      -- but the label is A
```

Labeled bindings must appear by themselves when typed, so the type `Set` needs to be repeated in this example:

```
const : {A = X : Set} {B = Y : Set} → A → B → A
const x y = x
```

When constructing implicit function spaces the implicit argument can be omitted, so both expressions below are valid expressions of type `{A : Set} → A → A`:

```
z1 = λ {A} x → x
z2 = λ x → x
```

The ∀ (or `forall`) syntax for function types also has implicit variants:

```
① : (∀ {x : A} → B)    is-the-same-as  ({x : A} → B)
② : (∀ {x} → B)        is-the-same-as  ({x : _} → B)
③ : (∀ {x y} → B)      is-the-same-as  (∀ {x} → ∀ {y} → B)
```

In very special situations it makes sense to declare *unnamed* hidden arguments `{A} → B`. In the following `example`, the hidden argument to `scons` of type `zero ≤ zero` can be solved by η-expansion, since this type reduces to ⊤.

```
data ⊥ : Set where

_≤_ : Nat → Nat → Set
zero ≤ _      = ⊤
suc m ≤ zero  = ⊥
suc m ≤ suc n = m ≤ n

data SList (bound : Nat) : Set where
  []    : SList bound
  scons : (head : Nat) → {head ≤ bound} → (tail : SList head) → SList bound

example : SList zero
example = scons zero []
```

There are no restrictions on when a function space can be implicit. Internally, explicit and implicit function spaces are treated in the same way. This means that there are no guarantees that implicit arguments will be solved. When there are unsolved implicit arguments the type checker will give an error message indicating which application contains the unsolved arguments. The reason for this liberal approach to implicit arguments is that limiting the use of implicit argument to the cases where we guarantee that they are solved rules out many useful cases in practice.

3.17.1 Tactic arguments

You can declare *tactics* to be used to solve a particular implicit argument using the `@(tactic t)` attribute, where `t : Term → TC ⊤`. For instance:

```
clever-search : Term → TC ⊤
clever-search hole = unify hole (lit (nat 17))

the-best-number : {@(tactic clever-search) n : Nat} → Nat
the-best-number {n} = n

check : the-best-number ≡ 17
check = refl
```

The tactic can be an arbitrary term of the right type and may depend on previous arguments to the function:

```
default : {A : Set} → A → Term → TC ⊤
default x hole = bindTC (quoteTC x) (unify hole)

search : (depth : Nat) → Term → TC ⊤

example : {@(tactic default 10)   depth : Nat}
          {@(tactic search depth) proof : Proof} →
          Goal
```

3.17.2 Metavariables

3.17.3 Unification

3.18 Instance Arguments

Instance arguments are a special kind of *implicit arguments* that get solved by a special *instance resolution* algorithm, rather than by the unification algorithm used for normal implicit arguments. Instance arguments are the Agda equivalent of Haskell type class constraints and can be used for many of the same purposes.

An instance argument will be resolved if its type is a *named type* (i.e. a data type or record type) or a *variable type* (i.e. a previously bound variable of type *Set ℓ*), and a unique *instance* of the required type can be built from *declared instances* and the current context.

3.18.1 Usage

Instance arguments are enclosed in double curly braces `{{ }}`, e.g. `{{x : T}}`. Alternatively they can be enclosed, with proper spacing, e.g. PDF TODO `x : T` PDF TODO, in the unicode braces PDF TODO PDF TODO (`U+2983` and `U+2984`, which can be typed as `\{{` and `\}}` in the *Emacs mode*).

For instance, given a function `_==_`

```
_==_ : {A : Set} {{eqA : Eq A}} → A → A → Bool
```

for some suitable type `Eq`, you might define

```
elem : {A : Set} {{eqA : Eq A}} → A → List A → Bool
elem x (y ∷ xs) = x == y || elem x xs
elem x []       = false
```

Here the instance argument to _==_ is solved by the corresponding argument to `elem`. Just like ordinary implicit arguments, instance arguments can be given explicitly. The above definition is equivalent to

```
elem : {A : Set} {{eqA : Eq A}} → A → List A → Bool
elem {{eqA}} x (y ∷ xs) = _==_ {{eqA}} x y || elem {{eqA}} x xs
elem         x []       = false
```

A very useful function that exploits this is the function `it` which lets you apply instance resolution to solve an arbitrary goal:

```
it : ∀ {a} {A : Set a} → {{A}} → A
it {{x}} = x
```

As the last example shows, the name of the instance argument can be omitted in the type signature:

```
_==_ : {A : Set} → {{Eq A}} → A → A → Bool
```

Defining type classes

The type of an instance argument should have the form `{Γ} → C vs`, where `C` is a postulated name, a bound variable, or the name of a data or record type, and `{Γ}` denotes an arbitrary number of implicit or instance arguments (see *Dependent instances* below for an example where `{Γ}` is non-empty).

Instances with explicit arguments are also accepted but will not be considered as instances because the value of the explicit arguments cannot be derived automatically. Having such an instance has no effect and thus raises a warning.

Instance arguments whose types end in any other type are currently also accepted but cannot be resolved by instance search, so they must be given by hand. For this reason it is not recommended to use such instance arguments. Doing so will also raise a warning.

Other than that there are no requirements on the type of an instance argument. In particular, there is no special declaration to say that a type is a "type class". Instead, Haskell-style type classes are usually defined as *record types*. For instance,

```
record Monoid {a} (A : Set a) : Set a where
  field
    mempty : A
    _<>_   : A → A → A
```

In order to make the fields of the record available as functions taking instance arguments you can use the special module application

```
open Monoid {{...}} public
```

This will bring into scope

```
mempty : ∀ {a} {A : Set a} → {{Monoid A}} → A
_<>_   : ∀ {a} {A : Set a} → {{Monoid A}} → A → A → A
```

Superclass dependencies can be implemented using *Records and instance search*.

See *Module application* and *Record modules* for details about how the module application is desugared. If defined by hand, `mempty` would be

```
mempty : ∀ {a} {A : Set a} → {{Monoid A}} → A
mempty {{mon}} = Monoid.mempty mon
```

Although record types are a natural fit for Haskell-style type classes, you can use instance arguments with data types to good effect. See the *Examples* below.

Declaring instances

As seen above, instance arguments in the context are available when solving instance arguments, but you also need to be able to define top-level instances for concrete types. This is done using the `instance` keyword, which starts a *block* in which each definition is marked as an instance available for instance resolution. For example, an instance `Monoid (List A)` can be defined as

```
instance
  ListMonoid : ∀ {a} {A : Set a} → Monoid (List A)
  ListMonoid = record { mempty = []; _<>_ = _++_ }
```

Or equivalently, using *copatterns*:

```
instance
  ListMonoid : ∀ {a} {A : Set a} → Monoid (List A)
  mempty {{ListMonoid}} = []
  _<>_   {{ListMonoid}} xs ys = xs ++ ys
```

Top-level instances must target a named type (`Monoid` in this case), and cannot be declared for types in the context.

You can define local instances in let-expressions in the same way as a top-level instance. For example:

```
mconcat : ∀ {a} {A : Set a} → {{Monoid A}} → List A → A
mconcat [] = mempty
mconcat (x :: xs) = x <> mconcat xs

sum : List Nat → Nat
sum xs =
  let instance
        NatMonoid : Monoid Nat
        NatMonoid = record { mempty = 0; _<>_ = _+_ }
  in mconcat xs
```

Instances can have instance arguments themselves, which will be filled in recursively during instance resolution. For instance,

```
record Eq {a} (A : Set a) : Set a where
  field
    _==_ : A → A → Bool

open Eq {{...}} public

instance
  eqList : ∀ {a} {A : Set a} → {{Eq A}} → Eq (List A)
  _==_ {{eqList}} []       []       = true
  _==_ {{eqList}} (x :: xs) (y :: ys) = x == y && xs == ys
  _==_ {{eqList}} _        _        = false
```

(continues on next page)

(continued from previous page)

```
  eqNat : Eq Nat
  _==_ {{eqNat}} = _≡ᵇ_  -- imported from Data.Nat.Base

ex : Bool
ex = (1 :: 2 :: 3 :: []) == (1 :: 2 :: []) -- false
```

Note the two calls to `_==_` in the right-hand side of the second clause. The first uses the `Eq A` instance and the second uses a recursive call to `eqList`. In the example `ex`, instance resolution, needing a value of type `Eq (List Nat)`, will try to use the `eqList` instance and find that it needs an instance argument of type `Eq Nat`, it will then solve that with `eqNat` and return the solution `eqList {{eqNat}}`.

Note

At the moment there is no termination check on instances, so it is possible to construct non-sensical instances like `loop : ∀ {a} {A : Set a} → {{Eq A}} → Eq A`. To prevent looping in cases like this, the search depth of instance search is limited, and once the maximum depth is reached, a type error will be thrown. You can set the maximum depth using the *--instance-search-depth* flag.

Restricting instance search

To restrict an instance to the current module, you can mark it as *private*. For instance,

```
record Default (A : Set) : Set where
  field default : A

open Default {{...}} public

module M where

  private
    instance
      defaultNat : Default Nat
      defaultNat .default = 6

  test₁ : Nat
  test₁ = default

  _ : test₁ ≡ 6
  _ = refl

open M

instance
  defaultNat : Default Nat
  defaultNat .default = 42

test₂ : Nat
test₂ = default

_ : test₂ ≡ 42
_ = refl
```

Alternatively, you can enable the *--no-qualified-instances* flag to make Agda only consider instances from modules that have been opened (see *below* for more details).

Constructor instances

Although instance arguments are most commonly used for record types, mimicking Haskell-style type classes, they can also be used with data types. In this case you often want the constructors to be instances, which is achieved by declaring them inside an `instance` block. Constructors can only be declared as instances if all their arguments are implicit or instance arguments. See *Instance resolution* below for the details.

A simple example of a constructor that can be made an instance is the reflexivity constructor of the equality type:

```
data _≡_ {a} {A : Set a} (x : A) : A → Set a where
  instance refl : x ≡ x
```

This allows trivial equality proofs to be inferred by instance resolution, which can make working with functions that have preconditions less of a burden. As an example, here is how one could use this to define a function that takes a natural number and gives back a `Fin n` (the type of naturals smaller than `n`):

```
data Fin : Nat → Set where
  zero : ∀ {n} → Fin (suc n)
  suc  : ∀ {n} → Fin n → Fin (suc n)

mkFin : ∀ {n} (m : Nat) → {{suc m - n ≡ 0}} → Fin n
mkFin {zero}  m {{}}
mkFin {suc n} zero    = zero
mkFin {suc n} (suc m) = suc (mkFin m)

five : Fin 6
five = mkFin 5 -- OK
```

In the first clause of `mkFin` we use an *absurd pattern* to discharge the impossible assumption `suc m ≡ 0`. See the *next section* for another example of constructor instances.

Record fields can also be declared instances, with the effect that the corresponding projection function is considered a top-level instance.

Qualified instances

By default, Agda considers all instances as candidates, even if they are only in scope under a qualified name. In particular, this means that instances from a module that is `import`-ed but not `open`-ed are still considered for instance search. You can use the *--no-qualified-instances* flag to make Agda instead only consider instances that are in scope under an unqualified name.

As an example, consider the following Agda code:

```
record MyClass (A : Set) : Set where
  field
    myFun : A → A
open MyClass {{...}}

module Instances where

  instance myNatInstance : MyClass Nat
  myFun {{myNatInstance}} = suc
```

(continues on next page)

(continued from previous page)

```
-- without --no-qualified-instances
test1 : Nat
test1 = myFun 41
```

By default, this example is accepted by Agda, but if *--no-qualified-instances* is enabled you have to open the module `Instances` first:

```
-- with --no-qualified-instances
open Instances

test2 : Nat
test2 = myFun 41
```

This flag can be especially useful if you want to import a module without necessarily using all of the instances that it exports.

Examples

Dependent instances

Consider a variant on the `Eq` class where the equality function produces a proof in the case the arguments are equal:

```
record Eq {a} (A : Set a) : Set a where
  field
    _==_ : (x y : A) → Maybe (x ≡ y)

open Eq {{...}} public
```

A simple boolean-valued equality function is problematic for types with dependencies, like the Σ-type

```
data Σ {a b} (A : Set a) (B : A → Set b) : Set (a ⊔ b) where
  _,_ : (x : A) → B x → Σ A B
```

since given two pairs `x , y` and $\mathtt{x}_1$ `,` $\mathtt{y}_1$, the types of the second components `y` and $\mathtt{y}_1$ can be completely different and not admit an equality test. Only when `x` and $\mathtt{x}_1$ are *really equal* can we hope to compare `y` and $\mathtt{y}_1$. Having the equality function return a proof means that we are guaranteed that when `x` and $\mathtt{x}_1$ compare equal, they really are equal, and comparing `y` and $\mathtt{y}_1$ makes sense.

An `Eq` instance for Σ can be defined as follows:

```
instance
  eqΣ : ∀ {a b} {A : Set a} {B : A → Set b} → {{Eq A}} → {{∀ {x} → Eq (B x)}} → Eq␣
↪(Σ A B)
  _==_ {{eqΣ}} (x , y) (x₁ , y₁) with x == x₁
  _==_ {{eqΣ}} (x , y) (x₁ , y₁)    | nothing = nothing
  _==_ {{eqΣ}} (x , y) (.x , y₁)    | just refl with y == y₁
  _==_ {{eqΣ}} (x , y) (.x , y₁)    | just refl    | nothing   = nothing
  _==_ {{eqΣ}} (x , y) (.x , .y)    | just refl    | just refl = just refl
```

Note that the instance argument for `B` states that there should be an `Eq` instance for `B x`, for any `x : A`. The argument `x` must be implicit, indicating that it needs to be inferred by unification whenever the `B` instance is used. See *Instance resolution* below for more details.

3.18.2 Overlap and backtracking

By default, instance resolution does not make unforced choices between instances. In practice, this means that instances may not *overlap*: if there are multiple candidates that could be used to solve an instance goal, a type error is raised.

For example, imagine that we have two separate printing classes: one for printing a debug representation, which we will call `Show`, and one for pretty printing, called `Pretty`. Since quite a few types (e.g. integers) have identical debug and pretty representations, we could try having a "default" instance for `Pretty`, in terms of `Show`:

```
record Show (A : Set) : Set where
  field show : A → String
open Show PDF TODO ... PDF TODO

record Pretty (A : Set) : Set where
  field pretty : A → String
open Pretty PDF TODO ... PDF TODO

instance
  pretty-show : ∀ {a} PDF TODO _ : Show a PDF TODO → Pretty a
  pretty-show = record { pretty = show }
```

Of course, some values have distinct representations. For example, we might want to pretty-print lists in square brackets, instead of as cons-cells. We write an instance:

```
postulate instance
  show-nat : Show Nat
  pretty-list : ∀ {a} PDF TODO _ : Pretty a PDF TODO → Pretty (List a)
```

However, if we try printing a list of numbers, Agda complains about overlap! While the `pretty-list` instance is strictly more specific than `pretty-show`, neither candidate is inapplicable in this situation, so Agda refuses to choose.

```
Failed to solve the following constraints:
  Resolve instance argument _r_273 : Pretty (List Nat)
  Candidates
    pretty-show : {a : Set} PDF TODO _ : Show a PDF TODO → Pretty a
    pretty-list : {a : Set} PDF TODO _ : Pretty a PDF TODO → Pretty (List a)
```

Overlapping instances

To support situations like `Pretty` above, Agda allows the user to specify, on a per-instance basis, what should happen when multiple candidates are available. This is done using one of the following four pragmas:

- An `OVERLAPPABLE` instance can be discarded in favour of a strictly *more* specific instance.
- An `OVERLAPPING` instance can cause strictly *less* specific instances to be discarded.
- The convenience pragma `OVERLAPS` is equivalent to `OVERLAPPABLE` and `OVERLAPPING`. This means that it can both cause less specific instances to be discarded, *and* it can be discarded if a more specific candidate is available.
- An `INCOHERENT` instance can be arbitrarily discarded in favour of another possible candidate.

An instance `c1 : ∀ {Γ} → C xs` is **more specific** than an instance `T2 : ∀ {Δ} → C ys` if there is an instantiation of the variables Δ which makes `ys` definitionally equal to `xs`. We say that `c1` is **strictly** more specific than `c2` if `c1` is more specific than `c2` and `c2` is *not* more specific than `c1`.

Returning to the `Pretty` example, we can make the more specific instance(s) be selected by marking the `pretty-show` instance `OVERLAPPABLE`:

```
{-# OVERLAPPABLE pretty-show #-}
_ : String
_ = pretty (1 :: 2 :: 3 :: [])
```

It would also have been possible to mark the `pretty-list` instance `OVERLAPPING`.

Overlap resolution considers *strict* specificity to keep Agda from making unforced choices. If multiple candidates have "the same specificity", then no matter whether they are both overlappable, the instance constraint still goes unsolved. An example is the following situation:

```
postulate
  C   : Set → Set → Set
  instance
    CIa : ∀ {a} → C Int a
    CaI : ∀ {a} → C a Int
  {-# OVERLAPS CIa CaI #-}
```

When solving the goal `C Int Int`, neither candidate can be discarded in favour of the other. You can make the choice yourself by marking the candidate that should **not** be used as `INCOHERENT` instead of `OVERLAPS`.

Backtracking

By default, Agda only considers an instance's final return type when considering whether an instance is applicable. In particular, the instance search algorithm does not backtrack, and whether or not an instance's constraints are satisfied does not factor into overlap resolution.

For example, in code below, the instances `zero` and `suc` overlap for the goal $\texttt{ex}_1$, because either one of them can be used to solve the goal when given appropriate arguments, so instance search will fail.

```
infix 4 _∈_
data _∈_ {A : Set} (x : A) : List A → Set where
  instance
    zero : ∀ {xs} → x ∈ x :: xs
    suc  : ∀ {y xs} → {{x ∈ xs}} → x ∈ y :: xs

ex₁ : 1 ∈ 1 :: 2 :: 3 :: 4 :: []
ex₁ = it  -- overlapping instances
```

However, *if* we looked for the appropriate arguments *before* checking for overlap, the goal above would have a unique solution. The *--backtracking-instance-search* option controls whether instance arguments *to instances* should be filled in before checking whether the instance is applicable.

> **Warning**
>
> Agda uses naïve backtracking to check instances' constraints, which has exponential performance in the worst case. Enabling *--backtracking-instance-search* might cause significant slowdown in instance search, and even apparent infinite loops.

3.18.3 Instance resolution

This section provides a precise specification of the instance resolution algorithm.

Verify the goal
: The first step is checking that the goal type has the right shape to be solved by instance resolution.

Instance search can only solve goals of the form `{Γ} → C vs`, where the target type `C` is either a variable, a data type, a record type, or a postulate; and `{Γ}` represents a sequence of implicit or instance arguments.

If this is not the case, instance resolution fails with an error message.

Find candidates

The second step is to compute a list of *initial candidates*.

Let-bound variables and top-level definitions in scope are candidates if they are defined in an `instance` block.

Local variables, i.e. variables bound in lambdas, function types, left-hand sides, or module parameters, are candidates if they are bound as instance arguments, using `{{ }}`. This includes any instance arguments to constructors of inductive or record types that have been matched on.

If local variables in the context have *superclass fields*, superclass expansion will apply. Note that determining whether any local instance variables are subject to expansion requires head-normalising their types.

Only candidates of type `{Δ} → C us`, where `C` is the target type computed in the previous stage, and `{Δ}` only contains implicit or instance arguments, are considered.

If it is not possible to determine whether a candidate's type is of the correct shape, or whether its type is an eta record, instance search will not run. This can happen in contexts where a local *instance* variable has a metavariable as its (return) type, or if its type is blocked on another metavariable, even if this metavariable would be solved by choosing one of the unblocked candidates.

Check the type of the candidates

The list of initial candidates is an overapproximation to the set of possible solutions. The next step is to check, in turn, whether the candidate could actually be used to solve the instance goal. If our goal is of the form `{Γ} → C vs`, we take the following steps:

1. The local context is extended by `{Γ}`. This may bring additional candidates into scope.
2. The candidate's type, say `c : {Δ} → A`, is instantiated with fresh metavariables, say α.
3. The target type `C vs` is unified with `A[`α`/Δ]`. If this results in a definite mismatch, the candidate is discarded.
4. Finally, if *`--backtracking-instance-search`* is enabled, we recursively apply instance search to any instance variables present in Δ.

If all of these steps succeed, we make note of the term λ `{Γ} → c {`α`}` as a potential solution.

Resolve overlaps

The previous step might have left us with multiple potential solutions, even if recursive instance search was enabled. We now remove any potential solutions which are overlapped by a *strictly more specific* candidate.

To wit, given a pair of candidates `c1 : {Δ} → C xs` and `c2 : {Γ} → C ys`, we remove `c1` from the list exactly when:

- There exists a substitution for the variables in Δ, in terms of those in Γ, which makes `C xs` and `C ys` definitionally equal. We say `c2` is *more specific* than `c1`.
- Such a substitution does *not* exist for Γ in terms of Δ. This makes `c2` *strictly* more specific than `c1`.
- Either `c1` is overlappable or `c2` is overlapping. Keep in mind that instances marked `OVERLAPS` (or `INCOHERENT`) are both overlappable and overlapping.

Compute the result

After resolving overlaps, we may be in five situations:

- There is exactly one non-incoherent candidate, along with some number of incoherent candidates. The non-incoherent candidate is chosen.
- All the potential solutions are incoherent. Agda makes an arbitrary choice.

- There are multiple candidates, and they all come from *instance fields* which are marked with the `overlap` keyword. Agda again makes an arbitrary choice.
- There are multiple, non-incoherent candidates. The instance constraint is postponed until we have more information available about either the goal or the candidates.
- There are no candidates at all. This is an immediate error.

If there are left-over instance problems at the end of type checking, the corresponding metavariables are printed in the Emacs status buffer, together with their types and source location. The candidates that gave rise to potential solutions can be printed with the *show constraints command* (`C-c C-=`).

3.19 Irrelevance

Since version 2.2.8 Agda supports irrelevancy annotations. The general rule is that anything prepended by a dot (`.`) is marked irrelevant, which means that it will only be typechecked but never evaluated or compared for equality. Arguments marked as irrelevant are erased by the *compiler*.

Irrelevance annotations are enabled by default. You can use the options *–irrelevance*/*–no-irrelevance* to enable/disable them.

Note

This section is about compile-time irrelevance. Agda also supports a weaker form of irrelevance called *Run-time Irrelevance* that can be used for arguments that should be erased by the compiler but can still be relevant at compile-time.

Note

The *Prop* universe provides an alternative form of irrelevance that does not require marking function arguments and declarations as irrelevant.

3.19.1 Motivating example

One intended use case of irrelevance is data structures with embedded proofs, like sorted lists.

```
data _≤_ : Nat → Nat → Set where
  zero≤   : {n : Nat} → zero ≤ n
  suc≤suc : {m n : Nat} → m ≤ n → suc m ≤ suc n

postulate
  p₁  : 0 ≤ 1
  p₂  : 0 ≤ 1

module No-Irrelevance where
  data SList (bound : Nat) : Set where
    []    : SList bound
    scons : (head : Nat)
          → (head ≤ bound)
          → (tail : SList head)
          → SList bound
```

Usually, when we define datatypes with embedded proofs we are forced to reason about the values of these proofs. For example, suppose we have two lists l_1 and l_2 with the same elements but different proofs:

```
l₁ : SList 1
l₁ = scons 0 p₁ []

l₂ : SList 1
l₂ = scons 0 p₂ []
```

Now suppose we want to prove that `l₁` and `l₂` are equal:

```
l₁≡l₂ : l₁ ≡ l₂
l₁≡l₂ = refl
```

It's not so easy! Agda gives us an error:

```
p₁ != p₂ of type 0 ≤ 1
when checking that the expression refl has type l₁ ≡ l₂
```

We can't show that `l₁ ≡ l₂` by `refl` when `p₁` and `p₂` are relevant. Instead, we need to reason about proofs of `0 ≤ 1`.

```
postulate
  proof-equality : p₁ ≡ p₂
```

Now we can prove `l₁ ≡ l₂` by rewriting with this equality:

```
l₁≡l₂ : l₁ ≡ l₂
l₁≡l₂ rewrite proof-equality = refl
```

Reasoning about equality of proofs becomes annoying quickly. We would like to avoid this kind of reasoning about proofs here - in this case we only care that a proof of `head ≤ bound` exists, i.e. any proof suffices. We can use irrelevance annotations to tell Agda we don't care about the values of the proofs:

```
data SList (bound : Nat) : Set where
  []    : SList bound
  scons : (head : Nat)
        → .(head ≤ bound)        -- note the dot!
        → (tail : SList head)
        → SList bound
```

The effect of the irrelevant type in the signature of scons is that scons's second argument is never inspected after Agda has ensured that it has the right type. The type-checker ignores irrelevant arguments when checking equality, so two lists can be equal even if they contain different proofs:

```
l₁ : SList 1
l₁ = scons 0 p₁ []

l₂ : SList 1
l₂ = scons 0 p₂ []

l₁≡l₂ : l₁ ≡ l₂
l₁≡l₂ = refl
```

3.19.2 Irrelevant function types

For starters, consider irrelevant non-dependent function types:

```
f : .A → B
```

This type implies that `f` does not depend computationally on its argument.

What can be done to irrelevant arguments

Example 1. We can prove that two applications of an unknown irrelevant function to two different arguments are equal.

```
-- an unknown function that does not use its second argument
postulate
  f : {A B : Set} -> A -> .B -> A

-- the second argument is irrelevant for equality
proofIrr : {A : Set}{x y z : A} -> f x y ≡ f x z
proofIrr = refl
```

Example 2. We can use irrelevant arguments as arguments to other irrelevant functions.

```
id : {A B : Set} -> (.A -> B) -> .A -> B
id g x = g x
```

Example 3. We can match on an irrelevant argument of an empty type with an absurd pattern `()`.

```
data ⊥ : Set where

zero-not-one : .(0 ≡ 1) → ⊥
zero-not-one ()
```

What can't be done to irrelevant arguments

Example 1. You can't use an irrelevant value in a non-irrelevant context.

```
bad-plus : Nat → .Nat → Nat
bad-plus n m = m + n
```

```
Variable m is declared irrelevant, so it cannot be used here
when checking that the expression m has type Nat
```

Example 2. You can't declare the function's return type as irrelevant.

```
bad : Nat → .Nat
bad n = 1
```

```
Invalid dotted expression
when checking that the expression .Nat has type Set _47
```

Example 3. You can't pattern match on an irrelevant value.

```
badMatching : Nat → .Nat → Nat
badMatching n zero    = n
badMatching n (suc m) = n
```

```
Cannot pattern match against irrelevant argument of type Nat
when checking that the pattern zero has type Nat
```

Example 4. We also can't match on an irrelevant record (see *Record Types*).

```
record Σ (A : Set) (B : A → Set) : Set where
  constructor _,_
  field
    fst : A
    snd : B fst

irrElim : {A : Set} {B : A → Set} → .(Σ A B) → _
irrElim (a , b) = ?
```

```
Cannot pattern match against irrelevant argument of type Σ A B
when checking that the pattern a , b has type Σ A B
```

If this were allowed, b would have type $B\ a$ but this type is not even well-formed because a is irrelevant!

3.19.3 Irrelevant declarations

Postulates and functions can be marked as irrelevant by prefixing the name with a dot when the name is declared. Irrelevant definitions can only be used as arguments of functions of an irrelevant function type `.A → B`.

Examples:

```
.irrFunction : Nat → Nat
irrFunction zero    = zero
irrFunction (suc n) = suc (suc (irrFunction n))

postulate
  .assume-false : (A : Set) → A
```

An important example is the irrelevance axiom `irrAx`:

```
postulate
  .irrAx : ∀ {ℓ} {A : Set ℓ} -> .A -> A
```

This axiom is not provable inside Agda, but it is often very useful when working with irrelevance. The irrelevance axiom is a form of non-constructive choice.

3.19.4 Irrelevant record fields

Record fields (see *Record Types*) can be marked as irrelevant by prefixing their name with a dot in the definition of the record type. Projections for irrelevant fields are only created if option *--irrelevant-projections* is supplied (since Agda > 2.5.4).

Example 1. A record type containing pairs of numbers satisfying certain properties.

```
record InterestingNumbers : Set where
  field
    n      : Nat
    m      : Nat
    .prop1 : n + m ≡ n * m + 2
    .prop2 : suc m ≤ n
```

Example 2. For any type `A`, we can define a 'squashed' version `Squash A` where all elements are equal.

```
record Squash (A : Set) : Set where
  constructor squash
  field
    .unsquash : A

open Squash

.example : ∀ {A} → Squash A → A
example x = unsquash x
```

Example 3. We can define the subset of `x : A` satisfying `P x` with irrelevant membership certificates.

```
record Subset (A : Set) (P : A -> Set) : Set where
  constructor _#_
  field
    elem         : A
    .certificate : P elem

.certificate : {A : Set}{P : A -> Set} -> (x : Subset A P) -> P (Subset.elem x)
certificate (a # p) = irrAx p
```

Example 4. Irrelevant projections are justified by the irrelevance axiom.

```
.unsquash' : ∀ {A} → Squash A → A
unsquash' (squash x) = irrAx x

.irrAx' : ∀ {A} → .A → A
irrAx' x = unsquash (squash x)
```

Like the irrelevance axiom, irrelevant projections cannot be reduced.

3.19.5 Dependent irrelevant function types

Just like non-dependent functions, we can also make dependent functions irrelevant. The basic syntax is as in the following examples:

```
f : .(x y : A) → B
f : .{x y z : A} → B
f : .(xs {ys zs} : A) → B
f : ∀ x .y → B
f : ∀ x .{y} {z} .v → B
f : .{{x : A}} → B
```

The declaration

```
f : .(x : A) → B[x]
f x = t[x]
```

requires that `x` is irrelevant both in `t[x]` and in `B[x]`. This is possible if, for instance, `B[x] = C x`, with `C : .A → Set`.

Dependent irrelevance allows us to define the eliminator for the Squash type:

```
elim-Squash : {A : Set} (P : Squash A → Set)
              (ih : .(a : A) → P (squash a)) →
              (a⁻ : Squash A) → P a⁻
elim-Squash P ih (squash a) = ih a
```

Note that this would not type-check with `(ih : (a : A) → P (squash a))`.

3.19.6 Irrelevant instance arguments

Contrary to normal instance arguments, irrelevant instance arguments (see *Instance Arguments*) are not required to have a unique solution.

```
record ⊤ : Set where
  instance constructor tt

NonZero : Nat → Set
NonZero zero    = ⊥
NonZero (suc _) = ⊤

pred´ : (n : Nat) .{{_ : NonZero n}} → Nat
pred´ zero {{}}
pred´ (suc n) = n

find-nonzero : (n : Nat) {{x y : NonZero n}} → Nat
find-nonzero n = pred´ n
```

3.20 Lambda Abstraction

3.20.1 Lambda expressions

Anonymous functions can be defined using a lambda expression `\x → u`:

```
myFun = \ x → x + x -- equivalent: `myFun x = x + x`
```

You can also use the Unicode symbol λ (type "\lambda" or "\Gl" in the Emacs Agda mode) instead of `\` (type "\\" in the Emacs Agda mode).

Lambda expressions can take several arguments and arguments can optionally be annotated with a type. For instance, both expressions below have type `(A : Set) → A → A` (the second expression checks against other types as well):

```
example₁ = \ (A : Set)(x : A) → x
example₂ = \ A x → x
```

A functions taking `n` (> 1) arguments is equivalent to a function that takes a single argument and returns another function with `n-1` arguments, i.e. functions are curried.

```
curry : (λ x y → x + y) ≡ (λ x → (λ y → x + y))
curry = refl
```

All functions in Agda satisfy η-equality: `f` is (definitionally) equal to `λ x → f x`.

```
etaFun : myFun ≡ λ x → myFun x
etaFun = refl
```

In particular, two lambda expressions with the same body up to renaming of the argument(s) are considered equal.

```
alpha : (λ x → x + 1) ≡ (λ y → y + 1)
alpha = refl
```

Lambda expressions can take *Implicit Arguments* and *Instance Arguments* by adding curly braces (resp. double curly braces) around the argument.

```
implicit-lambda = λ {A : Set} (x : A) → x

instance-lambda = λ (A : Set) {{monoid-A : Monoid A}} → mempty
```

Arguments to lambda expressions can also be annotated with any *modality*, for instance with *erasure status*.

Note that in Cubical Agda (see `--cubical`), many of the examples above do not pass because there the types of lambda expressions are not *inferred* in general; lambdas are only *checked* against given types. Thus, type signatures are needed for `myFun` etc.

3.20.2 Pattern lambda

Anonymous pattern matching functions can be defined by a *pattern lambda* using one of the two following syntaxes:

```
\ { p11 .. p1n -> e1 ; ... ; pm1 .. pmn -> em }

\ where
  p11 .. p1n -> e1
  ...
  pm1 .. pmn -> em
```

(where, as usual, `\` and `->` can be replaced by λ and →). Note that the `where` keyword introduces an *indented* block of clauses; if there is only one clause then it may be used inline.

Examples of pattern lambdas:

```
and : Bool → Bool → Bool
and = λ { true x → x ; false _ → false }

xor : Bool → Bool → Bool
xor = λ { true  true  → false
        ; false false → false
        ; _     _     → true
        }

eq : Bool → Bool → Bool
eq = λ where
  true  true  → true
  false false → true
  _ _ → false

myFst : {A : Set} {B : A → Set} → Σ A B → A
myFst = λ { (a , b) → a }

mySnd : {A : Set} {B : A → Set} (p : Σ A B) → B (fst p)
mySnd = λ { (a , b) → b }
```

(continues on next page)

(continued from previous page)

```
swap : {A B : Set} → A × B → B × A
swap = λ where (a , b) → (b , a)
```

Pattern lambdas can also use *Copatterns* by using projections in *postfix notation*.

```
swap' : {A B : Set} → A × B → B × A
swap' = λ where
  (a , b) .fst → b
  (a , b) .snd → a
```

It is not allowed to use `where` and `with` constructions in pattern lambdas.

Internal representation of pattern lambdas

Internally pattern lambdas are translated into a function definition of the following form:

```
extlam p11 .. p1n = e1
...
extlam pm1 .. pmn = em
```

where *extlam* is a fresh name. In other words, pattern lambdas are *generative*. In particular, two pattern lambdas with the same body are not considered equal by Agda (in contrast to regular lambda expressions).

```
(λ { true → true ; false → false }) ==
(λ { true → true ; false → false })
```

This type is equivalent to `extlam1 ≡ extlam2` for some distinct fresh names `extlam1` and `extlam2`, hence cannot be proven with `refl`.

3.20.3 Absurd lambda

An *absurd lambda* is a lambda expression that uses an *absurd pattern* `()`.

```
absurd-lambda : 0 ≡ 1 → ⊥
absurd-lambda = λ ()
```

Unlike general pattern lambdas, absurd lambdas do not require curly braces or the `where` keyword, although using them is still allowed.

```
absurd-lambda-curly : 0 ≡ 1 → ⊥
absurd-lambda-curly = λ { () }

absurd-lambda-where : 0 ≡ 1 → ⊥
absurd-lambda-where = λ where ()
```

It is also allowed to have regular arguments before or after the absurd pattern.

```
absurd-lambda-list : {A : Set} (x : A) (xs : List A) → x ∷ xs ≡ [] → ⊥
absurd-lambda-list = λ x xs ()
```

3.21 Local Definitions: let and where

There are two ways of declaring local definitions in Agda:

- let-expressions
- where-blocks

3.21.1 let-expressions

A let-expression defines an abbreviation. This means that let-bound functions have to make sense as pure lambda expressions: they can not be recursive, and can not be defined by pattern matching on inductive types.

Example:

```
f : Nat
f = let h : Nat → Nat
        h m = suc (suc m)
    in  h zero + h (suc zero)
```

However, it is possible to match on *record* types in the left-hand side of a let-bound function, as described *below*:

```
g : Nat
g = let h : Nat × Nat → Nat
        h (x , y) = x + y
     in h (1 , 2)
```

A let-expression has the general form

```
let f₁ : A₁₁ → ... → A₁ₙ → A₁
    f₁ x₁ ... xₙ = e₁
    ...
    fₘ : Aₘ₁ → ... → Aₘₖ → Aₘ
    fₘ x₁ ... xₖ = eₘ
in  e'
```

where previous definitions are in scope in later definitions. The type signatures can be left out if Agda can infer them. After type-checking, the meaning of this is simply the substitution `e'[f₁ := λ x₁ ... xₙ → e; ...; fₘ := λ x₁ ... xₖ → eₘ]`. Since Agda substitutes away let-bindings, they do not show up in terms Agda prints, nor in the goal display in interactive mode.

> **Warning**
>
> The *internal syntax used by Agda* does not have let-expressions as a construct. As a result, Agda inlines all let-bound variables during type checking. In particular, let-binding cannot be used to introduce sharing.

Let binding record patterns

For a record

```
record R : Set where
  constructor c
  field
    f : X
```

(continues on next page)

(continued from previous page)

```
    g : Y
    h : Z
```

a let expression of the form

```
let (c x y z) = t
in  u
```

will be translated internally to as

```
let x = f t
    y = g t
    z = h t
in  u
```

This is not allowed if `R` is declared `coinductive`. If `R` lacks eta-equality, e.g. by being declared with `no-eta-equality`, then warning *ShouldBeEtaRecordPattern* is triggered (since Agda 2.9.0).

Let binding record patterns

Let-expressions can be used to locally open a *module*. For example:

```
let z = x + y
    open M z
in  u          -- using definitions from M
```

3.21.2 where-blocks

where-blocks are much more powerful than let-expressions, as they support arbitrary local definitions. A `where` can be attached to any function clause.

where-blocks have the general form

```
clause
  where
  decls
```

or

```
clause
  module M where
  decls
```

A simple instance is

```
g ps = e
  where
  f : A₁ → ... → Aₙ → A
  f p₁₁ ... p₁ₙ= e₁
  ...
  ...
  f pₘ₁ ... pₘₙ= eₘ
```

Here, the $\mathtt{p}_{ij}$ are patterns of the corresponding types and $\mathtt{e}_i$ is an expression that can contain occurrences of `f`. Functions defined with a where-expression must follow the rules for general definitions by pattern matching.

Example:

```
reverse : {A : Set} → List A → List A
reverse {A} xs = rev-append xs []
  where
  rev-append : List A → List A → List A
  rev-append [] ys = ys
  rev-append (x :: xs) ys = rev-append xs (x :: ys)
```

Variable scope

The pattern variables of the parent clause of the where-block are in scope; in the previous example, these are `A` and `xs`. The variables bound by the type signature of the parent clause are not in scope. This is why we added the hidden binder `{A}`.

Scope of the local declarations

The `where`-definitions are not visible outside of the clause that owns these definitions (the parent clause). If the `where`-block is given a name (form `module M where`), then the definitions are available as qualified by `M`, since module `M` is visible even outside of the parent clause. The special form of an anonymous module (`module _ where`) makes the definitions visible outside of the parent clause without qualification.

If the parent function of a named `where`-block (form `module M where`) is `private`, then module `M` is also `private`. However, the declarations inside `M` are not private unless declared so explicitly. Thus, the following example scope checks fine:

```
module Parent₁ where
  private
    parent = local
      module Private where
      local = Set
  module Public = Private

test₁ = Parent₁.Public.local
```

Likewise, a `private` declaration for a parent function does not affect the privacy of local functions defined under a `module _ where`-block:

```
module Parent₂ where
  private
    parent = local
      module _ where
      local = Set

test₂ = Parent₂.local
```

They can be declared `private` explicitly, though:

```
module Parent₃ where
  parent = local
    module _ where
    private
      local = Set
```

Now, `Parent₃.local` is not in scope.

A `private` declaration for the parent of an ordinary `where`-block has no effect on the local definitions, of course. They are not even in scope.

No named `where`-modules under `with` or `rewrite`

Since Agda 2.9.0, a clause that uses *with-abstraction* — be it via `with`, *rewrite*, or *with* $p \leftarrow e$ — must not carry a named `where`-module. The following is rejected with error `NamedWhereModuleUnderWith`:

```
f : Bool → Bool
f x with x
... | true  = local
  module M where   -- rejected since Agda 2.9.0
  local = false
... | false = true
```

The reason is that with-abstraction can change the types of the module parameters that `M` inherits from its parent module. However, these changed parameters are not visible in the source code which might be confusing. Ignoring the changes, i.e., making `M` available outside of the clause with the original module parameters inherited from the parent module, even leads to inconsistencies.

Ordinary anonymous `where`-blocks are unaffected by this restriction, and so is *using* $p \leftarrow e$, which does not with-abstract but introduces a let-binding:

```
ok : Bool → Bool
ok x with x
... | true  = local
  where
  local = false
... | false = true

ok' : Bool → Bool
ok' x using y ← x = local
  module M' where
  local = y
```

If you need to refer to the local definitions from outside the clause, define them in a proper module instead, or move the with-abstraction into a helper function.

3.21.3 Proving properties

Sometimes one needs to refer to local definitions in proofs about the parent function. In this case, the `module ⋯ where` variant is preferable.

```
reverse : {A : Set} → List A → List A
reverse {A} xs = rev-append xs []
   module Rev where
   rev-append : List A → List A → List A
   rev-append [] ys = ys
   rev-append (x :: xs) ys = rev-append xs (x :: ys)
```

This gives us access to the local function as

```
Rev.rev-append : {A : Set} (xs : List A) → List A → List A → List A
```

Alternatively, we can define local functions as private to the module we are working in; hence, they will not be visible in any module that imports this module but it will allow us to prove some properties about them.

```
private
   rev-append : {A : Set} → List A → List A → List A
   rev-append []       ys = ys
   rev-append (x :: xs) ys = rev-append xs (x :: ys)

reverse' : {A : Set} → List A → List A
reverse' xs = rev-append xs []
```

3.21.4 More Examples (for Beginners)

Using a let-expression:

```
tw-map : {A : Set} → List A → List (List A)
tw-map {A} xs = let twice : List A → List A
                    twice xs = xs ++ xs
                in  map (\ x → twice [ x ]) xs
```

Same definition but with less type information:

```
tw-map' : {A : Set} → List A → List (List A)
tw-map' {A} xs = let twice : _
                     twice xs = xs ++ xs
                 in  map (\ x → twice [ x ]) xs
```

Same definition but with a where-expression

```
tw-map'' : {A : Set} → List A → List (List A)
tw-map'' {A} xs =  map (\ x → twice [ x ]) xs
   where twice : List A → List A
         twice xs = xs ++ xs
```

Even less type information using let:

```
h : Nat → List Nat
h zero    = [ zero ]
h (suc n) = let sing = [ suc n ]
            in  sing ++ h n
```

Same definition using where:

```
h' : Nat → List Nat
h' zero = [ zero ]
h' (suc n) = sing ++ h' n
   where  sing = [ suc n ]
```

More than one definition in a let:

```
i : Nat → Nat
i n = let add2 : Nat
          add2 = suc (suc n)

          twice : Nat → Nat
          twice m = m * m
```

(continues on next page)

(continued from previous page)

```
    in twice add2
```

More than one definition in a where:

```
fibfact : Nat → Nat
fibfact n = fib n + fact n
 where fib : Nat → Nat
       fib zero = suc zero
       fib (suc zero) = suc zero
       fib (suc (suc n)) = fib (suc n) + fib n

       fact : Nat → Nat
       fact zero = suc zero
       fact (suc n) = suc n * fact n
```

Combining let and where:

```
k : Nat → Nat
k n = let aux : Nat → Nat
          aux m = pred (i m) + fibfact m
      in aux (pred n)
  where pred : Nat → Nat
        pred zero = zero
        pred (suc m) = m
```

3.22 Lexical Structure

Agda code is written in UTF-8 encoded plain text files with the extension `.agda`; more file extensions are supported for *Literate Programming*. A UTF-8 byte order mark (BOM) is ignored at the beginning of a file.

Most unicode characters can be used in identifiers, see section *Names*. Whitespace is important, see section *Layout*.

3.22.1 Tokens

Keywords and special symbols

Most non-whitespace unicode can be used as part of an Agda name, but there are two kinds of exceptions:

special symbols
: Characters with special meaning that cannot appear at all in a name. These are `.;{}()@"`.

keywords
: Reserved words that cannot appear as a *name part*, but can appear in a name together with other characters.

  `=` `|` `->` `→` `:` `?` `\` `λ` `∀` `..` `...` `abstract` `coinductive` `constructor` `data` *do* `eta-equality` `field` *forall* `hiding` `import` `in` `inductive` `infix` `infixl` `infixr` `instance` `interleaved` `let` *macro* `module` `mutual` `no-eta-equality` `opaque` `open` *overlap* `pattern` `postulate` `primitive` `private` `public` *quote* *quoteTerm* `record` `renaming` `rewrite` `syntax` `tactic` `unfolding` *unquote* *unquoteDecl* *unquoteDef* `using` *variable* `where` `with`

keywords in `renaming` directives
: The word `to` is only reserved in `renaming` directives.

keywords in `import` statements
: The word `as` has a special meaning in `import` statements, although it is not reserved.

Names

A *qualified name* is a non-empty sequence of *names* separated by dots (`.`). A *name* is an alternating sequence of *name parts* and underscores (_), containing at least one name part. A *name part* is a non-empty sequence of unicode characters, excluding whitespace, _, and *special symbols*. A name part cannot be one of the *keywords* above, and cannot start with a single quote, `'` (which are used for character literals, see *Literals* below).

Examples

- Valid: `data?`, `::`, `if_then_else_`, `0b`, `_⊢_∈_`, `x=y`
- Invalid: `data_?`, `foo__bar`, `_`, `a;b`, `[_.._]`

The underscores in a name indicate where the arguments go when the name is used as an operator. For instance, the application `_+_ 1 2` can be written as `1 + 2`. See *Mixfix Operators* for more information. Since most sequences of characters are valid names, whitespace is more important than in other languages. In the example above the whitespace around + is required, since `1+2` is a valid name.

Qualified names are used to refer to entities defined in other modules. For instance `Prelude.Bool.true` refers to the name `true` defined in the module `Prelude.Bool`. See *Module System* for more information.

Literals

There are four types of literal values: integers, floats, characters, and strings. See *Built-ins* for the corresponding types, and *Literal Overloading* for how to support literals for user-defined types.

Integers

Integer values in decimal, hexadecimal (prefixed by `0x`), or binary (prefixed by `0b`) notation. The character _ can be used to separate groups of digits. Non-negative numbers map by default to *built-in natural numbers*, but can be overloaded. Negative numbers have no default interpretation and can only be used through *overloading*.

Examples: `123`, `0xF0F080`, `-42`, `-0xF`, `0b11001001`, `1_000_000_000`, `0b01001000_01001001`.

Floats

Floating point numbers in the standard notation (with square brackets denoting optional parts):

```
float    ::= [-] decimal . decimal [exponent]
           | [-] decimal exponent
exponent ::= (e | E) [+ | -] decimal
```

These map to *built-in floats* and cannot be overloaded.

Examples: `1.0`, `-5.0e+12`, `1.01e-16`, `4.2E9`, `50e3`.

Characters

Character literals are enclosed in single quotes (`'`). They can be a single (unicode) character, other than `'` or \, or an escaped character. Escaped characters start with a backslash \ followed by an escape code. Escape codes are natural numbers in decimal or hexadecimal (prefixed by `x`) between `0` and `0x10ffff` (`1114111`), or one of the following special escape codes:

Code	ASCII	Code	ASCII	Code	ASCII	Code	ASCII
a	7	b	8	t	9	n	10
v	11	f	12	\	\	'	'
"	"	NUL	0	SOH	1	STX	2
ETX	3	EOT	4	ENQ	5	ACK	6
BEL	7	BS	8	HT	9	LF	10
VT	11	FF	12	CR	13	SO	14
SI	15	DLE	16	DC1	17	DC2	18
DC3	19	DC4	20	NAK	21	SYN	22
ETB	23	CAN	24	EM	25	SUB	26
ESC	27	FS	28	GS	29	RS	30
US	31	SP	32	DEL	127		

Character literals map to the *built-in character type* and cannot be overloaded.

Examples: `'A'`, `'∀'`, `'\x2200'`, `'\ESC'`, `'\32'`, `'\n'`, `'\''`, `'"'`.

Strings

String literals are sequences of, possibly escaped, characters enclosed in double quotes `"`. They follow the same rules as *character literals* except that double quotes `"` need to be escaped rather than single quotes `'`. String literals map to the *built-in string type* by default, but can be *overloaded*.

Example: `"PDF TODOPDF TODOPDF TODOPDF TODOPDF TODOPDF TODO \ "PDF TODOPDF TODOPDF TODO\"\n"`.

Holes

Holes are an integral part of the interactive development supported by the *Emacs mode*. Any text enclosed in `{!` and `!}` is a hole and may contain nested holes. A hole with no contents can be written `?`. There are a number of Emacs commands that operate on the contents of a hole. The type checker ignores the contents of a hole and treats it as an unknown (see *Implicit Arguments*).

Example: `{! f {!x!} 5 !}`

Comments

Single-line comments are written with a double dash `--` followed by arbitrary text. Multi-line comments are enclosed in `{-` and `-}` and can be nested. Comments cannot appear in *string literals*.

Example:

```
{- Here is a {- nested -}
   comment -}
s : String --line comment {-
s = "{- not a comment -}"
```

Pragmas

Pragmas are special comments enclosed in `{-#` and `#-}` that have special meaning to the system. See *Pragmas* for a full list of pragmas.

3.22.2 Layout

Agda is layout sensitive using similar rules as Haskell, with the exception that layout is mandatory: you cannot use explicit `{`, `}` and `;` to avoid it.

A layout block contains a sequence of *statements* and is started by one of the layout keywords:

```
abstract
constructor
do
field
instance
let
macro
mutual
opaque
postulate
primitive
private
variable
where
```

The first token after the layout keyword decides the indentation of the block. Any token indented more than this is part of the previous statement, a token at the same level starts a new statement, and a token indented less lies outside the block.

```
data Nat : Set where -- starts a layout block
    -- comments are not tokens
  zero : Nat      -- statement 1
  suc  : Nat →    -- statement 2
         Nat      -- also statement 2

one : Nat -- outside the layout block
one = suc zero
```

Note that the indentation of the layout keyword does not matter.

If several layout blocks are started by layout keywords without line break in between (where line breaks inside block comments do not count), then those blocks indented *more* than the last block go passive, meaning they cannot be further extended by new statements:

```
private module M where postulate
          A : Set                -- module-block goes passive
          B : Set                -- postulate-block can still be extended
        module N where           -- private-block can still be extended
```

An Agda file contains one top-level layout block, with the special rule that the contents of the top-level module need not be indented.

```
module Example where
NotIndented : Set₁
NotIndented = Set
```

3.22.3 Literate Agda

Agda supports literate programming with multiple typesetting tools like LaTeX, Markdown and reStructuredText. For instance, with LaTeX, everything in a file is a comment unless enclosed in `\begin{code}`, `\end{code}`. Literate Agda files have special file extensions, like `.lagda` and `.lagda.tex` for LaTeX, `.lagda.md` for Markdown, `.lagda.rst` for reStructuredText instead of `.agda`. One use case for literate Agda files is to generate documents including Agda code. See *Generating HTML* and *Generating LaTeX* for more information.

```
\documentclass{article}
% some preamble stuff
\begin{document}
Introduction usually goes here
\begin{code}
module MyPaper where
  open import Prelude
  five : Nat
  five = 2 + 3
\end{code}
Now, conclusions!
\end{document}
```

3.23 Literal Overloading

3.23.1 Natural numbers

By default *natural number literals* are mapped to the *built-in natural number type*. This can be changed with the `FROMNAT` built-in, which binds to a function accepting a natural number:

```
{-# BUILTIN FROMNAT fromNat #-}
```

This causes natural number literals `n` to be desugared to `fromNat n`, whenever `fromNat` is in scope *unqualified* (renamed or not). Note that the desugaring happens before *implicit argument* are inserted so `fromNat` can have any number of implicit or *instance arguments*. This can be exploited to support overloaded literals by defining a *type class* containing `fromNat`:

```
module number-simple where

  record Number {a} (A : Set a) : Set a where
    field fromNat : Nat → A

  open Number {{...}} public
```

```
{-# BUILTIN FROMNAT fromNat #-}
```

This definition requires that any natural number can be mapped into the given type, so it won't work for types like `Fin n`. This can be solved by refining the `Number` class with an additional constraint:

```
record Number {a} (A : Set a) : Set (lsuc a) where
  field
    Constraint : Nat → Set a
    fromNat : (n : Nat) {{_ : Constraint n}} → A

open Number {{...}} public using (fromNat)
```

(continues on next page)

(continued from previous page)

```
{-# BUILTIN FROMNAT fromNat #-}
```

This is the definition used in `Agda.Builtin.FromNat`. A `Number` instance for `Nat` is simply this:

```
instance
  NumNat : Number Nat
  NumNat .Number.Constraint _ = ⊤
  NumNat .Number.fromNat    m = m
```

A `Number` instance for `Fin n` can be defined as follows:

```
_≤_ : (m n : Nat) → Set
zero  ≤ n     = ⊤
suc m ≤ zero  = ⊥
suc m ≤ suc n = m ≤ n

fromN≤ : ∀ m n → m ≤ n → Fin (suc n)
fromN≤ zero    _       _  = zero
fromN≤ (suc _) zero    ()
fromN≤ (suc m) (suc n) p  = suc (fromN≤ m n p)

instance
  NumFin : ∀ {n} → Number (Fin (suc n))
  NumFin {n} .Number.Constraint m         = m ≤ n
  NumFin {n} .Number.fromNat    m {{m≤n}} = fromN≤ m n m≤n

test : Fin 5
test = 3
```

It is important that the constraint for literals is trivial. Here, `3 ≤ 5` evaluates to ⊤ whose inhabitant is found by unification.

Using predefined function from the standard library and instance `NumNat`, the `NumFin` instance can be simply:

```
open import Data.Fin using (Fin; #_)
open import Data.Nat using (suc; _≤?_)
open import Relation.Nullary.Decidable using (True)

instance
  NumFin : ∀ {n} → Number (Fin n)
  NumFin {n} .Number.Constraint m         = True (suc m ≤? n)
  NumFin {n} .Number.fromNat    m {{m<n}} = #_ m {m<n = m<n}
```

Note

Overloading numeric literals only works in expressions, not in patterns. The following is rejected:

```
isZero : ∀ {n} → Fin n → Bool
isZero 0 = true   -- error: zero is not a constructor of the datatype Fin
isZero _ = false
```

3.23.2 Negative numbers

Negative integer literals have no default mapping and can only be used through the `FROMNEG` built-in. Binding this to a function `fromNeg` causes negative integer literals `-n` to be desugared to `fromNeg n`, where `n` is a *built-in natural number*. From `Agda.Builtin.FromNeg`:

```
record Negative {a} (A : Set a) : Set (lsuc a) where
  field
    Constraint : Nat → Set a
    fromNeg : (n : Nat) {{_ : Constraint n}} → A

open Negative {{...}} public using (fromNeg)
{-# BUILTIN FROMNEG fromNeg #-}
```

3.23.3 Strings

String literals are overloaded with the `FROMSTRING` built-in, which works just like `FROMNAT`. If it is not bound string literals map to *built-in strings*. From `Agda.Builtin.FromString`:

```
record IsString {a} (A : Set a) : Set (lsuc a) where
  field
    Constraint : String → Set a
    fromString : (s : String) {{_ : Constraint s}} → A

open IsString {{...}} public using (fromString)
{-# BUILTIN FROMSTRING fromString #-}
```

3.23.4 Restrictions

Currently only integer and string literals can be overloaded.

Overloading does not work in patterns yet.

3.24 Local Rewriting

Local rewrite rules is an experimental feature which enables parameterising modules over computation rules. Specifically, it allows declaring module parameters targetting a rewrite relation as local rewrite rules by annotating with them the `@rewrite` attribute. Consequently:

- Inside the module, local rewrite rules will automatically apply during reduction, rewriting instances of the left-hand side to the right-hand side, similarly to *global rewriting*.
- Outside the module, local rewrite rules act as constraints on instantiations of the module parameters. E.g. when opening the module, Agda will check that both sides are definitionally equal.

This feature is based on Local Rewriting Type Theory (LRTT) as introduced in Encode the Cake and Eat It Too, by Yann Leray and Théo Winterhalter. Unlike their presentation, we do not make a strong syntactic distinction between the "interface environment" and the "local context", but nonetheless, by restricting `@rewrite` attributes to module parameters, quantification over rewrites is prenex-only. For example, definitions cannot return local rewrite rules, or be parameterised over other definitions taking local rewrite rules, so `foo : (n : Nat) → ((@rewrite p : n ≡ 0) → Nat) → Nat` is not allowed.

Semantically, local rewrite rules can be eliminated by inlining all instantiations of modules with local rewrite rule parameters and so should be conservative over the rest of Agda's theory.

> **Note**
>
> This page is about the *--local-rewriting* option. This is is unrelated to *--local-confluence-check*, which enables a form of confluence checking for *global REWRITE rules* whilst using the *--rewriting* option.
>
> It is also (currently) distinct from the the *rewrite construct*, although there are plans to implement a new version of *with-abstraction* which internally desugars to local rewrite rules ("smart with").

3.24.1 Local rewrite rules by example

```
{-# OPTIONS --local-rewriting --rewriting #-}

module language.local-rewriting where

open import Agda.Builtin.Equality
open import Agda.Builtin.Equality.Rewrite
```

To motivate local rewrite rules, consider the following code which implements addition and proves associativity for Agda's built-in natural numbers.

```
module Addition where
  _+_ : Nat → Nat → Nat
  zero  + m = m
  suc n + m = suc (n + m)

  +-assoc : ∀ {n m l} → (n + m) + l ≡ n + (m + l)
  +-assoc {n = zero}  = refl
  +-assoc {n = suc n} = cong suc (+-assoc {n = n})
```

Now imagine we want to use a different encoding of natural numbers - for example, lists of unit values.

```
Nat' = List ⊤
```

To define addition and prove associativity for `Nat'` without duplication, we can parameterise our addition and the associativity definitions over an abstract type of natural numbers and an induction principle.

```
module ParametricAddition
  (Nat : Set) (zero : Nat) (suc : Nat → Nat)
  (ind : (P : Nat → Set) → P zero → (∀ n → P n → P (suc n)) → ∀ n → P n)
  (ind-zero : ∀ {P z s} → ind P z s zero ≡ z)
  (ind-suc  : ∀ {P z s n} → ind P z s (suc n) ≡ s n (ind P z s n))
  where
  _+_ : Nat → Nat → Nat
  n + m = ind (λ _ → Nat) m (λ _ → suc) n

  +-assoc : ∀ {n m l} → (n + m) + l ≡ n + (m + l)
  +-assoc {n = n} {m = m} {l = l}
    = ind (λ □ → (□ + m) + l ≡ □ + (m + l))
          (trans (cong (_+ l) ind-zero) (sym ind-zero))
          (λ _ h → trans (cong (_+ l) ind-suc)
                 ( trans ind-suc
                 ( trans (cong suc h) (sym ind-suc))))
          n
```

We have succeeded in writing a single parametric definition of addition and associativity, but at the cost of a much more convoluted proof. The parameterised-over induction principle no longer computes on zero and successor automatically, so we have to manually invoke `ind-zero` and `ind-suc` multiple times.

Local rewrite rules resolve this tedium. We can simply annotate the `ind-zero` and `ind-suc` equations with `@rewrite` and recover the simple associativity proof, whilst staying parametric over encoding details.

```
module ParametricAdditionRew
  (Nat : Set) (zero : Nat) (suc : Nat → Nat)
  (ind : (P : Nat → Set) → P zero → (∀ n → P n → P (suc n)) → ∀ n → P n)
  (@rewrite ind-zero : ∀ {P z s} → ind P z s zero ≡ z)
  (@rewrite ind-suc  : ∀ {P z s n} → ind P z s (suc n) ≡ s n (ind P z s n))
  where
  _+_ : Nat → Nat → Nat
  n + m = ind (λ _ → Nat) m (λ _ → suc) n

  +-assoc : ∀ {n m l} → (n + m) + l ≡ n + (m + l)
  +-assoc {n = n} {m = m} {l = l}
    = ind (λ □ → (□ + m) + l ≡ □ + (m + l)) refl (λ _ h → cong suc h) n
```

For more examples on where local rewrite rules can be useful, see the Encode the Cake and Eat It Too paper.

3.24.2 Limitations

Confluence and termination checking

Currently, no form of confluence or termination checking is implemented for local rewrite rules. The consequences of non-confluent or non-terminating local rewrite rules are similar to *global rewriting*: non-confluence endangers subject reduction and non-termination might cause the typechecker to loop, but logical soundness should never be threatened.

Refining local rewrite rules with pattern matching

Currently, inaccessible pattern matches on variables that occur in local rewrite rules are not allowed. This is to avoid typechecking under invalid rewrite rules (rewrite validity is unstable under substitution).

```
module _ (f : Nat → Nat) (@rewrite p : f 1 ≡ 0) where
  bad : f ≡ (λ x → x) → Nat
  bad refl = {!!} -- Substituting 'f' for 'λ x → x' here would invalidate the
                  -- local rewrite rule 'p'
```

Furthermore, matches on variables in local rewrite rules breaks the inlining-based semantic justification.

In spite of these downsides, there are plans to relax this restriction in the future under an additional flag. Refining local rewrite rules with pattern matches enables a restricted form of "local equality reflection", which has many interesting applications, including a (hopefully) better-behaved *with-abstraction* mechanism.

Parameterising datatypes over local rewrite rules with *--cubical*

In *Cubical Agda*, there is an additional limitation with local rewrite rules. Attempting to declare data or record types inside modules with local rewrite rule parameters will throw a `CannotGenerateTransportLocalRewrite` error:

```
module _ (n : Nat) (@rewrite _ : n ≡ 0) where
  data Foo : Set where
    mk : Foo
```

This restriction is due to how Cubical Agda automatically defines various primitives for datatypes for transport and path composition. It is not (currently) clear what these should look like for datatypes with local rewrite rule parameters.

If you run into this issue, try defining your data or record types outside of the module with the local rewrite rule.

3.25 Lossy Unification

The option *--lossy-unification* enables an experimental heuristic in the unification checker intended to improve its performance for unification problems of the form `f es₀ = f es₁`, i.e. unifying two applications of the same defined function, here `f`, to possibly different lists of arguments and projections `es₀` and `es₁`. The heuristic is sound but not complete. In particular if Agda accepts code with the flag enabled it should also accept it without the flag (with enough resources, and possibly needing extra type annotations).

The option can be used either globally or in an `OPTIONS` pragma, in the latter case it applies only to the current module. There is also a pragma `{-# INJECTIVE_FOR_INFERENCE f #-}` which enables lossy unification only for `f`.

3.25.1 Heuristic

When trying to solve the unification problem `f es₀ = f es₁` the heuristic proceeds by trying to solve `es₀ = es₁`, if that succeeds the original problem is also solved, otherwise unification proceeds as without the flag, likely by reducing both `f es₀` and `f es₁`.

3.25.2 Example

Suppose `f` adds `100` to its input as defined below

```
f : ℕ → ℕ
f n = 100 + n
```

then to unify `f 2` and `f (1 + 1)` the heuristic would proceed by unifying `2` with `(1 + 1)`, which quickly succeeds. Without the flag we might instead first reduce both `f 2` and `f (1 + 1)` to `102` and then compare those results.

The performance will improve most dramatically when reducing an application of `f` would produce a large term, perhaps an element of a record type with several fields and/or large embedded proof terms.

3.25.3 Drawbacks

One drawback is that in some cases performance of unification will be worse with the heuristic. Specifically, if the heuristic will repeatedly attempt to unify lists of arguments `es₀ = es₁` while failing.

The main drawback is that the heursitic is not complete, i.e. it will cause Agda to ignore some possible solutions to unification variables. For example if `f` is a constant function, then the constraint `f ?0 = f 1` does not uniquely determine `?0`, but the heuristic will end up assigning `1` to `?0`. However, if `f` is injective this heuristic is complete.

Such assignments can lead to Agda to report a type error which would not have been reported without the heuristic. This is because committing to `?0 = 1` might make other constraints unsatifiable.

Such assignments might also confuse readers. Note that with non-lossy unification you have the guarantee (in the absence of bugs) that, if the code type-checks, and you can find one consistent way to instantiate all meta-variables, then that is the way that the code is interpreted by the system. With lossy unification the solution you have in mind might not be the one the system uses.

Consider the following code, which is based on an example from López Juan's PhD thesis (see Listing 6.16):

```
{-# OPTIONS --lossy-unification #-}
```

```
open import Agda.Builtin.Bool
open import Agda.Builtin.Nat
```

(continues on next page)

(continued from previous page)

```
private variable
  m n : Nat

infixr 5 _::_ _++_

data Bit-vector : Nat → Set where
  []  : Bit-vector 0
  _::_ : Bool → Bit-vector n → Bit-vector (suc n)

_++_ : Bit-vector m → Bit-vector n → Bit-vector (m + n)
[]       ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)

replicate : ∀ n → Bool → Bit-vector n
replicate zero    x = []
replicate (suc n) x = x :: replicate n x

vector : Bit-vector (m + n)
vector {m = m} {n = n} = replicate m true ++ replicate n false
```

Can you confidently predict the values of all of the following four bit-vectors? Are you sure that readers of your code can do this?

```
ex₁ : Bit-vector (0 + 1)
ex₁ = vector

ex₂ : Bit-vector (1 + 0)
ex₂ = vector

ex₃ : Bit-vector ((0 + 1) + (1 + 0))
ex₃ = xs ++ xs
  where
  xs = vector

ex₄ : Bit-vector ((0 + 1) + (1 + 0))
ex₄ = xs ++ xs
  where
  xs = vector {m = _}
```

References

Slow typechecking of single one-line definition, issue (#1625).

3.26 Mixfix Operators

A type name, function name, or constructor name can comprise one or more name parts if we separate them with underscore characters _, and the resulting name can be used as an operator. From left to right, each argument goes in the place of each underscore _.

For instance, we can join with underscores the name parts `if`, `then`, and `else` into a single name `if_then_else_`. The application of the function name `if_then_else_` to some arguments named `x`, `y`, and `z` can still be written as:

- a standard application by using the full name `if_then_else_ x y z`

- an operator application by placing the arguments between the name parts `if x then y else z`, leaving a space between arguments and part names
- other *sections* of the full name, for instance leaving one or two underscores:
  - `(if_then y else z) x`
  - `(if x then_else z) y`
  - `if x then y else_ z`
  - `if x then_else_ y z`
  - `if_then y else_ x z`
  - `(if_then_else z) x y`

Examples of type names, function names, and constructor names as mixfix operators:

```
-- Example type name _⇒_
_⇒_   : Bool → Bool → Bool
true  ⇒ b = b
false ⇒ _ = true

-- Example function name _and_
_and_ : Bool → Bool → Bool
true and x = x
false and _ = false

-- Example function name if_then_else_
if_then_else_ : {A : Set} → Bool → A → A → A
if true then x else y = x
if false then x else y = y

-- Example constructor name _::_
data List (A : Set) : Set where
  nil  : List A
  _::_ : A → List A → List A
```

3.26.1 Precedence

Consider the expression `false and true ⇒ false`. Depending on which of `_and_` and `_⇒_` has more precedence, it can either be read as `(false and true) ⇒ false` (which is `true`), or as `false and (true ⇒ false)` (which is `false`).

Each operator is associated to a precedence, which is a floating point number (can be negative and fractional!). The default precedence for an operator is 20.

> **Note**
>
> Please note that `->` is directly handled in the parser. As a result, the precedence of `->` is lower than any precedence you may declare with `infixl` and `infixr`.

If we give `_and_` more precedence than `_⇒_`, then we will get the first result:

```
infix 30 _and_
-- infix 20 _⇒_ (default)
```

(continues on next page)

(continued from previous page)

```
p-and : {x y z : Bool} →  x and y ⇒ z  ≡  (x and y) ⇒ z
p-and = refl

e-and : false and true ⇒ false  ≡  true
e-and = refl
```

But, if we declare a new operator _and'_ and give it less precedence than _⇒_, then we will get the second result:

```
_and'_ : Bool → Bool → Bool
_and'_ = _and_
infix 15 _and'_
-- infix 20 _⇒_ (default)

p-⇒ : {x y z : Bool} →  x and' y ⇒ z  ≡  x and' (y ⇒ z)
p-⇒ = refl

e-⇒ : false and' true ⇒ false  ≡  false
e-⇒ = refl
```

Fixities can be changed when importing with a `renaming` directive:

```
open M using (_•_)
open M renaming (_•_ to infixl 10 _*_)
```

This code brings two instances of the operator _•_ in scope:

- the first named _•_ and with its original fixity
- the second named _*_ and with the fixity changed to act like a left associative operator of precedence 10.

3.26.2 Associativity

Consider the expression `true ⇒ false ⇒ false`. Depending on whether _⇒_ associates to the left or to the right, it can be read as `(false ⇒ true) ⇒ false = false`, or `false ⇒ (true ⇒ false) = true`, respectively.

If we declare an operator _⇒_ as `infixr`, it will associate to the right:

```
infixr 20 _⇒_

p-right : {x y z : Bool} →  x ⇒ y ⇒ z  ≡  x ⇒ (y ⇒ z)
p-right = refl

e-right : false ⇒ true ⇒ false  ≡  true
e-right = refl
```

If we declare an operator _⇒'_ as `infixl`, it will associate to the left:

```
infixl 20 _⇒'_

_⇒'_ : Bool → Bool → Bool
_⇒'_ = _⇒_

p-left : {x y z : Bool} →  x ⇒' y ⇒' z  ≡  (x ⇒' y) ⇒' z
```

(continues on next page)

(continued from previous page)

```
p-left = refl

e-left : false ⇒' true ⇒' false  ≡  false
e-left = refl
```

3.26.3 Ambiguity and Scope

If you have not yet declared the fixity of an operator, Agda will complain if you try to use ambiguously:

```
e-ambiguous : Bool
e-ambiguous = true ⇒ true ⇒ true
```

```
Could not parse the application true ⇒ true ⇒ true
Operators used in the grammar:
  ⇒ (infix operator, level 20)
```

Fixity declarations may appear anywhere in a module that other declarations may appear. They then apply to the entire scope in which they appear (i.e. before and after, but not outside).

3.26.4 Operators in telescopes

Agda does not yet support declaring the fixity of operators declared in *telescopes*, see *Issue #1235 <https://github.com/agda/agda/issues/1235>*.

However, the following hack currently works:

```
module _ {A : Set} (_+_ : A → A → A) (let infixl 5 _+_; _+_ = _+_) where
```

3.27 Modalities

Agda uses the machinery of modalities to implement a couple of features. While they all generally have similar structure, these modality systems don't all have the same behavior and obey the same typing rules, especially regarding definitions and modules. They can be grouped into two styles: *Positional modality systems* and *Pure modality systems*. Here is the list of the modality systems in Agda:

- *Irrelevance*, which is positional, using dot prefixes or annotations `@irr`/`@irrelevant`, `@shirr`/`@shape-irrelevant` and `@relevant`;
- *Run-time Irrelevance*, positional, using `@0` and `@ω`;
- *Flat Modality*, pure, using `@♭` and `@⊤`, although the latter can never be written explicitely by the user;
- *Polarity Annotations*, pure, using `@++`, `@+`, `@-`, `@mixed` and `@unused`.

3.27.1 General modalities

Modality systems let you add modality annotations to the domain of arrow types, and also to definitions (this includes local definitions, such as in where and let blocks).

For example, a function of type `@`μ `A → B` means that it uses its argument μ-modally, for μ=`irr`, that means that the argument is irrelevant for the function! Annotations on a definition are a bit more complicated (and behave differently depending on the modality system), but as a first approximation, you can think of a definition `@`μ `f : A` to be usable only μ-modally.

When a variable is bound by e.g. pattern matching, Agda remembers under which modality it is available, and checks that the variable is only used under compatible modalities: as an example, you can use a relevant variable irrelevantly, but you can't use an irrelevant variable relevantly.

More formally, a modality system is given by:

- an ordered set of modalities, which govern how variables of different modalities can be used;
- a way to compose modalities, with a binary operation * that is monotone in both arguments, along with an identity modality `id` for that operation;
- a way to left-divide modalities, with an operation \, such that $\mu \leq \delta * \nu$ if and only if $\delta \setminus \mu \leq \nu$.

When no annotation is specified by the user, a default modality (not always the identity) is assigned.

A variable `@`μ `x : A` in the context is usable if $\mu \leq$ `id`, and for `f` a term of type `(@`ν `x : A)` $\to$ `B`, the subterm `s` in `f s` is checked with all modalities in the context left-divided by ν.

3.27.2 Positional modality systems

For positional modality systems, a definition of the form

```
module M Γ where
  @μ f : A
```

can only be used if the context has been divided by ν so that $\nu \setminus \mu \leq$ `id`, and their types would appear to the Agda user as `@`μ `f : Γ` $\to$ `A`.

Having a "boxed unboxing" for a modal system is the ability to derive `@`μ `unbox : @`μ `A` $\to$ `A` with `unbox x = x`. The erasure system has boxed unboxings, as well as irrelevance if *`--irrelevant-projections`* is enabled.

If a positional modal system has "boxed unboxings", then a definition such as `@`μ `f : A` is checked by first left-dividing the context by μ. This is why the following only type-checks with *`--irrelevant-projections`*:

```
module M (@irr A : Set) where
  @irr B : Set
  B = A
```

These modality systems are called positional because the type-checker "remembers" in what modality the current position is.

3.27.3 Pure modality systems

As opposed to the previous systems, in pure modality systems, a definition of the form

```
module M Γ where
  @μ f : A
```

is actually equivalent to a top-level definition of type $\mu \setminus \Gamma \to$ `A`. This is why the following

```
module M (@unused A : Set) where
  @unused B : Set
  B = A
```

gives a top level definition `M.B : @mixed Set → Set`.

A top-level definition `@`μ `f : A` is checked by always first left-dividing the context by the modality μ.

Definitions can then only be used if all the implicitely applied arguments coming from the context telescope are actually available at the proper modalities. The following doesn't type-check

```
module M (@++ A : Set) where
  @unused B : Set
  B = A → ⊤

  @++ C : Set
  C = B
```

because at the point of use of `B`, it implicitely tries to apply it to the `@++ A` present in the context, but since `B` has top-level type `@mixed Set → Set`, this doesn't work.

3.28 Module System

3.28.1 Basics

First let us introduce some terminology. A **definition** is a syntactic construction defining an entity such as a function or a datatype. A name is a string used to identify definitions. The same definition can have many names and at different points in the program it will have different names. It may also be the case that two definitions have the same name. In this case there will be an error if the name is used.

The main purpose of the module system is to structure the way names are used in a program. This is done by organising the program in an hierarchical structure of modules where each module contains a number of definitions and submodules. For instance,

```
module Main where

  module B where
    f : Nat → Nat
    f n = suc n

  g : Nat → Nat → Nat
  g n m = m
```

Note that we use indentation to indicate which definitions are part of a module. In the example `f` is in the module `Main.B` and `g` is in `Main`. How to refer to a particular definition is determined by where it is located in the module hierarchy. Definitions from an enclosing module are referred to by their given names as seen in the type of f above. To access a definition from outside its defining module a qualified name has to be used.

```
module Main₂ where

  module B where
    f : Nat → Nat
    f n = suc n

  ff : Nat → Nat
  ff x = B.f (B.f x)
```

To be able to use the short names for definitions in a module the module has to be opened.

```
module Main₃ where

  module B where
    f : Nat → Nat
    f n = suc n
```

(continues on next page)

(continued from previous page)

```
  open B

  ff : Nat → Nat
  ff x = f (f x)
```

If `A.qname` refers to a definition `d`, then after `open A`, `qname` will also refer to `d`. Note that `qname` can itself be a qualified name. Opening a module only introduces new names for a definition, it never removes the old names. The policy is to allow the introduction of ambiguous names, but give an error if an ambiguous name is used.

Modules can also be opened within a local scope by putting the `open B` within a `where` clause:

```
ff₁ : Nat → Nat
ff₁ x = f (f x) where open B
```

3.28.2 Private definitions

To make a definition inaccessible outside its defining module it can be declared `private`. A private definition is treated as a normal definition inside the module that defines it, but outside the module the definition has no name. In a dependently type setting there are some problems with private definitions—since the type checker performs computations, private names might show up in goals and error messages. Consider the following (contrived) example

```
module Main₄ where
  module A where

    private
      IsZero' : Nat → Set
      IsZero' zero    = ⊤
      IsZero' (suc n) = ⊥

    IsZero : Nat → Set
    IsZero n = IsZero' n

  open A
  prf : (n : Nat) → IsZero n
  prf n = ?
```

The type of the goal `?0` is `IsZero n` which normalises to `IsZero' n`. The question is how to display this normal form to the user. At the point of `?0` there is no name for `IsZero'`. One option could be try to fold the term and print `IsZero n`. This is a very hard problem in general, so rather than trying to do this we make it clear to the user that `IsZero'` is something that is not in scope and print the goal as `;Main₄.A.IsZero' n`. The leading semicolon indicates that the entity is not in scope. The same technique is used for definitions that only have ambiguous names.

In effect using private definitions means that, from the user's perspective, we do not have subject reduction. This is just an illusion, however—the type checker has full access to all definitions.

3.28.3 Name modifiers

An alternative to making definitions private is to exert finer control over what names are introduced when opening a module. This is done by qualifying an `open` (or `open import` or `module X (args : Args) = ...`) statement with one or more of the modifiers `using`, `hiding`, or `renaming`.

- `using` is followed by a list of identifiers, separated by semicolons, and has the effect of introducing *only* those identifiers and the ones named in the `renaming` clause,

- `hiding` is equally followed by a list of identifiers, separated by semicolons, and has the effect of introducing *all* identifiers but the ones named in the `hiding` clause,
- `renaming` is followed by a list of `<identifier> to <identifier>`, separated by semicolons, and has the effect of introducing the mentioned identifiers by their new names. An omitted `renaming` modifier is equivalent to an empty renaming.

For example, the effect of

```
open A using (xs) renaming (ys to zs)
```

is to introduce the names `xs` and `zs` where `xs` refers to the same definition as `A.xs` and `zs` refers to `A.ys`. We do not permit `xs`, `ys` and `zs` to overlap.

Explicitly hiding `x` in a `hiding` clause and also using `x` in a `using` clause or renaming `x to y` in a `renaming` clause is an error. A `renaming` clause can be combined with either a `using` or a `hiding` clause. A `using` and a `hiding` clause can be combined, but the `using` clause takes precedence, hiding everything not mentioned, so except for a special situation with modules, there is nothing that the `hiding` clause can additionally hide.

For submodules of the module being opened, we need to distinguish three situations:

- If `M` is only a module (and not an object), then use `module M` to refer to it, and `module M to N` to rename it. Mentioning just `M` will be ignored with a warning. For instance,

  ```
  open A using (module M)
  ```

- If `M` is only an object (and not a module), then use `M` to refer to it, and `M to N` to renaming. Mentioning `module M` will be ignored with a warning.
- If `M` is both an object and a module (which happens automatically if `M` was introduced with a `data` or `record` definition), then `M` affects *both* the object *and* the module, *unless* `module M` is mentioned separately. In order to introduce only the module, you can write `using (module B)`. In order to introduce only the object, you can write `using (B) hiding (module B)`. In order to introduce all but the module, you can write `hiding (module B)`. It does not seem possible to introduce all but the object: if you write `hiding (B) using (module B)`, then the `using` clause takes precedence and only `module B` is introduced.

Since 2.6.1: The fixity of an operator can be set or changed in a `renaming` directive:

```
module ExampleRenamingFixity where

  module ArithFoo where
    postulate
      A : Set
      _&_ _^_ : A → A → A
    infixr 10 _&_

  open ArithFoo renaming (_&_ to infixl 8 _+_; _^_ to infixl 10 _^_)
```

Here, we change the fixity of `_&_` while renaming it to `_+_`, and assign a new fixity to `_^_` which has the default fixity in module `ArithFoo`.

3.28.4 Re-exporting names

A useful feature is the ability to re-export names from another module. For instance, one may want to create a module to collect the definitions from several other modules. This is achieved by qualifying the open statement with the public keyword:

```
module Example where

  module Nat₁ where

    data Nat₁ : Set where
      zero : Nat₁
      suc  : Nat₁ → Nat₁

  module Bool₁ where

    data Bool₁ : Set where
      true false : Bool₁

  module Prelude where

    open Nat₁  public
    open Bool₁ public

    isZero : Nat₁ → Bool₁
    isZero zero    = true
    isZero (suc _) = false
```

The module `Prelude` above exports the names `Nat`, `zero`, `Bool`, etc., in addition to `isZero`.

3.28.5 Parameterised modules

So far, the module system features discussed have dealt solely with scope manipulation. We now turn our attention to some more advanced features.

When declaring a module you can give a *telescope* of module parameters which are abstracted from all the definitions in the module. This allows us to temporarily work in a given signature.

For instance, when defining functions for sorting lists it is convenient to assume a set of list elements `A` and an ordering over `A`. Thus, a simple implementation of a sorting function looks like this:

```
module Sort (A : Set)(_≤_ : A → A → Bool) where
  insert : A → List A → List A
  insert x [] = x ∷ []
  insert x (y ∷ ys) with x ≤ y
  insert x (y ∷ ys)    | true  = x ∷ y ∷ ys
  insert x (y ∷ ys)    | false = y ∷ insert x ys

  sort : List A → List A
  sort []       = []
  sort (x ∷ xs) = insert x (sort xs)
```

As mentioned parametrising a module has the effect of abstracting the parameters over the definitions in the module, so outside the Sort module we have

```
Sort.insert : (A : Set)(_≤_ : A → A → Bool) →
              A → List A → List A
Sort.sort   : (A : Set)(_≤_ : A → A → Bool) →
              List A → List A
```

For function definitions, explicit module parameter become explicit arguments to the abstracted function, and implicit

parameters become implicit arguments. For constructors, however, the parameters are always implicit arguments. This is a consequence of the fact that module parameters are turned into datatype parameters, and the datatype parameters are implicit arguments to the constructors.

Module application

Parameterized modules can be instantiated via the module application statement. Continuing our example,

```
module SortNat = Sort Nat leqNat
```

This will define a new module SortNat as follows:

```
module SortNat where
  insert : Nat → List Nat → List Nat
  insert = Sort.insert Nat leqNat

  sort : List Nat → List Nat
  sort = Sort.sort Nat leqNat
```

The new module can also be parameterised, and you can use name modifiers to control what definitions from the original module are applied and what names they have in the new module. The general form of a module application is:

```
module M1 Δ = M2 terms modifiers
```

A common pattern is to apply a module to its arguments and then open the resulting module. To simplify this we introduce the short-hand

```
open module M1 Δ = M2 terms [public] modifiers
```

for:

```
module M1 Δ = M2 terms modifiers
open M1 [public]
```

No infix module application

While module names follow the same syntactic rules as ordinary names, they cannot be used in infix form (and neither in pre-, post- or mixfix form). Continuing the above example, even if you defined:

```
module _Sort_ (A : Set)(_≤_ : A → A → Bool) where
```

you could not instantiate it using infix notation:

```
module SortNat = Nat Sort leqNat
```

Anonymous modules

An anonymous module is a module that has the name _ (underscore). Anonymous modules are especially useful when many definitions share the same arguments. For example:

```
module _ (A : Set) where
  f : A → A
  -- ...
```

(continues on next page)

(continued from previous page)

```
  g : A → A → A
  -- ...
```

Anonymous modules are automatically opened immediately after their definition, and cannot be applied.

3.28.6 Splitting a program over multiple files

When building large programs it is crucial to be able to split the program over multiple files and to not have to type check and compile all the files for every change. The module system offers a structured way to do this. We define a program to be a collection of modules, each module being defined in a separate file. To gain access to a module defined in a different file you can import the module:

```
import M
```

In order to implement this we must be able to find the file in which a module is defined. To do this we require that the top-level module `A.B.C` is defined in the file `C.agda` in the directory `A/B/`. One could imagine instead to give a file name to the import statement, but this would mean cluttering the program with details about the file system which is not very nice.

When importing a module `M`, the module and its contents are brought into scope as if the module had been defined in the current file. In order to get access to the unqualified names of the module contents it has to be opened. Similarly to module application we introduce the short-hand

```
open import M
```

for

```
import M
open M
```

Sometimes the name of an imported module clashes with a local module. In this case it is possible to import the module under a different name.

```
import M as M'
```

It is also possible to attach modifiers to import statements, limiting or changing what names are visible from inside the module. Note that modifiers attached to `open import` statements apply to the `open` statement and not the `import` statement.

3.28.7 Datatype modules and record modules

When you define a datatype it also defines a module so constructors can now be referred to qualified by their data type. For instance, given:

```
module DatatypeModules where

  data Nat₂ : Set where
    zero : Nat₂
    suc  : Nat₂ → Nat₂

  data Fin : Nat₂ → Set where
    zero : ∀ {n} → Fin (suc n)
    suc  : ∀ {n} → Fin n → Fin (suc n)
```

you can refer to the constructors unambiguously as `Nat₂.zero`, `Nat₂.suc`, `Fin.zero`, and `Fin.suc` (`Nat₂` and `Fin` are modules containing the respective constructors). Example:

```
inj : (n m : Nat₂) → Nat₂.suc n ≡ suc m → n ≡ m
inj .m m refl = refl
```

Previously you had to write something like

```
inj₁ : (n m : Nat₂) → _≡_ {A = Nat₂} (suc n) (suc m) → n ≡ m
inj₁ .m m refl = refl
```

to make the type checker able to figure out that you wanted the natural number suc in this case.

Also record declarations define a corresponding module, see *Record modules*.

3.28.8 References

The initial design of Agda 2 module system is covered in Ulf Norell thesis.

Survey on the module system implementation and its current semantics and performance problems was done recently (2023) by Ivar de Bruin.

3.29 Mutual Recursion

Agda offers multiple ways to write mutually-defined data types, record types and functions.

- *Old-style mutual blocks*
- *Forward declaration*
- *Interleaved mutual blocks*

The last two are more expressive than the first one as they allow the interleaving of declarations and definitions thus making it possible for some types to refere to the constructors of a mutually-defined datatype.

3.29.1 Interleaved mutual blocks

Mutual recursive functions can be written by placing them inside an `interleaved mutual` block. The type signature of each function must come before its defining clauses and its usage sites on the right-hand side of other functions. The clauses for different functions can be interleaved e.g. for pedagogical purposes:

```
interleaved mutual

  -- Declarations:
  even : Nat → Bool
  odd  : Nat → Bool

  -- zero is even, not odd
  even zero = true
  odd  zero = false

  -- suc case: switch evenness on the predecessor
  even (suc n) = odd n
  odd  (suc n) = even n
```

You can mix arbitrary declarations, such as modules and postulates, with mutually recursive definitions. For data types and records the following syntax is used to separate the declaration from the introduction of constructors in one or many `data ... where` blocks:

```
interleaved mutual

  -- Declaration of a product record, a universe of codes, and a decoding function
  record _×_ (A B : Set) : Set
  data U : Set
  El : U → Set

  -- We have a code for the type of natural numbers in our universe
  data U where `Nat : U
  El `Nat = Nat

  -- Btw we know how to pair values in a record
  record _×_ A B where
    inductive; no-eta-equality; pattern
    constructor _,_
    field fst : A; snd : B

  -- And we have a code for pairs in our universe
  data _ where
    _`×_ : (A B : U) → U
  El (A `× B) = El A × El B

-- we can now build types of nested pairs of natural numbers
ty-example : U
ty-example = `Nat `× ((`Nat `× `Nat) `× `Nat)

-- and their values
val-example : El ty-example
val-example = 0 , ((1 , 2) , 3)
```

You can mix constructors for different data types in a `data _ where` block (underscore instead of name).

The `interleaved mutual` blocks get desugared into the *Forward declaration* blocks described below by:

- leaving the signatures where they are,
- grouping the clauses for a function together with the first of them, and
- grouping the constructors for a datatype together with the first of them.

3.29.2 Forward declaration

Mutual recursive functions can be written by placing the type signatures of all mutually recursive function before their definitions. The span of the mutual block will be automatically inferred by Agda:

```
f : A
g : B[f]
f = a[f, g]
g = b[f, g].
```

You can mix arbitrary declarations, such as modules and postulates, with mutually recursive definitions. For data types and records the following syntax is used to separate the declaration from the definition:

```
-- Declaration.
data Vec (A : Set) : Nat → Set  -- Note the absence of 'where'.

-- Definition.
data Vec A where                -- Note the absence of a type signature.
  []   : Vec A zero
  _::_ : {n : Nat} → A → Vec A n → Vec A (suc n)

-- Declaration.
record Sigma (A : Set) (B : A → Set) : Set

-- Definition.
record Sigma A B where
  constructor _,_
  field fst : A
        snd : B fst
```

The parameter lists in the second part of a data or record declaration behave like variables left-hand sides (although infix syntax is not supported). That is, they should have no type signatures, but implicit parameters can be omitted or bound by name.

Such a separation of declaration and definition is for instance needed when defining a set of codes for types and their interpretation as actual types (a so-called *universe*):

```
-- Declarations.
data TypeCode : Set
Interpretation : TypeCode → Set

-- Definitions.
data TypeCode where
  nat : TypeCode
  pi  : (a : TypeCode) (b : Interpretation a → TypeCode) → TypeCode

Interpretation nat      = Nat
Interpretation (pi a b) = (x : Interpretation a) → Interpretation (b x)
```

> **Note**
>
> In contrast to *Interleaved mutual blocks*, in forward-declaration style we can only have one `data ... where` block per data type.

When making separated declarations/definitions private or abstract you should attach the `private` keyword to the declaration and the `abstract` keyword to the definition. For instance, a private, abstract function can be defined as

```
private
  f : A
abstract
  f = e
```

3.29.3 Old-style `mutual` blocks

Mutual recursive functions can be written by placing the type signatures of all mutually recursive function before their definitions:

```
mutual
  f : A
  f = a[f, g]

  g : B[f]
  g = b[f, g]
```

Using the `mutual` keyword, the *universe* example from above is expressed as follows:

```
mutual
  data TypeCode : Set where
    nat : TypeCode
    pi  : (a : TypeCode) (b : Interpretation a → TypeCode) → TypeCode

  Interpretation : TypeCode → Set
  Interpretation nat      = Nat
  Interpretation (pi a b) = (x : Interpretation a) → Interpretation (b x)
```

This alternative syntax desugars into the new syntax by sorting the content of the mutual block into a *declaration* and a *definition* part and placing the declarations before the definitions.

Declarations comprise:

- Type signatures of functions, `data` and `record` declarations, `unquoteDecl`. (*Function* includes here `postulate` and `primitive` etc.)
- Module statements, such as `module` aliases, `import` and `open` statements.
- Pragmas that only need the name, but not the definition of the thing they affect (e.g. `INJECTIVE`).

Definitions comprise:

- Function clauses, `data` constructors and `record` definitions, `unquoteDef`.
- `pattern` synonym definitions.
- Pragmas that need the definition, e.g. `INLINE`, `REWRITE`, etc.
- Pragmas that are not needed for type checking, like compiler pragmas.

Module definitions with `module ... where` are not supported in old-style `mutual` blocks.

3.30 Opaque definitions

Opaque definitions are a mechanism for controlling unfolding of Agda definitions, to help with both goal readability and performance. Like *abstract definitions*, opaque definitions will not unfold in general, but *unlike* `abstract` definitions, opacity can be selectively controlled at use-sites.

Our implementation of unfolding control is based on the theory introduced by Gratzer et. al. in *Controlling unfolding in type theory*, but handled entirely at the elaborator level, without a dependency on our (cubical) extension types.

3.30.1 Overview

- Function definitions, whether user-written or generated by reflection, can be marked `opaque`. Outside of `opaque` blocks, these behave like postulates.
- Opaque blocks, even in unrelated modules, can have `unfolding` clauses, which allow the user to list their choice of names that should be locally treated as transparent.
- Opaque definitions do not reduce in type signatures, even inside opaque blocks where they would otherwise be unfolded.

3.30.2 Unfolding opaque definitions

Consider an implementation of the integers as an abstract setoid: The underlying representation is given by pairs of natural numbers, representing a difference, but day-to-day, we would like to treat $\mathbb{Z}$ as its own type.

Our core module might define these operations:

```
module Integer where
  opaque
    ℤ : Set
    ℤ = Nat × Nat

    _≡ℤ_ : (x y : ℤ) → Set
    (p , n) ≡ℤ (p' , n') = (p + n') ≡ (p' + n)

    infix 10 _≡ℤ_

    0ℤ : ℤ
    0ℤ = 0 , 0

    1ℤ : ℤ
    1ℤ = 1 , 0

    _+ℤ_ : (x y : ℤ) → ℤ
    (p , n) +ℤ (p' , n') = (p + p') , (n + n')

    _*ℤ_ : (x y : ℤ) → ℤ
    (a , b) *ℤ (c , d) = ((a * c) + (b * d)) , ((a * d) + (b * c))

    infixl 20 _+ℤ_
    infixl 30 _*ℤ_
```

(continues on next page)

(continued from previous page)

```
  -ℤ_ : ℤ → ℤ
  -ℤ (p , n) = (n , p)
```

We'd now like to prove that the integers form a ring, under the _≡ℤ_ notion of equality. Some of the equations on natural numbers involved are pretty nasty, though, so this would be very hard to do without a solver for semiring equations. However, such a solver would also depend on *reflection machinery*, bloating the dependency tree of the `Integer` module for people who do not care about it provably forming a ring.

Fortunately, since ℤ is *opaque* rather than *abstract*, a different module, say `Integer-ring`, can provide its own proofs, in an `opaque` block that unfolds the definition of ℤ:

```
module Integer-ring where
  open Integer

  opaque
    unfolding ℤ

    distlℤ : ∀ x y z → x *ℤ (y +ℤ z) ≡ℤ x *ℤ y +ℤ x *ℤ z
    distlℤ (a , b) (c , d) (e , f) = use-nat-solver where postulate
      use-nat-solver
        : a * (c + e) + b * (d + f) + (a * d + b * c + (a * f + b * e))
        ≡ a * c + b * d + (a * e + b * f) + (a * (d + f) + b * (c + e))
```

Since the definition of `distlℤ` is in an `opaque` block with an `unfolding ℤ` clause, it sees through the opacity of ℤ, and of all names unfolded by ℤ's opaque block (see below).

3.30.3 What actually unfolds?

When an `opaque` block is checked, Agda will compute ahead-of-time the set of names it is allowed to unfold. This set is per-*block*, not per-*definition*. An `unfolding` clause, if it mentions opaque names, will cause the unfolding sets associated with those names to be added to the current block.

The following illustrates the behaviour of these rules:

- Unfolding any name in an opaque block will cause any of the *other* names in that block to be unfolded as well. Example:

  ```
  module _ where private
    opaque
      x : Nat
      y : Nat

      x = 3
      y = 4

    opaque
      unfolding x

      _ : y ≡ 4
      _ = refl
  ```

  Here, even though only `x` was asked for, `y` is also available for unfolding.

- Since the unfolding sets brought in by clauses are associated with the block, unfolding is transitive:

```
module _ where private
  opaque
    x : Nat
    x = 3

  opaque
    unfolding x
    y : Nat
    y = 4 + x

  opaque
    unfolding y
    _ : y ≡ 7
    _ = refl
```

- Opaque blocks which are lexically nested can also unfold the names of their *parent* blocks, even if the name is not in scope when the child block is defined:

```
module _ where private
  opaque
    x : Nat
    x = 3

    opaque
      y : Nat
      y = 4

      _ : x ≡ 3
      _ = refl

    z : Nat
    z = 5

  opaque
    unfolding y
    _ : z ≡ 5
    _ = refl
```

  This is because the `x` and `z` are direct children of the same `opaque` block: the `opaque` block that defines `y` does not "split" its parent block.

Multiple unfolding clauses are supported, as well as unfolding more than one name per clause. The syntax for the latter is simply a space-separated list of names, which must refer to unambiguous functions:

```
module _ where private
  opaque
    x : Nat
    x = 3

  opaque
    y : Nat
    y = 4

  opaque
```

(continues on next page)

(continued from previous page)

```
    z : Nat
    z = 5

  opaque
    unfolding x y
    unfolding z

    _ : x + y + z ≡ 12
    _ = refl
```

Finally, `unfolding` clauses do not introduce new layout context, so that the following is legal: note that `y` appears to the left of `x`, but is still attached to the same `unfolding` clause. This allows the user their preference for how to lay out their unfolding sets:

```
opaque
  unfolding x
    y
  unfolding z

  _ : x + y + z ≡ 12
  _ = refl
```

Having an `unfolding` clause appear after other definitions, or outside of `opaque` blocks, is a syntax error.

Note that unlike `abstract` blocks, which are treated on a per-module basis, `opaque` blocks will only unfold names according to the rules above:

```
module _ where private
  opaque
    x : Nat
    x = 3

  -- opaque
    -- _ : x ≡ 3
    -- _ = refl
    -- Fails with: x != 3 of type Nat
```

3.30.4 Unfolding in types

Note that unfolding clauses do not apply to the *type signatures* inside an `opaque` block. Much like for `abstract` blocks, this prevents leakage of implementation details, but it is also necessary to ensure that the types of names defined by the opaque block remain valid outside the opaque block. Consider:

```
opaque
  S : Set₁
  S = Set

  foo´ : S
  foo´ = Nat

opaque
  unfolding foo´
```

(continues on next page)

(continued from previous page)

```
  -- bar´ : foo´
  -- bar´ = 123
  -- Error: S should be a sort, but it isn't
```

If the definition of `bar´` were allowed, we would have `bar´ : foo´` in the context. Outside of the relevant opaque blocks, `foo´` is not a type, for `foo´ : S`, and `S` is not a sort. In cases like this, using an auxiliary definition whose type *is* a sort is required:

```
-- Lift foo´ to a definition:
ty´ : Set
ty´ = foo´

bar´ : ty´
bar´ = 123
```

Since `ty´ : Set` is manifestly a well-formed type, even outside of this opaque block, there is no problem in adding `bar´ : ty´` to the context.

3.30.5 Bibliography

Daniel Gratzer, Jonathan Sterling, Carlo Angiuli, Thierry Coquand, and Lars Birkedal; "Controlling unfolding in type theory".

3.31 Pattern Synonyms

A **pattern synonym** is a declaration that can be used on the left hand side (when pattern matching) as well as the right hand side (in expressions). For example:

```
data Nat : Set where
  zero : Nat
  suc  : Nat → Nat

pattern z    = zero
pattern ss x = suc (suc x)

f : Nat → Nat
f z       = z
f (suc z) = ss z
f (ss n)  = n
```

Pattern synonyms are implemented by substitution on the abstract syntax, so definitions are scope-checked but *not type-checked*. They are particularly useful for universe constructions.

3.31.1 Overloading

Pattern synonyms can be overloaded as long as all candidates have the same *shape*. Two pattern synonym definitions have the same shape if they are equal up to variable and constructor names. Shapes are checked at resolution time and after expansion of nested pattern synonyms.

For example:

```
data List (A : Set) : Set where
  lnil  : List A
```

(continues on next page)

(continued from previous page)

```
  lcons : A → List A → List A

data Vec (A : Set) : Nat → Set where
  vnil  : Vec A zero
  vcons : ∀ {n} → A → Vec A n → Vec A (suc n)

pattern [] = lnil
pattern [] = vnil

pattern _::_ x xs = lcons x xs
pattern _::_ y ys = vcons y ys

lmap : ∀ {A B} → (A → B) → List A → List B
lmap f []       = []
lmap f (x :: xs) = f x :: lmap f xs

vmap : ∀ {A B n} → (A → B) → Vec A n → Vec B n
vmap f []       = []
vmap f (x :: xs) = f x :: vmap f xs
```

Flipping the arguments in the synonym for `vcons`, changing it to `pattern _::_ ys y = vcons y ys`, results in the following error when trying to use the synonym:

```
Cannot resolve overloaded pattern synonym _::_, since candidates
have different shapes:
  pattern _::_ x xs = lcons x xs
    at pattern-synonyms.lagda.rst:51,13-16
  pattern _::_ ys y = vcons y ys
    at pattern-synonyms.lagda.rst:52,13-16
(hint: overloaded pattern synonyms must be equal up to variable and
constructor names)
when checking that the clause lmap f (x :: xs) = f x :: lmap f xs has
type {A B : Set} → (A → B) → List A → List B
```

3.31.2 Refolding

For each pattern `pattern lhs = rhs`, Agda declares a `DISPLAY` pragma refolding `rhs` to `lhs` (see *The DISPLAY pragma* for more details).

3.32 Polarity Annotations

Agda supports explicitly annotating functions arguments and datatype parameters with their polarities, using a *modality system*, which the positivity checker can then use to infer positivity. This experimental feature has to be enabled by the *`--polarity`* option. Here are some sample uses of the different annotations which type-check:

```
strictly-positive : @++ Set → Set
strictly-positive A = Nat → A

positive : @+ Set → Set
positive A = (A → Nat) → A
```

(continues on next page)

(continued from previous page)

```
negative : @- Set → Set
negative A = A → Nat

mixed : @mixed Set → Set
mixed A = A → A

unused : @unused Set → Set
unused A = Nat

ex : @- Set → Set
ex A = negative (positive A)

ex2 : @mixed Set → Set
ex2 A = mixed (strictly-positive A)

ex3 : @++ Set → Set
ex3 A = unused (negative A)
```

The following code doesn't type-check:

```
ex3 : @++ Set → Set
ex3 A = mixed (strictly-positive A)
```

The standard use-case is defining the fix-point of any strictly positive type-former (which is already the criterion for an inductive type to be definable in Agda, see *Strict positivity*, but the polarity modality internalizes this criterion into the type system):

```
data Mu (F : @++ Set → Set) : Set where
  fix : F (Mu F) → Mu F
```

In the example above, because `F` is specified as using its argument strictly positively, the positivity checker allows `F (Mu F)` as an argument to a constructor since it knows the recursive call to `Mu F` is in strictly positive position.

When defining functions that take arguments annotated with `@++`, the typing rules ensure that those arguments may never actually appear to the left of an arrow. They can syntactically appear to the left of an arrow as arguments to functions that don't use their arguments, such as in the correct example below:

```
const : @unused Set → Set
const _ = Nat

typechecks : @++ Set → Set
typechecks A = const A → Nat
```

3.32.1 The polarity modality

Agda implements polarity annotations using a modal system. Here are the different modalities and their meaning:

Notation	Name	Possible use
@++	Strictly positive	Anywhere except in the domain of a pi/function type
@+	Positive	In an even number of nested domains of pi/function types
@-	Negative	In an odd number of nested domains of pi/function types
@mixed	Mixed	Anywhere
@unused	Unused	Nowhere

Strictly positive types are a syntactical condition described for example in section 2.3 of *[1]*. A very similar system (without strict positivity) was described in *[2]*, but didn't use the modality formalism.

When Agda type-checks any definition, it ensures that the variables bound by a lambda-abstraction with annotated types satisfy those restrictions. Note that there is no such restriction when checking the codomain of a pi type, so for example `(@++ A : Set) → (A → A)` is perfectly valid!

Polarity annotations can only appear on domains of function types and data/record type parameters. Pattern matching on annotated arguments is only supported for mixed arguments.

3.32.2 Positivity checking

The Agda positivity checker uses the polarity annotations in the typing information to enhance its analysis and accept types like `Mu` above. This can also help when the positivity checker is unable to automatically infer that information itself. Here is a contrived example that doesn't type-check without annotations:

```
apply-pattern-match : {A B : Set₁} → Nat → (@++ A → B) → @++ A → B
apply-pattern-match zero f = f
apply-pattern-match (suc n) f = f

id : {A : Set₁} → @++ A → A
id x = x

data D : Set where
  node : (u : Nat) → apply-pattern-match u id D → D
```

3.32.3 References

[1] Michael Abbott, Thorsten Altenkirch, Neil Ghani, "Containers: Constructing strictly positive types", In Theoretical Computer Science, Volume 342, Issue 1, 2005, https://doi.org/10.1016/j.tcs.2005.06.002

[2] Andreas Abel, "Polarized Subtyping for Sized Types", In: Mathematical Structures in Computer Science, 2006, https://doi.org/10.1007/11753728_39

[3] Josselin Poiret, Lucas Escot, Joris Ceulemans, Malin Altenmüller, and Andreas Nuyts. 2023. Read the Mode and Stay Positive. In 29th International Conference on Types for Proofs and Programs (TYPES), https://lirias.kuleuven.be/retrieve/720869

3.33 Positivity Checking

> **Note**
>
> This is a stub.

3.33.1 Occurrence analysis

By default Agda analyses how functions use their arguments. For instance, Agda can tell that `D` in the following code is strictly positive, because `Vec` uses its `Set` argument in a strictly positive way:

```
data _×_ (A B : Set) : Set where
  _,_ : A → B → A × B

Vec : Set → Nat → Set
Vec A zero    = ⊤
```

(continues on next page)

(continued from previous page)

```
Vec A (suc n) = A × Vec A n

data D : Set where
  c : ∀ n → Vec D n → D
```

However, this analysis can be slow, especially for big mutual blocks. It can be turned off with the *--no-occurrence-analysis* flag.

The analysis is also used to detect unused function arguments. For instance, Agda by default notices that the last argument of `F` in the following code is unused, and accepts the use of reflexivity:

```
F : Bool → Set → Set
F true  _ = Bool
F false _ = ⊤

_ : {b : Bool} → F b Bool ≡ F b ⊤
_ = refl
```

An alternative to the occurrence analysis is to use *polarities*:

```
data _×_ (@++ A B : Set) : Set where
  _,_ : A → B → A × B

Vec : @++ Set → Nat → Set
Vec A zero    = ⊤
Vec A (suc n) = A × Vec A n

data D : Set where
  c : ∀ n → Vec D n → D

F : Bool → @unused Set → Set
F true  _ = Bool
F false _ = ⊤

_ : {b : Bool} → F b Bool ≡ F b ⊤
_ = refl
```

The `NO_POSITIVITY_CHECK` pragma

The pragma switches off the positivity checker for data/record definitions and mutual blocks. This pragma was added in Agda 2.5.1

The pragma must precede a data/record definition or a mutual block. The pragma cannot be used in *--safe* mode.

Examples:

- Skipping a single data definition:

```
{-# NO_POSITIVITY_CHECK #-}
data D : Set where
  lam : (D → D) → D
```

- Skipping a single record definition:

```
{-# NO_POSITIVITY_CHECK #-}
record U : Set where
  inductive; no-eta-equality
  field ap : U → U
```

- Skipping an old-style mutual block. Somewhere within a mutual block before a data/record definition:

```
mutual
  data D : Set where
    lam : (D → D) → D

  {-# NO_POSITIVITY_CHECK #-}
  record U : Set where
    inductive; no-eta-equality
    field ap : U → U
```

- Skipping an old-style mutual block. Before the `mutual` keyword:

```
{-# NO_POSITIVITY_CHECK #-}
mutual
  data D : Set where
    lam : (D → D) → D

  record U : Set where
    inductive; no-eta-equality
    field ap : U → U
```

- Skipping a new-style mutual block. Anywhere before the declaration or the definition of a data/record in the block:

```
record U : Set
data D   : Set

record U where
  inductive; no-eta-equality
  field ap : U → U

{-# NO_POSITIVITY_CHECK #-}
data D where
  lam : (D → D) → D
```

POLARITY pragmas

Polarity pragmas can be attached to postulates. The polarities express how the postulate's arguments are used. The following polarities are available:

- _: Unused.
- ++: Strictly positive.
- +: Positive.
- -: Negative.
- *: Unknown/mixed.

Polarity pragmas have the form `{-# POLARITY name <zero or more polarities> #-}`, and can be given wherever fixity declarations can be given. The listed polarities apply to the given postulate's arguments (explicit/implicit/instance), from left to right. Polarities currently cannot be given for module parameters. If the postulate takes n arguments (excluding module parameters), then the number of polarities given must be between 0 and n (inclusive).

Polarity pragmas make it possible to use postulated type formers in recursive types in the following way:

```
postulate
  ‖_‖ : Set → Set

{-# POLARITY ‖_‖ ++ #-}

data D : Set where
  c : ‖ D ‖ → D
```

Note that one can use postulates that may seem benign, together with polarity pragmas, to prove that the empty type is inhabited:

```
postulate
  _⇒_    : Set → Set → Set
  lambda : {A B : Set} → (A → B) → A ⇒ B
  apply  : {A B : Set} → A ⇒ B → A → B

{-# POLARITY _⇒_ ++ #-}

data ⊥ : Set where

data D : Set where
  c : D ⇒ ⊥ → D

not-inhabited : D → ⊥
not-inhabited (c f) = apply f (c f)

d : D
d = c (lambda not-inhabited)

bad : ⊥
bad = not-inhabited d
```

Polarity pragmas are not allowed in safe mode.

3.34 Postulates

A postulate is a declaration of an element of some type without an accompanying definition. With postulates we can introduce elements in a type without actually giving the definition of the element itself.

The general form of a postulate declaration is as follows:

```
postulate
    c11 ... c1i : <Type>
    ...
    cn1 ... cnj : <Type>
```

Postulate blocks can include `instance` and `private` declarations.

Example for a basic postulate block:

```
postulate
  A B    : Set
  a      : A
  b      : B
  _=AB=_ : A → B → Set
  a==b   : a =AB= b
```

Introducing postulates is in general not recommended. Once postulates are introduced the consistency of the whole development is at risk, because there is nothing that prevents us from introducing an element in the empty set.

```
data False : Set where

postulate bottom : False
```

Postulates are forbidden in *Safe Agda* (option `--safe`) to prevent accidential inconsistencies.

A preferable way to work with assumptions is to define a module parametrised by the elements we need:

```
module Absurd (bt : False) where

  -- ...

module M (A B : Set) (a : A) (b : B)
         (_=AB=_ : A → B → Set) (a==b : a =AB= b) where

  -- ...
```

3.34.1 Postulated built-ins

Some *built-ins* such as *Float* and *Char* are introduced as a postulate and then given a meaning by the corresponding `{-# BUILTIN ... #-}` pragma.

3.34.2 Local uses of `postulate`

Postulates are declarations and can appear in positions where arbitrary declarations are allowed, e.g., in `where` blocks:

```
module PostulateInWhere where

  my-theorem : (A : Set) → A
  my-theorem A = I-prove-this-later
    where
      postulate I-prove-this-later : _
```

3.35 Pragmas

Pragmas are special declarations that pass extra information to Agda about how regular declarations are to be interpreted. They are written similar to block comments so that users may easily skip them in a first reading of an Agda document. The general format is:

```
{-# <PRAGMA_NAME> <arguments> #-}
```

3.35.1 Index of pragmas

See also *Command-line and pragma options*.

The `DISPLAY` pragma

Users can declare a display form via the `DISPLAY` pragma:

```
{-# DISPLAY f e1 .. en = e #-}
```

This causes `f e1 .. en` to be printed in the same way as `e`, where `ei` can bind variables used in `e`. The expressions `ei` and `e` are scope checked, but not type checked.

For example this can be used to print overloaded (instance) functions with the overloaded name:

```
instance
  NumNat : Num Nat
  NumNat = record { ..; _+_ = natPlus }

{-# DISPLAY natPlus a b = a + b #-}
```

Limitations:

- Left-hand sides of the display form are restricted to variables, constructors, defined functions or types, and literals. In particular, lambdas are not allowed in left-hand sides.
- Since display forms are not type checked, implicit argument insertion may not work properly if the type of *f* computes to an implicit function space after pattern matching.
- An ill-typed display form can make Agda crash with an internal error when Agda tries to use it (issue *#6476 <https://github.com/agda/agda/issues/6476>*).

The `INJECTIVE` pragma

Injective pragmas can be used to mark a definition as injective for the pattern matching unifier. This can be used as a version of `--injective-type-constructors` that only applies to specific datatypes.

Example:

```
open import Agda.Builtin.Equality
open import Agda.Builtin.Nat

data Fin : Nat → Set where
  zero : {n : Nat} → Fin (suc n)
  suc  : {n : Nat} → Fin n → Fin (suc n)

{-# INJECTIVE Fin #-}

Fin-injective : {m n : Nat} → Fin m ≡ Fin n → m ≡ n
Fin-injective refl = refl
```

Aside from datatypes, this pragma can also be used to mark other definitions as being injective (for example postulates).

At the moment it only gives you propositional injectivity, so you can pattern match on a proof of *Fin x* ≡ *Fin y* in example above, but does not give you definitional injectivity, so the constraint solver does not know how to solve the constraint *Fin x* = *Fin _*. Relevant issue: https://github.com/agda/agda/issues/4106#issuecomment-534904561

The `INJECTIVE_FOR_INFERENCE` pragma

Treats functions as injective for type inference. This behaves like a local version of `--lossy-unification` and has the same potential issues. Since Agda can not always infer whether a function is injective it can be used to get stronger unification for those functions.

The option `--no-require-unique-meta-solutions` needs to be active in the file where the function is used, but not necessarily in the file it is defined. When solving a constraint involving the function in a file where `--require-unique-meta-solutions` is in effect, the pragma is ignored.

Example:

```
open import Agda.Builtin.Equality
open import Agda.Builtin.List

module _ {A : Set} where
  _++_ : List A → List A → List A
  []       ++ ys = ys
  (x ∷ xs) ++ ys = x ∷ (xs ++ ys)

  reverse : List A → List A
  reverse []      = []
  reverse (x ∷ l) = reverse l ++ (x ∷ [])
```

(continues on next page)

(continued from previous page)

```
{-# INJECTIVE_FOR_INFERENCE reverse #-}

reverse-≡ : {l l' : List A} → reverse l ≡ reverse l' → reverse l ≡ reverse l'
reverse-≡ h = h

[]≡[] : {l l' : List A} → [] ≡ []
[]≡[] = reverse-≡ (refl {x = reverse []})
```

The INLINE and NOINLINE pragmas

A function definition marked with an INLINE pragma is inlined during compilation. If it is a simple function definition that does no pattern matching, it is also inlined in function bodies at type-checking time.

When the *--auto-inline command-line option* is enabled, function definitions are automatically marked INLINE if they satisfy the following criteria:

- No pattern matching.
- Uses each argument at most once.
- Does not use all its arguments.

Automatic inlining can be prevented using the NOINLINE pragma.

Example:

```
-- Would be auto-inlined since it doesn't use the type arguments.
_∘_ : {A B C : Set} → (B → C) → (A → B) → A → C
(f ∘ g) x = f (g x)

{-# NOINLINE _∘_ #-} -- prevents auto-inlining

-- Would not be auto-inlined since it's using all its arguments
_o_ : (Set → Set) → (Set → Set) → Set → Set
(F o G) X = F (G X)

{-# INLINE _o_ #-} -- force inlining
```

Inlining constructor right-hand sides

Added in version 2.6.4.

Constructors can also be marked INLINE (for types supporting copattern matching):

```
record Stream (A : Set) : Set where
  coinductive; constructor _::_
  field head : A
        tail : Stream A
open Stream
{-# INLINE _::_ #-}
```

Functions definitions using these constructors will be translated to use copattern matching instead, e.g.:

```
nats : Nat → Stream Nat
nats n = n :: nats (1 + n)
```

is translated to:

```
nats' : Nat → Stream Nat
nats' n .head = n
nats' n .tail = nats (n + 1)
```

which passes termination-checking.

This translation only works for fully-applied constructors at the root of a function definition's right-hand side.

If *--exact-split* is on, the inlining will trigger a *InlineNoExactSplit* warning for `nats`.

The `NON_COVERING` pragma

Added in version 2.6.1.

The `NON_COVERING` pragma can be placed before a function (or a block of mutually defined functions) which the user knows to be partial. To be used as a version of *--allow-incomplete-matches* that only applies to specific functions.

The `NOT_PROJECTION_LIKE` pragma

Added in version 2.6.3.

The `NOT_PROJECTION_LIKE` pragma disables projection-likeness analysis for a particular function, which must be defined before it can be affected by the pragma. To be used as a version of *--no-projection-like* that only applies to specific functions.

For example, suppose you have a function which projects a field from an instance argument, and instance selection depends on a visible argument. If an application of this function is generated by metaprogramming, and inserted in the source code by `elaborate-and-give` (`C-c C-m` in Emacs), the visible argument would instead be printed as `_`, because it was erased!

Example:

```
open import Agda.Builtin.Bool

record P (n : Nat) : Set where
  field the-bool : Bool
open P

-- Agda would normally mark this projection-like, so it would have its
-- (n : Nat) argument erased when printing, including by e.g.
-- elaborate-and-give
get-bool-from-p : (n : Nat) PDF TODO has-p : P n PDF TODO → Bool
get-bool-from-p _ PDF TODO p PDF TODO = p .the-bool
{-# NOT_PROJECTION_LIKE get-bool-from-p #-}

-- With the pragma, it gets treated as a regular function.
```

The `OPTIONS` pragma

Some options can be given at the top of .agda files in the form

`{-# OPTIONS --{opt$_1$} --{opt$_2$} ... #-}`

The possible options are listed in *Command-line and pragma options*.

The WARNING_ON_ pragmas

A library author can use a WARNING_ON_USAGE pragma to attach to a defined name a warning to be raised whenever this name is used (since Agda 2.5.4).

Similarly they can use a WARNING_ON_IMPORT pragma to attach to a module a warning to be raised whenever this module is imported (since Agda 2.6.1).

This would typically be used to declare a name or a module 'DEPRECATED' and advise the end-user to port their code before the feature is dropped.

Users can turn these warnings off by using the --warn=noUserWarning option. For more information about the warning machinery, see *Warnings*.

Example:

```
-- The new name for the identity
id : {A : Set} → A → A
id x = x

-- The deprecated name
λx→x = id

-- The warning
{-# WARNING_ON_USAGE λx→x "DEPRECATED: Use `id` instead of `λx→x`" #-}
{-# WARNING_ON_IMPORT "DEPRECATED: Use module `Function.Identity` rather than `Identity`"
↪#-}
```

3.36 Prop

Prop is Agda's built-in sort of *definitionally proof-irrelevant propositions*. It is similar to the sort Set, but all elements of a type in Prop are considered to be (definitionally) equal.

The implementation of Prop is based on the POPL 2019 paper Definitional Proof-Irrelevance without K by Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau.

This is an experimental extension of Agda guarded by option *--prop*.

3.36.1 Usage

Just as for Set, we can define new types in Prop's as data or record types:

```
data ⊥ : Prop where

record ⊤ : Prop where
  constructor tt
```

When defining a function from a data type in Prop to a type in Set, pattern matching is restricted to the *absurd pattern* ():

```
absurd : (A : Set) → ⊥ → A
absurd A ()
```

Unlike for Set, all elements of a type in Prop are definitionally equal. This implies all applications of absurd are the same:

```
only-one-absurdity : {A : Set} → (p q : ⊥) → absurd A p ≡ absurd A q
only-one-absurdity p q = refl
```

Since pattern matching on datatypes in Prop is limited, it is recommended to define types in Prop as recursive functions rather than inductive datatypes. For example, the relation _≤_ on natural numbers can be defined as follows:

```
_≤_ : Nat → Nat → Prop
zero  ≤ y     = ⊤
suc x ≤ zero  = ⊥
suc x ≤ suc y = x ≤ y
```

The induction principle for _≤_ can then be defined by matching on the arguments of type Nat:

```
module _ (P : (m n : Nat) → Set)
  (pzy : (y : Nat) → P zero y)
  (pss : (x y : Nat) → P x y → P (suc x) (suc y)) where

  ≤-ind : (m n : Nat) → m ≤ n → P m n
  ≤-ind zero    y       pf = pzy y
  ≤-ind (suc x) (suc y) pf = pss x y (≤-ind x y pf)
  ≤-ind (suc _) zero    ()
```

Note that while it is also possible to define _≤_ as a datatype in Prop, it is hard to use that version because of the limitations to matching.

When defining a record type in Set, the types of the fields can be both in Set and Prop. For example:

```
record Fin (n : Nat) : Set where
  constructor _[_]
  field
    ⟦_⟧   : Nat
    proof : suc ⟦_⟧ ≤ n
open Fin

Fin-≡ : ∀ {n} (x y : Fin n) → ⟦ x ⟧ ≡ ⟦ y ⟧ → x ≡ y
Fin-≡ x y refl = refl
```

3.36.2 The predicative hierarchy of Prop

Just as for Set, Agda has a predicative hierarchy of sorts $\mathtt{Prop}_0$ (= Prop), $\mathtt{Prop}_1$, $\mathtt{Prop}_2$, ..., $\mathtt{Prop}\omega_0$ (= $\mathtt{Prop}\omega$), $\mathtt{Prop}\omega_1$, $\mathtt{Prop}\omega_2$, ..., where $\mathtt{Prop}_0$: $\mathtt{Set}_1$, $\mathtt{Prop}_1$: $\mathtt{Set}_2$, $\mathtt{Prop}_2$: $\mathtt{Set}_3$, ..., $\mathtt{Prop}\omega_0$: $\mathtt{Set}\omega_1$, $\mathtt{Prop}\omega_1$: $\mathtt{Set}\omega_2$, $\mathtt{Prop}\omega_2$: $\mathtt{Set}\omega_3$, etc. Like Set, Prop also supports universe polymorphism (see *universe levels*), so for each ℓ : Level we have the sort Prop ℓ. For example:

```
True : ∀ {ℓ} → Prop (lsuc ℓ)
True {ℓ} = ∀ (P : Prop ℓ) → P → P
```

Note that ∀ {ℓ} → Prop (lsuc ℓ) (and likewise any ∀ {ℓ} → Prop (t ℓ)) lives in $\mathtt{Set}\omega$, not $\mathtt{Prop}\omega$.

3.36.3 The propositional squash type

When defining a datatype in Prop ℓ, it is allowed to have constructors that take arguments in Set ℓ' for any $\ell' \leq \ell$. For example, this allows us to define the propositional squash type and its eliminator:

```
data Squash {ℓ} (A : Set ℓ) : Prop ℓ where
  squash : A → Squash A

squash-elim : ∀ {ℓ₁ ℓ₂} (A : Set ℓ₁) (P : Prop ℓ₂) → (A → P) → Squash A → P
squash-elim A P f (squash x) = f x
```

This type allows us to simulate Agda's existing irrelevant arguments (see *irrelevance*) by replacing `.A` with `Squash A`.

3.36.4 Limitations

It is possible to define an equality type in Prop as follows:

```
data _≐_ {ℓ} {A : Set ℓ} (x : A) : A → Prop ℓ where
  refl : x ≐ x
```

However, the corresponding eliminator cannot be defined because of the limitations on pattern matching. As a consequence, this equality type is only useful for refuting impossible equations:

```
0≢1 : 0 ≐ 1 → ⊥
0≢1 ()
```

3.37 Record Types

Records are types for grouping values together. They generalise the dependent product type by providing named fields and (optional) further components.

3.37.1 Example: the Pair type constructor

Record types can be declared using the `record` keyword

```
record Pair (A B : Set) : Set where
  field
    fst : A
    snd : B
```

This defines a new type constructor `Pair : Set → Set → Set` and two projection functions

```
Pair.fst : {A B : Set} → Pair A B → A
Pair.snd : {A B : Set} → Pair A B → B
```

Note

The parameters `A` and `B` are implicit arguments to the projection functions.

```
test-fst : {A B : Set} → Pair A B → A
test-fst p = Pair.fst p

test-snd : {A B : Set} → Pair A B → B
test-snd p = Pair.snd p
```

You can `open` the record type to avoid the need to prefix projections by the name of the record type (see *record modules*):

```
open Pair

test-fst' : {A B : Set} → Pair A B → A
test-fst' p = fst p

test-snd' : {A B : Set} → Pair A B → B
test-snd' p = snd p
```

Elements of record types can be defined using a record expression, where the associations are simple `key = value` pairs;

```
p23 : Pair Nat Nat
p23 = record { fst = 2; snd = 3 }
```

Using a `record where` expression, where the associations are treated like *let bindings*, in that they may refer to previous bindings, may be parametrised, etc; Fields in a `record where` expression can also be inherited from a module, by mentioning all the bindings that should become fields in `using` or `renaming` clauses.

```
p23' : Pair Nat Nat
p23' = record where
   -- use the 'fst' binding in the module as the 'snd' field in this
   -- record:
   open Pair p23 using () renaming (fst to snd)
   fst = 2
```

or using *copatterns*. Copatterns may be used prefix

```
p34 : Pair Nat Nat
Pair.fst p34 = 3
Pair.snd p34 = 4
```

or postfix (in which case they are written prefixed with a dot)

```
p56 : Pair Nat Nat
p56 .Pair.fst = 5
p56 .Pair.snd = 6
```

or using an *pattern lambda* (you may only use the postfix form of copatterns in this case)

```
p78 : Pair Nat Nat
p78 = λ where
  .Pair.fst → 7
  .Pair.snd → 8
```

If you use the `constructor` keyword, you can also use the named constructor to define elements of the record type:

```
record Pair (A B : Set) : Set where
  constructor _,_
  field
    fst : A
    snd : B

p45 : Pair Nat Nat
p45 = 4 , 5
```

Even if you did *not* use the `constructor` keyword, then it's still possible to refer to the record's internally-constructor as a name, using the syntax `Record.constructor`; see *Records with anonymous constructors* below for the details of this syntax.

```
record Anon (A B : Set) : Set where
  field
    fst : A
    snd : B

a45 : Anon Nat Nat
a45 = Anon.constructor 4 5
```

In this sense, record types behave much like single constructor datatypes (but see *Eta-expansion* below).

3.37.2 Declaring, constructing and decomposing records

Declaring record types

The general form of a record declaration is as follows:

```
record <recordname> <parameters> : Set <level> where
  <directives>
  constructor <constructorname>
  field
    <fieldname1> : <type1>
    <fieldname2> : <type2>
```

(continues on next page)

(continued from previous page)

```
    -- ...
  <declarations>
```

All the components are optional, and can be given in any order. In particular, fields can be given in more than one block, interspersed with other declarations. Each field is a component of the record. Types of later fields can depend on earlier fields.

The directives available are `eta-equality`, `no-eta-equality`, `pattern` (see *Eta-expansion*), `inductive` and `coinductive` (see *Recursive records*).

Constructing record values

Record values are constructed by giving a value for each record field:

```
record { <fieldname1> = <term1> ; <fieldname2> = <term2> ; ... }
```

where the types of the terms match the types of the fields. If a constructor `<constructorname>` has been declared for the record, this can also be written

```
<constructorname> <term1> <term2> ...
```

For named definitions, this can also be expressed using copatterns:

```
<named-def> : <recordname> <parameters>
<recordname>.<fieldname1> <named-def> = <term1>
<recordname>.<fieldname2> <named-def> = <term2>
...
```

Records can also be constructed by *updating other records*.

Building records from modules

The `record { <fields> }` syntax also accepts module names. Fields are defined using the corresponding definitions from the given module. For instance assuming this record type R and module M:

```
record R : Set where
  field
    x : X
    y : Y
    z : Z

module M where
   x = ...
   y = ...

r : R
r = record { M; z = ... }
```

This construction supports any combination of explicit field definitions and applied modules. If a field is both given explicitly and available in one of the modules, then the explicit one takes precedence. If a field is available in more than one module then this is ambiguous and therefore rejected. As a consequence the order of assignments does not matter.

The modules can be both applied to arguments and have import directives such as hiding, using, and renaming. Here is a contrived example building on the example above:

```
module M2 (a : A) where
  w = ...
  z = ...

r2 : A → R
r2 a = record { M hiding (y); M2 a renaming (w to y) }
```

Records with anonymous constructors

Even if a record was not defined with a named `constructor` directive, Agda will still internally generate a constructor for the record. This name is used internally to implement `record{}` syntax, but it can still be obtained through using *Reflection*. Since Agda 2.8.0, it's possible to refer to this name from surface syntax as well:

```
_ : Name
_ = quote Anon.constructor
```

This syntax can be used wherever a name can be, and behaves exactly as though the constructor had been named.

```
{-# INLINE Anon.constructor #-}
```

Technically, `constructor` is a name bound in the *record module* `Anon`. However, since `constructor` is a keyword, it can never be written as an unqualified name: it is not possible to declare a function called `constructor`, opening the record module does not make the constructor accessible as `constructor`, and the pseudo-name cannot be listed in `using`, `hiding` *or* `renaming` directives (attempting to do so is a syntax error).

Note also that `constructor` is merely an alias. Unlike the projections, the record constructor is not itself a definition of the record module: it does not take an element of the record type as an argument. Since a *module application* copies the definitions of the module it instantiates, no `M.constructor` is available after `module M = Anon`.

The constructor of a record can be referred to whenever the record module itself is in scope, though note that if the record is abstract (see *Abstract definitions*), it's still an error to refer to the constructor:

```
module _ where private
  record R : Set where

abstract record S : Set where

_ = R.constructor
-- Name not in scope: R.constructor

_ = S.constructor
-- Constructor S.constructor is abstract, thus, not in scope here
```

Decomposing record values

With the field name, we can project the corresponding component out of a record value. Projections can be used either in prefix notation like a function, or in postfix notation by adding a dot to the field name:

```
sum-prefix : Pair Nat Nat → Nat
sum-prefix p = Pair.fst p + Pair.snd p

sum-postfix : Pair Nat Nat → Nat
sum-postfix p = p .Pair.fst + p .Pair.snd
```

It is also possible to pattern match against inductive records:

```
sum-match : Pair Nat Nat → Nat
sum-match (x , y) = x + y
```

Or, using a *let binding record pattern*:

```
sum-let : Pair Nat Nat → Nat
sum-let p = let (x , y) = p in x + y
```

Since Agda 2.9.0, the latter requires records with eta-equality (such as `Pair`) lest warning *ShouldBeEtaRecordPattern* is raised.

Note

Naming the constructor is not required to enable pattern matching against record values. Record expressions can appear as patterns.

```
sum-record-match : Pair Nat Nat → Nat
sum-record-match record { fst = x ; snd = y } = x + y
```

Record update

Assume that we have a record type and a corresponding value:

```
record MyRecord : Set where
  field
    a b c : Nat

old : MyRecord
old = record { a = 1; b = 2; c = 3 }
```

Then we can update (some of) the record value's fields in the following way:

```
new : MyRecord
new = record old { a = 0; c = 5 }
```

or using the `record where` syntax

```
new₁ : MyRecord
new₁ = record old where
  a = 0
  c = 5
```

Here `new` normalises to `record { a = 0; b = 2; c = 5 }`. Any expression yielding a value of type `MyRecord` can be used instead of `old`. Using that *records can be built from module names*, together with the fact that *all records define a module*, this can also be written as

```
new₂ : MyRecord
new₂  = record { MyRecord old; a = 0; c = 5}
```

Record updating is not allowed to change types: the resulting value must have the same type as the original one, including the record parameters. Thus, the type of a record update can be inferred if the type of the original record can be inferred.

The record update syntax is expanded before type checking. When the expression

```
record old { upd-fields }
```

is checked against a record type R, it is expanded to

```
let r = old in record { new-fields }
```

where `old` is required to have type `R` and `new-fields` is defined as follows: for each field `x` in `R`,

- if `x = e` is contained in `upd-fields` then `x = e` is included in `new-fields`, and otherwise
- if `x` is an explicit field then `x = R.x r` is included in `new-fields`, and
- if `x` is an *implicit* or *superclass* field, then it is omitted from `new-fields`.

The reason for treating implicit and superclass fields specially is to allow code like the following:

```
data Vec (A : Set) : Nat → Set where
  [] : Vec A zero
  _::_ : ∀{n} → A → Vec A n → Vec A (suc n)

record VList : Set where
  field
    {length} : Nat
    vec      : Vec Nat length
    -- More fields ...

xs : VList
xs = record { vec = 0 :: 1 :: 2 :: [] }

ys = record xs { vec = 0 :: [] }
```

Without the special treatment the last expression would need to include a new binding for `length` (for instance `length = _`).

3.37.3 Record modules

Along with a new type, a record declaration also defines a module with the same name, parameterised over an element of the record type containing the projection functions. This allows records to be "opened", bringing the fields into scope. For instance

```
swap : {A B : Set} → Pair A B → Pair B A
swap p = snd , fst
  where open Pair p
```

In the example, the record module `Pair` has the shape

```
module Pair {A B : Set} (p : Pair A B) where
  fst : A
  snd : B
```

> **Note**
>
> This is not quite right: The projection functions take the parameters as *erased* arguments. However, the parameters are not erased in the module telescope if they were not erased to start with.

It's possible to add arbitrary definitions to the record module, by defining them inside the record declaration

```
record Functor (F : Set → Set) : Set₁ where
  field
    fmap : ∀ {A B} → (A → B) → F A → F B

  _<$_ : ∀ {A B} → A → F B → F A
  x <$ fb = fmap (λ _ → x) fb
```

Note

In general new definitions need to appear after the field declarations, but simple non-recursive function definitions without pattern matching can be interleaved with the fields. The reason for this restriction is that the type of the record constructor needs to be expressible using *let-expressions*. In the example below `D₁` can only contain declarations for which the generated type of `mkR` is well-formed.

```
record R Γ : Setᵢ where
  constructor mkR
  field f₁ : A₁
  D₁
  field f₂ : A₂

mkR : ∀ {Γ} (f₁ : A₁) (let D₁) (f₂ : A₂) → R Γ
```

3.37.4 Eta-expansion

The eta (η) rule for a record type

```
record R : Set where
   field
     a : A
     b : B
     c : C
```

states that every `x : R` is definitionally equal to `record { a = R.a x ; b = R.b x ; c = R.c x }`.

```
eta-R : (x : R) → x ≡ record { a = R.a x ; b = R.b x ; c = R.c x }
eta-R r = refl
```

Record types enjoy η-equality by default (option:*–eta-equality*) with the exception of coinductive records. The keywords `eta-equality`/`no-eta-equality` enable/disable η rules for the record type being declared.

```
record R-noeta : Set where
  no-eta-equality
  field
    a : A
    b : B
    c : C
```

3.37.5 Recursive records

A recursive record is a record where the record type itself appears in the type of one of its fields. Recursive records need to be declared as either `inductive` or `coinductive`.

Inductive records

Inductive records are recursive records that only allow values of finite depth.

```
record Tree (A : Set) : Set where
  inductive
  constructor tree
  field
    elem     : A
    subtrees : List (Tree A)

open Tree
```

Inductive record types (see *Recursive records*) have η-equality enabled by default (unless `--no-eta-equality` is given). (Unguarded records should be annotated with `no-eta-equality`, see section *Unguarded records*.)

```
eta-Tree : {A : Set} (t : Tree A) → t ≡ tree (elem t) (subtrees t)
eta-Tree t = refl
```

It is possible to pattern match and recurse on an inductive record if it has η-equality:

```
map-Tree : {A B : Set} → (A → B) → Tree A → Tree B
map-Tree {A} {B} f (tree x ts) = tree (f x) (map-subtrees ts)
  where
    map-subtrees : List (Tree A) → List (Tree B)
    map-subtrees [] = []
    map-subtrees (t ∷ ts) = map-Tree f t ∷ map-subtrees ts
```

For inductive record types *without* η-equality, pattern matching is not allowed by default. Pattern matching can be turned on manually by using the `pattern` record directive:

```
record HereditaryList : Set where
  inductive
  no-eta-equality
  pattern
  field sublists : List HereditaryList

pred : HereditaryList → List HereditaryList
pred record{ sublists = ts } = ts
```

If both `eta-equality` and `pattern` are given for a record types, Agda will alert the user of a redundant `pattern` directive with warning *UselessPatternDeclarationForRecord*.

Note

It is not allowed to use copattern matching to define values of inductive record types with pattern matching enabled. This combination leads to either a loss of canonicity or a loss of subject reduction. For example, consider the following definitions:

```
record Rec : Set where
  constructor con
```

```
  no-eta-equality
  pattern
  field
    f : Nat
open Rec

eta : (r : Rec) → r ≡ con (f r)
eta (con n) = refl

bar : Rec
f bar = 0
```

If this code were allowed, then `eta bar` is a closed term of type `bar ≡ con 0`. Now either `eta bar` reduces to `refl : bar ≡ con 0` (contradicting the `no-eta-equality` directive) or else `eta bar` is a stuck term (breaking canonicity).

Unguarded records

η-equality is not always safe for recursive records as it could lead to infinite η-expansion. This is the case for so-called unguarded records where the recursive occurrence is not guarded by a type former that does not have η (and thus stops infinite expansion). η-equality should be turned off for such records:

```
record Empty : Set where
  inductive
  no-eta-equality; pattern
  field emp : Empty

isReallyEmpty : Empty → {A : Set} → A
isReallyEmpty record{ emp = x } = isReallyEmpty x
```

Agda points out unguarded records that have η, see warning *UnguardedEtaRecord*. Agda's unguarded-record detection is not perfect, so in some cases it is safe to have η despite Agda's warning. In such cases, one can prefix the `record` declaration with the *ETA_EQUALITY* pragma to silence the warning.

```
mutual
  {-# ETA_EQUALITY #-}
  record NonEmptyTuple (A : Set) (n : Nat) : Set where
    inductive; eta-equality
    field theTuple : FTuple A n

  FTuple : (A : Set) (n : Nat) → Set
  FTuple A zero    = A
  FTuple A (suc n) = Pair A (NonEmptyTuple A n)

nonEmptyTupleEta : {A : Set} {n : Nat} (t : NonEmptyTuple A n)
  → t ≡ record { theTuple = NonEmptyTuple.theTuple t }
nonEmptyTupleEta t = refl
```

Coinductive records

Coinductive records are recursive records that allow values of possibly infinite depth.

```
record Stream (A : Set) : Set where
  coinductive
  constructor _::_
  field
    head : A
    tail : Stream A

open Stream
```

Values of coinductive records can be defined using copatterns:

```
natsFrom : Nat → Stream Nat
head (natsFrom n) = n
tail (natsFrom n) = natsFrom (suc n)
```

Constructors of records supporting copattern matching may be marked with an `{-# INLINE #-}` *pragma*. This will automatically convert uses of the constructor to the equivalent definition using copatterns, which can be useful to assist the termination checker.

Eta equality for coinductive records is not allowed, since this combination could easily make Agda loop. This can be overridden at your own risk by using the *ETA_EQUALITY* instead. Pattern matching on coinductive records is likewise not allowed.

You can read more about coinductive records in the section on *coinduction*.

3.37.6 Records and instance search

The fields of a record type can interact with the *instance search* mechanism in two orthogonal ways.

1. A record field can be given instance *visibility*, by wrapping its name in double braces `{{ }}`. To disambiguate, we refer to record fields with instance visibility as **superclass fields**.

   Superclass fields are instance arguments to the record constructor. This means that they *can often be omitted*.

2. The field declaration itself can be nested in an `instance` block. We refer to these as (having) **instance projections**.

   Making a record field into an instance projection does *not* alter the visibility with which it is bound, meaning that, unless it is additionally made into a superclass field (or into a hidden argument to the record constructor), it must be explicitly specified when constructing a record value.

Any given record field can be made into both an instance projection *and* a superclass field. It will then be subject to *both* of the behaviours described in the following sections:

```
record Ex2 (A : Set) : Set where
  field
    instance PDF TODO both PDF TODO : Ex1 A
```

Both superclass fields and instance projections can appear any number of times in a record declaration, and in any position in the list of fields.

Superclass fields

As the name implies, superclass fields are used to model superclass relationships (in the Haskell sense). For example, we can define a class `Ord` that "extends" a class `Eq` by defining a pair of record types, where an `Ord` value has a superclass field of `Eq` type:

```
record Eq (A : Set) : Set where
  field
    _==_ : A → A → Bool

open Eq PDF TODO ... PDF TODO

record Ord (A : Set) : Set where
  field
    _<_     : A → A → Bool
    PDF TODO eqA PDF TODO : Eq A

open Ord PDF TODO ... PDF TODO hiding (eqA)
```

As long as `Ord` is an *eta record*, local *instance* variables of type `Ord A` will also bring `Eq A` instances into scope. For example, a function taking an `Ord A` instance argument can also use an `Eq` instance:

```
_≤_ : {A : Set} PDF TODO ordA : Ord A PDF TODO → A → A → Bool
x ≤ y = (x == y) || (x < y)
```

Superclass fields are **only** brought into scope if the "subclass" variable is in scope *as an instance*. The following will not work:

```
weird : {A : Set} (ordA : Ord A) → A → A → Bool
weird _ = x == y
```

However, since superclass fields are instance arguments to a constructor, they can be manually brought into scope by matching on the record value. Keep in mind that if the record value had non-instance visibility, then the **subclass** will not be available for instance search, even after pattern matching:

```
works : {A : Set} (ordA : Ord A) → A → A → Bool
works record{} x y = x == y
-- Matching on record{} makes all the fields into local variables,
-- including the superclass fields.
```

```
fails : {A : Set} (ordA : Ord A) → A → A → Bool
fails record{} x y = x ≤ y
-- No instance of type Ord A was found in scope.
-- Since the Ord argument is visible, it is totally ignored by
-- instance search.
```

Warning

Superclass fields are brought into the local instance table by **expanding local instance variables**. This means that Agda can fail to find an instance of a subclass *even if* the corresponding superclass is derivable. For example, we can explicitly provide the `eqA` field of an `Ord` instance when constructing it, even if there is no corresponding instance in scope:

```
instance
  OrdNat : Ord Nat
  OrdNat = record
    { _<_ = Agda.Builtin.Nat._<_
    ; eqA = record { _==_ = Agda.Builtin.Nat._==_ }
    -- no global Eq Nat instance!
    }
```

Attempting to search for an `Eq Nat` instance will then fail, because superclass field expansion *only* applies to local variables with instance visibility, and not to top-level instance declarations.

```
fails : Bool
fails = 1 == 2
-- No instance of type Eq Nat was found in scope.
```

Eta-expanding instance variables to find superclass fields will also work under binders, which means that a *family* of subclass instances gets expanded into a *family* of instances for the corresponding superclass:

```
fam
  : {Ix : Set} {F : Ix → Set} PDF TODO ords : ∀ {i} → Ord (F i) PDF TODO
  → (ix : Ix) → F ix → F ix → Bool
fam ix x y = x == y
```

If the type of a superclass field is *itself* a record type with superclass fields, then it will be expanded recursively:

```
data ORD : Set where
  lt le eq : ORD

record Cmp (A : Set) : Set where
  field
    PDF TODO ordA PDF TODO : Ord A
    compare  : A → A → ORD

cmp→eq : {A : Set} PDF TODO ordA : Cmp A PDF TODO → A → A → Bool
cmp→eq x y = x == y
```

Superclass fields are in scope as local instances in the types of subsequent fields, and in the types of any declarations nested within the record:

```
record Bounded (A : Set) : Set where
  field
    PDF TODO cmpA PDF TODO : Cmp A
    lo hi    : A

  -- Declaration *between* fields:
  ordered : Bool
  ordered = lo < hi

  field in-order : ordered ≡ true

  -- Declaration *after* fields:
  is-empty : Bool
  is-empty = lo == hi
```

Any subsequent declaration within the record can depend on its superclass fields through instance search, and these fields are themselves available for superclass expansion.

Warning

Opening a record module, even locally, will **not** make superclass fields into local instances. Only the *instance projections* will be brought into scope by a module application.

```
fails : {A : Set} (ordA : Ord A) → A → A → Bool
fails ord x y = let open Ord ord in x == y
-- no instance of Eq A in scope, since the superclass field eqA does
-- not have an instance projection
```

Overlapping superclass fields

When multiple subclasses "inherit" from the same class, superclass field expansion will produce distinct candidates for the superclass by expanding each of the subclasses. In this situation, instance search will fail with an unresolved overlap.

This can be remedied by marking **all** of the relevant superclass fields with the `overlap` keyword. For example, we can define a `Num` class that also "extends" `Eq`, as long as `Ord` is redefined to have its `Eq` superclass field also marked `overlap`:

```
record Ord (A : Set) : Set where
  field
    _<_     : A → A → Bool
    overlap PDF TODO eqA PDF TODO : Eq A

record Num (A : Set) : Set where
  field
    fromNat         : Nat → A
    overlap PDF TODO eqA PDF TODO : Eq A

open Ord PDF TODO ... PDF TODO hiding (eqA)
open Num PDF TODO ... PDF TODO hiding (eqA)
```

We can now define a function that takes both `Num` and `Ord` instances:

```
_≤3 : {A : Set} PDF TODO ordA : Ord A PDF TODO PDF TODO numA : Num A PDF TODO␣
↪→ A → Bool
x ≤3 = (x == fromNat 3) || (x < fromNat 3)
```

When all possible candidates for an instance constraint arise from superclass fields marked `overlap`, instance search will choose the field arising from the leftmost record value. In the function above, that is the candidate coming from the `Ord` argument:

```
_
  : {A : Set} PDF TODO ordA : Ord A PDF TODO PDF TODO numA : Num A PDF TODO
↪{arg : A}
  → (arg ≤3) ≡ ((_==_ PDF TODO Ord.eqA ordA PDF TODO arg (fromNat 3)) || _)
_ = refl
```

If overlapping candidates are introduced by *recursive* superclass expansion, resolution will prefer those arising from the earlier field in declaration order. Candidates expanded 'on the way' do not, themselves, need to be marked overlap; nor will this be sufficient for resolving an overlap in *their* superclass fields.

```
record NumOrd (A : Set) : Set where
  field
    PDF TODO numA PDF TODO : Num A
    PDF TODO ordA PDF TODO : Ord A
```

(continues on next page)

(continued from previous page)

```
_≤4 : {A : Set} PDF TODO numordA : NumOrd A PDF TODO → A → Bool
x ≤4 = (x == fromNat 4) || (x < fromNat 4)
```

Here, the `Eq` instance is selected from the `numA` field of the `NumOrd` argument:

```
_
  : {A : Set} PDF TODO numordA : NumOrd A PDF TODO {arg : A}
  → (arg ≤4) ≡ ((_==_ PDF TODO numordA .numA .eqA PDF TODO arg (fromNat 4)) || _)
_ = refl
```

Omitting superclass fields

Since superclass fields become instance arguments to the constructor, they can be omitted when the constructor is applied as a function, when using either form of `record` expression, and when defining a value of the record by copattern matching. In any of these cases, the missing fields will be filled by instance search:

```
instance
  EqNat : Eq Nat
  EqNat ._==_ = Agda.Builtin.Nat._==_

ex1 ex2 ex3 : Ord Nat
ex1 ._<_ = Agda.Builtin.Nat._<_
ex2 = record { _<_ = Agda.Builtin.Nat._<_ }
ex3 = record where
  _<_ = Agda.Builtin.Nat._<_
```

Instance projections

A record field defined in a nested `instance` block makes the projection function into a top-level instance declaration nested in the record module. It does not affect the visibility of the field in the record constructor's telescope:

```
record Eqtype : Set₁ where
  no-eta-equality -- (!)
  field
    carrier      : Set
    instance eqA : Eq carrier

open Eqtype renaming (carrier to [_])
```

A priori, this has no effect on the instance table, since definitions within the record module take the record as a **visible** argument. However, if the record module is instantiated, as long as the type of the resulting instantiation is a valid type for an instance, the projection will become usable as an instance, at the instantiated type:

```
example : (T : Eqtype) → [ T ] → [ T ] → Bool
example T x y = let open Eqtype T in x == y
```

In the example above, the local declarations introduced by the `let open Eqtype T` expression are equivalent to:

```
example' : (T : Eqtype) → [ T ] → [ T ] → Bool
example' T x y =
  let
    carrier = Eqtype.carrier T
```

(continues on next page)

(continued from previous page)

```
    instance
      eqA : Eq carrier
      eqA = Eqtype.eqA T
  in x == y
```

Note

Because instance projections are brought into scope by instantiations of the record module, they work even if they belong to a `no-eta-equality` record type.

Since instance projections are fully equivalent to defining a top-level `instance` in the record module, they can be annotated with one of the *overlap pragmas* for fine-grained control of the overlapping behaviour of the instances resulting from an application of the record module.

Warning

If an instance projection has a *modality* that prevents it from having a corresponding top-level projection (e.g., it is irrelevant, and *`--irrelevant-projections`* was not given), instantiating the record module will **not** bring it into scope as an instance:

```
postulate
  Nonzero : Nat → Set
  _/_ : Nat → (div : Nat) PDF TODO _ : Nonzero div PDF TODO → Nat

record Pos : Set where
  field
    num : Nat
    instance .pos : Nonzero num
```

Attempting to use the `pos` field as an instance by opening the module `Pos` will fail.

```
fails : (x : Nat) (y : Pos) → Nat
fails x y = let open Pos y in x / num
-- The field 'pos' has no projection, so:
-- No instance of type Nonzero num was found in scope.
```

3.38 Reflection

3.38.1 Builtin types

Names

The built-in `QNAME` type represents quoted names and comes equipped with equality, ordering, and a show function.

```
postulate Name : Set
{-# BUILTIN QNAME Name #-}

primitive
  primQNameEquality : Name → Name → Bool
  primQNameLess     : Name → Name → Bool
  primShowQName     : Name → String
```

The fixity of a name can also be retrived.

```
primitive
  primQNameFixity    : Name → Fixity
```

To define a decidable propositional equality with the option *--safe*, one can use the conversion to a pair of built-in 64-bit machine words

```
primitive
  primQNameToWord64s : Name → Σ Word64 (λ _ → Word64)
```

with the injectivity proof in the `Properties` module.:

```
primitive
  primQNameToWord64sInjective : ∀ a b → primQNameToWord64s a ≡ primQNameToWord64s b →␣
↪a ≡ b
```

Name literals are created using the `quote` keyword and can appear both in terms and in patterns

```
nameOfNat : Name
nameOfNat = quote Nat

isNat : Name → Bool
isNat (quote Nat) = true
isNat _           = false
```

Note that the name being quoted must be in scope.

Metavariables

Metavariables are represented by the built-in `AGDAMETA` type. They have primitive equality, ordering, show, and conversion to Nat:

```
postulate Meta : Set
{-# BUILTIN AGDAMETA Meta #-}

primitive
  primMetaEquality : Meta → Meta → Bool
  primMetaLess     : Meta → Meta → Bool
  primShowMeta     : Meta → String
  primMetaToNat    : Meta → Nat
```

Builtin metavariables show up in reflected terms. In `Properties`, there is a proof of injectivity of `primMetaToNat`

```
primitive
  primMetaToNatInjective : ∀ a b → primMetaToNat a ≡ primMetaToNat b → a ≡ b
```

which can be used to define a decidable propositional equality with the option *--safe*.

Literals

Literals are mapped to the built-in `AGDALITERAL` datatype. Given the appropriate built-in binding for the types `Nat`, `Float`, etc, the `AGDALITERAL` datatype has the following shape:

```
data Literal : Set where
  nat    : (n : Nat)    → Literal
  word64 : (n : Word64) → Literal
  float  : (x : Float)  → Literal
  char   : (c : Char)   → Literal
  string : (s : String) → Literal
  name   : (x : Name)   → Literal
  meta   : (x : Meta)   → Literal

{-# BUILTIN AGDALITERAL   Literal #-}
{-# BUILTIN AGDALITNAT    nat     #-}
{-# BUILTIN AGDALITWORD64 word64  #-}
{-# BUILTIN AGDALITFLOAT  float   #-}
{-# BUILTIN AGDALITCHAR   char    #-}
{-# BUILTIN AGDALITSTRING string  #-}
{-# BUILTIN AGDALITQNAME  name    #-}
{-# BUILTIN AGDALITMETA   meta    #-}
```

Arguments

Arguments can be (visible), {hidden}, or {{instance}}:

```
data Visibility : Set where
  visible hidden instance′ : Visibility

{-# BUILTIN HIDING   Visibility #-}
{-# BUILTIN VISIBLE  visible    #-}
{-# BUILTIN HIDDEN   hidden     #-}
{-# BUILTIN INSTANCE instance′  #-}
```

Arguments can be relevant or irrelevant:

```
data Relevance : Set where
  relevant irrelevant : Relevance

{-# BUILTIN RELEVANCE  Relevance  #-}
{-# BUILTIN RELEVANT   relevant   #-}
{-# BUILTIN IRRELEVANT irrelevant #-}
```

Arguments also have a quantity:

```
data Quantity : Set where
  quantity-0 quantity-ω : Quantity

{-# BUILTIN QUANTITY   Quantity   #-}
{-# BUILTIN QUANTITY-0 quantity-0 #-}
{-# BUILTIN QUANTITY-ω quantity-ω #-}
```

Relevance and quantity are combined into a modality:

```
data Modality : Set where
  modality : (r : Relevance) (q : Quantity) → Modality
```

(continues on next page)

(continued from previous page)

```
{-# BUILTIN MODALITY             Modality #-}
{-# BUILTIN MODALITY-CONSTRUCTOR modality #-}
```

The visibility and the modality characterise the behaviour of an argument:

```
data ArgInfo : Set where
  arg-info : (v : Visibility) (m : Modality) → ArgInfo

data Arg (A : Set) : Set where
  arg : (i : ArgInfo) (x : A) → Arg A

{-# BUILTIN ARGINFO    ArgInfo  #-}
{-# BUILTIN ARGARGINFO arg-info #-}
{-# BUILTIN ARG        Arg      #-}
{-# BUILTIN ARGARG     arg      #-}
```

Name abstraction

```
data Abs (A : Set) : Set where
  abs : (s : String) (x : A) → Abs A

{-# BUILTIN ABS    Abs #-}
{-# BUILTIN ABSABS abs #-}
```

Terms

Terms, sorts, patterns, and clauses are mutually recursive and mapped to the `AGDATERM`, `AGDASORT`, `AGDAPATTERN` and `AGDACLAUSE` built-ins respectively. Types are simply terms. Terms and patterns use de Bruijn indices to represent variables.

```
data Term : Set
data Sort : Set
data Pattern : Set
data Clause : Set
Type = Term
Telescope = List (Σ String λ _ → Arg Type)

data Term where
  var       : (x : Nat) (args : List (Arg Term)) → Term
  con       : (c : Name) (args : List (Arg Term)) → Term
  def       : (f : Name) (args : List (Arg Term)) → Term
  lam       : (v : Visibility) (t : Abs Term) → Term
  pat-lam   : (cs : List Clause) (args : List (Arg Term)) → Term
  pi        : (a : Arg Type) (b : Abs Type) → Term
  agda-sort : (s : Sort) → Term
  lit       : (l : Literal) → Term
  meta      : (x : Meta) → List (Arg Term) → Term
  unknown   : Term -- Treated as '_' when unquoting.

data Sort where
  set     : (t : Term) → Sort -- A Set of a given (possibly neutral) level.
  lit     : (n : Nat) → Sort  -- A Set of a given concrete level.
```

(continues on next page)

(continued from previous page)

```
  prop    : (t : Term) → Sort -- A Prop of a given (possibly neutral) level.
  propLit : (n : Nat) → Sort  -- A Prop of a given concrete level.
  inf     : (n : Nat) → Sort  -- Setωi of a given concrete level i.
  unknown : Sort

data Pattern where
  con    : (c : Name) (ps : List (Arg Pattern)) → Pattern
  dot    : (t : Term)    → Pattern
  var    : (x : Nat   )  → Pattern
  lit    : (l : Literal) → Pattern
  proj   : (f : Name)    → Pattern
  absurd : (x : Nat)     → Pattern  -- Absurd patterns have de Bruijn indices

data Clause where
  clause        : (tel : Telescope) (ps : List (Arg Pattern)) (t : Term) → Clause
  absurd-clause : (tel : Telescope) (ps : List (Arg Pattern)) → Clause

{-# BUILTIN AGDATERM    Term   #-}
{-# BUILTIN AGDASORT    Sort   #-}
{-# BUILTIN AGDAPATTERN Pattern #-}
{-# BUILTIN AGDACLAUSE  Clause #-}

{-# BUILTIN AGDATERMVAR         var       #-}
{-# BUILTIN AGDATERMCON         con       #-}
{-# BUILTIN AGDATERMDEF         def       #-}
{-# BUILTIN AGDATERMMETA        meta      #-}
{-# BUILTIN AGDATERMLAM         lam       #-}
{-# BUILTIN AGDATERMEXTLAM      pat-lam   #-}
{-# BUILTIN AGDATERMPI          pi        #-}
{-# BUILTIN AGDATERMSORT        agda-sort #-}
{-# BUILTIN AGDATERMLIT         lit       #-}
{-# BUILTIN AGDATERMUNSUPPORTED unknown   #-}

{-# BUILTIN AGDASORTSET         set     #-}
{-# BUILTIN AGDASORTLIT         lit     #-}
{-# BUILTIN AGDASORTPROP        prop    #-}
{-# BUILTIN AGDASORTPROPLIT     propLit #-}
{-# BUILTIN AGDASORTINF         inf     #-}
{-# BUILTIN AGDASORTUNSUPPORTED unknown #-}

{-# BUILTIN AGDAPATCON    con     #-}
{-# BUILTIN AGDAPATDOT    dot     #-}
{-# BUILTIN AGDAPATVAR    var     #-}
{-# BUILTIN AGDAPATLIT    lit     #-}
{-# BUILTIN AGDAPATPROJ   proj    #-}
{-# BUILTIN AGDAPATABSURD absurd  #-}

{-# BUILTIN AGDACLAUSECLAUSE clause        #-}
{-# BUILTIN AGDACLAUSEABSURD absurd-clause #-}
```

Absurd lambdas `λ ()` are quoted to extended lambdas with an absurd clause.

The built-in constructors `AGDATERMUNSUPPORTED` and `AGDASORTUNSUPPORTED` are translated to meta variables when

unquoting.

Declarations

There is a built-in type `AGDADEFINITION` representing definitions. Values of this type is returned by the `AGDATCMGETDEFINITION` built-in *described below*.

```
data Definition : Set where
  function    : (cs : List Clause) → Definition
  data-type   : (pars : Nat) (cs : List Name) → Definition  -- parameters and␣
↪constructors
  record-type : (c : Name) (fs : List (Arg Name)) →         -- c: name of record␣
↪constructor
                Definition                                  -- fs: fields
  data-cons   : (d : Name) (q : Quantity) → Definition      -- d: name of data type
                                                            -- q: constructor quantity
  axiom       : Definition
  prim-fun    : Definition

{-# BUILTIN AGDADEFINITION                Definition  #-}
{-# BUILTIN AGDADEFINITIONFUNDEF          function    #-}
{-# BUILTIN AGDADEFINITIONDATADEF         data-type   #-}
{-# BUILTIN AGDADEFINITIONRECORDDEF       record-type #-}
{-# BUILTIN AGDADEFINITIONDATACONSTRUCTOR data-cons   #-}
{-# BUILTIN AGDADEFINITIONPOSTULATE       axiom       #-}
{-# BUILTIN AGDADEFINITIONPRIMITIVE       prim-fun    #-}
```

Type errors

Type checking computations (see *below*) can fail with an error, which is a list of `ErrorParts`. This allows metaprograms to generate nice errors without having to implement pretty printing for reflected terms.

```
-- Error messages can contain embedded names and terms.
data ErrorPart : Set where
  strErr  : String → ErrorPart
  termErr : Term → ErrorPart
  pattErr : Pattern → ErrorPart
  nameErr : Name → ErrorPart

{-# BUILTIN AGDAERRORPART       ErrorPart #-}
{-# BUILTIN AGDAERRORPARTSTRING strErr    #-}
{-# BUILTIN AGDAERRORPARTTERM   termErr   #-}
{-# BUILTIN AGDAERRORPARTNAME   nameErr   #-}
```

Blockers

A blocker represents a set of metavariables that impedes the progress of a reflective computation. Using a blocker containing all the metas in (for example) a term traversed by a macro is a lot more efficient than blocking on individual metas as they are encountered.

```
data Blocker : Set where
  blockerAny  : List Blocker → Blocker
  blockerAll  : List Blocker → Blocker
  blockerMeta : Meta → Blocker
```

(continues on next page)

(continued from previous page)

```
{-# BUILTIN AGDABLOCKER     Blocker #-}
{-# BUILTIN AGDABLOCKERANY  blockerAny #-}
{-# BUILTIN AGDABLOCKERALL  blockerAll #-}
{-# BUILTIN AGDABLOCKERMETA blockerMeta #-}
```

Type checking computations

Metaprograms, i.e. programs that create other programs, run in a built-in type checking monad TC:

```
postulate
  TC       : ∀ {a} → Set a → Set a
  returnTC : ∀ {a} {A : Set a} → A → TC A
  bindTC   : ∀ {a b} {A : Set a} {B : Set b} → TC A → (A → TC B) → TC B

{-# BUILTIN AGDATCM       TC       #-}
{-# BUILTIN AGDATCMRETURN returnTC #-}
{-# BUILTIN AGDATCMBIND   bindTC   #-}
```

The TC monad provides an interface to the Agda type checker using the following primitive operations:

```
postulate
  -- Unify two terms, potentially solving metavariables in the process.
  unify : Term → Term → TC ⊤

  -- Throw a type error. Can be caught by catchTC.
  typeError : ∀ {a} {A : Set a} → List ErrorPart → TC A

  -- Block a type checking computation on a blocker. This will abort
  -- the computation and restart it (from the beginning) when the
  -- blocker has been solved.
  blockTC : ∀ {a} {A : Set a} → Blocker → TC A

  -- Prevent current solutions of metavariables from being rolled back in
  -- case 'blockOnMeta' is called.
  commitTC : TC ⊤

  -- Backtrack and try the second argument if the first argument throws a
  -- type error.
  catchTC : ∀ {a} {A : Set a} → TC A → TC A → TC A

  -- Infer the type of a given term
  inferType : Term → TC Type

  -- Check a term against a given type. This may resolve implicit arguments
  -- in the term, so a new refined term is returned. Can be used to create
  -- new metavariables: newMeta t = checkType unknown t
  checkType : Term → Type → TC Term

  -- Compute the normal form of a term.
  normalise : Term → TC Term
```

(continues on next page)

(continued from previous page)

```
-- Compute the weak head normal form of a term.
reduce : Term → TC Term

-- Get the current context. Returns the context in reverse order, so that
-- it is indexable by deBruijn index. Note that the types in the context are
-- valid in the rest of the context. To use in the current context they need
-- to be weakened by 1 + their position in the list.
getContext : TC Telescope

-- Extend the current context with a variable of the given type and its name.
extendContext : ∀ {a} {A : Set a} → String → Arg Type → TC A → TC A

-- Set the current context relative to the context the TC computation
-- is invoked from.  Takes a context telescope entries in reverse
-- order, as given by `getContext`. Each type should be valid in the
-- context formed by the remaining elements in the list.
inContext : ∀ {a} {A : Set a} → Telescope → TC A → TC A

-- Quote a value, returning the corresponding Term.
quoteTC : ∀ {a} {A : Set a} → A → TC Term

-- Unquote a Term, returning the corresponding value.
unquoteTC : ∀ {a} {A : Set a} → Term → TC A

-- Quote a value in Setω, returning the corresponding Term
quoteωTC : ∀ {A : Setω} → A → TC Term

-- Create a fresh name.
freshName : String → TC Name

-- Declare a new function of the given type. The function must be defined
-- later using 'defineFun'. Takes an Arg Name to allow declaring instances
-- and irrelevant functions. The Visibility of the Arg must not be hidden.
declareDef : Arg Name → Type → TC ⊤

-- Declare a new postulate of the given type. The Visibility of the Arg
-- must not be hidden. It fails when executed from command-line with --safe
-- option.
declarePostulate : Arg Name → Type → TC ⊤

-- Declare a new datatype. The second argument is the number of parameters.
-- The third argument is the type of the datatype, i.e. its parameters and
-- indices. The datatype must be defined later using 'defineData'.
declareData      : Name → Nat → Type → TC ⊤

-- Define a declared datatype. The datatype must have been declared using
-- 'declareData`. The second argument is a list of triples in which each triple
-- is the name of a constructor, its erasure status and its type.
defineData       : Name → List (Σ Name (λ _ → Σ Quantity (λ _ → Type))) → TC ⊤

-- Define a declared function. The function may have been declared using
-- 'declareDef' or with an explicit type signature in the program.
```

(continues on next page)

(continued from previous page)

```
  defineFun : Name → List Clause → TC ⊤

  -- Get the type of a defined name relative to the current
  -- module. Replaces 'primNameType'.
  getType : Name → TC Type

  -- Get the definition of a defined name relative to the current
  -- module. Replaces 'primNameDefinition'.
  getDefinition : Name → TC Definition

  -- Check if a name refers to a macro
  isMacro : Name → TC Bool

  -- Generate FOREIGN pragma with specified backend and top-level backend-dependent text.
  pragmaForeign : String → String → TC ⊤

  -- Generate COMPILE pragma with specified backend, associated name and backend-
↪dependent text.
  pragmaCompile : String → Name → String → TC ⊤

  -- Change the behaviour of inferType, checkType, quoteTC, getContext
  -- to normalise (or not) their results. The default behaviour is no
  -- normalisation.
  withNormalisation : ∀ {a} {A : Set a} → Bool → TC A → TC A
  askNormalisation  : TC Bool

  -- If 'true', makes the following primitives to reconstruct hidden arguments:
  -- getDefinition, normalise, reduce, inferType, checkType and getContext
  withReconstructed : ∀ {a} {A : Set a} → Bool → TC A → TC A
  askReconstructed  : TC Bool

  -- Whether implicit arguments at the end should be turned into metavariables
  withExpandLast : ∀ {a} {A : Set a} → Bool → TC A → TC A
  askExpandLast  : TC Bool

  -- White/blacklist specific definitions for reduction while executing the TC␣
↪computation
  -- 'true' for whitelist, 'false' for blacklist
  withReduceDefs : ∀ {a} {A : Set a} → (Σ Bool λ _ → List Name) → TC A → TC A
  askReduceDefs  : TC (Σ Bool λ _ → List Name)

  -- Parse and type check the given string against the given type, returning
  -- the resulting term (when successful).
  checkFromStringTC : String → Type → TC Term

  -- Prints the third argument to the debug buffer in Emacs
  -- if the verbosity level (set by the -v flag to Agda)
  -- is higher than the second argument. Note that Level 0 and 1 are printed
  -- to the info buffer instead. For instance, giving -v a.b.c:10 enables
  -- printing from debugPrint "a.b.c.d" 10 msg.
  debugPrint : String → Nat → List ErrorPart → TC ⊤
```

(continues on next page)

(continued from previous page)

```
  -- Return the formatted string of the argument using the internal pretty printer.
  formatErrorParts : List ErrorPart → TC String

  -- Fail if the given computation gives rise to new, unsolved
  -- "blocking" constraints.
  noConstraints : ∀ {a} {A : Set a} → TC A → TC A

  -- Run the given computation at the type level, allowing use of erased things.
  workOnTypes : ∀ {a} {A : Set a} → TC A → TC A

  -- Run the given TC action and return the first component. Resets to
  -- the old TC state if the second component is 'false', or keep the
  -- new TC state if it is 'true'.
  runSpeculative : ∀ {a} {A : Set a} → TC (Σ A λ _ → Bool) → TC A

  -- Get a list of all possible instance candidates for the given meta
  -- variable (it does not have to be an instance meta).
  getInstances : Meta → TC (List Term)

  -- Try to solve open instance constraints. When wrapped in `noConstraints`,
  -- fails if there are unsolved instance constraints left over that originate
  -- from the current macro invokation. Outside constraints are still attempted,
  -- but failure to solve them are ignored by `noConstraints`.
  solveInstanceConstraints : TC ⊤

{-# BUILTIN AGDATCMUNIFY                  unify                #-}
{-# BUILTIN AGDATCMTYPEERROR              typeError            #-}
{-# BUILTIN AGDATCMBLOCK                  blockTC              #-}
{-# BUILTIN AGDATCMCATCHERROR             catchTC              #-}
{-# BUILTIN AGDATCMINFERTYPE              inferType            #-}
{-# BUILTIN AGDATCMCHECKTYPE              checkType            #-}
{-# BUILTIN AGDATCMNORMALISE              normalise            #-}
{-# BUILTIN AGDATCMREDUCE                 reduce               #-}
{-# BUILTIN AGDATCMGETCONTEXT             getContext           #-}
{-# BUILTIN AGDATCMEXTENDCONTEXT          extendContext        #-}
{-# BUILTIN AGDATCMINCONTEXT              inContext            #-}
{-# BUILTIN AGDATCMQUOTETERM              quoteTC              #-}
{-# BUILTIN AGDATCMUNQUOTETERM            unquoteTC            #-}
{-# BUILTIN AGDATCMQUOTEOMEGATERM         quoteωTC             #-}
{-# BUILTIN AGDATCMFRESHNAME              freshName            #-}
{-# BUILTIN AGDATCMDECLAREDEF             declareDef           #-}
{-# BUILTIN AGDATCMDECLAREPOSTULATE       declarePostulate     #-}
{-# BUILTIN AGDATCMDECLAREDATA            declareData          #-}
{-# BUILTIN AGDATCMDEFINEDATA             defineData           #-}
{-# BUILTIN AGDATCMDEFINEFUN              defineFun            #-}
{-# BUILTIN AGDATCMGETTYPE                getType              #-}
{-# BUILTIN AGDATCMGETDEFINITION          getDefinition        #-}
{-# BUILTIN AGDATCMCOMMIT                 commitTC             #-}
{-# BUILTIN AGDATCMISMACRO                isMacro              #-}
{-# BUILTIN AGDATCMPRAGMAFOREIGN          pragmaForeign        #-}
{-# BUILTIN AGDATCMPRAGMACOMPILE          pragmaCompile        #-}
{-# BUILTIN AGDATCMWITHNORMALISATION      withNormalisation    #-}
```

(continues on next page)

(continued from previous page)

```
{-# BUILTIN AGDATCMWITHRECONSTRUCTED          withReconstructed          #-}
{-# BUILTIN AGDATCMWITHEXPANDLAST             withExpandLast             #-}
{-# BUILTIN AGDATCMWITHREDUCEDEFS             withReduceDefs             #-}
{-# BUILTIN AGDATCMASKNORMALISATION           askNormalisation           #-}
{-# BUILTIN AGDATCMASKRECONSTRUCTED           askReconstructed           #-}
{-# BUILTIN AGDATCMASKEXPANDLAST              askExpandLast              #-}
{-# BUILTIN AGDATCMASKREDUCEDEFS              askReduceDefs              #-}
{-# BUILTIN AGDATCMDEBUGPRINT                 debugPrint                 #-}
{-# BUILTIN AGDATCMNOCONSTRAINTS              noConstraints              #-}
{-# BUILTIN AGDATCMWORKONTYPES                workOnTypes                #-}
{-# BUILTIN AGDATCMRUNSPECULATIVE             runSpeculative             #-}
{-# BUILTIN AGDATCMGETINSTANCES               getInstances               #-}
{-# BUILTIN AGDATCMSOLVEINSTANCES             solveInstanceConstraints   #-}
```

3.38.2 Metaprogramming

There are three ways to run a metaprogram (`TC` computation). To run a metaprogram in a term position you use a *macro*. To run metaprograms to create top-level definitions you can use the `unquoteDecl` and `unquoteDef` primitives (see *Unquoting Declarations*).

Macros

Macros are functions of type $\texttt{t}_1 \to \texttt{t}_2 \to \texttt{..} \to \texttt{Term} \to \texttt{TC}\ \top$ that are defined in a `macro` block. The last argument is supplied by the type checker and will be the representation of a metavariable that should be instantiated with the result of the macro.

Macro application is guided by the type of the macro, where `Term` and `Name` arguments are quoted before passed to the macro. Arguments of any other type are preserved as-is.

For example, the macro application `f u v w` where $\texttt{f : Term} \to \texttt{Name} \to \texttt{Bool} \to \texttt{Term} \to \texttt{TC}\ \top$ desugars into:

```
unquote (f (quoteTerm u) (quote v) w)
```

where `quoteTerm u` takes a `u` of arbitrary type and returns its representation in the `Term` data type, and `unquote m` runs a computation in the `TC` monad. Specifically, when checking $\texttt{unquote m : A}$ for some type `A` the type checker proceeds as follows:

- Check $\texttt{m : Term} \to \texttt{TC}\ \top$.
- Create a fresh metavariable $\texttt{hole : A}$.
- Let $\texttt{qhole : Term}$ be the quoted representation of `hole`.
- Execute `m qhole`.
- Return (the now hopefully instantiated) `hole`.

Reflected macro calls are constructed using the `def` constructor, so given a macro $\texttt{g : Term} \to \texttt{TC}\ \top$ the term `def (quote g) []` unquotes to a macro call to `g`.

> **Note**
>
> The `quoteTerm` and `unquote` primitives are available in the language, but it is recommended to avoid using them in favour of macros.

Limitations:

- Macros cannot be recursive. This can be worked around by defining the recursive function outside the macro block and have the macro call the recursive function.
- Macros and `quoteTerm` can get blocked on quoting a term with an interactive hole inside it. Enable `--quote-metas` to disable this blocking, so that quoted holes can be constrained by macro unquoting.

Silly example:

```
macro
    plus-to-times : Term → Term → TC ⊤
    plus-to-times (def (quote _+_) (a :: b :: [])) hole =
      unify hole (def (quote _*_) (a :: b :: []))
    plus-to-times v hole = unify hole v

thm : (a b : Nat) → plus-to-times (a + b) ≡ a * b
thm a b = refl
```

Macros lets you write tactics that can be applied without any syntactic overhead. For instance, suppose you have a solver:

```
magic : Type → Term
```

that takes a reflected goal and outputs a proof (when successful). You can then define the following macro:

```
macro
  by-magic : Term → TC ⊤
  by-magic hole =
    bindTC (inferType hole) λ goal →
    unify hole (magic goal)
```

This lets you apply the magic tactic as a normal function:

```
thm : ¬ P ≡ NP
thm = by-magic
```

Tactic Arguments

You can declare tactics to be used to solve a particular implicit argument using a `@(tactic t)` annotation. The provided tactic should be a term `t : Term → TC ⊤`. For instance,

```
defaultTo : {A : Set} (x : A) → Term → TC ⊤
defaultTo x hole = bindTC (quoteTC x) (unify hole)

f : {@(tactic defaultTo true) x : Bool} → Bool
f {x} = x

test-f : f ≡ true
test-f = refl
```

At calls to *f*, *defaultTo true* is called on the metavariable inserted for *x* if it is not given explicitly. The tactic can depend on previous arguments to the function. For instance,

```
g : (x : Nat) {@(tactic defaultTo x) y : Nat} → Nat
g x {y} = x + y
```

(continues on next page)

(continued from previous page)

```
test-g : g 4 ≡ 8
test-g = refl
```

Record fields can also be annotated with a tactic, allowing them to be omitted in constructor applications, record constructions and co-pattern matches:

```
record Bools : Set where
  constructor mkBools
  field fst : Bool
        @(tactic defaultTo fst) {snd} : Bool
open Bools

tt₀ tt₁ tt₂ tt₃ : Bools
tt₀ = mkBools true {true}
tt₁ = mkBools true
tt₂ = record{ fst = true }
tt₃ .fst = true

test-tt : tt₁ ∷ tt₂ ∷ tt₃ ∷ [] ≡ tt₀ ∷ tt₀ ∷ tt₀ ∷ []
test-tt = refl
```

Unquoting Declarations

While macros let you write metaprograms to create terms, it is also useful to be able to create top-level definitions. You can do this from a macro using the `declareDef`, `declareData`, `defineFun` and `defineData` primitives, but there is no way to bring such definitions into scope. For this purpose there are two top-level primitives `unquoteDecl` and `unquoteDef` that runs a `TC` computation in a declaration position. They both have the same form for declaring function definitions:

```
unquoteDecl x₁ .. xₙ = m
unquoteDef  x₁ .. xₙ = m
```

except that the list of names can be empty for `unquoteDecl`, but not for `unquoteDef`. In both cases `m` should have type `TC ⊤`. The main difference between the two is that `unquoteDecl` requires `m` to both declare (with `declareDef`) and define (with `defineFun`) the x_i whereas `unquoteDef` expects the x_i to be already declared. In other words, `unquoteDecl` brings the x_i into scope, but `unquoteDef` requires them to already be in scope.

In `m` the x_i stand for the names of the functions being defined (i.e. x_i `: Name`) rather than the actual functions.

One advantage of `unquoteDef` over `unquoteDecl` is that `unquoteDef` is allowed in mutual blocks, allowing mutually recursion between generated definitions and hand-written definitions.

Example usage:

```
arg′ : {A : Set} → Visibility → A → Arg A
arg′ v = arg (arg-info v (modality relevant quantity-ω))

-- Defining: id-name {A} x = x
defId : (id-name : Name) → TC ⊤
defId id-name = do
  defineFun id-name
    [ clause
      ( ("A" , arg′ visible (agda-sort (lit 0)))
```

(continues on next page)

(continued from previous page)

```
      :: ("x" , arg´ visible (var 0 []))
      :: [])
      ( arg´ hidden (var 1)
      :: arg´ visible (var 0)
      :: [] )
      (var 0 [])
    ]

id : {A : Set} (x : A) → A
unquoteDef id = defId id

mkId : (id-name : Name) → TC ⊤
mkId id-name = do
  ty ← quoteTC ({A : Set} (x : A) → A)
  declareDef (arg´ visible id-name) ty
  defId id-name

unquoteDecl id´ = mkId id´
```

Another form of `unquoteDecl` is used to declare data types:

```
unquoteDecl data x constructor c₁ .. cₙ = m
```

`m` is a metaprogram required to declare and define a data type `x` and its constructors c_1 to c_n using `declareData` and `defineData`.

> **Note**
>
> To debug code generated by `unquoteDecl` and `unquoteDef` it can be useful to turn on the verbosity flags `-v tc.unquote.decl:10` (for type signatures) and `-v tc.unquote.def:10` (for definition bodies). This will cause the generated code to be printed on stdout when running from the command line or in the debug buffer when loading from an editor. Unlike other verbosity flags, these two are available even if Agda has been built with the `debug` Cabal flag.

System Calls

It is possible to run system calls as part of a metaprogram, using the `execTC` builtin. You can use this feature to implement type providers, or to call external solvers. For instance, the following example calls `/bin/echo` from Agda:

```
postulate
  execTC : (exe : String) (args : List String) (stdIn : String)
         → TC (Σ Nat (λ _ → Σ String (λ _ → String)))

{-# BUILTIN AGDATCMEXEC execTC #-}

macro
  echo : List String → Term → TC ⊤
  echo args hole = do
    (exitCode , (stdOut , stdErr)) ← execTC "echo" args ""
    unify hole (lit (string stdOut))
```

(continues on next page)

(continued from previous page)

```
_ : echo ("hello" :: "world" :: []) ≡ "hello world\n"
_ = refl
```

The `execTC` builtin takes three arguments: the basename of the executable (e.g., `"echo"`), a list of arguments, and the contents of the standard input. It returns a triple, consisting of the exit code (as a natural number), the contents of the standard output, and the contents of the standard error.

It would be ill-advised to allow Agda to make arbitrary system calls. Hence, the feature must be activated by passing the *--allow-exec* option, either on the command-line or using a pragma. (Note that *--allow-exec* is incompatible with *--safe*.) Furthermore, Agda can only call executables which are listed in the list of trusted executables, `~/.agda/executables`. For instance, to run the example above, you must add `/bin/echo` to this file:

```
# contents of ~/.agda/executables
/bin/echo
```

The executable can then be called by passing its basename to `execTC`, subtracting the `.exe` on Windows.

Quote Metas

The option *--quote-metas* enables term quotation constraints to be resolved even when the term has meta variables in it. This allows for typed hole-driven development for macros, where the expected type for an interactive hole given as a quoted argument to a macro can be constrained by the unquoting of the macro.

In the following example without, *--quote-metas* , the interactive hole will cause the constraint to be blocked, which looks something like `quoteTerm ?0 :  Term (blocked on _17)`.

```
open import Agda.Builtin.Unit
open import Agda.Builtin.Nat
open import Agda.Builtin.List

repeat-helper : Nat → Name → Term → Term
repeat-helper zero f a = a
repeat-helper (suc n) f a = con f ((arg (arg-info visible (modality relevant quantity-
↪ω)) (repeat-helper n f a)) :: [])

macro
  repeat : Nat → Name → Term → Term → TC ⊤
  repeat n f a hole = unify hole (repeat-helper n f a)

_ : Nat
_ = repeat 10 suc {! !}
```

However, actually unquoting the macro would reveal that the interactive hole should have type $\mathbb{N}$. Enabling *--quote-metas* allows exactly this.

```
{-# OPTIONS --quote-metas #-}

-- ...

ex1 = repeat 10 f {! !}
```

Now, the macro will be unquoted, which in this spliced code applies `f` to the hole's value. This constrains hole's type to be $\mathbb{N}$, and the user can inspect the expected context and type of the hole to learn this.

Note

In this example, the hole's type is not constrained by the spliced code, so it will have an unknown inferred type.

```
ex2 = repeat 0 f {! !}
```

3.39 Rewriting

Rewrite rules allow you to extend Agda's evaluation relation with new computation rules.

Rules are safe to use with `Agda.Builtin.Equality` if *–confluence-check* is enabled. Confluent but non-terminating rewrite rules can not break consistency, unlike to non-terminating functions. Those results were proven by Cockx, Tabareau, and Winterhalter, see section 3 for statements.

Note

This page is about the *--rewriting* option and the associated *REWRITE* builtin. You might be looking for the documentation on the *rewrite construct* instead.

3.39.1 Rewrite rules by example

To enable rewrite rules, you should run Agda with the flag *--rewriting* and import the modules `Agda.Builtin.Equality` and `Agda.Builtin.Equality.Rewrite`:

```
{-# OPTIONS --rewriting #-}

module language.rewriting where

open import Agda.Builtin.Equality
open import Agda.Builtin.Equality.Rewrite
```

Overlapping pattern matching

To start, let's look at an example where rewrite rules can solve a problem that is encountered by almost every newcomer to Agda. This problem usually pops up as the question why `0 + m` computes to `m`, but `m + 0` does not (and similarly, `(suc m) + n` computes to `suc (m + n)` but `m + (suc n)` does not). This problem manifests itself for example when trying to prove commutativity of `_+_`:

```
+comm : m + n ≡ n + m
+comm {m = zero}  = refl
+comm {m = suc m} = cong suc (+comm {m = m})
```

Here, Agda complains that `n != n + zero of type Nat`. The usual way to solve this problem is by proving the equations `m + 0 ≡ m` and `m + (suc n) ≡ suc (m + n)` and using an explicit `rewrite` statement in the main proof (N.B.: Agda's `rewrite` keyword should not be confused with rewrite rules, which are added by a `REWRITE` pragma.)

By using rewrite rules, we can simulate the solution from our paper. First, we need to prove that the equations we want hold as propositional equalities:

```
+zero : m + zero ≡ m
+zero {m = zero}  = refl
```

(continues on next page)

(continued from previous page)

```
+zero {m = suc m} = cong suc +zero

+suc : m + (suc n) ≡ suc (m + n)
+suc {m = zero}  = refl
+suc {m = suc m} = cong suc +suc
```

Next we mark the equalities as rewrite rules with a `REWRITE` pragma:

```
{-# REWRITE +zero +suc #-}
```

Now the proof of commutativity works exactly as we wrote it before:

```
+comm : m + n ≡ n + m
+comm {m = zero}  = refl
+comm {m = suc m} = cong suc (+comm {m = m})
```

Note that there is no way to make this proof go through without rewrite rules: it is essential that `_+_` computes both on its first and its second argument, but there's no way to define `_+_` in such a way using Agda's regular pattern matching.

More examples

Additional examples of how to use rewrite rules can be found in a blog post by Jesper Cockx.

Controlling rewrite rule matching with `primRewriteNoMatch`

Rewrite rule matching does not reduce pattern-matching definitions. This can cause seemingly harmless rewrite rules to be non-confluent. For example, consider the rewrite rule for associativity of vector concatenation (which itself depends on associativity of addition).

```
_++_ : Vec A n → Vec A m → Vec A (n + m)
[]       ++ ys = ys
(x ∷ xs) ++ ys = x ∷ (xs ++ ys)

++-assoc : {xs : Vec A n} {ys : Vec A m} {zs : Vec A l}
         → (xs ++ ys) ++ zs ≡ xs ++ (ys ++ zs)
++-assoc {xs = []}     = refl
++-assoc {xs = x ∷ xs} = cong (x ∷_) (++-assoc {xs = xs})
```

Unfortunately, if we specialise the length of the first vector to `zero`, the `++-assoc` rewrite rule does not apply correctly.

```
{-# REWRITE ++-assoc #-}

++-assoc-fail : {xs : Vec A zero} {ys : Vec A m} {zs : Vec A l}
              → (xs ++ ys) ++ zs ≡ xs ++ (ys ++ zs)
++-assoc-fail = refl -- error: [UnequalTerms]
                     -- The terms
                     --   _++_ {n = m} (xs ++ ys) zs
                     -- and
                     --   _++_ {n = zero} xs (ys ++ zs)
                     -- are not equal at type Vec A (m + l)
```

The problem is that the outer `_++_` on `++-assoc`'s left-hand side takes an implicit length argument `{n = n + m}` but in our special-case, `n` is `zero` and so the implicit argument reduces to just `{n = m}`. Agda does not reduce pattern-matching definitions during rewrite rule matching, so it fails to match `m` against `n + m` and the rewrite does not apply.

The `primRewriteNoMatch` primitive (exported by `Agda.Builtin.Equality.Rewrite`) enables manually working around this limitation. Wrapping subterms of rewrite rule left-hand sides with the primitive tells Agda to not strictly match against those subterms (only check conversion after matching the rest of the rewrite rule LHS has succeeded). All variables which freely occur in `primRewriteNoMatch`-wrapped subterms must be bound somewhere else on the left-hand side.

If we wrap the implicit length argument to the outer `_++_` on the left-hand side of the associativity of vector concatenation rewrite rule with `primRewriteNoMatch`, then we find the rewrite rule works correctly.

```
++-assoc' : {xs : Vec A n} {ys : Vec A m} {zs : Vec A l}
          → _++_ {n = primRewriteNoMatch (n + m)} (xs ++ ys) zs
          ≡ xs ++ (ys ++ zs)
++-assoc' {xs = xs} = ++-assoc {xs = xs}

{-# REWRITE ++-assoc' #-}

++-assoc-succeed : {xs : Vec A zero} {ys : Vec A m} {zs : Vec A l}
                 → (xs ++ ys) ++ zs ≡ xs ++ (ys ++ zs)
++-assoc-succeed = refl
```

Definitional singletons and subject reduction

Some useful rewrite rules break subject reduction in the presence of definitional singletons (such as the unit type, $\top$).

For example, consider the following axiomatisation of the circle as a higher inductive type:

```
IdOver : (P : A → Set) → x ≡ y → P x → P y → Set
IdOver P refl x y = x ≡ y

syntax IdOver P p x y = x ≡[ P ↓ p ]≡ y

dcong : ∀ {B : A → Set} (f : (x : A) → B x) (p : x ≡ y) → f x ≡[ B ↓ p ]≡ f y
dcong f refl = refl

postulate
  Circle : Set
  base   : Circle
  loop   : base ≡ base

  circle-elim : ∀ (P : Circle → Set) (baseP : P base)
              → baseP ≡[ P ↓ loop ]≡ baseP
              → ∀ x → P x

  circle-elim-base : ∀ (P : Circle → Set) (baseP : P base)
                       (loopP : baseP ≡[ P ↓ loop ]≡ baseP)
                   → circle-elim P baseP loopP base ≡ baseP
  {-# REWRITE circle-elim-base #-}

  circle-elim-loop : ∀ (P : Circle → Set) (baseP : P base)
                       (loopP : baseP ≡[ P ↓ loop ]≡ baseP)
                   → dcong (circle-elim P baseP loopP) loop ≡ loopP
```

Ideally, we would also turn the computation rule for `circle-elim` applied to the path constructor `loop` (`circle-elim-loop`) into a rewrite rule:

```
{-# REWRITE circle-elim-loop #-}
```

Unfortunately, this causes subject reduction issues when eliminating into $\top$. Note that, by η-equality, we have `circle-elim (λ _ → ⊤) tt p = circle-elim (λ _ → ⊤) tt q` definitionally for any two identity proofs `p` and `q`.

```
open import Agda.Builtin.Unit

circle-elim-pq : ∀ {p q}
               → circle-elim (λ _ → ⊤) tt p
               ≡ circle-elim (λ _ → ⊤) tt q
circle-elim-pq = refl
```

The computation rule `circle-elim-loop` should therefore also imply `p ≡ q` definitionally, but the inductive identity type does not satisfy such an η rule.

Agda tries to detect such problematic rewrite rules and will throw a `RewriteVariablesBoundInSingleton` warning if it is not certain that the rewrite is safe. If you are confident your rewrite rule is actually safe or you know you are not using definitional singletons, or you are simply not worried about weirdly behaving definitional equality, you can silence the warning with `-WnoRewriteVariablesBoundInSingleton` and continue to rely on the rewrite rule.

3.39.2 General shape of rewrite rules

In general, an equality proof `eq` may be registered as a rewrite rule using the pragma `{-# REWRITE eq #-}`, provided the following requirements are met:

- The type of `eq` is of the form `eq : (x₁ : A₁) ... (xₖ : Aₖ) → f p₁ ... pₙ ≡ v`
- `f` is a postulate, a defined function symbol, or a constructor applied to fully general parameters (i.e. the parameters must be distinct variables)
- Each variable $\mathtt{x}_1, \ldots, \mathtt{x}_k$ occurs at least once in a pattern position in `p₁ ... pₙ` (see below for the definition of pattern positions)
- The left-hand side `f p₁ ... pₙ` should be neutral, i.e. it should not reduce.

The following patterns are supported:

- `x y₁ ... yₙ`, where `x` is a pattern variable and $\mathtt{y}_1, \ldots, \mathtt{y}_n$ are distinct variables that are bound locally in the pattern
- `f p₁ ... pₙ`, where `f` is a postulate, a defined function, a constructor, or a data/record type, and $\mathtt{p}_1, \ldots, \mathtt{p}_n$ are again patterns
- `λ x → p`, where `p` is again a pattern
- `(x : P) → Q`, where `P` and `Q` are again patterns
- `y p₁ ... pₙ`, where `y` is a variable bound locally in the pattern and $\mathtt{p}_1, \ldots, \mathtt{p}_n$ are again patterns
- `Set p` or `Prop p`, where `p` is again a pattern
- Any other term `v` (here the variables in `v` are not considered to be in a pattern position)

Once a rewrite rule has been added, Agda automatically rewrites all instances of the left-hand side to the corresponding instance of the right-hand side during reduction. More precisely, a term (definitionally equal to) $\mathtt{f}\ \mathtt{p}_1\sigma\ \ldots\ \mathtt{p}_n\sigma$ is rewritten to $\mathtt{v}\sigma$, where σ is any substitution on the pattern variables $\mathtt{x}_1, \ldots\ \mathtt{x}_k$.

Since rewriting happens after normal reduction, rewrite rules are only applied to terms that would otherwise be neutral.

3.39.3 Confluence checking

Agda can optionally check confluence of rewrite rules by enabling the `--confluence-check` flag. Concretely, it does so by enforcing two properties:

1. For any two left-hand sides of the rewrite rules that overlap (either at the root position or at a subterm), the most general unifier of the two left-hand sides is again a left-hand side of a rewrite rule. For example, if there are two rules `suc m + n = suc (m + n)` and `m + suc n = suc (m + n)`, then there should also be a rule `suc m + suc n = suc (suc (m + n))`.
2. Each rewrite rule should satisfy the *triangle property*: For any rewrite rule `u = w` and any single-step parallel unfolding `u => v`, we should have another single-step parallel unfolding `v => w`.

There is also a flag `--local-confluence-check` that is less restrictive but only checks local confluence of rewrite rules. In case the rewrite rules are terminating (currently not checked), these two properties are equivalent.

3.39.4 Advanced usage

Instead of importing `Agda.Builtin.Equality.Rewrite`, a different type may be chosen as the rewrite relation by registering it as the `REWRITE` builtin. For example, using the pragma `{-# BUILTIN REWRITE _~_ #-}` registers the type `_~_` as the rewrite relation. To qualify as the rewrite relation, the type must take at least two arguments, and the final two arguments should be visible.

3.39.5 Importing rewrite rules

With `import M`, all of `M`'s rewrite rules get imported no matter whether they are `private` or in submodules of `M`. Further, all the rewrite rules that `M` imports will also get imported, transitively. Thus, say module `M0` declares or imports some rewrite rules, and `M1` imports `M0`, then any module `M2` importing `M1` will import these rewrite rules even if `M2` does not directly import `M0`.

The reason for this transitive import behavior of rewrite rules is to ensure the subject reduction (aka type preservation) property. Say `M0` exports a type `A` and a rule that rewrites `A` to `Nat`, then `M1` can declare a constant `a = zero` of type `A`. If `M2` could import `A` and `a` from `M1` but not the rewrite rule for `A`, then in the context of `M2` the reduction from `a` to `zero` would change its type from `A` to `Nat` which is not equal to `A` in absence of the rewrite rule. Thus, type preservation under reduction would fail.

In summary, the scoping rules for rewrite rules are same as the scoping rules for instances in Haskell 2010.

3.40 Run-time Irrelevance

From version 2.6.1 Agda supports run-time irrelevance (or erasure) annotations. Values marked as erased are not present at run time, and consequently the type checker enforces that no computations depend on erased values.

3.40.1 Syntax

A function or constructor argument is declared erased using the `@0` or `@erased` annotation. (These annotations may only be used if the option `--erasure` is active.) For example, the following definition of vectors guarantees that the length argument to `_::_` is not present at runtime:

```
data Vec (A : Set a) : @0 Nat → Set a where
  []  : Vec A 0
  _::_ : ∀ {@0 n} → A → Vec A n → Vec A (suc n)
```

The *GHC backend* compiles this to a datatype where the cons constructor takes only two arguments.

Note

In this particular case, the compiler identifies that the length argument can be erased also without the annotation, using Brady et al's forcing analysis *[1]*. Marking it erased explictly, however, ensures that it is erased without relying on the analysis.

Note

If *--erasure* is used, then parameters are marked as erased in the type signatures of constructors and record fields, even if the parameters are not marked as erased in the data or record type's telescope, with one exception: for indexed data types this only happens if the *--with-K* flag is active.

Erasure annotations can also appear in function arguments (both first-order and higher-order). For instance, here is an implementation of `foldl` on vectors:

```
foldl : (B : @0 Nat → Set b)
      → (f : ∀ {@0 n} → B n → A → B (suc n))
      → (z : B 0)
      → ∀ {@0 n} → Vec A n → B n
foldl B f z []       = z
foldl B f z (x ∷ xs) = foldl (λ n → B (suc n)) (λ {n} → f {suc n}) (f z x) xs
```

Here the length arguments to `foldl` and to `f` have been marked erased. As a result it gets compiled to the following Haskell code (modulo renaming):

```
foldl f z xs
  = case xs of
      []     -> z
      x ∷ xs -> foldl f (f _ z x) xs
```

In contrast to constructor arguments, erased arguments to higher-order functions are not removed completely, but instead replaced by a placeholder value `_`. The crucial optimization enabled by the erasure annotation is compiling `λ {n} → f {suc n}` to simply `f`, removing a terrible space leak from the program. Compare to the result of compiling without erasure:

```
foldl f z xs
  = case xs of
      []     -> z
      x ∷ xs -> foldl (\ n -> f (1 + n)) (f 0 z x) xs
```

Erased definitions and fields

It is also possible to mark top-level function definitions as erased. This guarantees that they are only used in erased arguments and can be useful to ensure that code intended only for compile-time evaluation is not executed at run time. (One can also use erased things in the bodies of erased definitions.) For instance,

```
@0 spec : Nat → Nat   -- slow, but easy to verify
impl    : Nat → Nat   -- fast, but hard to understand
proof   : ∀ n → spec n ≡ impl n
```

Erased record fields become erased arguments to the record constructor and the projection functions are treated as erased definitions.

Erased constructors

Constructors can also be marked as erased:

```
data D : Set where
  always : D
  @0 compile-time-only : D

caseD : A → @0 A → D → A
caseD a c always            = a
caseD a c compile-time-only = c
```

In the code above the constructor `compile-time-only` is only available at compile-time, whereas `always` is also available at run-time. Clauses that match on erased constructors in non-erased positions are omitted by (at least some) compiler backends, so one can use erased names in the bodies of such clauses. (There is an exception for constructors that were not originally declared as erased, but that are currently treated as erased.)

A more meaningful example for erased constructors involves higher inductive types, see *Erased constructors*.

Erased data types

One can also mark data and record types as erased. Such types can only be used in erased positions, their constructors and projections are erased, and definitions in record modules for erased record types are erased. A data or record type is marked as erased by writing `@0` or `@erased` right after the `data` or `record` keyword of the data or record type's declaration:

```
data @0 D₁ : Set where
  c : D₁

data @0 D₂ : Set

data D₂ where
  c : D₁ → D₂

interleaved mutual

  data @0 D₃ : Set where

  data D₃ where
    c : D₃

record @0 R₁ : Set where
  field
    x : D₁

record @0 R₂ : Set

record R₂ where
  field
    x : R₁
```

Erased modules

Finally one can mark modules as erased. The module identifier itself does not become erased, but all definitions inside the module. A module is marked as erased by writing `@0` or `@erased` right after the `module` keyword:

```
module @0 _ where

  F : @0 Set → Set
  F A = A

module M (@0 A : Set) where

  record R : Set where
    field
      @0 x : A

module @0 N (@0 A : Set) = M A

G : (@0 A : Set) → let module @0 M₂ = M A in Set
G A = M.R C
  module @0 _ where
    C : Set
    C = A
```

Lambda abstraction

Lambda-bound variables can be annotated with erasure status `@0` and `@ω`. If the type of the lambda expression is known, such annotations are superfluous, they are inherited from the type in this case:

```
checkedLambda : _ → _
checkedLambda = λ x → x + x

const : {A B : Set} → A → @0 B → A
const = λ x y → x
```

If the type is unknown and shall be inferred by Agda, the annotation is mandatory. For instance, the following definition is not accepted by Agda:

```
inferredLambda = λ x → x + x
```

It is accepted if you communicate that `x` is not erased:

```
inferredLambda = λ (@ω x) → x + x
```

You have to annotate all lambda-bound variables with their erasure status:

```
Const = λ (@ω A : Set) (@0 B : Set) → A
```

Note, however, that with *`--cubical`* lambdas are in general not inferred.

Pattern lambdas

Regular pattern lambdas are treated as non-erased function definitions. One can make a pattern lambda erased by writing `@0` or `@erased` before the lambda:

```
@0 _ : @0 Set → Set
_ = λ @0 { A → A }

@0 _ : @0 Set → Set
_ = λ @erased where
  A → A
```

3.40.2 Rules

The typing rules are based on Conor McBride's "I Got Plenty o'Nuttin'" *[2]* and Bob Atkey's "The Syntax and Semantics of Quantitative Type Theory" *[3]*. In essence the type checker keeps track of whether it is running in *run-time mode*, checking something that is needed at run time, or *compile-time mode*, checking something that will be erased. In compile-time mode everything to do with erasure can safely be ignored, but in run-time mode the following restrictions apply:

- Cannot use erased variables or definitions.
- Cannot pattern match on erased arguments, unless there is at most one valid case. If *--without-K* is enabled and there is one valid case, then there are further restrictions:
  - The constructor's data or record type must not be indexed.
  - If the type is anything but a record type with η-equality, then the option *--erased-matches* must be enabled.

Consider the function `foo` taking an erased vector argument:

```
foo : (n : Nat) (@0 xs : Vec Nat n) → Nat
foo zero    []       = 0
foo (suc n) (x ∷ xs) = foo n xs
```

This is okay (when the K rule is on), since after matching on the length, the matching on the vector does not provide any computational information, and any variables in the pattern (`x` and `xs` in this case) are marked erased in turn. On the other hand, if we don't match on the length first, the type checker complains:

```
foo : (n : Nat) (@0 xs : Vec Nat n) → Nat
foo n []       = 0
foo n (x ∷ xs) = foo _ xs
-- Error: Cannot branch on erased argument of datatype Vec Nat n
```

The type checker enters compile-time mode when

- checking erased arguments to a constructor, function or module application,
- checking the body of an erased definition (including an erased module application),
- checking the body of a clause that matches (in a non-erased position) on a constructor that was originally defined as erased (it does not suffice for the constructor to be currently treated as erased),
- checking the domain of an erased Π type, or
- checking a type, i.e. when moving to the right of a `:`, with some exceptions:
  - Compile-time mode is not entered for the domains of non-erased Π types.
  - If the K rule is off then compile-time mode is not entered for non-erased constructors (of fibrant type) or record fields.

Note that the type checker does not enter compile-time mode based on the type a term is checked against (except that a distinction is sometimes made between fibrant and non-fibrant types). In particular, checking a term against `Set` does not trigger compile-time mode.

There is also a *hard compile-time mode*. In this mode all definitions are treated as erased. The hard compile-time mode is entered when an erased definition is checked.

Unnamed and named *where* modules in erased context are always checked in hard compile-time mode.

The type-checker switches from compile-time mode to run-time mode for certain expressions/declarations if it is not in the hard compile-time mode:

- Absurd lambdas.
- Non-erased pattern lambdas.
- Non-erased module definitions ("`module M ... = ...`") or applications ("`M ...`").
- Applications of ♯ (see *Old Coinduction*).

The reflection API provides a primitive function $workOnTypes : TC\ A \to TC\ A$ that manually switches the type-checker from run-time mode to compile-time mode.

3.40.3 References

[1] Brady, Edwin, Conor McBride, and James McKinna. "Inductive Families Need Not Store Their Indices." International Workshop on Types for Proofs and Programs. Springer, Berlin, Heidelberg, 2003.

[2] McBride, Conor. "I Got Plenty o'Nuttin'." A List of Successes That Can Change the World. Springer, Cham, 2016.

[3] Atkey, Robert. "The Syntax and Semantics of Quantitative Type Theory". In LICS '18: Oxford, United Kingdom. 2018.

3.41 Safe Agda

By using the option `--safe` (as a pragma option, or on the command-line), a user can specify that Agda should ensure that features leading to possible inconsistencies should be disabled.

Here is a list of the features `--safe` is incompatible with:

- `postulate`; can be used to assume any axiom.
- `--allow-unsolved-metas`; forces Agda to accept unfinished proofs.
- `--allow-incomplete-matches` and pragma *NON_COVERING*; allows to prove false using a partial function or through a partial proof.
- `--no-positivity-check` and pragmas *NO_POSITIVITY_CHECK* and *POLARITY*; make it possible to write non-terminating programs via datatypes that are not strictly positive.
- `--no-termination-check` and pragmas *TERMINATING* and *NON_TERMINATING*; give loopy programs any type.
- `--type-in-type` and `--omega-in-omega` and pragma *NO_UNIVERSE_CHECK*; allow the user to encode the Girard-Hurken paradox.
- pragma *INJECTIVE*; allows to prove false by declaring a non-injective function as injective.
- `--injective-type-constructors`; together with excluded middle leads to an inconsistency via Chung-Kil Hur's construction.
- `--sized-types`; lacks some checks that rule out improper, inconsistent uses of sizes.

- `--experimental-irrelevance` and `--irrelevant-projections`; enables potentially unsound irrelevance features (irrelevant levels, irrelevant data matching, and projection of irrelevant record fields, respectively).
- `--rewriting`; turns any equation into one that holds definitionally. It can at the very least break convergence.
- `--cubical=compatible` together with `--with-K`; the univalence axiom is provable using cubical constructions, which falsifies the K axiom.
- `--without-K` together with `--flat-split`
- The `primEraseEquality` primitive together with `--without-K`; using `primEraseEquality`, one can derive the K axiom.
- `--allow-exec`; allows system calls during type checking.
- `--no-load-primitives`; allows the user to bind the sort and level primitives manually.
- `--cumulativity`; due to its poor heuristic for solving universe levels.
- `--large-indices` together with `--without-K` or `--forced-argument-recursion`; both of these combinations are known to be inconsistent.
- pragma *COMPILE*; allows to change the meaning of code during compilation.

The option `--safe` is coinfective (see *Checking options for consistency*); if a module is declared safe, then all its imported modules must also be declared safe.

3.42 Sized Types

> **Note**
>
> This is a stub.

Sizes help the termination checker by tracking the depth of data structures across definition boundaries.

The built-in combinators for sizes are described in *Sized types*.

3.42.1 Example for coinduction: finite languages

See *Abel 2017* and *Traytel 2017*.

Decidable languages can be represented as infinite trees. Each node has as many children as the number of characters in the alphabet `A`. Each path from the root of the tree to a node determines a possible word in the language. Each node has a boolean label, which is `true` if and only if the word corresponding to that node is in the language. In particular, the root node of the tree is labelled `true` if and only if the word ϵ belongs to the language.

These infinite trees can be represented as the following coinductive data-type:

```
record Lang (i : Size) (A : Set) : Set where
  coinductive
  field
    ν : Bool
    δ : ∀{j : Size< i} → A → Lang j A

open Lang
```

As we said before, given a language `a : Lang A`, ν `a` $\equiv$ `true` iff $\epsilon \in$ `a`. On the other hand, the language δ `a x : Lang A` is the Brzozowski derivative of `a` with respect to the character `x`, that is, `w` $\in \delta$ `a x` iff `xw` $\in$ `a`.

With this data type, we can define some regular languages. The first one, the empty language, contains no words; so all the nodes are labelled `false`:

```
∅ : ∀ {i A}  → Lang i A
ν ∅   = false
δ ∅ _ = ∅
```

The second one is the language containing a single word; the empty word. The root node is labelled `true`, and all the others are labelled `false`:

```
ϵ : ∀ {i A} → Lang i A
ν ϵ   = true
δ ϵ _ = ∅
```

To compute the union (or sum) of two languages, we do a point-wise `or` operation on the labels of their nodes:

```
_+_ : ∀ {i A} → Lang i A → Lang i A → Lang i A
ν (a + b)   = ν a   ∨ ν b
δ (a + b) x = δ a x + δ b x

infixl 10 _+_
```

Now, lets define concatenation. The base case (ν) is straightforward: $\epsilon \in$ `a · b` iff $\epsilon \in$ `a` and $\epsilon \in$ `b`.

For the derivative (δ), assume that we have a word `w`, `w` $\in \delta$ `(a · b) x`. This means that `xw` $= \alpha\beta$, with $\alpha \in$ `a` and $\beta \in$ `b`.

We have to consider two cases:

1. $\epsilon \in$ `a`. Then, either:
   - $\alpha =$ `x`α`'`, with α`'` $\in \delta$ `a x`, and `w` $= \alpha$`'`$\beta \in \delta$ `a x · b`.
2. $\epsilon \notin$ `a`. Then, only the second case above is possible:
   - $\alpha =$ `x`α`'`, with α`'` $\in \delta$ `a x`, and `w` $= \alpha$`'`$\beta \in \delta$ `a x · b`.

```
_·_ : ∀ {i A} → Lang i A → Lang i A → Lang i A
ν (a · b)   = ν a ∧ ν b
δ (a · b) x = if ν a then δ a x · b + δ b x else δ a x · b

infixl 20 _·_
```

Here is where sized types really shine. Without sized types, the termination checker would not be able to recognize that `_+_` or `if_then_else` are not inspecting the tree, which could render the definition non-productive. By contrast, with sized types, we know that the `a + b` is defined to the same depth as `a` and `b` are.

In a similar spirit, we can define the Kleene star:

```
_* : ∀ {i A} → Lang i A → Lang i A
ν (a *)   = true
δ (a *) x = δ a x · a *

infixl 30 _*
```

Again, because the types tell us that `_·_` preserves the size of its inputs, we can have the recursive call to `a *` under a function call to `_·_`.

Testing

First, we want to give a precise notion of membership in a language. We consider a word as a `List` of characters.

```
_∈_ : ∀ {i} {A} → List i A → Lang i A → Bool
[]       ∈ a = ν a
(x :: w) ∈ a = w ∈ δ a x
```

Note how the size of the word we test for membership cannot be larger than the depth to which the language tree is defined.

If we want to use regular, non-sized lists, we need to ask for the language to have size ∞.

```
_∈_ : ∀ {A} → List A → Lang ∞ A → Bool
[]       ∈ a = ν a
(x :: w) ∈ a = w ∈ δ a x
```

Intuitively, ∞ is a `Size` larger than the size of any term than one could possibly define in Agda.

Now, let's consider binary strings as words. First, we define the languages ⟦ x ⟧ containing the single word "x" of length 1, for alphabet `A = Bool`:

```
⟦_⟧ : ∀ {i} → Bool → Lang i Bool
ν ⟦ _     ⟧       = false

δ ⟦ false ⟧ false = ϵ
δ ⟦ true  ⟧ true  = ϵ
δ ⟦ false ⟧ true  = ∅
δ ⟦ true  ⟧ false = ∅
```

Now we can define the bip-bop language, consisting of strings of even length alternating letters "true" and "false".

```
bip-bop = (⟦ true ⟧ · ⟦ false ⟧)*
```

Let's test a few words for membership in the language `bip-bop`!

```
test₁ : (true :: false :: true :: false :: true :: false :: []) ∈ bip-bop ≡ true
test₁ = refl

test₂ : (true :: false :: true :: false :: true :: []) ∈ bip-bop ≡ false
test₂ = refl

test₃ : (true :: true :: false :: []) ∈ bip-bop ≡ false
test₃ = refl
```

3.42.2 References

Equational Reasoning about Formal Languages in Coalgebraic Style, Andreas Abel.

Formal Languages, Formally and Coinductively, Dmitriy Traytel, LMCS Vol. 13(3:28)2017, pp. 1–22 (2017).

3.43 Sort System

Sorts (also known as universes) are types whose members themselves are again types. The fundamental sort in Agda is named `Set` and it denotes the universe of small types. But for some applications, other sorts are needed. This page explains the need for additional sorts and describes all the sorts that are used by Agda.

The theoretical foundation for Agda's sort system are *Pure Type Systems* (PTS). A PTS has, besides the set of supported sorts, two parameters:

1. A set of *axioms* of the form `s : s´`, stating that sort `s` itself has sort `s´`.
2. A set of *rules* of the form (s_1, s_2, s_3) stating that if $A : s_1$ and $B(x) : s_2$ then $(x : A) \to B(x) : s_3$.

Agda is a *functional* PTS in the sense that s_3 is uniquely determined by s_1 and s_2. Axioms are implemented internally by the `univSort` function, see *univSort*. Rules are implemented by the `funSort` and `piSort` functions, see *funSort*.

3.43.1 Introduction to universes

Russell's paradox implies that the collection of all sets is not itself a set. Namely, if there were such a set `U`, then one could form the subset $A \subseteq U$ of all sets that do not contain themselves. Then we would have $A \in A$ if and only if $A \notin A$, a contradiction.

Likewise, Martin-Löf's type theory had originally a rule `Set : Set` but Girard showed that it is inconsistent. This result is known as Girard's paradox. Hence, not every Agda type is a `Set`. For example, we have

```
Bool : Set
Nat  : Set
```

but not `Set : Set`. However, it is often convenient for `Set` to have a type of its own, and so in Agda, it is given the type $\texttt{Set}_1$:

```
Set : Set₁
```

In many ways, expressions of type $\texttt{Set}_1$ behave just like expressions of type `Set`; for example, they can be used as types of other things. However, the elements of $\texttt{Set}_1$ are potentially *larger*; when $A : \texttt{Set}_1$, then `A` is sometimes called a **large set**. In turn, we have

```
Set₁ : Set₂
Set₂ : Set₃
```

and so on. A type whose elements are types is called a **sort** or a **universe**; Agda provides an infinite number of universes `Set`, $\texttt{Set}_1$, $\texttt{Set}_2$, $\texttt{Set}_3$, ..., each of which is an element of the next one. In fact, `Set` itself is just an abbreviation for $\texttt{Set}_0$. The subscript `n` is called the **level** of the universe $\texttt{Set}_n$.

> **Note**
>
> You can also write `Set1`, `Set2`, etc., instead of $\texttt{Set}_1$, $\texttt{Set}_2$. To enter a subscript in the Emacs mode, type "_1".

Universe example

So why are universes useful? Because sometimes it is necessary to define and prove theorems about functions that operate not just on sets but on large sets. In fact, most Agda users sooner or later experience an error message where Agda complains that $\texttt{Set}_1$ `!= Set`. These errors usually mean that a small set was used where a large one was expected, or vice versa.

For example, suppose you have defined the usual datatypes for lists and cartesian products:

```
data List (A : Set) : Set where
  [] : List A
  _::_ : A → List A → List A

data _×_ (A B : Set) : Set where
 _,_ : A → B → A × B

infixr 5 _::_
infixr 4 _,_
infixr 2 _×_
```

Now suppose you would like to define an operator `Prod` that inputs a list of `n` sets and outputs their cartesian product, like this:

```
Prod (A :: B :: C :: []) = A × B × C
```

There is only one small problem with this definition. The type of `Prod` should be

```
Prod : List Set → Set
```

However, the definition of `List A` specified that `A` had to be a `Set`. Therefore, `List Set` is not a valid type. The solution is to define a special version of the `List` operator that works for large sets:

```
data List₁ (A : Set₁) : Set₁ where
  []   : List₁ A
  _::_ : A → List₁ A → List₁ A
```

With this, we can indeed define:

```
Prod : List₁ Set → Set
Prod []        = ⊤
Prod (A :: As) = A × Prod As
```

Universe polymorphism

To allow definitions of functions and datatypes that work for all possible universes $\texttt{Set}_i$, Agda provides a type `Level` of universe levels and level-polymorphic universes `Set ℓ` where `ℓ : Level`. For more information, see the page on *universe levels*.

3.43.2 Agda's sort system

The implementation of Agda's sort system is based on the theory of pure type systems. The full sort system of Agda consists of the following sorts:

1. Standard small sorts (universe-polymorphic).
   - $\texttt{Set}_i$ and its universe-polymorphic variant `Set ℓ`
   - $\texttt{Prop}_i$ and its universe-polymorphic variant `Prop ℓ` (with *--prop*)
   - $\texttt{SSet}_i$ and its universe-polymorphic variant `SSet ℓ` (with *--two-level*)
2. Standard large sorts (non polymorphic).
   - $\texttt{Set}\omega_i$
   - $\texttt{Prop}\omega_i$ (with *--prop*)

- SSetω_i (with *--two-level*)

3. Special sorts.
   - SizeUniv (with *--sized-types*)
   - IUniv, short for *interval universe* (with *--cubical*)
   - primLockUniv (with *--guarded*)
   - LevelUniv (with *--level-universe*)

Only the small standard sort hierarchies Set and Prop are in scope by default (see *--import-sorts*). They and most other sorts are defined in the system module Agda.Primitive. Sorts, even though they might enjoy the priviledge of numeric suffixes, are brought into scope just as any Agda definition, by open Agda.Primitive. Note that sorts can also be renamed, e.g., you might want to open Agda.Primitive renaming (Set to Type).

Some special sorts are defined in other system modules, see *Special sorts*.

Sorts Set$_i$ and Set ℓ

As explained in the introduction, Agda has a hierarchy of sorts Set$_i$: Set$_{i+1}$, where i is any concrete natural number, i.e. 0, 1, 2, 3, ... The sort Set is an abbreviation for Set$_0$.

You can also refer to these sorts with the alternative syntax Seti. That means that you can also write Set0, Set1, Set2, etc., instead of Set$_0$, Set$_1$, Set$_2$.

In addition, Agda supports the universe-polymorphic version Set ℓ where ℓ : Level (see *universe levels*).

Sorts Prop$_i$ and Prop ℓ

In addition to the hierarchy Set$_i$, Agda also supports a second hierarchy Prop$_i$: Set$_{i+1}$ (or Propi) of *proof-irrelevant propositions*. Like Set, Prop also has a universe-polymorphic version Prop ℓ where ℓ : Level.

Sorts SSet$_i$ and SSet ℓ

These experimental universes SSet$_0$: SSet$_1$: SSet$_2$: ... of *strict sets* or non-fibrant sets are described in *Two-Level Type Theory*.

Sorts Setω_i

To assign a sort to types such as (ℓ : Level) → Set ℓ, Agda further supports an additional sort Setω that stands above all sorts Set$_i$.

Just as for Set and Prop, Setω is the lowest level at an infinite hierarchy Setω_i : Setω_{i+1} where Setω = Setω_0. You can also refer to these sorts with the alternative syntax Setωi. That means that you can also write Setω0, Setω1, Setω2, etc., instead of Setω_0, Setω_1, Setω_2.

However, unlike the standard hierarchy of universes Set$_i$, the second hierarchy Setω_i does not support universe polymorphism. This means that it is not possible to quantify over *all* Setω_i at once. For example, the expression ∀ {i} (A : Setω i) → A → A would not be a well-formed agda term. See the section on Setω on the page on *universe levels* for more information.

Concerning other applications, it should not be necessary to refer to these sorts during normal usage of Agda, but they might be useful for defining *reflection-based macros*. And it is allowed to define data types in Setω_i.

Note

When *--omega-in-omega* is enabled, Setω_i is considered to be equal to Setω for all i (thus rendering Agda inconsistent).

Sorts `Prop`ω_i

This transfinite extension of the `Prop` hierarchy works analogous to `Set`ω_i. However, it is not motivated by typing `(ℓ : Level) → Prop ℓ`, because that lives in `Set`ω. Instead, it may be used to host large inductive propositions, where constructors can have fields that live at any finite level `ℓ`.

The sorting rules for finite levels extend to the transfinite hierarchy, so we have `Prop`ω_i `:` `Set`ω_{i+1}.

Sorts `SSet`ω_i

This is a transfinite extension of the `SSet` hierarchy.

Special sorts

Special sorts host special types that are not placed in a standard universe for technical reasons, typically because they require special laws for function type formation (see *funSort*).

With *--sized-types* and `open import Agda.Builtin.Size` we have `SizeUniv` which hosts the special type `Size` and the special family `Size<`.

With *--cubical* and `open import Agda.Primitive.Cubical` we get `IUniv` which hosts the interval `I`.

With *--guarded* we can define `primitive primLockUniv : Set`$_1$ in which we can postulate the `Tick` type.

With *--level-universe* the type `Level` no longer lives in `Set` but in its own sort `LevelUniv`. It is still defined in `Agda.Primitive`.

3.43.3 Sort metavariables and unknown sorts

Under universe polymorphism, levels can be arbitrary terms, e.g., a level that contains free variables. Sometimes, we will have to check that some expression has a valid type without knowing what sort it has. For this reason, Agda's internal representation of sorts implements a constructor (sort metavariable) representing an unknown sort. The constraint solver can compute these sort metavariables, just like it does when computing regular term metavariables.

However, the presence of sort metavariables also means that sorts of other types can sometimes not be computed directly. For this reason, Agda's internal representation of sorts includes three additional constructors `univSort`, `funSort`, and `piSort`. These constructors compute to the proper sort once enough metavariables in their arguments have been solved.

Note

`univSort`, `funSort` and `piSort` are *internal* constructors that may be printed when evaluating a term. The user cannot enter them, nor introduce them in Agda code. All these constructors do not represent new sorts but instead, they compute to the right sort once their arguments are known.

univSort

`univSort` returns the successor sort of a given sort. In PTS terminology, it implements the *axioms* `s : univSort s`.

Table 1: `univSort`

sort	successor sort
`Prop a`	`Prop (lsuc a)`
`Set a`	`Set (lsuc a)`
`SSet a`	`SSet (lsuc a)`
`Prop`ω_i	`Prop`ω_{i+1}
`Set`ω_i	`Set`ω_{i+1}
`SSet`ω_i	`SSet`ω_{i+1}
`SizeUniv`	`Set`ω
`IUniv`	$\texttt{SSet}_1$
`LockUniv`	$\texttt{Set}_1$
`LevelUniv`	$\texttt{Set}_1$
`_1`	`univSort _1`

funSort

The constructor `funSort` computes the sort of a function type even if the sort of the domain and the sort of the codomain are still unknown.

To understand how `funSort` works in general, let us assume the following scenario:

- `sA` and `sB` are two (possibly different) sorts.
- `A : sA`, meaning that `A` is a type that has sort `sA`.
- `B : sB`, meaning that `B` is a (possibly different) type that has sort `sB`.

Under these conditions, we can build the function type `A → B : funSort sA sB`. This type signature means that the function type `A → B` has a (possibly unknown) but well-defined sort `funSort sA sB`, specified in terms of the sorts of its domain and codomain.

Example: the sort of the function type `∀ {A} → A → A` with normal form `{A : _5} → A → A` evaluates to `funSort (univSort _5) (funSort _5 _5)` where:

- `_5` is a metavariable that represents the sort of `A`.
- `funSort _5 _5` is the sort of `A → A`.

If `sA` and `sB` happen to be known, then `funSort sA sB` can be computed to a sort value.

To specify how `funSort` computes, let `U` range over `Prop`, `Set`, `SSet` and let `U ⇝ U'` be `SSet` if one of `U`, `U'` is `SSet`, and `U'` otherwise. E.g. `SSet ⇝ Prop` is `SSet` and `Set ⇝ Prop` is `Prop`. Also, let `L` range over levels `a` and transfinite numbers ω_i (which is ω `+ i`) and let us generalize $\sqcup$ to `L ⊔ L'`, e.g. `a ⊔` ω_i `=` ω_i and $\omega_i \sqcup \omega_j = \omega_k$ where `k = max i j`. We write standard universes as pairs `U L`, e.g. `Prop`ω_i as pair `Prop` ω_i. Let `S` range over special universes `SizeUniv`, `IUniv`, `LockUniv`, `LevelUniv`.

In the following table we specify how `funSort` $\texttt{s}_1$ $\texttt{s}_2$ computes on known sorts $\texttt{s}_1$ and $\texttt{s}_2$, excluding interactions between different special sorts. In PTS terminology, these are the *rules* ($\texttt{s}_1$, $\texttt{s}_2$, `funSort` $\texttt{s}_1$ $\texttt{s}_2$).

Table 2: `funSort`

s_1	s_2	`funSort` s_1 s_2
`U L`	`U' L'`	`(U ⇝ U') (L ⊔ L')`
`U L`	`IUniv`	`SSet L`
`U` ω_i	`S` $\neq$ `IUniv`	`Set` ω_i
`U a`	`SizeUniv`	`SizeUniv`
`S`	`U` ω_i	`U` ω_i
`S` $\neq$ `LevelUniv`	`U a`	`U a`
`LevelUniv`	`U a`	`U` ω_0
`LevelUniv`	`LevelUniv`	`LevelUniv`
`SizeUniv`	`SizeUniv`	`SizeUniv`
`IUniv`	`IUniv`	$\mathtt{SSet}_0$

Here are some examples for the standard universes `U L`:

```
funSort Setω_i    Setω_j    = Setω_k          (where k = max(i,j))
funSort Setω_i    (Set b)   = Setω_i
funSort Setω_i    (Prop b)  = Setω_i
funSort (Set a)   Setω_j    = Setω_j
funSort (Prop a)  Setω_j    = Setω_j
funSort (Set a)   (Set b)   = Set (a ⊔ b)
funSort (Prop a)  (Set b)   = Set (a ⊔ b)
funSort (Set a)   (Prop b)  = Prop (a ⊔ b)
funSort (Prop a)  (Prop b)  = Prop (a ⊔ b)
```

Note

`funSort` can admit just two arguments, so it will be iterated when the function type has multiple arguments. E.g. the function type `∀ {A} → A → A → A` evaluates to `funSort (univSort _5) (funSort _5 (funSort _5 _5))`

piSort

Similarly, `piSort s1 (λ x → s2)` is a more general version of `funSort` that computes the sort of a Π-type given the sort `s1` of its domain and the sort `s2` of its codomain as arguments. It is used in cases where the sort of the codomain of a Π-type might be dependent on the variable bound by the Π-type. It computes to either `funSort` (when the codomain sort is not dependent) or to `Set`ω (when it is dependent).

To understand how `piSort` works in general, we set the following scenario:

- `sA` and `sB` are two (possibly different) sorts.
- `A : sA`, meaning that `A` is a type that has sort `sA`.
- `x : A`, meaning that `x` has type `A`.
- `B : sB`, meaning that `B` is a type (possibly different than `A`) that has sort `sB`.

Under these conditions, we can build the dependent function type `(x : A) → B : piSort sA (λ x → sB)`. This type signature means that the dependent function type `(x : A) → B` has a (possibly unknown) but well-defined sort `piSort sA (λ x → sB)`, specified in terms of the sorts of its domain and codomain.

Here are some examples how `piSort` computes:

```
piSort s1       (λ x → s2)    = funSort s1 s2          (if x does not occur freely in␣
↪s2)
piSort (Set ℓ)  (λ x → Set ℓ') = Setω                  (if x occurs rigidly in ℓ')
piSort (Prop ℓ) (λ x → Set ℓ') = Setω                  (if x occurs rigidly in ℓ')
piSort Setωi    (λ x → Set ℓ') = Setωi                 (if x occurs rigidly in ℓ')
```

With these rules, we can compute the sort of the function type `∀ {A} → ∀ {B} → B → A → B` (or more explicitly, `{A : _9} {B : _7} → B → A → B`) to be `piSort (univSort _9) (λ A → funSort (univSort _7) (funSort _7 (funSort _9 _7)))`.

More examples:

- `piSort LevelUniv (λ l → Set l)` evaluates to `Setω` (see also *level universe*)
- `piSort (Set l) (λ _ → Set l')` evaluates to `Set (l ⊔ l')`
- `piSort s (λ _ → Setωi)` evaluates to `funSort s Setωi`

3.44 Syntactic Sugar

3.44.1 Hidden argument puns

If the option *--hidden-argument-puns* is used, then the pattern `{x}` is interpreted as `{x = x}`, and the pattern PDF TODO `x` PDF TODO is interpreted as PDF TODO `x = x` PDF TODO. Here `x` must be an unqualified name that does not refer to a constructor that is in scope: if `x` is qualified, then the pattern is not interpreted as a pun, and if `x` is unqualified and refers to a constructor that is in scope, then the code is rejected.

Note that `{(x)}` and PDF TODO `(x)` PDF TODO are not interpreted as puns.

Note also that `{x}` is not interpreted as a pun in `λ {x} → ...` or `syntax f {x} = ...`. However, `{x}` is interpreted as a pun in `λ (c {x}) → ...`.

3.44.2 Do-notation

A *do-block* consists of the *layout keyword* `do` followed by a sequence of *do-statements*, where

```
do-stmt    ::= pat ← expr [where lam-clauses]
             | let decls
             | expr
lam-clause ::= pat → expr
```

The `where` clause of a bind is used to handle the cases not matched by the pattern left of the arrow. See *details below*.

> **Note**
>
> Arrows can use either unicode (←/→) or ASCII (<-/->) variants.

For example:

```
filter : {A : Set} → (A → Bool) → List A → List A
filter p xs = do
  x    ← xs
  true ← p x ∷ []
    where false → []
  x ∷ []
```

Do-notation is desugared before scope checking and is translated into calls to _>>=_ and _>>_, which can be arbitrary user-defined functions. This means that do-blocks are not tied to any particular notion of monad. In fact, if there are no monadic statements in the do block, it can be used as sugar for a `let`:

```
pure-do : Nat → Nat
pure-do n = do
  let p2 m = m * m
      p4 m = p2 (p2 m)
  p4 n

check-pure-do : pure-do 5 ≡ 625
check-pure-do = refl
```

The operators used for desugaring do notation are resolved as if the user had written the desugared version, and so will refer to whatever functions are in scope by those names. To control this, the do keyword may appear qualified by a *module* name, in which case the operators will be drawn from that module instead:

```
module Add where
  _>>_ : Nat → Nat → Nat
  x >> y = x + y

qualified-example : Nat
qualified-example = Add.do
  1
  2
  3

_ : qualified-example ≡ 6
_ = refl
```

Note that the qualifying module is **not** opened in the scope of the do expression, i.e. no bindings are brought into scope. This means that explicit usage of _>>=_ and _>>_ inside a qualified do expression will still refer to the **unqualified** versions.

Desugaring

Statement	Sugar	Desugars to
Simple bind	`do x ← m` `  m'`	`m >>= λ x →` `m'`

Pattern bind	`do p ← m`
	`  where` $p_i \to m_i$
	`  m'`
		`m >>= λ where`
		`  p  → m'`
		$p_i \to m_i$

Absurd match	`do () ← m`	`m >>= λ ()`
Non-binding statement	`do m` `  m'`	`m >>` `m'`
Let	`do let ds` `  m'`	`let ds in` `m'`

If the pattern in the bind is exhaustive, the where-clause can be omitted.

Example

Do-notation becomes quite powerful together with pattern matching on indexed data. As an example, let us write a correct-by-construction type checker for simply typed λ-calculus.

First we define the raw terms, using de Bruijn indices for variables and explicit type annotations on the lambda:

```
infixr 6 _=>_
data Type : Set where
  nat  : Type
  _=>_ : (A B : Type) → Type

data Raw : Set where
  var : (x : Nat) → Raw
  lit : (n : Nat) → Raw
  suc : Raw
  app : (s t : Raw) → Raw
  lam : (A : Type) (t : Raw) → Raw
```

Next up, well-typed terms:

```
Context = List Type

-- A proof of x ∈ xs is the index into xs where x is located.
infix 2 _∈_
data _∈_ {A : Set} (x : A) : List A → Set where
  zero : ∀ {xs} → x ∈ x :: xs
  suc  : ∀ {y xs} → x ∈ xs → x ∈ y :: xs

data Term (Γ : Context) : Type → Set where
  var : ∀ {A} (x : A ∈ Γ) → Term Γ A
  lit : (n : Nat) → Term Γ nat
  suc : Term Γ (nat => nat)
```

(continues on next page)

(continued from previous page)

```
  app : ∀ {A B} (s : Term Γ (A => B)) (t : Term Γ A) → Term Γ B
  lam : ∀ A {B} (t : Term (A :: Γ) B) → Term Γ (A => B)
```

Given a well-typed term we can mechanically erase all the type information (except the annotation on the lambda) to get the corresponding raw term:

```
rawIndex : ∀ {A} {x : A} {xs} → x ∈ xs → Nat
rawIndex zero    = zero
rawIndex (suc i) = suc (rawIndex i)

eraseTypes : ∀ {Γ A} → Term Γ A → Raw
eraseTypes (var x)   = var (rawIndex x)
eraseTypes (lit n)   = lit n
eraseTypes suc       = suc
eraseTypes (app s t) = app (eraseTypes s) (eraseTypes t)
eraseTypes (lam A t) = lam A (eraseTypes t)
```

Now we're ready to write the type checker. The goal is to have a function that takes a raw term and either fails with a type error, or returns a well-typed term that erases to the raw term it started with. First, lets define the return type. It's parameterised by a context and the raw term to be checked:

```
data WellTyped Γ e : Set where
  ok : (A : Type) (t : Term Γ A) → eraseTypes t ≡ e → WellTyped Γ e
```

We're going to need a corresponding type for variables:

```
data InScope Γ n : Set where
  ok : (A : Type) (i : A ∈ Γ) → rawIndex i ≡ n → InScope Γ n
```

Lets also have a type synonym for the case when the erasure proof is `refl`:

```
infix 2 _ofType_
pattern _ofType_ x A = ok A x refl
```

Since this is a do-notation example we had better have a monad. Lets use the either monad with string errors:

```
TC : Set → Set
TC A = Either String A

typeError : ∀ {A} → String → TC A
typeError = left
```

For the monad operations, we are using *instance arguments* to infer which monad is being used.

We are going to need to compare types for equality. This is our first opportunity to take advantage of pattern matching binds:

```
_=?=_ : (A B : Type) → TC (A ≡ B)
nat      =?= nat      = pure refl
nat      =?= (_ => _) = typeError "type mismatch: expected nat, got _ => _"
(_ => _) =?= nat      = typeError "type mismatch: expected _ => _, got nat"
(A => B) =?= (A₁ => B₁) = do
  refl ← A =?= A₁
```

(continues on next page)

(continued from previous page)

```
  refl ← B =?= B₁
  pure refl
```

We will also need to look up variables in the context:

```
lookupVar : ∀ Γ n → TC (InScope Γ n)
lookupVar []      n       = typeError "variable out of scope"
lookupVar (A :: Γ) zero    = pure (zero ofType A)
lookupVar (A :: Γ) (suc n) = do
  i ofType B ← lookupVar Γ n
  pure (suc i ofType B)
```

Note how the proof obligation that the well-typed deBruijn index erases to the given raw index is taken care of completely under the hood (in this case by the `refl` pattern in the `ofType` synonym).

Finally we are ready to implement the actual type checker:

```
infer : ∀ Γ e → TC (WellTyped Γ e)
infer Γ (var x)    = do
  i ofType A ← lookupVar Γ x
  pure (var i ofType A)
infer Γ (lit n)    = pure (lit n ofType nat)
infer Γ suc        = pure (suc ofType nat => nat)
infer Γ (app e e₁) = do
  s ofType A => B ← infer Γ e
    where _ ofType nat → typeError "numbers cannot be applied to arguments"
  t ofType A₁     ← infer Γ e₁
  refl            ← A =?= A₁
  pure (app s t ofType B)
infer Γ (lam A e)  = do
  t ofType B ← infer (A :: Γ) e
  pure (lam A t ofType A => B)
```

In the `app` case we use a where-clause to handle the error case when the function to be applied is well-typed, but does not have a function type.

3.44.3 Idiom brackets

Idiom brackets is a notation used to make it more convenient to work with applicative functors, i.e. functors `F` equipped with two operations

```
pure  : ∀ {A} → A → F A
_<*>_ : ∀ {A B} → F (A → B) → F A → F B
```

Name resolution for idiom brackets works as for `do`-notation: the operators are resolved as though the user had written them, unless the brackets appear qualified, in which case they are looked up in the qualifying module.

The syntax for idiom brackets is

```
(| e a₁ .. aₙ |)
```

or using unicode lens brackets ⦇ (U+2987) and ⦈ (U+2988):

```
⦇ e a₁ .. aₙ ⦈
```

This expands to (assuming left associative _<*>_)

```
pure e <*> a₁ <*> .. <*> aₙ
```

Idiom brackets work well with operators, for instance

```
(| if a then b else c |)
```

desugars to

```
pure if_then_else_ <*> a <*> b <*> c
```

Idiom brackets also support none or multiple applications. If the applicative functor has an additional binary operation

```
_<|>_ : ∀ {A B} → F A → F A → F A
```

then idiom brackets support multiple applications separated by a vertical bar `|`, i.e.

```
(| e₁ a₁ .. aₙ | e₂ a₁ .. aₘ | .. | eₖ a₁ .. aₗ |)
```

which expands to (assuming right associative _<|>_)

```
(pure e₁ <*> a₁ <*> .. <*> aₙ) <|> ((pure e₂ <*> a₁ <*> .. <*> aₘ) <|> (pure eₖ <*> a₁ <*>
↪ .. <*> aₗ))
```

Idiom brackets without any application `(|)` or `⦇⦈` expend to `empty` if

```
empty : ∀ {A} → F A
```

is in scope. An applicative functor with `empty` and _<|>_ is typically called `Alternative`.

Note that `pure`, _<*>_, and _<|>_ need not be in scope to use `(|)`.

Limitations:

- Binding syntax and operator sections cannot appear immediately inside idiom brackets.
- The top-level application inside idiom brackets cannot include implicit applications, so

  ```
  (| foo {x = e} a b |)
  ```

  is illegal. In case the `e` is pure you can write

  ```
  (| (foo {x = e}) a b |)
  ```

  which desugars to

  ```
  pure (foo {x = e}) <*> a <*> b
  ```

3.45 Syntax Declarations

> **Note**
>
> This is a stub

It is now possible to declare user-defined syntax that binds identifiers. Example:

```
record Σ (A : Set) (B : A → Set) : Set where
  constructor _,_
  field fst : A
        snd : B fst

syntax Σ A (λ x → B) = [ x ∈ A ] × B

witness : ∀ {A B} → [ x ∈ A ] × B → A
witness (x , _) = x
```

The syntax declaration for Σ implies that `x` is in scope in `B`, but not in `A`.

You can give fixity declarations along with syntax declarations:

```
infix 5 Σ
syntax Σ A (λ x → B) = [ x ∈ A ] × B
```

The fixity applies to the syntax, not the name; syntax declarations are also restricted to ordinary, non-operator names. The following declaration is disallowed:

```
syntax _==_ x y = x === y
```

Syntax declarations must also be linear; the following declaration is disallowed:

```
syntax wrong x = x + x
```

Syntax declarations can have implicit arguments. For example:

```
id : ∀ {a}{A : Set a} -> A -> A
id x = x

syntax id {A} x = x ∈ A
```

Unlike *mixfix operators* that can be used unapplied using the name including all the underscores, or partially applied by replacing only some of the underscores by arguments, syntax must be fully applied.

3.46 Telescopes

A telescope is a non-empty sequence of variable bindings annotated by their types, with each variable surrounded by parentheses. For example, `(x : Nat) (y : Bool) (z : Bool)` is a telescope. Adjacent variables that have the same type can share a type annotation. For example, the same telescope can be written equivalently as `(x : Nat) (y z : Bool)`. The type of each variable can depend on the previous variables in the telescope, for example `(A : Set) (n : Nat) (v : Vec A n)`.

> **Note**

> The terminology is due to de Bruijn *[1]*: "The word was inspired, of course, by the old-fashioned instrument consisting of segments that slide one into another." Each variable binding corresponds to a segment of the telescope, which can slide into (i.e. depend on) the previous ones.

Telescopes appear in the following parts of the Agda syntax:

- *Function types*
- Declarations of *data types* and *record types*
- Declarations of *parameterised modules*

```
postulate
  f : (A : Set) (n : Nat) (v : Vec A n) → Nat

data D (A : Set) (n : Nat) (v : Vec A n) : Set where
  -- ...

module M (A : Set) (n : Nat) (v : Vec A n) where
  -- ...
```

In telescopes of data and record types as well as parameterised modules, it is allowed to omit the type of a variable binding. This is equivalent to giving the variable the type _ (see *implicit arguments*).

```
data D' A n (v : Vec A n) : Set where
  -- ...

module M' A n (v : Vec A n) where
  -- ...
```

In function types, the type of a variable binding can be omitted if the module telescope starts with `forall` or ∀.

```
postulate
  f' : ∀ A n (v : Vec A n) → Nat
```

When binding a variable of a *record type* (but not a data type), it is possible to deconstruct the bound variable by *pattern matching*:

```
module N ((x , y) : Nat × Bool) where
  -- ...
```

Variable bindings in a telescope can be *implicit* or *instance* arguments. For example:

```
postulate
  mconcat : {A : Set} {{monoidA : Monoid A}} → List A → A
```

They can also be *irrelevant* or carry another *modality*. For example:

```
postulate
  div : (m n : Nat) .(nz : NonZero n) → Nat
```

3.46.1 Irrefutable Patterns in Binding Positions

Since Agda 2.6.1, irrefutable patterns can be used at every binding site in a telescope to take the bound value of record type apart. The type of the second projection out of a dependent pair will for instance naturally mention the value of the first projection. Its type can be defined directly using an irrefutable pattern as follows:

```
proj₂ : ((a , _) : Σ A B) → B a
```

And this second projection can be implemented with a lamba-abstraction using one of these irrefutable patterns taking the pair apart:

```
proj₂ = λ (_ , b) → b
```

Using an as-pattern makes it possible to name the argument and to take it apart at the same time. We can for instance prove that any pair is equal to the pairing of its first and second projections, a property commonly called eta-equality:

```
eta : (p@(a , b) : Σ A B) → p ≡ (a , b)
eta p = refl
```

Since Agda 2.9.0, irrefutable patterns require that the deconstructed record has eta-equality. This is the case for Σ, but e.g. not for `coinductive` records or records declared with `no-eta-equality`. In the absence of eta-equality, irrefutable patterns trigger the warning *ShouldBeEtaRecordPattern*.

3.46.2 Let Bindings in Telescopes

Telescopes of function types and parameterised modules (but not of data and record types) can also contain *let bindings*. When used in this manner, the let-binding should be surrounded by parentheses and the `in` part of the syntax is omitted. For example:

```
postulate
  g : (x : Nat) (let y = x + x) (v : Vec Nat y) → Nat
```

Let-bound variables in a module telescope are available in the whole module. For example:

```
module O (X : Set) (let LX = List X) (l : LX) where

  extend : LX → LX
  extend m = l ++ m
```

In general, any valid let-binding can also be used in a telescope. For example, it is possible to pattern match on a record type with a let-binding:

```
postulate
  h : (f : Nat → (Bool × Bool)) (let (x0 , y0) = f 0) (tx : IsTrue x0) → IsTrue y0
```

Another notable example is opening a *module* in a telescope:

```
module M1 (X : Set) (let open M X) where
```

This can also be written more compactly with just `open` (without the `let`):

```
module M2 (X : Set) (open M X) where
```

References

[1] N.G. de Bruijn. "Telescopic mappings in typed lambda calculus." Information and Computation, Volume 91, Issue 2, 1991.

3.47 Termination Checking

Not all recursive functions are permitted - Agda accepts only these recursive schemas that it can mechanically prove terminating.

3.47.1 Primitive recursion

In the simplest case, a given argument must be exactly one constructor smaller in each recursive call. We call this scheme primitive recursion. A few correct examples:

```
plus : Nat → Nat → Nat
plus zero    m = m
plus (suc n) m = suc (plus n m)

natEq : Nat → Nat → Bool
natEq zero    zero    = true
natEq zero    (suc m) = false
natEq (suc n) zero    = false
natEq (suc n) (suc m) = natEq n m
```

Both `plus` and `natEq` are defined by primitive recursion.

The recursive call in `plus` is OK because `n` is a subexpression of `suc n` (so `n` is structurally smaller than `suc n`). So every time plus is recursively called the first argument is getting smaller and smaller. Since a natural number can only have a finite number of suc constructors we know that plus will always terminate.

`natEq` terminates for the same reason, but in this case we can say that both the first and second arguments of natEq are decreasing.

3.47.2 Structural recursion

Agda's termination checker allows more definitions than just primitive recursive ones – it allows structural recursion.

This means that we require recursive calls to be on a (strict) subexpression of the argument (see `fib` below) - this is more general that just taking away one constructor at a time.

```
fib : Nat → Nat
fib zero          = zero
fib (suc zero)    = suc zero
fib (suc (suc n)) = plus (fib n) (fib (suc n))
```

It also means that arguments may decrease in an lexicographic order - this can be thought of as nested primitive recursion (see `ack` below).

```
ack : Nat → Nat → Nat
ack zero    m       = suc m
ack (suc n) zero    = ack n (suc zero)
ack (suc n) (suc m) = ack n (ack (suc n) m)
```

In `ack` either the first argument decreases or it stays the same and the second one decreases. This is the same as a lexicographic ordering.

3.47.3 With-functions

3.47.4 Pragmas and Options

- The `NON_TERMINATING` pragma

  This is a safer version of *TERMINATING* which doesn't treat the affected functions as terminating. This means that `NON_TERMINATING` functions do not reduce during type checking. Though, they do reduce at run-time and when invoking `C-u C-c C-n` interactively. The pragma was added in Agda 2.4.2.

- The `TERMINATING` pragma

  Switches off termination checker for individual function definitions and mutual blocks and marks them as terminating. Since Agda 2.4.2.1 replaced the `NO_TERMINATION_CHECK` pragma.

  The pragma must precede a function definition or a mutual block. The pragma cannot be used in *--safe* mode.

  Examples:

  – Skipping a single definition: before type signature:

  ```
  {-# TERMINATING #-}
  a : A
  a = a
  ```

  – Skipping a single definition: before first clause:

  ```
  b : A
  {-# TERMINATING #-}
  b = b
  ```

  – Skipping an old-style mutual block: Before *mutual* keyword:

  ```
  {-# TERMINATING #-}
  mutual
    c : A
    c = d

    d : A
    d = c
  ```

  – Skipping an old-style mutual block: Somewhere within *mutual* block before a type signature or first function clause:

  ```
  mutual
   {-# TERMINATING #-}
   e : A
   e = f

   f : A
   f = e
  ```

  – Skipping a new-style mutual block: Anywhere before a type signature or first function clause in the block:

  ```
  g : A
  h : A

  g = h
  ```

(continues on next page)

(continued from previous page)

```
{-# TERMINATING #-}
h = g
```

- Increasing the maximal analysis depth with `--termination-depth`.

  The following mutual functions need a termination depth of 2 to be accepted by the termination checker:

```
mutual

  f : Nat → Nat
  f zero = zero
  f (suc zero) = suc zero
  f (suc (suc x)) = g x

  g : Nat → Nat
  g y = f (suc y)
```

  With a termination depth of 1, the termination checker would only register that the call from `f` to `g` *decreases* the argument and the call from `g` to `f` *increases* the argument, but not by *how much*. Thus, it has no evidence that the call sequence `f` → `g` → `f` decreases the argument.

  With termination depth 2, it will see that the call `f` → `g` decreases by 2 and the call `g` → `f` increases only by 1, so the overall decrease in `f` → `g` → `f` is still 1.

  In general termination depth *N* can track decrease up to *N* and increase up to *N-1*.

  Agda will first check termination with a termination depth of 1 and increase this value until the termination check succeeds or the maximum termination depth has been reached. The maximum depth is by default 3 (since Agda 2.9.0) and can be set by the `--termination-depth`.

  In practice, it should not be necessary to increase the maximum termination depth, as examples using depth 2 are already rare. A high termination depth can make the termination checker slow and memory hungry. Rather then increasing the termination depth, functions should be reformulated such that they are structurally recursive, i.e., only match one level deep.

3.47.5 References

Andreas Abel, Foetus – termination checker for simple functional programs

3.48 Two-Level Type Theory

3.48.1 Basics

Two-level type theory (2LTT) refers to versions of Martin-Löf type theory that combine two type theories: one "inner" level that is potentially a homotopy type theory or cubical type theory, which may include univalent universes and higher inductive types, and a second "outer" level that validates uniqueness of identity proofs.

Since version 2.6.2, Agda enables 2LTT with the `--two-level` flag. The two levels are distinguished with two hierarchies of universes: the usual universes `Set` for the inner level, and a new hierarchy of universes denoted `SSet` (for "strict sets") for the outer level.

Note

The types in `SSet` have various names in the literature. They are called *non-fibrant types* in HTS (2017), *outer types* in 2LTT (2017), and *exo-types* in UP (2021). Similarly, these references refer to the types in `Set` as *fibrant types*, *inner types*, and *types*, respectively.

Function-types belong to `Set` if both their domain and codomain do, and to `SSet` otherwise. Records and datatypes can always be declared to belong to `SSet`, and can be declared to belong to `Set` instead if all their inputs belong to `Set`. In particular, any type in `Set` can be lifted to `SSet` using a trivial record:

```
record c (A : Set) : SSet where
  constructor ↑
  field
    ↓ : A

open c
```

The main differences between the two levels are that, firstly, homotopical flags such as `--without-K` and `--cubical` apply only to the `Set` level (the `SSet` level is never homotopical); and secondly, datatypes belonging to the inner level cannot be pattern-matched on when the motive belongs to the outer level (this is necessary to maintain the previous distinction).

As a primary example, we can define separate inductive equality types for both levels:

```
infix 4 _≡ˢ_ _≡_

data _≡ˢ_ {a} {A : SSet a} (x : A) : A → SSet a where
  reflˢ : x ≡ˢ x

data _≡_ {a} {A : Set a} (x : A) : A → Set a where
  refl : x ≡ x
```

With these definitions, we can prove uniqueness of identity proofs for the strict equality even if `--without-K` or `--cubical` is enabled:

```
UIP : {a : Level} {A : SSet a} {x y : A} (p q : x ≡ˢ y) → p ≡ˢ q
UIP reflˢ reflˢ = reflˢ
```

We can also prove that strictly equal elements are also non-strictly equal:

```
≡ˢ-to-≡ : {A : Set} {x y : c A} → (x ≡ˢ y) → (↓ x ≡ ↓ y)
≡ˢ-to-≡ reflˢ = refl
```

The opposite implication, however, fails because, as noted above, we cannot pattern-match against a datatype in `Set` when the motive lies in `SSet`. Similarly, we can map from the strict natural numbers into the ordinary ones:

```
data ℕ : Set where
  zero : ℕ
  succ : ℕ → ℕ

data ℕˢ : SSet where
  zeroˢ : ℕˢ
  succˢ : ℕˢ → ℕˢ

ℕˢ-to-ℕ : ℕˢ → ℕ
```

(continues on next page)

(continued from previous page)

```
ℕˢ-to-ℕ zeroˢ = zero
ℕˢ-to-ℕ (succˢ n) = succ (ℕˢ-to-ℕ n)
```

but not vice versa. (Agda does currently allow mapping from the empty `SSet` to the empty `Set`, but this feature is disputed.)

If the `--two-level` flag is combined with `--cumulativity`, then each universe `Set a` becomes a subtype of `SSet a`. In this case we can instead define the coercion `c` to be the identity function:

```
c' : Set → SSet
c' A = A
```

and replace the coercions ↑ and ↓ with the identity function. However, this combination currently allows some functions to be defined that shouldn't be allowed; see Agda issue #5761 for details.

3.49 Universe Levels

Agda's type system includes an infinite hierarchy of universes $\texttt{Set}_i$ `:` $\texttt{Set}_{i+1}$. This hierarchy enables quantification over arbitrary types without running into the inconsistency that follows from `Set :  Set`. These universes are further detailed on the page on *Agda's sort system*.

However, when working with this hierarchy it can quickly get tiresome to repeat the same definition at different universe levels. For example, we might be forced to define new datatypes `data List (A : Set) :  Set`, `data` $\texttt{List}_1$ `(A :` $\texttt{Set}_1$`) :` $\texttt{Set}_1$, etc. Also every function on lists (such as `append`) must be re-defined, and every theorem about such functions must be re-proved, for every possible level.

The solution to this problem is universe polymorphism. Agda provides a special primitive type `Level`, whose elements are possible levels of universes. In fact, the notation for the `n` th universe, $\texttt{Set}_n$, is just an abbreviation for `Set n`, where `n :  Level` is a level. We can use this to write a polymorphic `List` operator that works at any level. The library `Agda.Primitive` must be imported to access the `Level` type. The definition then looks like this:

```
open import Agda.Primitive

data List {n : Level} (A : Set n) : Set n where
  []   : List A
  _::_ : A → List A → List A
```

This new operator works at all levels; for example, we have

```
List Nat : Set
List Set : Set₁
List Set₁ : Set₂
```

3.49.1 Level arithmetic

Even though we don't have the number of levels specified, we know that there is a lowest level `lzero`, and for each level `n`, there exists some higher level `lsuc n`; therefore, the set of levels is infinite. In addition, we can also take the least upper bound `n ⊔ m` of two levels. In summary, the following (and only the following) operations on levels are provided:

```
lzero : Level
lsuc  : (n : Level) → Level
_⊔_   : (n m : Level) → Level
```

This is sufficient for most purposes; for example, we can define the Cartesian product of two types of arbitrary (and not necessarily equal) levels like this:

```
data _×_ {n m : Level} (A : Set n) (B : Set m) : Set (n ⊔ m) where
  _,_ : A → B → A × B
```

With this definition, we have, for example:

```
Nat × Nat : Set
Nat x Set : Set₁
Set × Set : Set₁
```

3.49.2 Intrinsic level properties

Levels and their associated operations have some properties which are internally and automatically solved by the compiler. This means that we can replace some expressions with others, without worrying about the expressions for their corresponding levels matching exactly.

For example, we can write:

```
_ : {F : (l : Level) → Set l} {l1 l2 : Level} → F (l1 ⊔ l2) → F (l2 ⊔ l1)
_ = λ x → x
```

and Agda does the conversion from `F (l1 ⊔ l2)` to `F (l2 ⊔ l1)` automatically.

Here is a list of the level properties:

- Idempotence: `a ⊔ a` is the same as `a`.
- Associativity: `(a ⊔ b) ⊔ c` is the same as `a ⊔ (b ⊔ c)`.
- Commutativity: `a ⊔ b` is the same as `b ⊔ a`.
- Distributivity of `lsuc` over `⊔`: `lsuc (a ⊔ b)` is the same as `lsuc a ⊔ lsuc b`.
- Neutrality of `lzero`: `a ⊔ lzero` is the same as `a`.
- Subsumption: `a ⊔ lsuc a` is the same as `lsuc a`. Notably, this also holds for arbitrarily many `lsuc` usages: `a ⊔ lsuc (lsuc a)` is also the same as `lsuc (lsuc a)`.

3.49.3 `forall` notation

From the fact that we write `Set n`, it can always be inferred that `n` is a level. Therefore, when defining universe-polymorphic functions, it is common to use the `∀` (or *forall*) notation. For example, the type of the universe-polymorphic `map` operator on lists can be written

```
map : ∀ {n m} {A : Set n} {B : Set m} → (A → B) → List A → List B
```

which is equivalent to

```
map : {n m : Level} {A : Set n} {B : Set m} → (A → B) → List A → List B
```

3.49.4 Expressions of sort `Set`ω

In a sense, universes were introduced to ensure that every Agda expression has a type, including expressions such as `Set`, `Set`$_1$, etc. However, the introduction of universe polymorphism inevitably breaks this property again, by creating some new terms that have no type. Consider the polymorphic singleton set `Unit n :  Set`$_n$, defined by

```
data Unit (n : Level) : Set n where
  <> : Unit n
```

It is well-typed, and has type

```
Unit : (n : Level) → Set n
```

However, the type `(n : Level) → Set n`, which is a valid Agda expression, does not belong to any universe in the `Set` hierarchy. Indeed, the expression denotes a function mapping levels to sorts, so if it had a type, it should be something like `Level → Sort`, where `Sort` is the collection of all sorts. But if Agda were to support a sort `Sort` of all sorts, it would be a sort itself, so in particular we would have `Sort : Sort`. Just like `Type : Type`, this would leads to circularity and inconsistency.

Instead, Agda introduces a new sort `Set`ω that stands above all sorts `Set` ℓ, but is not itself part of the hierarchy. For example, Agda assigns the expression `(n : Level) → Set n` to be of type `Set`ω.

`Set`ω is itself the first step in another infinite hierarchy `Set`ω_i `:` `Set`ω_{i+1}. However, this hierarchy does not support universe polymorphism, i.e. there are no sorts `Set`ω ℓ for ℓ `:` `Level`. Allowing this would require a new universe `Set2`ω, which would then naturally lead to `Set2`ω_1, and so on. Disallowing universe polymorphism for `Set`ω_i avoids the need for such even larger sorts. This is an intentional design decision.

3.49.5 Pragmas and options

- The option *--type-in-type* disables the checking of universe level consistency for the whole file.
- The option *--omega-in-omega* enables the typing rule `Set`ω `:` `Set`ω (thus making Agda inconsistent) but otherwise leaves universe checks intact.
- The option *--level-universe* makes `Level` live in its own universe `LevelUniv` and disallows having levels depend on terms that are not levels themselves. When this option is turned off, `LevelUniv` still exists, but reduces to `Set`.

  Note: While compatible with the *--cubical* option, this option is currently not compatible with cubical builtin files, and an error will be raised when trying to import them in a file using *--level-universe*.

  ```
  {-# OPTIONS --level-universe #-}
  open import Agda.Primitive
  open import Agda.Builtin.Nat

  toLevel : Nat → Level
  toLevel _ = lzero
  ```

  ```
  funSort Set LevelUniv is not a valid sort
  when checking that the expression Nat → Level is a type
  ```

- The pragma `{-# NO_UNIVERSE_CHECK #-}` can be put in front of a data or record type to disable universe consistency checking locally. Example:

  ```
  {-# NO_UNIVERSE_CHECK #-}
  data U : Set where
    el : Set → U
  ```

  This pragma applies only to the check that the universe level of the type of each constructor argument is less than or equal to the universe level of the datatype, not to any other checks.

  Added in version 2.6.0.

The options `--type-in-type` and `--omega-in-omega` and the pragma `{-# NO_UNIVERSE_CHECK #-}` cannot be used with *–safe*.

3.50 With-Abstraction

With-abstraction was first introduced by Conor McBride [McBride2004] and lets you pattern match on the result of an intermediate computation by effectively adding an extra argument to the left-hand side of your function.

3.50.1 Usage

In the simplest case the `with` construct can be used just to discriminate on the result of an intermediate computation. For instance

```
filter : {A : Set} → (A → Bool) → List A → List A
filter p [] = []
filter p (x :: xs) with p x
filter p (x :: xs)    | true  = x :: filter p xs
filter p (x :: xs)    | false = filter p xs
```

The clause containing the with-abstraction has no right-hand side. Instead it is followed by a number of clauses with an extra argument on the left, separated from the original arguments by a vertical bar (`|`).

When the original arguments are the same in the new clauses you can use the `...` syntax:

```
filter : {A : Set} → (A → Bool) → List A → List A
filter p [] = []
filter p (x :: xs) with p x
```

(continues on next page)

(continued from previous page)

```
...                  | true  = x :: filter p xs
...                  | false = filter p xs
```

In this case `...` expands to `filter p (x :: xs)`. There are three cases where you have to spell out the left-hand side:

- If you want to do further pattern matching on the original arguments.
- When the pattern matching on the intermediate result refines some of the other arguments (see *Dot patterns*).
- To disambiguate the clauses of nested with-abstractions (see *Nested with-abstractions* below).

Generalisation

The power of with-abstraction comes from the fact that the goal type and the type of the original arguments are generalised over the value of the scrutinee. See *Technical details* below for the details. This generalisation is important when you have to prove properties about functions defined using `with`. For instance, suppose we want to prove that the `filter` function above satisfies some property `P`. Starting out by pattern matching of the list we get the following (with the goal types shown in the holes)

```
postulate P : ∀ {A} → List A → Set
postulate p-nil : ∀ {A} → P {A} []
postulate Q : Set
postulate q-nil : Q
```

```
proof : {A : Set} (p : A → Bool) (xs : List A) → P (filter p xs)
proof p []       = {! P [] !}
proof p (x :: xs) = {! P (filter p (x :: xs) | p x) !}
```

In the cons case we have to prove that `P` holds for `filter p (x :: xs) | p x`. This is the syntax for a stuck with-abstraction—`filter` cannot reduce since we don't know the value of `p x`. This syntax is used for printing, but is not accepted as valid Agda code. Now if we with-abstract over `p x`, but don't pattern match on the result we get:

```
proof : {A : Set} (p : A → Bool) (xs : List A) → P (filter p xs)
proof p [] = p-nil
proof p (x :: xs) with p x
...                  | r   = {! P (filter p (x :: xs) | r) !}
```

Here the `p x` in the goal type has been replaced by the variable `r` introduced for the result of `p x`. If we pattern match on `r` the with-clauses can reduce, giving us:

```
proof : {A : Set} (p : A → Bool) (xs : List A) → P (filter p xs)
proof p [] = p-nil
proof p (x :: xs) with p x
...                  | true  = {! P (x :: filter p xs) !}
...                  | false = {! P (filter p xs) !}
```

Both the goal type and the types of the other arguments are generalised, so it works just as well if we have an argument whose type contains `filter p xs`.

```
proof₂ : {A : Set} (p : A → Bool) (xs : List A) → P (filter p xs) → Q
proof₂ p [] _ = q-nil
proof₂ p (x :: xs) H with p x
...                     | true  = {! H : P (x :: filter p xs) !}
...                     | false = {! H : P (filter p xs) !}
```

The generalisation is not limited to scrutinees in other with-abstractions. All occurrences of the term in the goal type and argument types will be generalised.

Note that this generalisation is not always type correct and may result in a (sometimes cryptic) type error. See *Ill-typed with-abstractions* below for more details.

Nested with-abstractions

With-abstractions can be nested arbitrarily. The only thing to keep in mind in this case is that the `...` syntax applies to the closest with-abstraction. For example, suppose you want to use `...` in the definition below.

```
compare : Nat → Nat → Comparison
compare x y with x < y
compare x y    | false with y < x
compare x y    | false    | false = equal
compare x y    | false    | true  = greater
compare x y    | true = less
```

You might be tempted to replace `compare x y` with `...` in all the with-clauses as follows.

```
compare : Nat → Nat → Comparison
compare x y with x < y
...            | false with y < x
...                       | false = equal
...                       | true  = greater
...            | true = less    -- WRONG
```

This, however, would be wrong. In the last clause the `...` is interpreted as belonging to the inner with-abstraction (the whitespace is not taken into account) and thus expands to `compare x y | false | true`. In this case you have to spell out the left-hand side and write

```
compare : Nat → Nat → Comparison
compare x y with x < y
...            | false with y < x
...                       | false = equal
...                       | true  = greater
compare x y    | true = less
```

Simultaneous abstraction

You can abstract over multiple terms in a single with-abstraction. To do this you separate the terms with vertical bars (`|`).

```
compare : Nat → Nat → Comparison
compare x y with x < y | y < x
...            | true  | _     = less
...            | _     | true  = greater
...            | false | false = equal
```

In this example the order of abstracted terms does not matter, but in general it does. Specifically, the types of later terms are generalised over the values of earlier terms. For instance

```
postulate plus-commute : (a b : Nat) → a + b ≡ b + a
postulate P : Nat → Set
```

```
thm : (a b : Nat) → P (a + b) → P (b + a)
thm a b t with a + b | plus-commute a b
thm a b t    | ab    | eq = {! t : P ab, eq : ab ≡ b + a !}
```

Note that both the type of `t` and the type of the result `eq` of `plus-commute a b` have been generalised over `a + b`. If the terms in the with-abstraction were flipped around, this would not be the case. If we now pattern match on `eq` we get

```
thm : (a b : Nat) → P (a + b) → P (b + a)
thm a b t with   a + b  | plus-commute a b
thm a b t    | .(b + a) | refl = {! t : P (b + a) !}
```

and can thus fill the hole with `t`. In effect we used the commutativity proof to rewrite `a + b` to `b + a` in the type of `t`. This is such a useful thing to do that there is special syntax for it. See *Rewrite* below. A limitation of generalisation is that only occurrences of the term that are visible at the time of the abstraction are generalised over, but more instances of the term may appear once you start filling in the right-hand side or do further matching on the left. For instance, consider the following contrived example where we need to match on the value of `f n` for the type of `q` to reduce, but we then want to apply `q` to a lemma that talks about `f n`:

```
postulate
  R     : Set
  P     : Nat → Set
  f     : Nat → Nat
  lemma : ∀ n → P (f n) → R

Q : Nat → Set
Q zero    = ⊥
Q (suc n) = P (suc n)
```

```
proof : (n : Nat) → Q (f n) → R
proof n q with f n
proof n ()    | zero
proof n q     | suc fn = {! q : P (suc fn) !}
```

Once we have generalised over `f n` we can no longer apply the lemma, which needs an argument of type `P (f n)`. To solve this problem we can add the lemma to the with-abstraction:

```
proof : (n : Nat) → Q (f n) → R
proof n q with f n     | lemma n
proof n ()    | zero    | _
proof n q     | suc fn | lem = lem q
```

In this case the type of `lemma n` (`P (f n) → R`) is generalised over `f n` so in the right-hand side of the last clause we have `q : P (suc fn)` and `lem : P (suc fn) → R`.

See *With-abstraction equality* below for an alternative approach.

Making with-abstractions hidden and/or irrelevant

It is possible to add hiding and relevance annotations to *with* expressions. For example:

```
module _ (A B : Set) (recompute : .B → .{{A}} → B) where

  _$_ : .(A → B) → .A → B
```

(continues on next page)

(continued from previous page)

```
  f $ x with .{f} | .(f x) | .{{x}}
  ... | y = recompute y
```

This can be useful for hiding with-abstractions that you do not need to match on but that need to be abstracted over for the result to be well-typed. It can also be used to abstract over the fields of a record type with irrelevant fields, for example:

```
record EqualBools : Set₁ where
  field
    bool1  : Bool
    bool2  : Bool
    .same : bool1 ≡ bool2
open EqualBools

example : EqualBools → EqualBools
example x with bool1 x | bool2 x | .(same x)
...      | true  | y′ | eq′ = record { bool1 = true;  bool2 = y′; same = eq′ }
...      | false | y′ | eq′ = record { bool1 = false; bool2 = y′; same = eq′ }
```

Using underscores and variables in pattern repetition

If an ellipsis ... cannot be used, the with-clause has to repeat (or refine) the patterns of the parent clause. Since Agda 2.5.3, such patterns can be replaced by underscores _ if the variables they bind are not needed. Here is a (slightly contrived) example:

```
record R : Set where
  coinductive -- disallows matching
  field  f  :  Bool
         n  :  Nat

data P (r : R) : Nat → Set where
  fTrue  :  R.f r ≡ true  →  P r zero
  nSuc   :                   P r (suc (R.n r))

data Q : (b : Bool) (n : Nat) →  Set where
  true!  :             Q true zero
  suc!   :  ∀{b n}  →  Q b (suc n)

test : (r : R) {n : Nat} (p : P r n) → Q (R.f r) n
test  r  nSuc       = suc!
test  r  (fTrue p)  with  R.f r
test  _  (fTrue ())    |  false
test  _  _             |  true  = true!  -- underscore instead of (isTrue _)
```

Since Agda 2.5.4, patterns can also be replaced by a variable:

```
f : List Nat → List Nat
f [] = []
f (x ∷ xs) with f xs
f xs0 | r = ?
```

The variable *xs0* is treated as a let-bound variable with value *.x* ∷ *.xs* (where *.x : Nat* and *.xs : List Nat* are out of scope). Since with-abstraction may change the type of variables, the instantiation of such let-bound variables are type checked

again after with-abstraction.

Irrefutable With

When a pattern is irrefutable, we can use a pattern matching `with` instead of a traditional `with` block. This gives us a lightweight syntax to make a lot of observations before using a "proper" `with` block. For a basic example of such an irrefutable pattern, see this unfolding lemma for `pred`

```
pred : Nat → Nat
pred zero    = zero
pred (suc n) = n

NotNull : Nat → Set
NotNull zero    = ⊥ -- false
NotNull (suc n) = ⊤ -- trivially true

pred-correct : ∀ n (pr : NotNull n) → suc (pred n) ≡ n
pred-correct n pr with suc p ← n = refl
```

In the above code snippet we do not need to entertain the idea that `n` could be equal to `zero`: Agda detects that the proof `pr` allows us to dismiss such a case entirely.

The patterns used in such an inversion clause can be arbitrary. We can for instance have deep patterns, e.g. projecting out the second element of a vector whose length is neither 0 nor 1:

```
infixr 5 _::_
data Vec {a} (A : Set a) : Nat → Set a where
  []  : Vec A zero
  _::_ : ∀ {n} → A → Vec A n → Vec A (suc n)

second : ∀ {n} {pr : NotNull (pred n)} → Vec A n → A
second vs with (_ :: v :: _) ← vs = v
```

Remember the example of *simultaneous abstraction* from above. A simultaneous rewrite / pattern matching `with` is to be understood as being nested. That is to say that the type refinements introduced by the first case analysis may be necessary to type the following ones.

In the following example, in `focusAt` we are only able to perform the `splitAt` we are interested in because we have massaged the type of the vector argument using `suc-+` first.

```
suc-+ : ∀ m n → suc m + n ≡ m + suc n
suc-+ zero    n                  = refl
suc-+ (suc m) n rewrite suc-+ m n = refl

infixr 1 _×_
_×_ : ∀ {a b} (A : Set a) (B : Set b) → Set _
A × B = Σ A (λ _ → B)

splitAt : ∀ m {n} → Vec A (m + n) → Vec A m × Vec A n
splitAt zero    xs       = ([] , xs)
splitAt (suc m) (x :: xs) with (ys , zs) ← splitAt m xs = (x :: ys , zs)

-- focusAt m (x₀ :: ⋯ :: x_{m-1} :: x_m :: x_{m+1} :: ⋯ :: x_{m+n})
-- returns ((x₀ :: ⋯ :: x_{m-1}) , x_m , (x_{m+1} :: ⋯ :: x_{m+n}))
focusAt : ∀ m {n} → Vec A (suc (m + n)) → Vec A m × A × Vec A n
```

(continues on next page)

(continued from previous page)

```
focusAt m {n} vs rewrite suc-+ m n
                 with (before , focus ∷ after) ← splitAt m vs
                 = (before , focus , after)
```

You can alternate arbitrarily many `rewrite` and pattern matching `with` clauses and still perform a `with` abstraction afterwards if necessary.

Left-hand side let-bindings

An alternative to an irrefutable `with`, when you just need to bind a variable or do simple unpacking of record values, is to use a `using`-binding. This is the left-hand side counterpart of a *let-binding* and supports the same limited form of pattern matching.

For instance, the irrefutable `with` used in `splitAt` in the section above can be changed to `using`:

```
splitAt : ∀ m {n} → Vec A (m + n) → Vec A m × Vec A n
splitAt zero    xs       = ([] , xs)
splitAt (suc m) (x ∷ xs) using (ys , zs) ← splitAt m xs = (x ∷ ys , zs)
```

Variables bound with `using` are in scope in following `with` clauses, allowing you to reuse bindings across multiple nested `with` s:

```
contrived : ∀ m {n} → Vec A (m + n) → (Vec A m → Bool) → (Vec A n → Bool) → Bool
contrived m xs p q using (ys , zs) ← splitAt m xs
                  with p ys
... | true = true
... | false with q zs
...   | true  = false
...   | false = true
```

For convenience, multiple bindings can be separated by `|`, and this has the same meaning as repeating the `using` keyword: bindings to the left are in scope to the right.

Contrary to `with` and `rewrite`, `using` does not perform any abstraction over the bound terms, but simply introduces a local binding. This can make it much cheaper to use than an irrefutable `with` in situations where the goal type and context are big and expensive to normalise, and the abstraction isn't required.

Rewrite

Remember example of *simultaneous abstraction* from above.

```
postulate plus-commute : (a b : Nat) → a + b ≡ b + a

thm : (a b : Nat) → P (a + b) → P (b + a)
thm a b t with   a + b  | plus-commute a b
thm a b t    | .(b + a) | refl = t
```

This pattern of rewriting by an equation by with-abstracting over it and its left-hand side is common enough that there is special syntax for it:

```
thm : (a b : Nat) → P (a + b) → P (b + a)
thm a b t rewrite plus-commute a b = t
```

The `rewrite` construction takes a term `eq` of type `lhs ≡ rhs`, where `_≡_` is the *built-in equality type*, and expands to a with-abstraction of `lhs` and `eq` followed by a match of the result of `eq` against `refl`:

```
f ps rewrite eq = v

  -->

f ps with lhs | eq
...    | .rhs | refl = v
```

One limitation of the `rewrite` construction is that you cannot do further pattern matching on the arguments *after* the rewrite, since everything happens in a single clause. You can however do with-abstractions after the rewrite. For instance,

```
postulate T : Nat → Set

isEven : Nat → Bool
isEven zero = true
isEven (suc zero) = false
isEven (suc (suc n)) = isEven n

thm₁ : (a b : Nat) → T (a + b) → T (b + a)
thm₁ a b t rewrite plus-commute a b with isEven a
thm₁ a b t | true  = t
thm₁ a b t | false = t
```

Note that the with-abstracted arguments introduced by the rewrite (`lhs` and `eq`) are not visible in the code.

With-abstraction equality

When you with-abstract a term `t` you lose the connection between `t` and the new argument representing its value. That's fine as long as all instances of `t` that you care about get generalised by the abstraction, but as we saw *above* this is not always the case. In that example we used simultaneous abstraction to make sure that we did capture all the instances we needed.

An alternative to that is to get Agda to remember in an equality proof that the patterns in the with clauses come from the expression you abstracted over. This is possible using the `in` keyword.

In the following artificial example, we try to prove that there exists two numbers such that one equals the double of the other. We start by computing the double of our input `m` and call it `n`. We can then return the nested pair containing `m`, `n`, and we now need a proof that `m + m ≡ n`. Luckily we used `in eq` when computing `n` as `m + m` and this `eq` is exactly the proof we need.

```
double : Nat → Σ Nat (λ m → Σ Nat (λ n → m + m ≡ n))
double m with n ← m + m in eq = m , n , eq
```

For a more natural example, we prove that `filter` (defined at the top of this page) is idempotent. That is to say that applying it twice to an input list is the same as only applying it once.

In the `filter-filter p (x ∷ xs)` case, abstracting over and then matching on the result of `p x` allows the first call to `filter p (x ∷ xs)` to reduce.

In case the element `x` is kept (i.e. `p x` is `true`), the second call to `filter` on the LHS goes on to performs the same `p x` test. Because we have retained the proof that `p x ≡ true` in `eq`, we are able to rewrite by this equality and get it to reduce too.

This leads to just enough computation that we can finish the proof with an appeal to congruence and the induction hypothesis.

```
filter-filter : ∀ {A} p (xs : List A) → filter p (filter p xs) ≡ filter p xs
filter-filter p []       = refl
filter-filter p (x ∷ xs) with p x in eq
... | false = filter-filter p xs -- easy
... | true -- second filter stuck on `p x`: rewrite by `eq`!
  rewrite eq = cong (x ∷_) (filter-filter p xs)
```

Alternatives to with-abstraction

Although with-abstraction is very powerful there are cases where you cannot or don't want to use it. For instance, you cannot use with-abstraction if you are inside an expression in a right-hand side. In that case there are a couple of alternatives.

Pattern lambdas

Agda does not have a primitive `case` construct, but one can be emulated using *pattern lambdas*. First you define a function `case_of_` as follows:

```
case_of_ : ∀ {a b} {A : Set a} {B : Set b} → A → (A → B) → B
case x of f = f x
```

You can then use this function with a pattern lambda as the second argument to get a Haskell-style case expression:

```
filter : {A : Set} → (A → Bool) → List A → List A
filter p [] = []
filter p (x ∷ xs) =
  case p x of
  λ { true  → x ∷ filter p xs
    ; false → filter p xs
    }
```

This version of `case_of_` only works for non-dependent functions. For dependent functions the target type will in most cases not be inferrable, but you can use a variant with an explicit `B` for this case:

```
case_returning_of_ : ∀ {a b} {A : Set a} (x : A) (B : A → Set b) → (∀ x → B x) → B x
case x returning B of f = f x
```

The dependent version will let you generalise over the scrutinee, just like a with-abstraction, but you have to do it manually. Two things that it will not let you do is

- further pattern matching on arguments on the left-hand side, and
- refine arguments on the left by the patterns in the case expression. For instance if you matched on a `Vec A n` the `n` would be refined by the nil and cons patterns.

Helper functions

Internally with-abstractions are translated to auxiliary functions (see *Technical details* below) and you can always write these functions manually. The downside is that the type signature for the helper function needs to be written out explicitly, but fortunately the *Emacs Mode* has a command (`C-c C-h`) to generate it using the same algorithm that generates the type of a with-function.

Termination checking

The termination checker runs on the translated auxiliary functions, which means that some code that looks like it should pass termination checking does not. Specifically this happens in call chains like c_1 `(`c_2 `x)` $\longrightarrow$ c_1 `x` where the recursive call is under a with-abstraction. The reason is that the auxiliary function only gets passed `x`, so the call chain is actually c_1 `(`c_2 `x)` $\longrightarrow$ `x` $\longrightarrow$ c_1 `x`, and the termination checker cannot see that this is terminating. For example:

```
data D : Set where
  [_] : Nat → D
```

```
fails : D → Nat
fails [ zero  ] = zero
fails [ suc n ] with some-stuff
... | _ = fails [ n ]
```

The easiest way to work around this problem is to perform a with-abstraction on the recursive call up front:

```
fixed : D → Nat
fixed [ zero  ] = zero
fixed [ suc n ] with fixed [ n ] | some-stuff
... | rec | _ = rec
```

If the function takes more arguments you might need to abstract over a partial application to just the structurally recursive argument. For instance,

```
fails : Nat → D → Nat
fails _ [ zero  ] = zero
fails _ [ suc n ] with some-stuff
... | m = fails m [ n ]

fixed : Nat → D → Nat
fixed _ [ zero  ] = zero
fixed _ [ suc n ] with (λ m → fixed m [ n ]) | some-stuff
... | rec | m = rec m
```

A possible complication is that later with-abstractions might change the type of the abstracted recursive call:

```
T      : D → Set
suc-T  : ∀ {n} → T [ n ] → T [ suc n ]
zero-T : T [ zero ]
```

```
fails : (d : D) → T d
fails [ zero  ] = zero-T
fails [ suc n ] with some-stuff
... | _ with [ n ]
...   | z = suc-T (fails [ n ])
```

Trying to abstract over the recursive call as before does not work in this case.

```
still-fails : (d : D) → T d
still-fails [ zero ] = zero-T
still-fails [ suc n ] with still-fails [ n ] | some-stuff
... | rec | _ with [ n ]
...   | z = suc-T rec -- Type error because rec : T z
```

To solve the problem you can add `rec` to the with-abstraction messing up its type. This will prevent it from having its type changed:

```
fixed : (d : D) → T d
fixed [ zero ] = zero-T
fixed [ suc n ] with fixed [ n ] | some-stuff
... | rec | _ with rec | [ n ]
...   | _ | z = suc-T rec
```

Performance considerations

The *generalisation step* of a with-abstraction needs to normalise the scrutinee and the goal and argument types to make sure that all instances of the scrutinee are generalised. The generalisation also needs to be type checked to make sure that it's not *ill-typed*. This makes it expensive to type check a with-abstraction if

- the normalisation is expensive,
- the normalised form of the goal and argument types are big, making finding the instances of the scrutinee expensive,
- type checking the generalisation is expensive, because the types are big, or because checking them involves heavy computation.

In these cases it is worth looking at the *alternatives to with-abstraction* from above.

3.50.2 Technical details

Internally with-abstractions are translated to auxiliary functions—there are no with-abstractions in the *Core language*. This translation proceeds as follows. Given a with-abstraction

$$\begin{aligned} &f : \Gamma \to B \\ &f\ ps \quad \textbf{with}\ t_1 \quad | \ldots | \ t_m \\ &f\ ps_1 \qquad | \ q_{11} \ | \ldots | \ q_{1m} = v_1 \\ &\vdots \\ &f\ ps_n \qquad | \ q_{n1} \ | \ldots | \ q_{nm} = v_n \end{aligned}$$

where $\Delta \vdash ps : \Gamma$ (i.e. Δ types the variables bound in ps), we

- Infer the types of the scrutinees $t_1 : A_1, \ldots, t_m : A_m$.
- Partition the context Δ into Δ_1 and Δ_2 such that Δ_1 is the smallest context where $\Delta_1 \vdash t_i : A_i$ for all i, i.e., where the scrutinees are well-typed. Note that the partitioning is not required to be a split, $\Delta_1\Delta_2$ can be a (well-formed) reordering of Δ.
- Generalise over the t_i s, by computing

  $$C = (w_1 : A_1)(w_1 : A_2') \ldots (w_m : A_m') \to \Delta_2' \to B'$$

  such that the normal form of C does not contain any t_i and

  $$\begin{aligned} A_i'[w_1 := t_1 \ldots w_{i-1} := t_{i-1}] &\simeq A_i \\ (\Delta_2' \to B')[w_1 := t_1 \ldots w_m := t_m] &\simeq \Delta_2 \to B \end{aligned}$$

  where $X \simeq Y$ is equality of the normal forms of X and Y. The type of the auxiliary function is then $\Delta_1 \to C$.
- Check that $\Delta_1 \to C$ is type correct, which is not guaranteed (see *below*).

- Add a function f_{aux}, mutually recursive with f, with the definition

$$\begin{aligned} &f_{aux} : \Delta_1 \to C \\ &f_{aux}\ ps_{11}\ \ qs_1\ ps_{21} = v_1 \\ &\vdots \\ &f_{aux}\ ps_{1n}\ qs_n\ ps_{2n} = v_n \end{aligned}$$

  where $qs_i = q_{i1} \ldots q_{im}$, and $ps_{1i} : \Delta_1$ and $ps_{2i} : \Delta_2$ are the patterns from ps_i corresponding to the variables of ps. Note that due to the possible reordering of the partitioning of Δ into Δ_1 and Δ_2, the patterns ps_{1i} and ps_{2i} can be in a different order from how they appear ps_i.

- Replace the with-abstraction by a call to f_{aux} resulting in the final definition

$$\begin{aligned} &f : \Gamma \to B \\ &f\ ps = f_{aux}\ xs_1\ ts\ xs_2 \end{aligned}$$

  where $ts = t_1 \ldots t_m$ and xs_1 and xs_2 are the variables from Δ corresponding to Δ_1 and Δ_2 respectively.

Examples

Below are some examples of with-abstractions and their translations.

```
postulate
   A     : Set
   _+_   : A → A → A
   T     : A → Set
   mkT   : ∀ x → T x
   P     : ∀ x → T x → Set

-- the type A of the with argument has no free variables, so the with
-- argument will come first
f₁ : (x y : A) (t : T (x + y)) → T (x + y)
f₁ x y t with x + y
f₁ x y t    | w = {!!}

-- Generated with function
f-aux₁ : (w : A) (x y : A) (t : T w) → T w
f-aux₁ w x y t = {!!}

-- x and p are not needed to type the with argument, so the context
-- is reordered with only y before the with argument
f₂ : (x y : A) (p : P y (mkT y)) → P y (mkT y)
f₂ x y p with mkT y
f₂ x y p    | w = {!!}

f-aux₂ : (y : A) (w : T y) (x : A) (p : P y w) → P y w
f-aux₂ y w x p = {!!}

postulate
  H : ∀ x y → T (x + y) → Set

-- Multiple with arguments are always inserted together, so in this case
-- t ends up on the left since it's needed to type h and thus x + y isn't
-- abstracted from the type of t
f₃ : (x y : A) (t : T (x + y)) (h : H x y t) → T (x + y)
```

(continues on next page)

(continued from previous page)

```
f₃ x y t h with x + y | h
f₃ x y t h    | w₁    | w₂ = {! t : T (x + y), goal : T w₁ !}

f-aux₃ : (x y : A) (t : T (x + y)) (h : H x y t) (w₁ : A) (w₂ : H x y t) → T w₁
f-aux₃ x y t h w₁ w₂ = {!!}

-- But earlier with arguments are abstracted from the types of later ones
f₄ : (x y : A) (t : T (x + y)) → T (x + y)
f₄ x y t with x + y | t
f₄ x y t    | w₁    | w₂ = {! t : T (x + y), w₂ : T w₁, goal : T w₁ !}

f-aux₄ : (x y : A) (t : T (x + y)) (w₁ : A) (w₂ : T w₁) → T w₁
f-aux₄ x y t w₁ w₂ = {!!}

-- With-abstraction equality
g : (x : A) → T x
g x with mkT x in eq
g x | w = {!!}

g-aux : (x : A) (w : T x) → mkT x ≡ w → T x
g-aux x w eq = {!!}

-- The equality argument is generalised over by further with-abstractions
g₁ : (x : A) → P x (g x)
g₁ x with mkT x in eq
g₁ x | w = {!!}

g-aux₁ : (x : A) (w : T x) (eq : mkT x ≡ w) → P x (g-aux x w eq)
g-aux₁ x w eq = {!!}
```

Ill-typed with-abstractions

As mentioned above, generalisation does not always produce well-typed results. This happens when you abstract over a term that appears in the *type* of a subterm of the goal or argument types. The simplest example is abstracting over the first component of a dependent pair. For instance,

```
postulate
  A : Set
  B : A → Set
  H : (x : A) → B x → Set
```

```
bad-with : (p : Σ A B) → H (fst p) (snd p)
bad-with p with fst p
...           | _ = {!!}
```

Here, generalising over `fst p` results in an ill-typed application `H w (snd p)` and you get the following type error:

```
fst p != w of type A
when checking that the type (p : Σ A B) (w : A) → H w (snd p) of
the generated with function is well-formed
```

This message can be a little difficult to interpret since it only prints the immediate problem (`fst p != w`) and the full type of the with-function. To get a more informative error, pointing to the location in the type where the error is, you

can copy and paste the with-function type from the error message and try to type check it separately.

3.51 Without K

The option *--without-K* adds some restrictions to Agda's typechecking algorithm in order to ensure compatability with versions of type theory that do not support UIP (uniqueness of identity proofs), such as HoTT (homotopy type theory).

The option *--with-K* can be used to override a global *--without-K* in a file, by adding a pragma `{-# OPTIONS --with-K #-}`. This option is enabled by default.

Note

Prior to Agda 2.6.3, the *--cubical-compatible* flag did not exist, and *--without-K* also implied the (internal) generation of Cubical Agda-specific code. See *Cubical compatible* for the specifics, and *#5843 <https://github.com/agda/agda/issues/5843>* for the rationale.

Note

When *–without-K* is used, it is not safe to postulate erased univalence: the theory is perhaps consistent, but one can get incorrect results at run-time. You should use the *Cubical compatible* flag instead. See *#4784 <https://github.com/agda/agda/issues/4784>* for more details on this restriction.

3.51.1 Restrictions on pattern matching

When the option *--without-K* is enabled, then Agda only accepts certain case splits. More specifically, the unification algorithm for checking case splits cannot make use of the deletion rule to solve equations of the form `x = x`.

For example, the obvious implementation of the K rule is not accepted:

```
K : {A : Set} {x : A} (P : x ≡ x → Set) →
    P refl → (x≡x : x ≡ x) → P x≡x
K P p refl = p
```

Pattern matching with the constructor `refl` on the argument `x≡x` causes `x` to be unified with `x`, which fails because the deletion rule cannot be used when *--without-K* is enabled.

On the other hand, the obvious implementation of the J rule is accepted:

```
J : {A : Set} (P : (x y : A) → x ≡ y → Set) →
    ((x : A) → P x x refl) → (x y : A) (x≡y : x ≡ y) → P x y x≡y
J P p x .x refl = p x
```

Pattern matching with the constructor `refl` on the argument `x≡y` causes `x` to be unified with `y`. The same applies to Christine Paulin-Mohring's version of the J rule:

```
J´ : {A : Set} {x : A} (P : (y : A) → x ≡ y → Set) →
     P x refl → (y : A) (x≡y : x ≡ y) → P y x≡y
J´ P p ._ refl = p
```

For more details, see Jesper Cockx's PhD thesis *Dependent Pattern Matching and Proof-Relevant Unification* [Cockx (2017)].

3.51.2 Restrictions on termination checking

When *--without-K* is enabled, Agda's termination checker restricts structural descent to arguments ending in data types or `Size`. Likewise, guardedness is only tracked when result type is data or record type:

```
data ⊥ : Set where

mutual
  data WOne : Set where wrap : FOne → WOne
  FOne = ⊥ → WOne

postulate iso : WOne ≡ FOne

noo : (X : Set) → (WOne ≡ X) → X → ⊥
noo .WOne refl (wrap f) = noo FOne iso f
```

`noo` is rejected since at type `X` the structural descent `f < wrap f` is discounted *--without-K*:

```
data Pandora : Set where
  C : ∞ ⊥ → Pandora

postulate foo : ⊥ ≡ Pandora

loop : (A : Set) → A ≡ Pandora → A
loop .Pandora refl = C (♯ (loop ⊥ foo))
```

`loop` is rejected since guardedness is not tracked at type `A` *--without-K*.

See issues #1023, #1264, #1292.

3.51.3 Restrictions on universe levels

When *--without-K* is enabled, some indexed datatypes must be defined in a higher universe level. In particular, the types of all indices should fit in the sort of the datatype.

For example, usually (i.e. *--with-K*) Agda allows the following definition of equality:

```
data _≡₀_ {ℓ} {A : Set ℓ} (x : A) : A → Set where
  refl : x ≡₀ x
```

However, with *--without-K* it must be defined at a higher universe level:

```
data _≡′_ {ℓ} {A : Set ℓ} : A → A → Set ℓ where
  refl : {x : A} → x ≡′ x
```

CHAPTER

FOUR

TOOLS

4.1 Automatic Proof Search (Auto)

Agda supports (since version 2.2.6) the command `Auto`, that searches for type inhabitants and fills a hole when one is found. The type inhabitant found is not necessarily unique.

Auto can be used as an aid when interactively constructing terms in Agda. In a system with dependent types it can be meaningful to use such a tool for finding fragments of, not only proofs, but also programs. One should not expect it to handle large problems of any particular kind, but small enough problems of almost any kind.

Any solution coming from Auto is checked by Agda. Also, the main search algorithm has a timeout mechanism. Therefore, there is little harm in trying Auto and it might save you key presses.

Auto was completely rewritten for Agda version 2.7.0.

4.1.1 Usage

The tool is invoked by placing the cursor on a hole and choosing `Auto` in the goal menu or pressing `C-c C-a`. Auto's behaviour can be changed by using various options which are passed directly in the hole.

Option	Meaning
`-t` *N*	Set timeout to *N* seconds
ID	Use definition *ID* as a hint
`-m`	Use the definitions in the current module as hints
`-u`	Use the unqualified definitions in scope as hints
`-l`	List up to ten solutions, does not commit to any
`-s` *N*	Skip the *N* first solutions

Giving no arguments is fine and results in a search with default parameters. The search carries on until either a (not necessarily unique) solution is found, the search space is fully (and unsuccessfully) explored or it times out (one second by default). Here follows a list of the different modes and parameters.

Hints

Auto does not by default try using constants in scope. If there is a lemma around that might help in constructing the term you can include it in the search by giving hints. There are two ways of doing this. One way is to provide the exact list of constants to include. Such a list is given by writing a number of constant names separated by space: `<hint1> <hint2> ...`.

You can also use `-m` to use all constants defined in the innermost module containing the current hole, or `-u` to use all constants that are in scope unqualified. Both options can be combined with an explicit list of named constants.

There are a few exceptions to what you have to specify as hints:

- Datatypes and constants that can be deduced by unifying the two sides of an equality constraint can be omitted.

  E.g., if the constraint `? = List A` occurs during the search, then refining `?` to `List ...` will happen without having to provide `List` as a hint. The constants that you can leave out overlap more or less with the ones appearing in hidden arguments, i.e. you wouldn't have written them when giving the term by hand either.
- Constructors and projection functions are automatically tried, so should never be given as hints.
- Recursive calls, although currently only the function itself, not all functions in the same mutual block.

Timeout

The timeout is one second by default but can be changed by adding `-t <n>` to the parameters, where <n> is the number of seconds.

Listing and choosing among several solutions

Normally, Auto replaces the hole with the first solution found. If you are not happy with that particular solution, you can list the ten (at most) first solutions encountered by including the flag `-l`.

You can then pick a particular solution by writing `-s <n>` where <n> is the number of solutions to skip (as well as the number appearing before the solution in the list). The options `-l` and `-s <n>` can be combined to list solutions other than the ten first ones.

Dependencies between meta variables

The following feature is missing from Agda 2.7.0's implementation of Auto: *If the goal type or type of local variables contain meta variables, then the constraints for these are also included in the search. If a solution is found it means that Auto has also found solutions for the occurring meta variables. Those solutions will be inserted into your file along with that of the hole from where you called Auto. Also, any unsolved equality constraints that contain any of the involved meta variables are respected in the search.*

4.1.2 Limitations

- Literals other than natural numbers are not supported.

4.1.3 User feedback

When sending bug reports, please use Agda's bug tracker. Apart from that, receiving nice examples (via the bug tracker) would be much appreciated. Both such examples which Auto does not solve, but you have a feeling it's not larger than for that to be possible. And examples that Auto only solves by increasing timeout. The examples sent in will be used for tuning the heuristics and hopefully improving the performance.

4.2 Command-line options

4.2.1 Command-line options

Agda accepts the following options on the command line. Where noted, these options can also serve as *pragma options*, i.e., be supplied in a file via the `{-# OPTIONS ... #-}` pragma or in the `flags` section of an `.agda-lib` file.

Setup and information

Some options cause Agda to perform tasks at startup like mandatory setup or printing some information. These options are not exclusive, they can be used with other options, albeit this seldom makes sense. They are not executed in the order given on the command line, but in the fixed order listed in the following:

--setup

Added in version 2.8.0.

Extract Agda's data files (primitive library, emacs mode etc.) to the data directory (see *--print-agda-data-dir*).

--version, -V

Show version number and cabal flags used in this build of Agda.

Overwrites *--numeric-version*.

--numeric-version

Show just the version number.

Overwrites *--version*.

--help[={TOPIC}], **-?[{TOPIC}]**

Show basically this help, or more help about `TOPIC`. Available topics:

- `emacs-mode`: Explain the option *--emacs-mode*.
- `error`: List the names of Agda's errors.
- `warning`: List warning groups and individual warnings and their default status. Instruct how to toggle benign warnings.

Overwrites itself, i.e., only the last of several *--help* options is effective.

--print-options

Added in version 2.9.0.

Print a simple list of all options, suitable for implementing bash completion.

--build-library

Added in version 2.8.0.

Expects an `.agda-lib` file in the current directory (or in a parent directory) and type-checks all Agda files found in the `include` directories of the library or in subdirectories thereof.

--print-agda-app-dir

Added in version 2.6.4.1.

Outputs the (`AGDA_DIR`) directory containing Agda's application configuration files, such as the `defaults` and `libraries` files, as described in *Library Management*.

--print-agda-dir

Added in version 2.6.2.

Alias of *--print-agda-data-dir*.

--print-agda-data-dir

Added in version 2.6.4.1.

Outputs the root of the directory structure holding Agda's data files such as core libraries, style files for the backends, etc.

Since 2.8.0, the data directory is determined as follows:

- The *default data directory* is defined at build time, either as the standard data directory defined by Cabal, or, if the `use-xdg-data-home` Cabal flag is enabled, as `$XDG_DATA_HOME/agda/$AGDA_VERSION`.
- The *data directory* can be set at runtime using the `Agda_datadir` environment variable and defaults to the default data directory. It can be printed with this flag.

`--emacs-mode={COMMAND}`

Added in version 2.8.0.

Administer the Agda Emacs mode, a task previously managed by the `agda-mode` executable.

Available commands:

- `setup`: Install the Emacs mode into `.emacs`.
- `compile`: Compile the Elisp files of the Emacs mode.
- `locate`: Print the path to the Emacs mode.

More information in *Section Emacs*.

This option can be given several times to perform several commands.

General options

`--interaction`

For use with the Emacs mode (no need to invoke yourself).

`--parallel[=N], -j[N]`

Added in version 2.9.0.

Type check in parallel. N is optional, and controls the number of threads to use for checking. If `N` is omitted, or explicitly set to 0, the number of processors is used. The default, `-j1`, means type checking is sequential. Parallelism has module granularity.

Parallel type checking trades space for time, with (generally) a decrease in wall-clock time *larger* than the corresponding increase in total memory usage. Type-checking of a module can not start until all of its dependencies have been checked, so wider dependency graphs will see more speedup than taller dependency graphs.

Using *too many* cores can make the space-time tradeoff worse. The appropriate number will depend on the specifics of the project being checked.

A reasonable number to start with is `-j8`.

`--interaction-json`

Added in version 2.6.1.

For use with other editors such as Atom (no need to invoke yourself).

`--interaction-exit-on-error`

Added in version 2.6.3.

Makes Agda exit with a non-zero exit code if *`--interaction`* or *`--interaction-json`* are used and a type error is encountered. The option also makes Agda exit with exit code 113 if Agda fails to parse a command.

This option might for instance be used if Agda is controlled from a script.

`--interactive, -I`

Start in interactive mode (not maintained).

`--trace-imports[=(0|1|2|3)]`

Added in version 2.6.4.

Configure printing of messages when an imported module is accessed during type-checking.

`0`	Do not print any messages about checking a module.
`1`	Print only *Checking ...* when an access to an uncompiled module occurs. This is the default behavior if `--trace-imports` is not specified.
`2`	Use the effect of `1`, but also print *Finished ...* when a compilation of an uncompiled module is finished. This is the behavior if `--trace-imports` is specified without a value.
`3`	Use the effect of `2`, but also print *Loading ...* when a compiled module (interface) is accessed during the type-checking.

`--colour``[=(auto|always|never)]`, **`--color`**`[=(auto|always|never)]`

Added in version 2.6.4: Configure whether or not Agda's standard output diagnostics should use ANSI terminal colours for syntax highlighting (e.g. error messages, warnings).

`always`	Always print diagnostic in colour.
`auto`	Automatically determine whether or not it is safe for standard output to include colours. Colours will be used when writing directly to a terminal device on Linux and macOS. This is the default value.
`never`	Never print output in colour.

The American spelling, `--color`, is also accepted.

Note: Currently, the colour scheme for terminal output can not be configured. If the colours are not legible on your terminal, please use `--colour=never` for now.

`--only-scope-checking`

Added in version 2.5.3.

Only scope-check the top-level module, do not type-check it (see *Quicker generation without typechecking*).

`--transliterate`

Added in version 2.6.3.

When writing to stdout or stderr Agda will (hopefully) replace code points that are not supported by the current locale or code page by something else, perhaps question marks.

This option is not supported when *`--interaction`* or *`--interaction-json`* are used, because when those options are used Agda uses UTF-8 when writing to stdout (and when reading from stdin).

Literate programming

`--literate-markdown-only-agda-blocks`

Added in version 2.9.0.

In literate Markdown (`.lagda.md`) and Typst (`.lagda.typ`) files, only treat code blocks explicitly marked with ```` ```agda ```` as Agda code. Unmarked code blocks (```` ``` ````) are treated as verbatim text and are not type-checked.

See *Only agda code blocks* for more details.

--no-literate-markdown-only-agda-blocks

Added in version 2.9.0.

Treat also unmarked code blocks as Agda code in literate Markdown and Typst files (default).

Compilation

See *Compilers* for backend-specific options.

--compile-dir=DIR

Set DIR as directory for compiler output (default: the project root).

--no-main

Do not treat the requested/current module as the main module of a program when compiling.

Pragma option since 2.5.3.

--main

Added in version 2.6.4.

Default, opposite of *--no-main*.

Generating highlighted source code

--dependency-graph=FILE

Added in version 2.3.0.

Generate a Dot file FILE with a module dependency graph.

--dependency-graph-include=LIBRARY

Added in version 2.6.3.

Include modules from the given library in the dependency graph. This option can be used multiple times to include modules from several libraries. If this option is not used at all, then all modules are included. (Note that the module given on the command line might not be included.)

A module M is considered to be in the library L if L is the name of an .agda-lib file *associated* to M (even if M's file cannot be found via the include paths given in the .agda-lib file).

--html

Added in version 2.2.0.

Generate HTML files with highlighted source code (see *Generating HTML* for description and further options).

--latex

Added in version 2.3.2.

Generate LaTeX with highlighted source code (see *Generating LaTeX* for description and further options).

--vim

Generate Vim highlighting files.

Imports and libraries

(see *Library Management*)

--ignore-all-interfaces

Added in version 2.6.0.

Don't read *any* interface files, including builtin and primitive modules; only use this if you know what you are doing!

`--ignore-interfaces`
: Don't read interface files (re-type check everything, except for builtin and primitive modules).

`--no-write-interfaces`
: Added in version 2.9.0.

  Don't write out interface files after type-checking a module.

`--include-path`=`{DIR}`, `-i` `{DIR}`
: Look for imports in `DIR`. This option can be given multiple times.

`--library`=`{DIR}`, `-l` `{LIB}`
: Added in version 2.5.1.

  Use library `LIB`.

`--library-file`=`{FILE}`
: Added in version 2.5.1.

  Use `FILE` instead of the standard `libraries` file.

`--no-default-libraries`
: Added in version 2.5.1.

  Don't use default library files.

`--no-libraries`
: Added in version 2.5.2.

  Don't use any library files.

4.2.2 Command-line and pragma options

The following options can also be given in Agda files using the *OPTIONS* pragma.

Performance

`--auto-inline`
: Added in version 2.6.2.

  Turn on automatic compile-time inlining. See *The INLINE and NOINLINE pragmas* for more information.

`--no-auto-inline`
: Added in version 2.5.4.

  Disable automatic compile-time inlining (default). Only definitions marked `INLINE` will be inlined. Default since 2.6.2.

`--caching`, `--no-caching`
: Added in version 2.5.4.

  Enable or disable caching of typechecking.

  Default: `--caching`.

`--call-by-name`
: Added in version 2.6.2.

  Disable call-by-need evaluation in the Agda Abstract Machine.

`--no-call-by-name`

Added in version 2.6.4.

Default, opposite of *--call-by-name*.

`--no-fast-reduce`

Added in version 2.6.0.

Disable reduction using the Agda Abstract Machine.

`--fast-reduce`

Added in version 2.6.4.

Default, opposite of *--no-fast-reduce*.

`--no-forcing`

Added in version 2.2.10.

Disable the forcing optimisation. Since Agda 2.6.1 it is a pragma option.

`--forcing`

Added in version 2.6.4.

Default, opposite of *--no-forcing*.

`--no-projection-like`

Added in version 2.6.1.

Turn off the analysis whether a type signature likens that of a projection.

Projection-likeness is an optimization that reduces the size of terms by dropping parameter-like reconstructible function arguments. Thus, it is advisable to leave this optimization on, the flag is meant for debugging Agda.

See also the *NOT_PROJECTION_LIKE* pragma.

`--projection-like`

Added in version 2.6.4.

Default, opposite of *--no-projection-like*.

Printing and debugging

`--no-unicode`

Added in version 2.5.4.

Do not use unicode characters to print terms.

`--unicode`

Added in version 2.6.4.

Default, opposite of *--no-unicode*.

`--show-identity-substitutions`

Added in version 2.6.2.

Show all arguments of metavariables when pretty-printing a term, even if they amount to just applying all the variables in the context.

`--no-show-identity-substitutions`

Added in version 2.6.4.

Default, opposite of *--show-identity-substitutions*.

--show-implicit

Show implicit arguments when printing.

--no-show-implicit

Added in version 2.6.4.

Default, opposite of *--show-implicit*.

--show-irrelevant

Added in version 2.3.2.

Show irrelevant arguments when printing.

--no-show-irrelevant

Added in version 2.6.4.

Default, opposite of *--show-irrelevant*.

--verbose={N}, **-v** {N}

Set verbosity level to N. This only has an effect if Agda was installed with the debug Cabal flag. See *Debugging* for more information.

--profile={PROF}

Added in version 2.6.3.

Turn on profiling option PROF. Available options are

internal	Measure time taken by various parts of the system (type checking, serialization, etc)
modules	Measure time spent on individual (Agda) modules
definitions	Measure time spent on individual (Agda) definitions
sharing	Measure things related to sharing
serialize	Collect detailed statistics about serialization
constraints	Collect statistics about constraint solving
metas	Count number of created metavariables
interactive	Measure time of interactive commands
conversion	Count number of times various steps of the conversion algorithm are used (reduction, eta-expansion, syntactic equality, etc)

Only one of internal, modules, and definitions can be turned on at a time. You can also give --profile=all to turn on all profiling options (choosing internal over modules and definitions, use --profile=modules --profile=all to pick modules instead).

Copatterns and projections

--copatterns, --no-copatterns

Added in version 2.4.0.

Enable or disable definitions by copattern matching (see *Copatterns*).

Default: --copatterns (since 2.4.2.4).

--postfix-projections

Added in version 2.5.2.

Make postfix projection notation the default. On by default since 2.7.0.

--no-postfix-projections

Added in version 2.6.4.

Opposite of *--postfix-projections*.

Experimental features

--allow-exec

Added in version 2.6.2.

Enable system calls during type checking (see *Reflection*).

--no-allow-exec

Added in version 2.6.4.

Default, opposite of *--allow-exec*.

--confluence-check, --local-confluence-check, --no-confluence-check

Added in version 2.6.1.

Enable optional (global or local) confluence checking of REWRITE rules (see *Confluence checking*).

Default is *--no-confluence-check*.

--cubical, --cubical=full

Added in version 2.6.0.

Enable cubical features. Turns on *--cubical=compatible* and *--without-K* (see *Cubical*).

--cubical=erased, **--erased-cubical**

Added in version 2.6.3.

Enable a *variant* of Cubical Agda, and turn on *--cubical=compatible* and *--without-K*.

--cubical=no-glue

Added in version 2.9.0.

Enable a *variant* of Cubical Agda without the *Glue types*. Turns on *--cubical=compatible* and *--without-K*.

--experimental-irrelevance

Added in version 2.3.0.

Enable potentially unsound irrelevance features (irrelevant levels, irrelevant data matching) (see *Irrelevance*).

Implies *--irrelevance*.

--no-experimental-irrelevance

Added in version 2.6.4.

Default, opposite of *--experimental-irrelevance*.

--guarded

Added in version 2.6.2.

Enable locks and ticks for guarded recursion (see *Guarded Type Theory*).

--no-guarded

Added in version 2.6.4.

Default, opposite of *--guarded*.

`--injective-type-constructors`

Added in version 2.2.8.

Enable injective type constructors.

This makes Agda anti-classical: injective type constructors are incompatible with the law of excluded middle, see theorem 93 (attributed to Chung-Kil Hur) by Cockx and Devriese *[1]*.

It is also incompatible with univalence, by theorem 92 of the same paper.

Additionally, this possibly makes Agda inconsistent.

`--no-injective-type-constructors`

Added in version 2.6.4.

Default, opposite of *--injective-type-constructors*.

`--irrelevant-projections, --no-irrelevant-projections`

Added in version 2.5.4.

Enable [disable] projection of irrelevant record fields (see *Irrelevance*). The option `--irrelevant-projections` makes Agda inconsistent.

Implies *--irrelevance*.

Default (since version 2.6.1): `--no-irrelevant-projections`.

`--lossy-unification, --no-lossy-unification`

Added in version 2.6.2.

Enable a constraint-solving heuristic akin to first-order unification, see *Lossy Unification*. Implies *--no-require-unique-meta-solutions*.

`--no-lossy-unification`

Added in version 2.6.4.

Default, opposite of *--lossy-unification*.

`--require-unique-meta-solutions, --no-require-unique-meta-solutions`

Added in version 2.7.0.

When turned off, type checking is allowed to use heuristics to solve meta variables that do not necessarily guarantee unique solutions. In particular, it can make use of *INJECTIVE_FOR_INFERENCE* pragmas.

`--no-require-unique-meta-solutions` is implied by the *--lossy-unification* flag.

Default: `--require-unique-meta-solutions`

`--irrelevance, --no-irrelevance`

Added in version 2.9.0.

Enable or disable declaration and use of irrelevant function spaces, record fields, and declarations (see *irrelevance*).

Default: `--irrelevance`.

`--prop, --no-prop`

Added in version 2.6.0.

Enable or disable declaration and use of definitionally proof-irrelevant propositions (see *proof-irrelevant propositions*).

Default: `--no-prop`. In this case, `Prop` is since 2.6.4 not in scope by default (*--import-sorts*).

`--rewriting`

Added in version 2.4.2.4.

Enable declaration and use of REWRITE rules (see *Rewriting*).

`--no-rewriting`

Added in version 2.6.4.

Default, opposite of `--rewriting`.

`--local-rewriting`

Added in version 2.9.0.

Enable declaring local rewrite rules with the `@rewrite` attribute (see *Local Rewriting*).

`--no-local-rewriting`

Added in version 2.9.0.

Default, opposite of `--local-rewriting`.

`--two-level`

Added in version 2.6.2.

Enable the use of strict (non-fibrant) type universes `SSet` *(two-level type theory)*. Since 2.6.4, brings `SSet` into scope unless `--no-import-sorts`.

`--no-two-level`

Added in version 2.6.4.

Default, opposite of `--two-level`.

Errors and warnings

`--allow-incomplete-matches`

Added in version 2.6.1.

Succeed and create interface file regardless of incomplete pattern-matching definitions. See also the *NON_COVERING* pragma.

`--no-allow-incomplete-matches`

Added in version 2.6.4.

Default, opposite of `--allow-incomplete-matches`.

`--allow-unsolved-metas`

Succeed and create interface file regardless of unsolved meta variables (see *Metavariables*).

`--no-allow-unsolved-metas`

Added in version 2.6.4.

Default, opposite of `--allow-unsolved-metas`.

`--no-positivity-check`

Do not warn about not strictly positive data types (see *Positivity Checking*).

`--positivity-check`

Added in version 2.6.4.

Default, opposite of `--no-positivity-check`.

`--no-termination-check`
: Do not warn about possibly nonterminating code (see *Termination Checking*).

`--termination-check`
: Added in version 2.6.4.

  Default, opposite of `--no-termination-check`.

`--warning={GROUP|FLAG}, -W {GROUP|FLAG}`
: Added in version 2.5.3.

  Set warning group or flag (see *Warnings*).

Pattern matching and equality

`--exact-split, --no-exact-split`
: Added in version 2.5.1.

  Require [do not require] all clauses in a definition to hold as definitional equalities unless marked `CATCHALL` (see *Case trees*).

  Turns on the following warnings:

  - `CoverageNoExactSplit`
  - `InlineNoExactSplit`
  - `ShouldBeEtaRecordPattern`

  Default: `--no-exact-split`.

`--hidden-argument-puns, --no-hidden-argument-puns`
: Added in version 2.6.4.

  Enable [disable] *hidden argument puns*.

  Default: `--no-hidden-argument-puns`.

`--no-eta-equality`
: Added in version 2.5.1.

  Default records to `no-eta-equality` (see *Eta-expansion*).

`--eta-equality`
: Added in version 2.6.4.

  Default, opposite of `--no-eta-equality`.

`--cohesion`
: Added in version 2.6.3.

  Enable the cohesion modalities, in particular `@♭` (see *Flat Modality*).

`--no-cohesion`
: Added in version 2.6.4.

  Default, opposite of `--cohesion`.

`--flat-split`
: Added in version 2.6.1.

  Enable pattern matching on `@♭` arguments (see *Pattern Matching on @♭*). Implies `--cohesion`.

--no-flat-split

Added in version 2.6.4.

Default, opposite of *--flat-split*.

--polarity, --no-polarity

Added in version 2.8.0.

Enables the use of modal polarity annotations, and their interaction with the positivity checker. See *Polarity Annotations*.

Default: *--no-polarity*.

--occurrence-analysis, --no-occurrence-analysis

Added in version 2.9.0.

Turns on or off automated occurrence analysis for functions. See *Occurrence analysis*.

Default: *--occurrence-analysis*.

--no-pattern-matching

Added in version 2.4.0.

Disable pattern matching completely.

--pattern-matching

Added in version 2.6.4.

Default, opposite of *--no-pattern-matching*.

--with-K

Added in version 2.4.2.

Overrides a global *--without-K* in a file (see *Without K*).

--without-K

Added in version 2.2.10.

Disables reasoning principles incompatible with univalent type theory, most importantly Streicher's K axiom (see *Without K*).

--cubical=compatible, **--cubical-compatible**

Added in version 2.6.3.

Generate internal support code necessary for use from Cubical Agda (see *Cubical compatible*). Implies *--without-K*.

--keep-pattern-variables

Added in version 2.6.1.

Prevent interactive case splitting from replacing variables with dot patterns (see *Dot patterns*).

Default since 2.7.0.

--no-keep-pattern-variables

Added in version 2.6.4.

Opposite of *--keep-pattern-variables*.

`--infer-absurd-clauses, --no-infer-absurd-clauses`

Added in version 2.6.4.

`--no-infer-absurd-clauses` prevents interactive case splitting and coverage checking from automatically filtering out absurd clauses. This means that these absurd clauses have to be written out in the Agda text. Try this option if you experience type checking performance degradation with omitted absurd clauses.

Default: `--infer-absurd-clauses`.

`--large-indices, --no-large-indices`

Added in version 2.6.4.

Allow constructors to store values of types whose sort is larger than that being defined, when these arguments are forced by the constructor's type.

When *--safe* is given, this flag can not be combined with *--without-K* or *--forced-argument-recursion*, since both of these combinations are known to be inconsistent.

When *--no-forcing* is given, this option is redundant.

Default: `--no-large-indices`.

Recursion

`--forced-argument-recursion, --no-forced-argument-recursion`

Added in version 2.6.4.

Allow the use of forced constructor arguments as termination metrics. This flag may be necessary for Agda to accept nontrivial uses of induction-induction.

Default: `--forced-argument-recursion`.

`--guardedness, --no-guardedness`

Added in version 2.6.0.

Enable [disable] constructor-based guarded corecursion (see *Coinduction*).

The option `--guardedness` is inconsistent with sized types, thus, it cannot be used with both *--safe* and *--sized-types*.

Default: `--no-guardedness` (since 2.6.2).

`--sized-types, --no-sized-types`

Added in version 2.2.0.

Enable [disable] sized types (see *Sized Types*).

The option `--sized-types` is inconsistent with constructor-based guarded corecursion, thus, it cannot be used with both *--safe* and *--guardedness*.

Default: `--no-sized-types` (since 2.6.2).

`--termination-depth={N}`

Added in version 2.2.8.

Allow termination checker to count decrease/increase upto `N`, see *Termination Checking*.

Defaults to 3 since 2.9.0.

Sorts and universes

--type-in-type

Ignore universe levels (this makes Agda inconsistent; see *type-in-type*).

--no-type-in-type

Added in version 2.6.4.

Default, opposite of *--type-in-type*.

--omega-in-omega

Added in version 2.6.0.

Enable typing rule `Set`ω `:  Set`ω (this makes Agda inconsistent; see *omega-in-omega*).

--no-omega-in-omega

Added in version 2.6.4.

Default, opposite of *--omega-in-omega*.

--level-universe, --no-level-universe

Added in version 2.6.4.

Makes `Level` live in its own universe `LevelUniv` and disallows having levels depend on terms that are not levels themselves. When this option is turned off, `LevelUniv` still exists, but reduces to `Set` (see *level-universe*).

Note: While compatible with the *--cubical* option, this option is currently not compatible with cubical builtin files.

Default: *--no-level-universe*.

--universe-polymorphism, --no-universe-polymorphism

Added in version 2.3.0.

Enable [disable] universe polymorphism (see *Universe Levels*).

Default: `--universe-polymorphism`.

--cumulativity, --no-cumulativity

Added in version 2.6.1.

Enable [disable] cumulative subtyping of universes, i.e., if `A : Set i` then also `A : Set j` for all `j >= i`.

Default: `--no-cumulativity`.

Search depth and instances

--instance-search-depth=`{N}`

Added in version 2.5.2.

Set instance search depth to `N` (default: 500; see *Instance Arguments*).

--inversion-max-depth=`{N}`

Added in version 2.5.4.

Set maximum depth for pattern match inversion to `N` (default: 50). Should only be needed in pathological cases.

--backtracking-instance-search, --no-backtracking-instance-search

Added in version 2.7.0.

Consider [do not consider] recursive instance arguments during pruning of instance candidates, see *Backtracking*

Default: `--no-backtracking-instance-search`.

This option used to be called `--overlapping-instances`.

`--qualified-instances`, `--no-qualified-instances`

Added in version 2.6.2.

Consider [do not consider] instances that are (only) in scope under a qualified name.

Default: `--qualified-instances`.

`--experimental-lazy-instances`, `--no-experimental-lazy-instances`

Added in version 2.8.0.

Opt into the experimental, faster implementation of instance search. This is presently optional since it may potentially introduce regressions in code which relies on the order of constraint solving.

Default: `--no-experimental-lazy-instances`.

The `--experimental-lazy-instances` behaviour will be made the default and this flag will be removed in the future.

Other features

`--double-check`

Enable double-checking of all terms using the internal typechecker. Off by default.

`--no-double-check`

Added in version 2.6.2.

Opposite of *`--double-check`*. On by default.

`--keep-covering-clauses`

Added in version 2.6.3.

Save function clauses computed by the coverage checker to the interface file. Required by some external backends.

`--no-keep-covering-clauses`

Added in version 2.6.4.

Opposite of *`--keep-covering-clauses`*, default.

`--no-print-pattern-synonyms`

Added in version 2.5.4.

Always expand *Pattern Synonyms* during printing. With this option enabled you can use pattern synonyms freely, but Agda will not use any pattern synonyms when printing goal types or error messages, or when generating patterns for case splits.

`--print-pattern-synonyms`

Added in version 2.6.4.

Default, opposite of *`--no-print-pattern-synonyms`*.

`--no-syntactic-equality`

Added in version 2.6.0.

Disable the syntactic equality shortcut in the conversion checker.

`--syntactic-equality`=`{N}`

Added in version 2.6.3.

Give the syntactic equality shortcut `N` units of fuel (`N` must be a natural number).

If `N` is omitted, then the syntactic equality shortcut is enabled without any restrictions. (This is the default.)

If `N` is given, then the syntactic equality shortcut is given `N` units of fuel. The exact meaning of this is implementation-dependent, but successful uses of the shortcut do not affect the amount of fuel.

Note that this option is experimental and subject to change.

`--safe`

Added in version 2.3.0.

Disable postulates, unsafe *OPTIONS* pragmas and `primTrustMe`. Prevents to have both *--sized-types* and *--guardedness* on. Further reading: *Safe Agda*.

`--no-import-sorts`

Added in version 2.6.2.

Disable the implicit statement `open import Agda.Primitive using (Set; ...)` at the start of each top-level Agda module.

`--import-sorts`

Added in version 2.6.4.

Default, opposite of *--no-import-sorts*.

Brings `Set` into scope, and if *--prop* is active, also `Prop`, and if *--two-level* is active, even `SSet`.

`--no-load-primitives`

Added in version 2.6.3.

Do not load the primitive modules (`Agda.Primitive`, `Agda.Primitive.Cubical`) when type-checking this program. This is useful if you want to declare Agda's very magical primitives in a Literate Agda file of your choice.

If you are using this option, it is your responsibility to ensure that all of the `BUILTIN` things defined in those modules are loaded. Agda will not work otherwise.

Implies *--no-import-sorts*.

Incompatible with *--safe*.

`--load-primitives`

Added in version 2.6.4.

Default, opposite of *--no-load-primitives*.

`--save-metas, --no-save-metas`

Added in version 2.6.3.

Save [or do not save] meta-variables in `.agdai` files. Not saving means that all meta-variable solutions are inlined into the interface. Currently, even if *--save-metas* is used, very few meta-variables are actually saved, and this option is more like an anticipation of possible future optimizations.

Default: *--no-save-metas*.

Erasure

--erasure, --no-erasure

Added in version 2.6.4.

Allow use of the annotations `@0` and `@erased`; allow use of names defined in Cubical Agda in Erased Cubical Agda; and mark parameters as erased in the type signatures of constructors and record fields (if *--with-K* is not active this is not done for indexed data types).

Default: *--no-erasure*.

--erased-matches, --no-erased-matches

Added in version 2.6.4.

Allow matching in erased positions for single-constructor, non-indexed data/record types. (This kind of matching is always allowed for record types with η-equality.)

Default: *--erased-matches* when *--with-K* is active, either by explicit activation or the absence of options like *--without-K*; otherwise *--no-erased-matches*.

If *--erased-matches* is given explicitly, it implies *--erasure*.

--erase-record-parameters

Added in version 2.6.3.

Mark parameters as erased in record module telescopes.

Implies *--erasure*.

--no-erase-record-parameters

Added in version 2.6.4.

Default, opposite of *--erase-record-parameters*.

--erased-funext, --no-erased-funext

Added in version 2.9.0.

Enables use of *Agda.Builtin.Erased.Funext*. This module contains an erased postulate of function extensionality. The idea is that it should be safe to use this postulate (in the absence of any Agda bugs):

- If *–erased-matches* is not used, then canonicity should hold for non-erased terms (if all opaque definitions are made transparent, the context only contains erased assumptions, and the context plus the postulates are jointly consistent).
- If *–erased-matches* is used, then reduction might get stuck, but compiled programs should still run correctly.

Implies *--erasure*. Default: `--no-erased-funext`.

--erased-propext, --no-erased-propext

Added in version 2.9.0.

Enables use of *Agda.Builtin.Erased.Propext*. This module contains an erased postulate of propositional extensionality. The idea is that it should be safe to use this postulate, see *--erased-funext*.

Implies *--erasure*. Default: `--no-erased-propext`.

--erased-quotients, --no-erased-quotients

Added in version 2.9.0.

Enables use of *Agda.Builtin.Erased.Quotient*. This module gives access to an implementation of set quotients with an eliminator that computes for the point constructor. The higher "constructors" are erased postulates. The idea is that it should be safe to use these postulates, see *--erased-funext*.

Implies *--erased-funext*. Default: `--no-erased-quotients`.

`--lossy-unification`

Added in version 2.6.4.

Enable lossy unification, see *Lossy Unification*.

`--quote-metas`

Added in version 2.9.0.

Allow typechecking to quote terms that contain metas, see *Quote Metas*.

`--no-quote-metas`

Added in version 2.9.0.

Block typechecking when attempting to quote terms that contain metas.

Opposite of *--quote-metas*, default.

4.2.3 Warnings

The *-W* or *--warning* option can be used to disable or enable different warnings. The flag `-W error` (or `--warning=error`) can be used to turn all warnings into errors, while `-W noerror` turns this off again.

A group of warnings can be enabled by `-W {GROUP}`, where `GROUP` is one of the following:

`all`

All of the existing warnings.

`warn`

Default warning level.

`ignore`

Ignore all warnings.

The command `agda --help=warning` provides information about which warnings are turned on by default.

Benign warnings

Individual non-fatal warnings can be turned on and off by `-W {NAME}` and `-W no{NAME}` respectively. The list containing any warning `NAME` can be produced by `agda --help=warning`:

`AbsurdPatternRequiresAbsentRHS`

Added in version 2.8.0.

(Previously named `AbsurdPatternRequiresNoRHS`.)

RHS given despite an absurd pattern in the LHS.

`BuiltinDeclaresIdentifier`

Added in version 2.7.0.

A `BUILTIN` pragma that declares an identifier, but has been given an existing one.

`AsPatternShadowsConstructorOrPatternSynonym`

Added in version 2.6.2.

`@`-patterns that shadow constructors or pattern synonyms.

`CantGeneralizeOverSorts`

Added in version 2.6.0.

Attempts to generalize over sort metas in `variable` declaration.

ClashesViaRenaming

Added in version 2.6.1.

Clashes introduced by `renaming`.

ConflictingPragmaOptions

Added in version 2.7.0.

Conflicting pragma options. For instance, both `--this` and `--no-that` when `--this` implies `--that`.

ConfluenceCheckingIncompleteBecauseOfMeta

Added in version 2.7.0.

Incomplete confluence checks because of unsolved metas.

ConfluenceForCubicalNotSupported

Added in version 2.7.0.

Attempts to check confluence with *`--cubical`*.

CoverageNoExactSplit

Added in version 2.5.3.

Failed exact split checks.

DeprecationWarning

Added in version 2.5.3.

Deprecated features.

DefinitionBeforeDeclaration

Added in version 2.9.0.

Definitions that occur in `mutual` blocks before their declarations.

DivergentModalityInClause

Added in version 2.9.0.

Modalities of clauses that diverge from the modality of the function.

DuplicateFields

Added in version 2.6.4.

`record` expression with duplicate field names.

DuplicateRecordDirective

Added in version 2.7.0.

Conflicting directives in a record declaration.

DuplicateRewriteRule

Added in version 2.7.0.

Duplicate declaration of a name as *REWRITE* rule.

DuplicateUsing

Added in version 2.6.2.

Repeated names in `using` directive.

EmptyAbstract

Added in version 2.5.4.

Empty `abstract` blocks.

EmptyConstructor

Added in version 2.6.2.

Empty `data _ where` blocks.

EmptyField

Added in version 2.6.1.

Empty `field` blocks.

EmptyGeneralize

Added in version 2.6.0.

Empty `variable` blocks.

EmptyInstance

Added in version 2.5.4.

Empty `instance` blocks.

EmptyMacro

Added in version 2.5.4.

Empty `macro` blocks.

EmptyMutual

Added in version 2.5.4.

Empty `mutual` blocks.

EmptyPolarityPragma

Added in version 2.8.0.

POLARITY pragmas not giving any polarities.

EmptyPostulate

Added in version 2.5.4.

Empty `postulate` blocks.

EmptyPrimitive

Added in version 2.6.0.

Empty `primitive` blocks.

EmptyPrivate

Added in version 2.5.4.

Empty `private` blocks.

EmptyRewritePragma

Added in version 2.5.2.

Empty `REWRITE` pragmas.

EmptyWhere

Added in version 2.6.2.

Empty `where` blocks.

FaceConstraintCannotBeHidden

Added in version 2.6.4.

Face constraint patterns that are given as implicit arguments.

FaceConstraintCannotBeNamed

Added in version 2.6.4.

Face constraint patterns that are given as named arguments.

FixingCohesion

Added in version 2.8.0.

Invalid cohesion annotations, automatically corrected.

FixingPolarity

Added in version 2.8.0.

Invalid polarity annotations, automatically corrected.

FixingRelevance

Added in version 2.8.0.

Invalid relevance annotations, automatically corrected.

FixityInRenamingModule

Added in version 2.6.1.

Fixity annotations in `renaming` directives for a `module`.

HiddenGeneralize

Added in version 2.6.3.

Hidden identifiers in `variable` blocks.

IllegalDeclarationInDataDefinition

Added in version 2.6.4.

Declarations inside of a `data` definition that are not constructor type signatures.

IllformedAsClause

Added in version 2.6.0.

Illformed `as`-clauses in `import` statements.

InlineNoExactSplit

Added in version 2.6.4.

Failed exact splits after inlining a constructor, see *The INLINE and NOINLINE pragmas*.

InstanceNoOutputTypeName

Added in version 2.6.0.

Instance arguments whose type does not end in a named or variable type; such are never considered by instance search.

InstanceArgWithExplicitArg

Added in version 2.6.0.

Instance arguments with explicit arguments; such are never considered by instance search.

InstanceWithExplicitArg

Added in version 2.6.0.

Instance declarations with explicit arguments; such are never considered by instance search.

InteractionMetaBoundaries

Added in version 2.6.4.

Interaction meta variables that have unsolved boundary constraints.

InvalidCatchallPragma

Added in version 2.5.4.

CATCHALL pragmas before a non-function clause.

InvalidCharacterLiteral

Added in version 2.6.4.

Illegal character literals such as surrogate code points.

InvalidConstructorBlock

Added in version 2.6.2.

`constructor` blocks outside of `interleaved mutual` blocks.

InvalidCoverageCheckPragma

Added in version 2.6.1.

NON_COVERING pragmas before non-function or `mutual` blocks.

InvalidDataOrRecDefParameter

Added in version 2.9.0.

A `data/record D parameters where` definition where the parameters do not match up with the previously given signature or contain more than just names with hiding information.

InvalidNoPositivityCheckPragma

Added in version 2.5.4.

NO_POSITIVITY_CHECK pragmas before something that is neither a `data` nor `record` declaration nor a `mutual` block.

InvalidNoUniverseCheckPragma

Added in version 2.6.0.

NO_UNIVERSE_CHECK pragmas before declarations other than `data` or `record` declarations.

InvalidEtaEqualityPragma

Added in version 2.9.0.

ETA_EQUALITY pragmas before declarations other than `record` declarations.

InvalidTacticAttribute

Added in version 2.9.0.

@(tactic ...) attributes where they are not supported.

`InvalidTerminationCheckPragma`

Added in version 2.5.4.

Termination checking pragmas before non-function or `mutual` blocks.

`InversionDepthReached`

Added in version 2.5.4.

Inversions of pattern-matching failed due to exhausted inversion depth.

`LibUnknownField`

Added in version 2.6.0.

Unknown fields in library files.

`LocalRewritingConfluenceCheck`

Added in version 2.9.0.

Confluence checking (*`--confluence-check`* or *`--local-confluence-check`*) is not yet implemented for local rewrite rules (*`--local-rewriting`*).

`MisplacedAttributes`

Added in version 2.8.0.

Attributes where they cannot appear.

`MisplacedRewrite`

Added in version 2.9.0.

Invalid local rewrite annotations, automatically ignored.

`MissingTypeSignatureForOpaque`

Added in version 2.7.0.

`abstract` or `opaque` definitions that lack a type signature.

`ModuleDoesntExport`

Added in version 2.6.0.

Names mentioned in an import statement which are not exported by the module in question.

`MultipleAttributes`

Added in version 2.6.2.

Multiple attributes given where only erasure is accepted.

`NoMain`

Added in version 2.7.0.

Invoking the compiler on a module without a `main` function. See also *`--no-main`*.

`NotAffectedByOpaque`

Added in version 2.6.4.

Declarations that should not be inside `opaque` blocks.

`NotARewriteRule`

Added in version 2.8.0.

`REWRITE` pragmas referring to identifiers that are neither definitions nor constructors.

`NotInScope`
: Added in version 2.6.1.

  Out of scope names.

`OldBuiltin`
: Added in version 2.5.2.

  Deprecated *BUILTIN* pragmas.

`OpenImportAbstract`
: Added in version 2.8.0.

  `open` or `import` statements in `abstract` blocks.

`OpenImportPrivate`
: Added in version 2.8.0.

  `open` or `import` statements in `private` blocks.

`OptionRenamed`
: Added in version 2.6.3.

  Renamed options.

`PatternShadowsConstructor`
: Added in version 2.6.4.

  Pattern variables that shadow constructors.

`PlentyInHardCompileTimeMode`
: Added in version 2.6.4.

  Use of attributes `@ω` or `@plenty` in hard compile-time mode.

`PolarityPragmasButNotPostulates`
: Added in version 2.5.4.

  Polarity pragmas for non-postulates.

`PragmaCompileErased`
: Added in version 2.6.1.

  COMPILE pragma targeting an erased symbol.

`PragmaCompileList`
: Added in version 2.7.0.

  COMPILE pragma for GHC backend targeting lists.

`PragmaCompileMaybe`
: Added in version 2.7.0.

  COMPILE pragma for GHC backend targeting `MAYBE`.

`PragmaCompileUnparsable`
: Added in version 2.8.0.

  Unparsable *COMPILE* GHC pragmas.

`PragmaCompileWrong`

Added in version 2.8.0.

Ill-formed *COMPILE* GHC pragmas.

`PragmaCompileWrongName`

Added in version 2.8.0.

COMPILE pragmas referring to identifiers that are neither definitions nor constructors.

`PragmaExpectsDefinedSymbol`

Added in version 2.8.0.

Pragmas referring to identifiers that are not defined symbols.

`PragmaExpectsUnambiguousConstructorOrFunction`

Added in version 2.8.0.

Pragmas referring to identifiers that are not unambiguous constructors or functions.

`PragmaExpectsUnambiguousProjectionOrFunction`

Added in version 2.8.0.

Pragmas referring to identifiers that are not unambiguous projections or functions.

`PragmaNoTerminationCheck`

Added in version 2.6.0.

NO_TERMINATION_CHECK pragmas; such are deprecated.

`InvalidDisplayForm`

Added in version 2.8.0.

An illegal *DISPLAY* form; it will be ignored.

`RewriteLHSNotDefinitionOrConstructor`

Added in version 2.7.0.

Rewrite rule head symbol is not a defined symbol or constructor.

`RewriteVariablesNotBoundByLHS`

Added in version 2.7.0.

Rewrite rule does not bind all of its variables.

`RewriteVariablesBoundMoreThanOnce`

Added in version 2.7.0.

Constructor-headed rewrite rule has non-linear parameters.

`RewriteVariablesBoundInSingleton`

Added in version 2.9.0.

Rewrite rule binds some variables in possibly definitionally singular contexts.

`RewriteLHSReduces`

Added in version 2.7.0.

Rewrite rule LHS is not in weak-head normal form.

`RewriteHeadSymbolIsProjectionLikeFunction`

Added in version 2.7.0.

Rewrite rule head symbol is a projection-like function.

`RewriteHeadSymbolIsTypeConstructor`

Added in version 2.7.0.

Rewrite rule head symbol is a type constructor.

`RewriteHeadSymbolContainsMetas`

Added in version 2.7.0.

Definition of rewrite rule head symbol contains unsolved metas.

`RewriteConstructorParametersNotGeneral`

Added in version 2.7.0.

Constructor-headed rewrite rule parameters are not fully general.

`RewriteContainsUnsolvedMetaVariables`

Added in version 2.7.0.

Rewrite rule contains unsolved metas.

`RewriteBlockedOnProblems`

Added in version 2.7.0.

Checking rewrite rule blocked by unsolved constraint.

`RewriteRequiresDefinitions`

Added in version 2.7.0.

Checking rewrite rule blocked by missing definition.

`RewriteDoesNotTargetRewriteRelation`

Added in version 2.7.0.

Rewrite rule does not target the rewrite relation.

`RewriteBeforeFunctionDefinition`

Added in version 2.7.0.

Rewrite rule is not yet defined.

`RewriteBeforeMutualFunctionDefinition`

Added in version 2.7.0.

Mutually declaration with the rewrite rule is not yet defined.

`RewritesNothing`

Added in version 2.8.0.

`rewrite` clauses that do not fire.

`ShadowingInTelescope`

Added in version 2.6.1.

Repeated variable name in telescope.

ShouldBeEtaRecordPattern

Added in version 2.9.0.

Irrefutable record pattern without eta.

Off by default, enabled with *--exact-split*.

TooManyArgumentsToSort

Added in version 2.8.0.

E.g. `Set` used with more than one argument.

TooManyFields

Added in version 2.6.4.

Record expression with invalid field names.

TooManyPolarities

Added in version 2.8.0.

POLARITY pragma with too many polarities given.

UnfoldingWrongName

Added in version 2.8.0.

Names in an `unfolding` clause that are not unambiguous definitions.

UnfoldTransparentName

Added in version 2.6.4.

Non-`opaque` names mentioned in an `unfolding` clause.

UnguardedEtaRecord

Added in version 2.9.0.

A record with `eta-equality` that Agda inferred as unguarded, meaning it has recursive occurences that are not protected by a type former that does not have eta equality (such as a `data` or a `no-eta-equality` `record` type).

UnknownAttribute

Added in version 2.8.0.

Unknown attributes.

UnknownFixityInMixfixDecl

Added in version 2.5.4.

Mixfix names without an associated fixity declaration.

UnknownJSPrimitive

Added in version 2.9.0.

A primitive compiled to `Undefined` by the JS backend because it is not in the list of known primitives.

UnknownPolarity

Added in version 2.8.0.

Unknown polarities.

UnreachableClauses

Added in version 2.5.3.

Unreachable function clauses.

UnsupportedAttribute

Added in version 2.6.2.

Unsupported attributes.

UnsupportedIndexedMatch

Added in version 2.6.3.

Failures to compute full equivalence when splitting on indexed family.

UnusedImports

Added in version 2.9.0.

Warn about openings of modules that do not bring identifiers into scope that are subsequently used. If the `open` comes with an explicit `using` or `renaming` directive, warn about individual unused identifiers (typically those mentioned in the directive). There is no warning about `public` openings. In the presence of option:*–no-qualified-instances*, there are also no warnings about unused instances brought into scope.

This warning is off by default.

UnusedImports=all

Added in version 2.9.0.

Same as *UnusedImports*, but warn about each unused identifier also when no `using` or `renaming` directive is given.

Option `-WnoUnusedImports` disables both *UnusedImports* and *UnusedImports=all*.

UnusedVariablesInDisplayForm

Added in version 2.8.0.

DISPLAY forms that bind variables they do not use.

UselessAbstract

Added in version 2.5.4.

`abstract` blocks where they have no effect.

UselessHiding

Added in version 2.6.2.

Names in `hiding` directive that are anyway not imported.

UselessInline

Added in version 2.5.3.

INLINE pragmas where they have no effect.

UselessImport

Added in version 2.9.0.

`import` statements that do not bring anything into scope.

UselessInstance

Added in version 2.5.4.

`instance` blocks where they have no effect.

UselessMacro

Added in version 2.7.0.

`macro` blocks where they have no effect.

UselessOpaque

Added in version 2.6.4.

`opaque` blocks that have no effect.

UselessPatternDeclarationForRecord

Added in version 2.6.2.

`pattern` directives where they have no effect.

UselessPragma

Added in version 2.6.4.

Pragmas that get ignored.

UselessPrivate

Added in version 2.5.4.

`private` blocks where they have no effect.

UselessPublic

Added in version 2.5.3.

`public` directives where they have no effect.

UselessTactic

Added in version 2.8.0.

`@tactic` attributes in non-hidden and instance arguments.

UserWarning

Added in version 2.5.4.

User-defined warnings added using one of the `WARNING_ON_*` pragmas.

WarningProblem

Added in version 2.7.0.

Problem encountered with option `-W`, like an unknown warning or the attempt to switch off a non-benign warning.

WithClauseProjectionFixityMismatch

Added in version 2.8.0.

Projection fixity different in with-clause compared to its parent clause.

WithoutKFlagPrimEraseEquality

Added in version 2.6.0.

`primEraseEquality` used with the without-K flags.

WrongInstanceDeclaration

Added in version 2.6.0.

Terms marked as eligible for instance search whose type does not end with a name.

CustomBackendWarning

Added in version 2.7.0.

Warnings from custom backends.

Error warnings

Some warnings are fatal; those are errors Agda first ignores but eventually raises. Such *error warnings* are always on, they cannot be toggled by `-W`.

AbstractInLetBindings

Added in version 2.8.0.

Let bindings can not be made abstract.

CoinductiveEtaRecord

Added in version 2.7.0.

Declaring a `record` type as both `coinductive` and having `eta-equality`.

CoInfectiveImport

Added in version 2.6.0.

Importing a file not using e.g. `--safe` from one which does.

ConstructorDoesNotFitInData

Added in version 2.7.0.

Constructor with arguments in a universe higher than the one of its data type.

CoverageIssue

Added in version 2.5.3.

Failed coverage checks.

HiddenNotInArgumentPosition

Added in version 2.8.0.

Hidden arguments `{ x }` can only appear as arguments to functions, not as expressions by themselves.

InfectiveImport

Added in version 2.6.0.

Importing a file using e.g. `--cubical` into one which does not.

InferredLocalRewrite

Added in version 2.9.0.

Tried to solve a meta with an '@rewrite' function.

InstanceNotInArgumentPosition

Added in version 2.8.0.

Instance arguments PDF TODO `x` PDF TODO can only appear as arguments to functions, not as expressions by themselves.

LocalRewriteOutsideTelescope

Added in version 2.9.0.

'@rewrite' arguments are (currently) only allowed in module telescopes.

MacroInLetBindings

Added in version 2.8.0.

Macros can not be let-bound.

`MismatchedBrackets`

Added in version 2.9.0.

An idiom bracket opened with unicode (resp. ASCII) syntax must also be closed with unicode (resp. ASCII) syntax.

`MissingDataDeclaration`

Added in version 2.8.0.

Constructor definitions not associated to a data declaration.

`MissingDefinitions`

Added in version 2.6.0.

Names declared without an accompanying definition.

`NotAllowedInMutual`

Added in version 2.6.0.

Declarations that are not allowed in a mutual block.

`NotStrictlyPositive`

Added in version 2.5.2.

Failed strict positivity checks.

`OverlappingTokensWarning`

Added in version 2.5.4.

Multi-line comments spanning one or more literate text blocks.

`PragmaCompiled`

Added in version 2.6.0.

COMPILE pragmas not allowed in safe mode.

`RecursiveRecordNeedsInductivity`

Added in version 2.7.0.

Recursive records that are neither declared `inductive` nor `coinductive`.

`RewriteAmbiguousRules`

Added in version 2.6.2.

Failed global confluence checks because of overlapping rules.

`RewriteMaybeNonConfluent`

Added in version 2.6.1.

Failed confluence checks while computing overlap.

`RewriteMissingRule`

Added in version 2.6.2.

Failed global confluence checks because of missing rules.

`RewriteNonConfluent`

Added in version 2.6.1.

Failed confluence checks while joining critical pairs.

`SafeFlagInjective`
: Added in version 2.6.1.

  INJECTIVE pragmas with the `--safe` flag.

`SafeFlagNoCoverageCheck`
: Added in version 2.6.1.

  NON_COVERING pragmas with the `--safe` flag.

`SafeFlagNonTerminating`
: Added in version 2.5.3.

  NON_TERMINATING pragmas with the `--safe` flag.

`SafeFlagNoPositivityCheck`
: Added in version 2.5.3.

  NO_POSITIVITY_CHECK pragmas with the `--safe` flag.

`SafeFlagNoUniverseCheck`
: Added in version 2.6.0.

  NO_UNIVERSE_CHECK pragmas with the `--safe` flag.

`SafeFlagPolarity`
: Added in version 2.5.3.

  POLARITY pragmas with the `--safe` flag.

`SafeFlagPostulate`
: Added in version 2.5.3.

  `postulate` blocks with the `--safe` flag.

`SafeFlagPragma`
: Added in version 2.5.3.

  Unsafe *OPTIONS* pragmas with the `--safe` flag.

`SafeFlagTerminating`
: Added in version 2.5.3.

  TERMINATING pragmas with the `--safe` flag.

`SafeFlagWithoutKFlagPrimEraseEquality`
: Added in version 2.6.0.

  `primEraseEquality` used with the `--safe` and `--without-K` flags.

`TerminationIssue`
: Added in version 2.5.2.

  Failed termination checks.

`TopLevelPolarity`
: Added in version 2.8.0.

  Declaring definitions with an explicit polarity annotation.

`UnknownNamesInFixityDecl`

Added in version 2.5.4.

Names not declared in the same scope as their syntax or fixity declaration.

`UnknownNamesInPolarityPragmas`

Added in version 2.5.4.

Names not declared in the same scope as their polarity pragmas.

`UnsolvedConstraints`

Added in version 2.5.2.

Unsolved constraints.

`UnsolvedInteractionMetas`

Added in version 2.5.2.

Unsolved interaction meta variables.

`UnsolvedMetaVariables`

Added in version 2.5.2.

Unsolved meta variables.

4.2.4 Command-line examples

Run Agda with all warnings enabled, except for warnings about empty `abstract` blocks:

```
agda -W all --warning=noEmptyAbstract file.agda
```

Run Agda on a file which uses the standard library. Note that you must have already created a `libraries` file as described in *Library Management*.

```
agda -l standard-library -i. file.agda
```

(Or if you have added `standard-library` to your `defaults` file, simply `agda file.agda`.)

4.2.5 Checking options for consistency

Agda checks that options used in imported modules are consistent with each other.

An *infective* option is an option that if used in one module, must be used in all modules that depend on this module. The following options are infective:

- *--cohesion*
- *--erased-funext*
- *--erased-matches*
- *--erased-propext*
- *--erased-quotients*
- *--erasure*
- *--flat-split*
- *--guarded*
- *--irrelevance*

- *--polarity*
- *--prop*
- *--rewriting*
- *--local-rewriting*
- *--two-level*

Furthermore, the Cubical options are *jointly infective* in the following sense: if one of them is used in one module, modules depending on it should use a Cubical option at least as strong as the imported module. The Cubical options are listed below in increasing strength:

- *Cubical without Glue* *--cubical=no-glue*
- *Cubical with Erased Glue* *--cubical=erased* *
- (Full) *Cubical* *--cubical[=full]* *

* : Exceptionally, modules using Full Cubical *--cubical[=full]* can be imported from modules using Erased Cubical *--cubical=erased* if *--erasure* is enabled and imports are only used in erased positions. See *here* for a summary.

A *coinfective* option is an option that if used in one module, must be used in all modules that this module depends on. The following options are coinfective:

- *--level-universe*
- *--no-guardedness*
- *--no-sized-types*
- *--no-universe-polymorphism*
- *--safe*
- *--without-K*

Furthermore the option *--cubical=compatible* is mostly coinfective. If a module uses *--cubical=compatible* then all modules that this module imports (directly) must also use *--cubical=compatible*, with the following exception: if a module uses both *--cubical=compatible* and *--with-K*, then it is not required to use *--cubical=compatible* in (directly) imported modules that use *--with-K*. (Note that one cannot use *--cubical=compatible* and *--with-K* at the same time if *--safe* is used.)

Agda records the options used when generating an interface file. If any of the following options differ when trying to load the interface again, the source file is re-typechecked instead:

- *--allow-exec*
- *--allow-incomplete-matches*
- *--allow-unsolved-metas*
- *--backtracking-instance-search*
- *--call-by-name*
- *--cohesion*
- *--confluence-check*
- *--copatterns*
- *--cubical*
- *--cubical=compatible*

- *--cubical=erased*
- *--cumulativity*
- *--double-check*
- *--erase-record-parameters*
- *--erased-funext*
- *--erased-matches*
- *--erased-propext*
- *--erased-quotients*
- *--erasure*
- *--exact-split*
- *--experimental-irrelevance*
- *--flat-split*
- *--guarded*
- *--hidden-argument-puns*
- *--infer-absurd-clauses*
- *--injective-type-constructors*
- *--instance-search-depth*
- *--inversion-max-depth*
- *--irrelevance*
- *--irrelevant-projections*
- *--keep-covering-clauses*
- *--local-confluence-check*
- *--lossy-unification*
- *--no-auto-inline*
- *--no-eta-equality*
- *--no-fast-reduce*
- *--no-forcing*
- *--no-guardedness*
- *--no-import-sorts*
- *--no-load-primitives*
- *--no-occurrence-analysis*
- *--no-pattern-matching*
- *--no-positivity-check*
- *--no-projection-like*
- *--no-sized-types*
- *--no-termination-check*

- *--no-unicode*
- *--no-universe-polymorphism*
- *--omega-in-omega*
- *--polarity*
- *--prop*
- *--qualified-instances*
- *--quote-metas*
- *--rewriting*
- *--local-rewriting*
- *--safe*
- *--save-metas*
- *--syntactic-equality*
- *--termination-depth*
- *--two-level*
- *--type-in-type*
- *--warning*
- *--without-K*

4.2.6 References

[1] Jesper Cockx and Dominique Devriese. "Proof-relevant unification: Dependent pattern matching with only the axioms of your type theory." In Journal of Functional Programming 28, 2018.

4.3 Compilers

See also *Foreign Function Interface*.

4.3.1 Backends

GHC Backend

The GHC backend translates Agda programs into GHC Haskell programs.

Usage

The GHC backend can be invoked from the command line using the flag *--compile* or *--ghc*:

```
agda --compile
  [--compile-dir=<DIR>]
  [--ghc-flag=<FLAG>]
  [--ghc-strict-data]
  [--ghc-strict]
  [--ghc-trace]
  <FILE>.agda
```

When the flag *--ghc-strict-data* is used, inductive data and record constructors are compiled to constructors with strict arguments. (This does not apply to certain builtin types—lists, the maybe type, and some types related to reflection—and might not apply to types with `COMPILE GHC ... = data ...` pragmas.)

When the flag *--ghc-strict* is used, the GHC backend generates mostly strict code. Note that functions might not be strict in unused arguments, and that function definitions coming from `COMPILE GHC` pragmas are not affected. This flag implies *--ghc-strict-data*, and the exceptions of that flag applies to this flag as well. (Note that this option requires the use of GHC 9 or later.)

Options

`--compile`, `--ghc`

Compile to GHC Haskell placing the files in subdirectory `MAlonzo` or the directory given by *--compile-dir*. Then invoke `ghc` (or the compiler given by *--with-compiler*) on the main file, unless option *--ghc-dont-call-ghc* is given.

`--with-compiler`={PATH}

Set `PATH` as the executable to call to compile the backend's output, default: `ghc`.

`--ghc-dont-call-ghc`

Only produce Haskell files, skip the compilation to binary.

`--ghc-flag`={GHC-FLAG}

Pass flag *GHC-FLAG* to the Haskell compiler. This option can be given several times.

`--ghc-strict-data`

Compile Agda constructor to strict Haskell constructors.

`--ghc-strict`

Generate strict Haskell code.

`--ghc-trace`

Instrument the code to trace function calls, inserting a `Debug.Trace.trace` statement at the beginning of each function.

See *Compilation* for options common to the compiler backends.

Pragmas

Example

The following "Hello, World!" example requires some *Built-ins* and uses the *Foreign Function Interface*:

```
module HelloWorld where

open import Agda.Builtin.IO
open import Agda.Builtin.Unit
open import Agda.Builtin.String

postulate
  putStrLn : String → IO ⊤

{-# FOREIGN GHC import qualified Data.Text.IO as Text #-}
{-# COMPILE GHC putStrLn = Text.putStrLn #-}

main : IO ⊤
main = putStrLn "Hello, World!"
```

After compiling the example

```
agda --compile HelloWorld.agda
```

you can run the HelloWorld program which prints `Hello, World!`.

> **Warning**
>
> Frequent error when compiling: `Float` requires the ieee754 haskell library. Usually `cabal v1-install ieee754` or `cabal v2-install --lib ieee754` in the command line does the trick.

JavaScript Backend

The JavaScript backend translates Agda code to JavaScript code.

Usage

The JavaScript backend can be invoked from the command line using the flag *--js*:

```
agda --js --js-es6 [--js-optimize] [--js-minify] [--compile-dir=<DIR>] <FILE>.agda
```

The *--js-es6* flag makes the generated JavaScript code use ES6-style module syntax.

The *--js-optimize* flag makes the generated JavaScript code typically faster and less readable.

The *--js-minify* flag makes the generated JavaScript code smaller and less readable.

Options

`--js`

Compile to JavaScript, placing translation of module *M* into file `jAgda.M.js` (or `jAgda.M.mjs`, if the option *--js-es6* is passed). The files will be placed into the root directory of the compiled Agda project, or into the directory given by *--compile-dir*.

`--js-es6`

Added in version 2.8.0.

Produce ES6 style modules (supported natively by browsers and NodeJS since 2020).

`--js-amd`

..deprecated:: 2.9.0

Produce AMD style modules (for in-browser usage with a wrapper like *require.js*).

`--js-cjs`

..deprecated:: 2.9.0

Produce CommonJS style modules (supported natively by NodeJS). This is the default.

`--js-minify`

Produce minified JavaScript (e.g. omitting whitespace where possible).

`--js-optimize`

Produce optimized JavaScript.

`--js-verify`

Except for the main module, run the generated modules through `node`, to verify absence of syntax errors.

See *Compilation* for options common to the compiler backends.

4.3.2 Optimizations

Builtin natural numbers

Builtin natural numbers are represented as arbitrary-precision integers. The builtin functions on natural numbers are compiled to the corresponding arbitrary-precision integer functions.

Note that pattern matching on an Integer is slower than on an unary natural number. Code that does a lot of unary manipulations and doesn't use builtin arithmetic likely becomes slower due to this optimization. If you find that this is the case, it is recommended to use a different, but isomorphic type to the builtin natural numbers.

Irrelevant fields and constructor arguments

Record fields and constructor arguments marked *irrelevant* or *runtime irrelevant* are completely erased from the compiled record or data type. For instance,

```
postulate Parity : Nat → Set

record PNat : Set where
  field
    n    : Nat
    .p   : Parity n
    @0 q : Parity (suc n)
```

gets compiled by the GHC backend to (up to naming)

```
newtype PNat = PNat'constructor Integer
```

Erasable types

A data type is considered *erasable* if it has a single constructor whose arguments are all erasable types, or functions into erasable types. The compilers will erase

- calls to functions into erasable types
- pattern matches on values of erasable type

At the moment the compilers only have enough type information to erase calls of top-level functions that can be seen to return a value of erasable type without looking at the arguments of the call. In other words, a function call will not be erased if it calls a lambda bound variable, or the result is erasable for the given arguments, but not for others.

Typical examples of erasable types are the equality type and the accessibility predicate used for well-founded recursion:

```
data _≡_ {a} {A : Set a} (x : A) : A → Set a where
  refl : x ≡ x

data Acc {a} {A : Set a} (_<_ : A → A → Set a) (x : A) : Set a where
  acc : (∀ y → y < x → Acc _<_ y) → Acc _<_ x
```

The erasure means that equality proofs will (mostly) be erased, and never looked at, and functions defined by well-founded recursion will ignore the accessibility proof.

4.4 Debugging

> **Warning**
>
> The following page contains information that is mostly of interest to developers of Agda, or those who would like to get a deeper understanding of the implementation of Agda.

4.4.1 Verbose mode

If Agda was installed with the `debug` Cabal flag (e.g. using `cabal install Agda -fdebug`), it can print internal information by setting the *`--verbose={N}`* flag (or *`-v {N}`*) with a verbosity tag and a verbosity level in form `tag:level`. For example, running Agda with `--verbose=tc.term:30` turns on debug printing for the verbosity key `tc.term` at verbosity level `30`. Verbosity levels range between 0 and 100.

- Activating a verbosity key will also enable all the verbosity keys that are nested under it, for example `-v tc:30` will also print debugging information with key `tc.term`.
- The higher the verbosity level, the more detailed debugging information will be printed, for example `-v tc.term:50` will include debugging information at verbosity level 30.

By convention, very gory details will be printed only with verbosity of at least 50, so it is advisable in most cases to keep the level below 50.

Verbosity tags and levels can be found by inspecting the source code of Agda by searching for calls to `reportSLn` and `reportSDoc`. Below are a few common debug flags that might be useful for developers:

- `import`: import statements
- `interaction`: interactive commands
  - `interaction.case`: case splitting
  - `interaction.eval`
  - `interaction.give`

    - interaction.helper
    - interaction.intro
    - interaction.refine
- mimer: automatic proof search
- rewriting: rewrite rules
- scope: scope checking
- tc: type checking
    - tc.abstract: abstract definitions
    - tc.cc: compiling clauses to a case tree
    - tc.conv: conversion checking
    - tc.constr: constraint solving
    - tc.cover: coverage checking
    - tc.data: data types
    - tc.def: function definitions
    - tc.generalize: variable generalization
    - tc.instance: instance arguments
    - tc.irr: irrelevance
    - tc.lhs: left-hand sides of function clauses
    - tc.meta: metavariables
    - tc.mod: modules and module parameters
    - tc.term: type checking of terms
    - tc.opaque: opaque definitions
    - tc.pos: positivity analysis
    - tc.reduce: evaluation of terms and types
    - tc.size: sized types
    - tc.sort: checking sorts
    - tc.with: with abstraction
- term: termination checking
- warning: warnings

4.5 Emacs Mode

Agda programs can be edited in Emacs with support by the agda-mode, which is maintained by the Agda developers in the main Agda repository and offers many advanced features. To use it, first ensure you have *installed Agda*, installed Emacs, and *installed agda-mode*.

To edit a module in Emacs, open a file ending in .agda and load it by pressing C-c C-l (other commands are listed under *Notation for key combinations* below). This will apply syntax highlighting to the code and display any errors in a separate buffer. Agda uses certain background colors to indicate specific issues with the code, see *Background highlighting* below.

4.5.1 Installation

After installing the `agda` program and Emacs, run the following command:

```
agda --emacs-mode setup
```

The above command will first write the Agda data files to the Agda data directory (see *`--print-agda-data-dir`*) if this directory does not exist yet. (To force writing the data files there use the *`--setup`* option of `agda`.)

It then tries to set up Emacs for use with Agda. As an alternative you can copy the following text to your *.emacs* file:

```
(load-file (let ((coding-system-for-read 'utf-8))
                (shell-command-to-string "agda --emacs-mode locate")))
```

It is also possible (but not necessary) to compile the Emacs mode's files:

```
agda --emacs-mode compile
```

This can, in some cases, give a noticeable speedup.

> **Warning**
>
> If you reinstall the Agda mode without recompiling the Emacs Lisp files, then Emacs may continue using the old, compiled files.

4.5.2 Menus

There are two main menus in the system:

- A main menu called **Agda2** which is used for global commands.
- A context sensitive menu which appears if you right-click in a hole.

The menus contain more commands than the ones listed above. See *global* and *context sensitive* commands.

4.5.3 Configuration

If you want to you can customise the Emacs mode. Just start Emacs and type the following:

```
M-x load-library RET agda2-mode RET
M-x customize-group RET agda2 RET
```

If you want some specific settings for the Emacs mode you can add them to `agda2-mode-hook`. For instance, if you do not want to use the Agda input method (for writing various symbols like $\forall \geq \mathbb{N} \to \pi [\![]$) you can add the following to your *.emacs*:

```
(add-hook 'agda2-mode-hook
          '(lambda ()
            ; If you do not want to use any input method:
            (deactivate-input-method)
            ; (In some versions of Emacs you should use
            ; inactivate-input-method instead of
            ; deactivate-input-method.)
```

Note that, on some systems, the Emacs mode changes the default font of the current frame in order to enable many Unicode symbols to be displayed. This only works if the right fonts are available, though. If you want to turn off this feature, then you should customise the `agda2-fontset-name` variable.

The colors that are used to highlight Agda syntax and errors can be adjusted by typing `M-x customize-group RET agda2-highlight RET` in Emacs and following the instructions.

4.5.4 Keybindings

Notation for key combinations

The following notation is used when describing key combinations:

C-c
: means hitting the `c` key while pressing the `Ctrl` key.

M-x
: means hitting the `x` key while pressing the `Meta` key, which is called `Alt` on many systems. Alternatively one can type `Escape` followed by `x` (in separate key strokes).

RET
: is the `Enter`, `Return` or  key.

SPC
: is the space bar.

By default, terms and types printed by interactive commands will be simplified by evaluating certain functions. Specifically, only those functions that match on a constructor pattern or a projection copattern will be reduced, as well as primitive functions of Agda. For example, `(suc x) + y` will be simplified to `suc (x + y)` but `id 42` will be printed as is. The level of normalisation of commands that print a term or type can be modified by prefixing the command by one or more repetitions of `C-u`:

- `C-u` to print the term without any simplification.
- `C-u C-u` to print the full normal form.
- `C-u C-u C-u` to print the weak-head normal form.

Global commands

These commands can be invoked both from within or outside of a hole. When invoked from within a hole and whenever it makes sense, they limit their action to the hole and take context and content of the hole into account.

C-c C-l
: **L**oad file. This type-checks the contents of the file, and replaces each occurrence of a question mark `?` or a hole marker `{! !}` by a freshly created hole.

C-c C-x C-c
: **C**ompile file. This will compile an Agda program with a `main` function using a given backend (the `GHC` backend is used by default).

C-c C-i
: Call a given backend's top-level (or hole) **i**nteraction command (if any).

C-c C-x C-q
: **Q**uit, kill the Agda process.

C-c C-x C-r
: Kill and **r**estart the Agda process.

C-c C-x C-s
: **S**witch to a different Agda version.

C-c C-x C-a
: **A**bort a command.

C-c C-x C-d
Remove goals and highlighting (**d**eactivate).

C-c C-x C-h
Toggle display of **h**idden arguments.

C-c C-x C-i
Toggle display of **i**rrelevant arguments.

C-c C-f
Move to next goal (**f**orward).

C-c C-b
Move to previous goal (**b**ackwards).

C-c C-?
Show all goals.

C-c C-=
Show constraints.

C-c C-s
Solve constraints. Tries to fill holes with existing meta variable solutions (as displayed by C-c C-=).

C-c C-a
Automatic Proof Search (Auto) Tries to fill holes by **a**utomatic type-directed term synthesis.

C-c C-d
Infer (**d**educe) type. The system asks for a term and infers its type. When executed inside a hole, it will instead take the contents of the hole as input (if any).

C-c C-n
Compute **n**ormal form. The system asks for a term which is then evaluated. When executed inside a hole, it will instead take the contents of the hole as input (if any).

C-u C-c C-n
Compute normal form, ignoring *abstract* and *NON_TERMINATING*.

C-u C-u C-c C-n
Compute and print normal form of `show <expression>`.

C-u C-u C-u C-c C-n
Compute weak head normal form.

C-c C-o
Display c**o**ntents of the given m**o**dule.

C-c C-w
Why in scope, given a defined name returns how it was brought into scope and its definition.

C-c C-z
Search Definitions in Scope

C-c C-x M-;
Comment/uncomment rest of buffer.

Commands in context of a goal

The following commands only work (and make sense) inside of a hole.

Commands expecting input (for example which variable to case split) will either use the text inside the goal or ask the user for input.

C-c C-SPC
Give (fill goal)

C-c C-r
Refine. Checks whether the return type of the expression `e` in the hole matches the expected type. If so, the hole is replaced by `e { }1 ... { }n`, where a sufficient number of new holes have been inserted. If the hole is empty, then the refine command instead inserts a lambda or constructor (if there is a unique type-correct choice).

C-c C-m
Elaborate and Give (fill goal with normalized expression). Takes the same `C-u` prefixes as `C-c C-n`.

C-c C-c
Case split. If the cursor is positioned in a hole which denotes the right hand side of a definition, then this command automatically performs pattern matching on variables of your choice. When given several variables (separated by spaces) it will case split on the first and then continue by case splitting on the remaining variables in each newly created clause. When given no variables, it will introduce a new variable if the target type is a function type, or introduce a new copattern match if the target type is a record type (see *Copatterns*). When given the special symbol `.`, it will expand the ellipsis `...` in the clause (see *With-Abstraction*).

C-c C-h
Compute type of **h**elper function and add type signature to kill ring (clipboard).

C-c C-t
Goal **t**ype.

C-c C-e
Context (**e**nvironment).

C-c C-,
Goal type and context. Shows the goal type, i.e. the type expected in the current hole, along with the types of locally defined identifiers.

C-c C-.
Goal type, context and inferred type.

C-c C-;
Goal type, context and checked term.

Other commands

TAB
Indent current line, cycles between points.

S-TAB
Indent current line, cycles in opposite direction.

M-.
Go to definition of identifier under point.

Middle mouse button
Go to definition of identifier clicked on.

C-u M-.
Go to definition of a prompted identifier.

M-?
Query a list of references in loaded files

C-M-.
Query a list of identifiers that match a prompt. The prompt may consist of multiple words that can occur in any order or a regular expression.

`M-,`
: Go back to previous location.

4.5.5 Unicode input

How can I write Unicode characters using Emacs?

The Agda Emacs mode comes with an input method for easily writing Unicode characters. Most Unicode character can be input by typing their corresponding TeX/LaTeX commands, eg. typing `\lambda` will input λ. Some characters have key bindings which have not been taken from TeX/LaTeX (typing `\bN` results in $\mathbb{N}$ being inserted, for instance), but all bindings start with `\`.

To see all characters you can input using the Agda input method type `M-x describe-input-method RET Agda` or type `M-x agda-input-show-translations RET RET` (with some exceptions in certain versions of Emacs).

If you know the Unicode name of a character you can input it using `M-x ucs-insert RET` (which supports tab-completion) or `C-x 8 RET`. Example: Type `C-x 8 RET not SPACE a SPACE sub TAB RET` to insert the character "NOT A SUBSET OF" ($\not\subset$).

(The Agda input method has one drawback: if you make a mistake while typing the name of a character, then you need to start all over again. If you find this terribly annoying, then you can use Abbrev mode instead. However, note that Abbrev mode cannot be used in the minibuffer, which is used to give input to many Agda and Emacs commands.)

The Agda input method can be customised via `M-x customize-group RET agda-input`.

OK, but how can I find out what to type to get the ... character?

To find out how to input a specific character, eg from the standard library, position the cursor over the character and type `M-x describe-char` or `C-u C-x =`.

For instance, for :: I get the following:

```
            character: ∷ (displayed as ∷) (codepoint 8759, #o21067, #x2237)
    preferred charset: unicode (Unicode (ISO10646))
code point in charset: 0x2237
               script: symbol
               syntax: w      which means: word
             category: .:Base, c:Chinese
             to input: type "\::" with Agda input method
          buffer code: #xE2 #x88 #xB7
            file code: #xE2 #x88 #xB7 (encoded by coding system utf-8-unix)
              display: by this font (glyph code)
    x:-misc-fixed-medium-r-normal--20-200-75-75-c-100-iso10646-1 (#x2237)

Character code properties: customize what to show
  name: PROPORTION
  general-category: Sm (Symbol, Math)
  decomposition: (8759) ('∷')

There are text properties here:
  fontified            t
```

Here it says that I can type `\::` to get a ::. If there is no "to input" line, then you can add a key binding to the Agda input method by using `M-x customize-variable RET agda-input-user-translations`.

Show me some commonly used characters

Many common characters have a shorter input sequence than the corresponding TeX command:

- **Arrows**: `\r-` for →. You can replace `r` with another direction: `u`, `d`, `l`. Eg. `\d-` for ↓. Replace `-` with `=` or `==` to get a double and triple arrows.
- **Greek letters** can be input by `\G` followed by the first character of the letters Latin name. Eg. `\Gl` will input λ while `\GL` will input Λ.
- **Negation**: you can get the negated form of many characters by appending `n` to the name. Eg. while `\ni` inputs ∋, `\nin` will input ∉.
- **Subscript** and **superscript**: you can input subscript or superscript forms by prepending the character with `\_` (subscript) or `\^` (superscript). Eg. `g\_1` will input g_1. Note that not all characters have a subscript or superscript counterpart in Unicode.

Note: to introduce multiple characters involving greek letters, subscripts or superscripts, you need to prepend `\G`, `\_` or `\^` respectively before each character.

Some characters which were used in this documentation or which are commonly used in the standard library (sorted by hexadecimal code):

Hex code	Character	Short key-binding	TeX command
00AC	¬		`\neg`
00D7	×	`\x`	`\times`
02E2	^{s}	`\^s`	
03BB	λ	`\Gl`	`\lambda`
041F	PDF TODO		
0432	PDF TODO		
0435	PDF TODO		
0438	PDF TODO		
043C	PDF TODO		
0440	PDF TODO		
0442	PDF TODO		
1D62	$_i$	`\_i`	
2032	′	`\'1`	`\prime`
207F	^{n}	`\^n`	
2081	$_1$	`\_1`	
2082	$_2$	`\_2`	
2083	$_3$	`\_3`	
2084	$_4$	`\_4`	
2096	$_k$	`\_k`	
2098	$_m$	`\_m`	
2099	$_n$	`\_n`	

Hex code	Character	Short key-binding	TeX command
2113	ℓ		`\ell`

Hex code	Character	Short key-binding	TeX command
2115	ℕ	\bN	\Bbb{N}
2191	↑	\u	\uparrow
2192	→	\r-	\to
21A6	↦	\r-|	\mapsto
2200	∀	\all	\forall
2208	∈		\in
220B	∋		\ni
220C	∌	\nin	
2218	∘	\o	\circ
2237	∷	\::	
223C	∼	\~	\sim
2248	≈	\~~	\approx
2261	≡	\==	\equiv
2264	≤	\<=	\le
2284	⊄	\subn	
228E	⊎	\u+	\uplus
2294	⊔	\lub	
22A2	⊢	\\|-	\vdash
22A4	⊤		\top
22A5	⊥		\bot
266D	♭	\b	
266F	♯	\#	
27E8	⟨	\<	
27E9	⟩	\>	

Hex code	Character	Short key-binding	TeX command
2983	PDF TODO	\{{	
2984	PDF TODO	\}}	
2985	PDF TODO	\((	
2986	PDF TODO	\))	

Hex code	Character	Short key-binding	TeX command
2C7C	$_j$	_j	

4.5.6 Background highlighting

Agda uses various background colors to indicate specific errors or warnings in your code. Specifically, the following colors are used:

- A *yellow* background indicates unsolved metavariables (see *Metavariables*) or unsolved constraints.
- A *light salmon* (pink-orange) background indicates an issue with termination or productivity checking (see *Termination Checking*).
- A *wheat* (light yellow) background indicates an issue with coverage checking (see *Coverage Checking*).
- A *peru* (brown) background indicates an issue with positivity checking (see *Positivity Checking*).
- An *orange* background indicates a type signature with a missing definition.
- A *light coral* (darker pink) background indicates a fatal warning
- A *grey* background indicates unreachable or dead code, and for shadowed variable names in telescopes.

- A *white smoke* (light grey) background indicates a clauses that does not hold definitionally (see *Case trees*).
- A *pink* background indicates an issue with confluence checking of rewrite rules (see *Confluence checking*).

4.6 Literate Programming

Agda supports a limited form of literate programming, i.e. code interspersed with prose, if the corresponding filename extension is used.

4.6.1 Literate TeX

Files ending in `.lagda` or `.lagda.tex` are interpreted as literate TeX files. All code has to appear in code blocks:

```
Ignored by Agda.

\begin{code}[ignored by Agda]
module Whatever where
-- Agda code goes here
\end{code}
```

Text outside of code blocks is ignored, as well as text right after `\begin{code}`, on the same line.

Agda finds code blocks by looking for the first instance of `\begin{code}` that is not preceded on the same line by `%` or `\` (not counting `\` followed by any code point), then (starting on the next line) the first instance of `\end{code}` that is preceded by nothing but spaces or tab characters (`\t`), and so on (always starting on the next line). Note that Agda does not try to figure out if, say, the LaTeX code changes the category code of `%`.

If you provide a suitable definition for the code environment, then literate Agda files can double as LaTeX document sources. Example definition:

```
\usepackage{fancyvrb}

\DefineVerbatimEnvironment
  {code}{Verbatim}
  {} % Add fancy options here if you like.
```

The *LaTeX backend* or the preprocessor lhs2TeX can also be used to produce LaTeX code from literate Agda files. See *Known pitfalls and issues* for how to make LaTeX accept Agda files using the UTF-8 character encoding.

4.6.2 Literate reStructuredText

Files ending in `.lagda.rst` are interpreted as literate reStructuredText files. Agda will parse code following a line ending in `::`, as long as that line does not start with `..`:

```
This line is ordinary text, which is ignored by Agda.

::

  module Whatever where
  -- Agda code goes here

Another non-code line.
::
.. This line is also ignored
```

reStructuredText source files can be turned into other formats such as HTML or LaTeX using Sphinx.

- Code blocks inside an rST comment block will be type-checked by Agda, but not rendered.
- Code blocks delimited by `.. code-block:: agda` or `.. code-block:: lagda` will be rendered, but not type-checked by Agda.
- All lines inside a codeblock must be further indented than the first line of the code block.
- Indentation must be consistent between code blocks. In other words, the file as a whole must be a valid Agda file if all the literate text is replaced by white space.

4.6.3 Literate Markdown and Typst

Files ending in `.lagda.md` are interpreted as literate Markdown files, while files ending in `.lagda.typ` are interpreted as literate Typst files. They use the same syntax for code blocks, and they are parsed the same way by Agda. Code blocks start with ``` or ```agda on its own line, and end with ```, also on its own line:

````
This line is ordinary text, which is ignored by Agda.

```
module Whatever where
-- Agda code goes here
```

Here is another code block:

```agda
data ℕ : Set where
 zero : ℕ
 suc  : ℕ → ℕ
```
````

For Typst, Agda does not yet support highlighting the code blocks.

Markdown source files can be turned into many other formats such as HTML or LaTeX using PanDoc.

- Code blocks which should be type-checked by Agda but should not be visible when the Markdown is rendered may be enclosed in HTML comment delimiters (`<!--` and `-->`).
- Code blocks which should be ignored by Agda, but rendered in the final document may be indented by four spaces.
- Note that inline code fragments are not supported due to the difficulty of interpreting their indentation level with respect to the rest of the file.

Only `agda` code blocks

Added in version 2.9.0.

By default, both unmarked code blocks (```) and explicitly marked code blocks (```agda) are treated as Agda code.

With the *`--literate-markdown-only-agda-blocks`* command-line option (off by default), only code blocks explicitly marked with ```agda are treated as Agda code. Unmarked code blocks are treated as verbatim text and are not type-checked. This allows including other code examples in the document without Agda attempting to parse them.

Example with `--literate-markdown-only-agda-blocks`:

````
This is prose.

Here is some Agda code:
````

(continues on next page)

(continued from previous page)

````
```agda
data Bool : Set where
  true false : Bool
```

Here is a JavaScript example that is NOT type-checked:

```
function hello() { return "world"; }
```

Here is another verbatim block with a language tag:

```haskell
main = putStrLn "Hello, World!"
```
````

This option is not available as pragma since it affects parsing before any pragma options are processed. It must be set via command line (`agda --literate-markdown-only-agda-blocks`) or passed under `flags:` in the `.agda-lib` file.

4.6.4 Literate Org

Files ending in `.lagda.org` are interpreted as literate Org files. Code blocks are surrounded by two lines including only `` `#+begin_src agda2` `` and `` `#+end_src` `` (case-insensitive).

````
This line is ordinary text, which is ignored by Agda.

#+begin_src agda2
module Whatever where
-- Agda code goes here
#+end_src

Another non-code line.
````

- Code blocks which should be ignored by Agda, but rendered in the final document may be placed in source blocks without the `agda2` label.

4.6.5 Literate Forester

Files ending in `.lagda.tree` are interpreted as literate Forester files. Literate forester uses `` `\agda{...}` `` for code blocks.

- Run `agda --html --html-highlight=code example.lagda.tree` to generate `html/example.tree`.
- Add `html/` to the `trees` list in `forest.toml` so Forester can find the generated trees.
- Modify `theme/tree.xsl` of your forester project to include `Agda.css` in the linked stylesheets.

Running `forester build` produce file `output/example/index.xml`.

- Run `cp html/Agda.css output/`, now you get Agda syntax highlighting.

```
\p{This line is ordinary text, which is ignored by Agda.}

\agda{
module Whatever where
-- Agda code goes here
}

\p{Here is another code block:}

\agda{
data ℕ : Set where
 zero : ℕ
 suc  : ℕ → ℕ
}
```

- Self-link issue with Agda + Forester: When compiling `.lagda.tree` files, Agda generates links to local definitions using the module name (e.g., `bool.html#232`). However, Forester outputs pages as `output/bool/index.html`. This mismatch causes self-referential links to resolve to `bool/bool.html#232` instead of `#232` on the current page, resulting in 404s. This cannot be fixed in Agda's HTML backend - it has no awareness of Forester's output structure. A post-processing script is needed: for each generated page, copy `output/i/index.html` to `output/i/i.html` so the incorrect paths become valid redirects.
- A similar problem occurs with references to Agda modules not compiled as part of your forest - whether standard library modules or local `.agda` files without corresponding trees. A script could rewrite these as root-relative paths (e.g., `/Agda.Primitive.html#388`), which works if you host at a domain root. But this isn't general - on GitHub Pages, for example, your site lives at `your-id.github.io/your-repo/`, so the correct path would be `/your-repo/Agda.Primitive.html#388` - requiring the script to know your deployment prefix. Either way, you also need to copy the generated HTML files from `html/` to your output directory - Forester won't include them automatically.

4.7 Generating HTML

To generate highlighted, hyperlinked web pages from source code, run the following command in a shell:

```
$ agda --html --html-dir={output directory} {root module}
```

You can change the way in which the code is highlighted by providing your own CSS file instead of the `default, included one` (use the *`--css`* option).

> **Note**
>
> The `Agda.css` shipped with Agda is located at `html/Agda.css` in the Agda data directory. Since version 2.6.2, the Agda data directory can be printed using the option *`--print-agda-dir`*, which has been an alias of *`--print-agda-data-dir`* since 2.6.4.1. Thus, you can get hold of the CSS file via `cat $(agda --print-agda-data-dir)/html/Agda.css`.

You can also get highlighting for all occurrences of the symbol the mouse pointer is hovering over in the HTML by adding the *`--highlight-occurrences`* option. The default behaviour is to only highlight the single symbol under the mouse pointer.

If you're using Literate Agda with Markdown or reStructedText and you want to highlight your Agda codes with Agda's HTML backend and render the rest of the content (let's call it "literate" part for convenience) with some another renderer, you can use the `--html-highlight=code` option, which makes the Agda compiler:

- not wrapping the literate part into <a class="Background"> tags
- not wrapping the generated document with a <html> tag, which means you'll have to specify the CSS location somewhere else, like <link rel="stylesheet" type="text/css" href="Agda.css">
- converting <a class="Markup"> tags into <pre class="agda-code"> tags that wrap the complete Agda code block below
- generating files with extension as-is (i.e. *.lagda.md* becomes *.md*, *.lagda.rst* becomes *.rst*)
- for reStructuredText, a `.. raw:: html` will be inserted before every code blocks

This will affect all the files involved in one compilation, making pure Agda code files rendered without HTML footer/header as well. To use `code` with literate Agda files and `all` with pure Agda files, use `--html-highlight=auto`, which means auto-detection.

4.7.1 Options

--html

Generate HTML files with highlighted source code.

--html-dir={DIR}

Set directory in which HTML files are placed to `DIR` (default: `html`).

--html-highlight=[code,all,auto]

Added in version 2.6.0.

Whether to highlight non-Agda code as comments in generated HTML files (default: `all`).

--css={URL}

Set URL of the CSS file used by the HTML files to `URL` (can be relative).

--highlight-occurrences

Added in version 2.6.2.

When *generating HTML*, place the `highlight-hover.js` script in the output directory (see *--html-dir*). In the presence of the script, hovering over an identifier in the rendering of the HTML will highlight all occurrences of the same identifier on the page.

4.8 Generating LaTeX

The LaTeX backend was added in Agda 2.3.2. It can be used as follows:

```
$ agda --latex {file}.lagda
$ cd latex
$ {latex-compiler} {file}.tex
```

where *latex-compiler* could be **pdflatex**, **xelatex** or **lualatex**, and *file*`.lagda` is a *literate Agda TeX file* (it could also be called *file*`.lagda.tex`). The source file is expected to import the LaTeX package `agda` by including the code `\usepackage{agda}` (possibly with some options). Unlike the *HTML backend* only the top-most module is processed. Imported modules can be processed by invoking `agda --latex` manually on each of them.

The LaTeX backend checks if `agda.sty` is found by the LaTeX environment. If it isn't, a default `agda.sty` is copied into the LaTeX output directory (by default `latex`). Note that the appearance of typeset code can be modified by overriding definitions from `agda.sty`.

Note

The `agda.sty` shipped with Agda is located at `latex/agda` in the Agda data directory. Since version 2.6.2, the Agda data directory can be printed using the option *--print-agda-dir*, which has been an alias of *--print-agda-data-dir* since 2.6.4.1. Thus, you can get hold of the class file via `cat $(agda --print-agda-data-dir)/latex/agda.sty`.

4.8.1 Options

The following command-line options change the behaviour of the LaTeX backend:

--latex

Generate LaTeX with highlighted source code.

--only-scope-checking

Generates highlighting without typechecking the file. See *Quicker generation without typechecking*.

--latex-dir={DIR}

Added in version 2.5.2.

Set directory where `agda.sty` and the generated LaTeX files are placed to `DIR` (default: `latex`).

--count-clusters

Added in version 2.5.3.

Count extended grapheme clusters when generating LaTeX code (see *Counting Extended Grapheme Clusters*). Available only when Agda was built with Cabal flag `enable-cluster-counting`.

This option can be given in *OPTIONS* pragmas since 2.5.4.

--no-count-clusters

Added in version 2.6.4.

Opposite of *--count-clusters*. Default.

The following options can be given when loading `agda.sty` by using `\usepackage[options]{agda}`:

bw

Colour scheme which highlights in black and white.

conor

Colour scheme similar to the colours used in Epigram 1.

references

Enables *inline typesetting* of referenced code.

links

Enables *hyperlink support*.

4.8.2 Known pitfalls and issues

- Unicode characters may not be typeset properly out of the box. How to address this problem depends on what LaTeX engine is used.
  - pdfLaTeX:

    The pdfLaTeX program does not by default understand the UTF-8 character encoding. You can tell it to treat the input as UTF-8 by using the inputenc package:

```
\usepackage[utf8]{inputenc}
```

If the inputenc package complains that some Unicode character is "not set up for use with LaTeX", then you can give your own definition. Here is one example:

```
\usepackage{newunicodechar}
\newunicodechar{λ}{\ensuremath{\mathnormal\lambda}}
\newunicodechar{←}{\ensuremath{\mathnormal\from}}
\newunicodechar{→}{\ensuremath{\mathnormal\to}}
\newunicodechar{∀}{\ensuremath{\mathnormal\forall}}
```

– XeLaTeX or LuaLaTeX:

It can sometimes be easier to use LuaLaTeX or XeLaTeX. When these engines are used it might suffice to choose a suitable font, as long as it contains all the right symbols in all the right shapes. If it does not, then `\newunicodechar` can be used as above. Here is one example:

```
\usepackage{unicode-math}
\setmathfont{XITS Math}

\usepackage{newunicodechar}
\newunicodechar{λ}{\ensuremath{\mathnormal\lambda}}
```

In recent versions of LuaLaTeX, you can avoid using `\newunicodechar` at all by instead setting up a chain of fallback fonts, e.g.

```
\usepackage{luaotfload}
\directlua{luaotfload.add_fallback
  ("mycustomfallback",
    { "SymbolamonospacifiedforSourceCodePro:style=Regular;"
    , "NotoSansMono:style=Regular;"
    , "NotoSansMath:style=Regular;"
    }
  )}
\defaultfontfeatures{RawFeature={fallback=mycustomfallback}}
```

- If < and > are typeset like ¡ and ¿, then the problem might be that you are using pdfLaTeX and have not selected a suitable font encoding.

  Possible workaround:

```
\usepackage[T1]{fontenc}
```

- If a regular text font is used, then `--` might be typeset as an en dash (–).

  Possible workarounds:

  – Use a monospace font.

  – Turn off ligatures. With pdfLaTeX the following code (which also selects a font encoding, and only turns off ligatures for character sequences starting with `-`) might work:

```
\usepackage[T1]{fontenc}
\usepackage{microtype}
\DisableLigatures[-]{encoding=T1}
```

With LuaLaTeX or XeLaTeX the following code (which also selects a font) might work:

```
\usepackage{fontspec}
\defaultfontfeatures[\rmfamily]{}
\setmainfont{Latin Modern Roman}
```

Note that you might not want to turn off all kinds of ligatures in the entire document. See the *Examples* below for information on how to set up special font families without TeX ligatures that are only used for Agda code.

- The unicode-math package and older versions of the polytable package are incompatible, which can result in errors in generated LaTeX code.

  Possible workaround: Download a more up-to-date version of polytable and put it together with the generated files or install it globally.

4.8.3 Quicker generation without typechecking

A faster variant of the backend is available by invoking `QuickLaTeX` from the Emacs mode, or using `agda --latex --only-scope-checking`. When this variant of the backend is used the top-level module is not type-checked, only scope-checked. Note that this can affect the generated document. For instance, scope-checking does not resolve overloaded constructors.

If the module has already been type-checked successfully, then this information is reused; in this case QuickLaTeX behaves like the regular LaTeX backend.

4.8.4 Features

Vertical space

Code blocks are by default surrounded by vertical space. Use `\AgdaNoSpaceAroundCode{}` to avoid this vertical space, and `\AgdaSpaceAroundCode{}` to reenable it.

Note that, if `\AgdaNoSpaceAroundCode{}` is used, then empty lines before or after a code block will not necessarily lead to empty lines in the generated document. However, empty lines inside the code block do (by default, with or without `\AgdaNoSpaceAroundCode{}`) lead to empty lines in the output. The height of such empty lines can be controlled by the length `\AgdaEmptySkip`, which by default is `\abovedisplayskip`.

Alignment

Tokens preceded by two or more space characters, as in the following example, are aligned in the typeset output:

```
\begin{code}
data ℕ : Set where
  zero  : ℕ
  suc   : ℕ → ℕ

_+_ : ℕ → ℕ → ℕ
zero   + n = n
suc m  + n = suc (m + n)
\end{code}
```

In the case of the first token on a line a single space character sometimes suffices to get alignment. A constraint on the indentation of the first token t on a line is determined as follows:

- Let T be the set containing every previous token (in any code block) that is either the initial token on its line or preceded by at least one whitespace character.
- Let S be the set containing all tokens in T that are not *shadowed* by other tokens in T. A token t_1 is shadowed by t_2 if t_2 is further down than t_1 and does not start to the right of t_1.

- Let L be the set containing all tokens in S that start to the left of t, and E be the set containing all tokens in S that start in the same column as t.
- The constraint is that t must be indented further than every token in L, and aligned with every token in E.

Note that if any token in L or E belongs to a previous code block, then the constraint may not be satisfied unless (say) the `AgdaAlign` *environment* is used in an appropriate way. If custom settings are used, for instance if `\AgdaIndent` is redefined, then the constraint discussed above may not be satisfied.

Examples:

- Here `C` is indented further than `B`:

```
postulate
  A  B
      C : Set
```

- Here `C` is not (necessarily) indented further than `B`, because `X` shadows `B`:

```
postulate
  A  B  : Set
  X
      C : Set
```

These rules are inspired by, but not identical to, the one used by lhs2TeX's poly mode (see Section 8.4 of the manual for lhs2TeX version 1.17).

Counting Extended Grapheme Clusters

The alignment feature regards the string +_, containing + and a combining character, as having length two. However, it seems more reasonable to treat it as having length one, as it occupies a single column, if displayed "properly" using a monospace font. The flag *--count-clusters* is an attempt at fixing this. When this flag is enabled the backend counts "extended grapheme clusters" rather than code points.

Note that this fix is not perfect: a single extended grapheme cluster might be displayed in different ways by different programs, and might, in some cases, occupy more than one column. Here are some examples of extended grapheme clusters, all of which are treated as a single character by the alignment algorithm:

```
+_
0̈̂
PDF TODOPDF TODO
PDF TODOPDF TODOPDF TODO
PDF TODOPDF TODOPDF TODOPDF TODOPDF TODOPDF TODOPDF TODOPDF TODOPDF TODOPDF TODO
```

Note also that the layout machinery does not count extended grapheme clusters, but code points. The following code is syntactically correct, but if *--count-clusters* is used, then the LaTeX backend does not align the two field keywords:

```
record +_ : Set₁ where  field A : Set
                        field B : Set
```

The *--count-clusters* flag is not enabled in all builds of Agda, because the implementation depends on the ICU library, the installation of which could cause extra trouble for some users. The presence of this flag is controlled by the Cabal flag `enable-cluster-counting`.

Breaking up code blocks

Sometimes one might want to break up a code block into multiple pieces, but keep code in different blocks aligned with respect to each other. Then one can use the `AgdaAlign` environment. Example usage:

```
 \begin{AgdaAlign}
 \begin{code}
   code
     code  (more code)
 \end{code}
 Explanation...
 \begin{code}
   aligned with "code"
     code  (aligned with (more code))
\end{code}
\end{AgdaAlign}
```

Note that `AgdaAlign` environments should not be nested.

Sometimes one might also want to hide code in the middle of a code block. This can be accomplished in the following way:

```
\begin{AgdaAlign}
\begin{code}
  visible
\end{code}
\begin{code}[hide]
  hidden
\end{code}
\begin{code}
  visible
\end{code}
\end{AgdaAlign}
```

However, the result may be ugly: extra space is perhaps inserted around the code blocks. The `AgdaSuppressSpace` environment ensures that extra space is only inserted before the first code block, and after the last one (but not if `\AgdaNoSpaceAroundCode{}` is used). Example usage:

```
\begin{AgdaAlign}
\begin{code}
  code
    more code
\end{code}
Explanation...
\begin{AgdaSuppressSpace}
\begin{code}
  aligned with "code"
    aligned with "more code"
\end{code}
\begin{code}[hide]
  hidden code
\end{code}
\begin{code}
    also aligned with "more code"
\end{code}
```

(continues on next page)

(continued from previous page)

```
\end{AgdaSuppressSpace}
\end{AgdaAlign}
```

Note that `AgdaSuppressSpace` environments should not be nested. There is also a combined environment, `AgdaMultiCode`, that combines the effects of `AgdaAlign` and `AgdaSuppressSpace`.

Hiding code

Code that you do not want to show up in the output can be hidden by giving the argument `hide` to the code block:

```
\begin{code}[hide]
-- the code here will not be part of the final document
\end{code}
```

Hyperlinks (experimental)

If the hyperref latex package is loaded before the agda package and the `links` option is passed to the agda package, then the agda package provides a function called `\AgdaTarget`. Identifiers which have been declared targets, by the user, will become clickable hyperlinks in the rest of the document. Here is a small example:

```
\documentclass{article}
\usepackage{hyperref}
\usepackage[links]{agda}
\begin{document}

\AgdaTarget{ℕ}
\AgdaTarget{zero}
\begin{code}
data ℕ : Set where
  zero  : ℕ
  suc   : ℕ → ℕ
\end{code}

See next page for how to define \AgdaFunction{two} (doesn't turn into a
link because the target hasn't been defined yet). We could do it
manually though; \hyperlink{two}{\AgdaDatatype{two}}.

\newpage

\AgdaTarget{two}
\hypertarget{two}{}
\begin{code}
two : ℕ
two = suc (suc zero)
\end{code}

\AgdaInductiveConstructor{zero} is of type
\AgdaDatatype{ℕ}. \AgdaInductiveConstructor{suc} has not been defined to
be a target so it doesn't turn into a link.

\newpage

Now that the target for \AgdaFunction{two} has been defined the link
```

(continues on next page)

(continued from previous page)

```
works automatically.

\begin{code}
data Bool : Set where
  true false : Bool
\end{code}

The AgdaTarget command takes a list as input, enabling several targets
to be specified as follows:

\AgdaTarget{if, then, else, if\_then\_else\_}
\begin{code}
if_then_else_ : {A : Set} → Bool → A → A → A
if true then t else f = t
if false then t else f = f
\end{code}

\newpage

Mixfix identifier need their underscores escaped:
\AgdaFunction{if\_then\_else\_}.

\end{document}
```

The borders around the links can be suppressed using hyperref's `hidelinks` option:

```
\usepackage[hidelinks]{hyperref}
```

> **Warning**
>
> The current approach to links does not keep track of scoping or types, and hence overloaded names might create links which point to the wrong place. Therefore it is recommended to not overload names when using the links option at the moment. This might get fixed in the future.

Numbered code listings

When the option `number` is used an equation number is generated for the code listing. The number is set to the right, centered vertically. By default the number is set in parentheses, but this can be changed by redefining `\AgdaFormatCodeNumber`.

The option can optionally be given an argument: when `number=l` is used a label `l`, referring to the code listing, is generated. It is possible to use this option several times with different labels.

An example:

```
\begin{code}[number=code:lemma]
  a proof
\end{code}
%
A consequence of Lemma~\ref{code:lemma} is that...
```

The option `number` has no effect if used together with `hide`, `inline` or `inline*`.

Inline code

Code can be typeset inline by giving the argument `inline` to the code block:

```
Assume that we are given a type
%
\begin{code}[hide]
  module _ (
\end{code}
\begin{code}[inline]
    A : Set
\end{code}
\begin{code}[hide]
    ) where
\end{code}
%
.
```

There is also a variant of `inline`, `inline*`. If `inline*` is used, then space (`\AgdaSpace{}`) is added at the end of the code, and when `inline` is used space is not added.

The implementation of these options is a bit of a hack. Only use these options for typesetting a single line of code without multiple consecutive whitespace characters (except at the beginning of the line).

Another way to typeset inline code

An alternative to using `inline` and `inline*` is to typeset code manually. Here is an example:

```
Below we postulate the existence of a type called
\AgdaPostulate{apa}:
%
\begin{code}
  postulate apa : Set
\end{code}
```

You can find all the commands used by the backend (and which you can use manually) in the `agda.sty` file.

Semi-automatically typesetting inline code (experimental)

Since Agda version 2.4.2 there is experimental support for semi-automatically typesetting code inside text, using the `references` option. After loading the agda package with this option, inline Agda snippets will be typeset in the same way as code blocks—after post-processing—if referenced using the `\AgdaRef` command. Only the current module is used; should you need to reference identifiers in other modules, then you need to specify which other module manually by using `\AgdaRef[module]{identifier}`.

In order for the snippets to be typeset correctly, they need to be post-processed by the **`postprocess-latex.pl`** script from the Agda data directory. You can copy it into the current directory by issuing the command

```
$ cp $(agda --print-agda-data-dir)/latex/postprocess-latex.pl .
```

In order to generate a PDF, you can then do the following:

```
$ agda --latex {file}.lagda
$ cd latex/
$ perl ../postprocess-latex.pl {file}.tex > {file}.processed
```

(continues on next page)

(continued from previous page)

```
$ mv {file}.processed {file}.tex
$ xelatex {file}.tex
```

Here is a full example, consisting of a Literate Agda file `Example.lagda` and a makefile `Makefile`.

Listing 1: Example.lagda

```
\documentclass{article}
\usepackage[references]{agda}

\begin{document}

Here we postulate \AgdaRef{apa}.
%
\begin{code}
  postulate apa : Set
\end{code}

\end{document}
```

Listing 2: Makefile

```
AGDA=agda
AFLAGS=-i. --latex
SOURCE=Example
POSTPROCESS=postprocess-latex.pl
LATEX=latexmk -pdf -use-make -xelatex

all:
    $(AGDA) $(AFLAGS) $(SOURCE).lagda
    cd latex/ && \
    perl ../$(POSTPROCESS) $(SOURCE).tex > $(SOURCE).processed && \
    mv $(SOURCE).processed $(SOURCE).tex && \
    $(LATEX) $(SOURCE).tex && \
    mv $(SOURCE).pdf ..
```

See Issue #1054 on the bug tracker for implementation details.

Warning

Overloading identifiers should be avoided. If multiple identifiers with the same name exist, then AgdaRef will typeset according to the first one it finds.

Controlling the typesetting of individual tokens

The typesetting of (certain) individual tokens can be controlled by redefining the `\AgdaFormat` command. Example:

```
\usepackage{ifthen}

% Insert extra space before some tokens.
\DeclareRobustCommand{\AgdaFormat}[2]{%
  \ifthenelse{
```

(continues on next page)

(continued from previous page)

```
    \equal{#1}{≡⟨} \OR
    \equal{#1}{≡⟨⟩} \OR
    \equal{#1}{■}
  }{\ }{}#2}
```

Note the use of `\DeclareRobustCommand`. The first argument to `\AgdaFormat` is the token, and the second argument the thing to be typeset.

Emulating %format rules

The LaTeX backend has no feature directly comparable to lhs2TeX's `%format` rules. However, one can hack up something similar by using a program like **sed**. For instance, let us say that `replace.sed` contains the following text:

```
# Turn Σ[ x ∈ X ] into (x : X) ×.
s/\\AgdaRecord{Σ\[} \(.*\) \\AgdaRecord{∈} \(.*\) \\AgdaRecord{]}/\\AgdaSymbol\{(\}\1 \\
↪AgdaSymbol\{:\} \2\\AgdaSymbol\{)\} \\AgdaFunction\{×\}/g
```

The output of the LaTeX backend can then be postprocessed in the following way:

```
$ sed -f replace.sed {file}.tex > {file}.sedded
$ mv {file}.sedded {file}.tex
```

Including Agda code in a larger LaTeX document

Sometimes you might want to include a bit of code without making the whole document a literate Agda file. There are two ways in which this can be accomplished.

(The following technique was probably invented by Anton Setzer.) Put the code in a separate file, and use `\newcommand` to give a name to each piece of code that should be typeset:

Listing 3: Code.lagda.tex

```
\newcommand{\nat}{%
\begin{code}
data ℕ : Set where
  zero  : ℕ
  suc   : (n : ℕ) → ℕ
\end{code}}
```

Preprocess this file using Agda, and then include it in another file in the following way:

Listing 4: Main.tex

```
% In the preamble:
\usepackage{agda}
% Further setup related to Agda code.

% The Agda code can be included either in the preamble or in the
% document's body.
\input{Code}

% Then one can refer to the Agda code in the body of the text:
The natural numbers can be defined in the following way in Agda:
\nat{}
```

Here it is assumed that `agda.sty` is available in the current directory (or on the TeX search path).

Note that this technique can also be used to present code in a different order, if the rules imposed by Agda are not compatible with the order that you would prefer.

There is another technique that uses the catchfilebetweentags latex package. Assuming you have some code in `Code.lagda` and want to include it in `Paper.tex`, you first add tags to your code as follows:

Listing 5: Code.lagda

```
%<*nat>
\begin{code}
data ℕ : Set where
  zero  : ℕ
  suc   : (n : ℕ) → ℕ
\end{code}
%</nat>

%<*plus>
\begin{code}
_+_ : ℕ → ℕ → ℕ
zero   + n = n
suc m  + n = suc (m + n)
\end{code}
%</plus>
```

You can then use `\ExecuteMetaData`, as provided by catchfilebetweentags, to include the code. Note that the code does not have to be in the same order (or from the same files). This method is particularly convenient when you want to write a paper or presentation about a library of code.

Listing 6: Paper.tex

```
% Other setup related to Agda...
\usepackage{catchfilebetweentags}

\begin{document}

  \begin{itemize}
    \item The natural numbers
  \end{itemize}

  \ExecuteMetaData[latex/Code.tex]{nat}

  \begin{itemize}
    \item Addition (\AgdaFunction{\_+\_})
  \end{itemize}

  \ExecuteMetaData[latex/Code.tex]{plus}
```

4.8.5 Examples

Some examples that can be used for inspiration (in the HTML version of the manual you see links to the source code and in the PDF version of the manual you see inline source code).

- For the `article` class and pdfLaTeX:

```
\documentclass{article}

% Use the input encoding UTF-8 and the font encoding T1.
\usepackage[utf8]{inputenc}
\usepackage[T1]{fontenc}

% Support for Agda code.
\usepackage{agda}

% Customised setup for certain characters.
\usepackage{newunicodechar}
\newunicodechar{∀}{\ensuremath{\mathnormal{\forall}}}
\newunicodechar{→}{\ensuremath{\mathnormal{\to}}}
\newunicodechar{₁}{\ensuremath{{}_1}}

% Support for Greek letters.
\usepackage{alphabeta}

% Disable ligatures that start with '-'. Note that this affects the
% entire document!
\usepackage{microtype}
\DisableLigatures[-]{encoding=T1}

\begin{document}

Some code:
\begin{code}
{-# OPTIONS --without-K --count-clusters #-}

open import Agda.Builtin.String

-- A comment with some TeX ligatures:
-- --, ---, ?`, !`, `, ``, ', '', <<, >>.

Θ₁ : Set → Set
Θ₁ = λ A → A

a-name-with--hyphens : ∀ {A : Set} → A → A
a-name-with--hyphens ff--fl = ff--fl

ffi : String
ffi = "--"
\end{code}
Note that the code is indented.

\end{document}
```

- For the `article` class and LuaLaTeX or XeLaTeX:
  - If you want to use the default fonts (with—at the time of writing—bad coverage of non-ASCII characters):

```
\documentclass{article}
```

(continues on next page)

(continued from previous page)

```
% Support for Agda code.
\usepackage{agda}

% Use special font families without TeX ligatures for Agda code. (This
% code is inspired by a comment by Enrico Gregorio/egreg:
% https://tex.stackexchange.com/a/103078.)
\usepackage{fontspec}
\newfontfamily{\AgdaSerifFont}{Latin Modern Roman}
\newfontfamily{\AgdaSansSerifFont}{Latin Modern Sans}
\newfontfamily{\AgdaTypewriterFont}{Latin Modern Mono}
\renewcommand{\AgdaFontStyle}[1]{{\AgdaSansSerifFont{}#1}}
\renewcommand{\AgdaKeywordFontStyle}[1]{{\AgdaSansSerifFont{}#1}}
\renewcommand{\AgdaStringFontStyle}[1]{{\AgdaTypewriterFont{}#1}}
\renewcommand{\AgdaCommentFontStyle}[1]{{\AgdaTypewriterFont{}#1}}
\renewcommand{\AgdaBoundFontStyle}[1]{\textit{\AgdaSerifFont{}#1}}

% Workarounds for the fact that the Latin Modern Sans font does not
% support certain characters. An alternative would be to use another
% font.
\usepackage{newunicodechar}
\newunicodechar{λ}{\ensuremath{\mathnormal{\lambda}}}
\newunicodechar{∀}{\ensuremath{\mathnormal{\forall}}}
\newunicodechar{₁}{\ensuremath{{}_1}}

\begin{document}

Some code:
\begin{code}
{-# OPTIONS --without-K --count-clusters #-}

open import Agda.Builtin.String

-- A comment with some TeX ligatures:
-- --, ---, ?`, !`, `, ``, ', '', <<, >>.

Θ₁ : Set → Set
Θ₁ = λ A → A

a-name-with--hyphens : ∀ {A : Set} → A → A
a-name-with--hyphens ff--fl = ff--fl

ffi : String
ffi = "--"
\end{code}
Note that the code is indented.

\end{document}
```

– If you would prefer to use other fonts (with possibly better coverage):

```
\documentclass{article}
```

(continues on next page)

(continued from previous page)

```
% Support for Agda code.
\usepackage{agda}

% Use fonts with a decent coverage of non-ASCII characters.
\usepackage{fontspec}
\setmainfont{DejaVu Serif}
\setsansfont{DejaVu Sans}
\setmonofont{DejaVu Sans Mono}

% Use special font families without TeX ligatures for Agda code. (This
% code is inspired by a comment by Enrico Gregorio/egreg:
% https://tex.stackexchange.com/a/103078.)
\newfontfamily{\AgdaSerifFont}{DejaVu Serif}
\newfontfamily{\AgdaSansSerifFont}{DejaVu Sans}
\newfontfamily{\AgdaTypewriterFont}{DejaVu Sans Mono}
\renewcommand{\AgdaFontStyle}[1]{{\AgdaSansSerifFont{}#1}}
\renewcommand{\AgdaKeywordFontStyle}[1]{{\AgdaSansSerifFont{}#1}}
\renewcommand{\AgdaStringFontStyle}[1]{{\AgdaTypewriterFont{}#1}}
\renewcommand{\AgdaCommentFontStyle}[1]{{\AgdaTypewriterFont{}#1}}
\renewcommand{\AgdaBoundFontStyle}[1]{\textit{\AgdaSerifFont{}#1}}

\begin{document}

Some code:
\begin{code}
{-# OPTIONS --without-K --count-clusters #-}

open import Agda.Builtin.String

-- A comment with some TeX ligatures:
-- --, ---, ?`, !`, `, ``, ', '', <<, >>.

Θ₁ : Set → Set
Θ₁ = λ A → A

a-name-with--hyphens : ∀ {A : Set} → A → A
a-name-with--hyphens ff--fl = ff--fl

ffi : String
ffi = "--"
\end{code}
Note that the code is indented.

\end{document}
```

- For the beamer class and pdfLaTeX:

```
\documentclass{beamer}

% Use the input encoding UTF-8 and the font encoding T1.
\usepackage[utf8]{inputenc}
\usepackage[T1]{fontenc}
```

(continues on next page)

(continued from previous page)

```
% Support for Agda code.
\usepackage{agda}

% Decrease the indentation of code.
\setlength{\mathindent}{1em}

% Customised setup for certain characters.
\usepackage{newunicodechar}
\newunicodechar{∀}{\ensuremath{\mathnormal{\forall}}}
\newunicodechar{→}{\ensuremath{\mathnormal{\to}}}
\newunicodechar{₁}{\ensuremath{{}_1}}

% Support for Greek letters.
\usepackage{alphabeta}

% Disable ligatures that start with '-'. Note that this affects the
% entire document!
\usepackage{microtype}
\DisableLigatures[-]{encoding=T1}

\begin{document}

\begin{frame}
  Some code:
  \begin{code}
  {-# OPTIONS --without-K --count-clusters #-}

  open import Agda.Builtin.String

  -- A comment with some TeX ligatures:
  -- --, ---, ?`, !`, `, ``, ', '', <<, >>.

  Θ₁ : Set → Set
  Θ₁ = λ A → A

  a-name-with--hyphens : ∀ {A : Set} → A → A
  a-name-with--hyphens ff--fl = ff--fl

  ffi : String
  ffi = "--"
  \end{code}
  Note that the code is indented.
\end{frame}

\end{document}
```

- For the beamer class and LuaLaTeX or XeLaTeX:

```
\documentclass{beamer}

% Support for Agda code.
```

(continues on next page)

(continued from previous page)

```
\usepackage{agda}

% Decrease the indentation of code.
\setlength{\mathindent}{1em}

% Use special font families without TeX ligatures for Agda code. (This
% code is inspired by a comment by Enrico Gregorio/egreg:
% https://tex.stackexchange.com/a/103078.)
\usepackage{fontspec}
\newfontfamily{\AgdaSerifFont}{Latin Modern Roman}
\newfontfamily{\AgdaSansSerifFont}{Latin Modern Sans}
\newfontfamily{\AgdaTypewriterFont}{Latin Modern Mono}
\renewcommand{\AgdaFontStyle}[1]{{\AgdaSansSerifFont{}#1}}
\renewcommand{\AgdaKeywordFontStyle}[1]{{\AgdaSansSerifFont{}#1}}
\renewcommand{\AgdaStringFontStyle}[1]{{\AgdaTypewriterFont{}#1}}
\renewcommand{\AgdaCommentFontStyle}[1]{{\AgdaTypewriterFont{}#1}}
\renewcommand{\AgdaBoundFontStyle}[1]{\textit{\AgdaSansSerifFont{}#1}}

% Workarounds for the fact that the Latin Modern Sans font does not
% support certain characters.
\usepackage{newunicodechar}
\newunicodechar{λ}{\ensuremath{\mathnormal{\lambda}}}
\newunicodechar{∀}{\ensuremath{\mathnormal{\forall}}}
\newunicodechar{₁}{\ensuremath{{}_1}}

\begin{document}

\begin{frame}
  Some code:
  \begin{code}
  {-# OPTIONS --without-K --count-clusters #-}

  open import Agda.Builtin.String

  -- A comment with some TeX ligatures:
  -- --, ---, ?`, !`, `, ``, ', '', <<, >>.

  Θ₁ : Set → Set
  Θ₁ = λ A → A

  a-name-with--hyphens : ∀ {A : Set} → A → A
  a-name-with--hyphens ff--fl = ff--fl

  ffi : String
  ffi = "--"
  \end{code}
  Note that the code is indented.
\end{frame}

\end{document}
```

- For the `acmart` class and pdfLaTeX:

```
\documentclass[acmsmall]{acmart}

% Use the UTF-8 encoding.
\usepackage[utf8]{inputenc}

% Support for Agda code.
\usepackage{agda}

% Code should be indented.
\setlength{\mathindent}{1em}

% Customised setup for certain characters.
\usepackage{newunicodechar}
\newunicodechar{∀}{\ensuremath{\mathnormal{\forall}}}
\newunicodechar{₁}{\ensuremath{{}_{\textsf{1}}}}

% Support for Greek letters.
\usepackage{alphabeta}

% Disable ligatures that start with '-'. Note that this affects the
% entire document! Note also that if all you want to do is to ensure
% that the comment starter '--' is typeset with two characters, then
% you do not need this command, because '--' is not typeset as an en
% dash (-) when the typewriter font is used.
\DisableLigatures[-]{encoding=T1}

\begin{document}
\acmConference{Some conference}
\maketile

Some code:
\begin{code}
{-# OPTIONS --without-K --count-clusters #-}

open import Agda.Builtin.String

-- A comment with some TeX ligatures:
-- --, ---, ?`, !`, `, ``, ', '', <<, >>.

Θ₁ : Set → Set
Θ₁ = λ A → A

a-name-with--hyphens : ∀ {A : Set} → A → A
a-name-with--hyphens ff--fl = ff--fl

ffi : String
ffi = "--"
\end{code}
Note that the code is indented.

\end{document}
```

- For the `acmart` class and XeLaTeX:

```
\documentclass[acmsmall]{acmart}

% Support for Agda code.
\usepackage{agda}

% Code should be indented.
\setlength{\mathindent}{1em}

% Use special font families without TeX ligatures for Agda code. (This
% code is inspired by a comment by Enrico Gregorio/egreg:
% https://tex.stackexchange.com/a/103078.)
\usepackage{fontspec}
\newfontfamily{\AgdaSerifFont}{Linux Libertine O}
\newfontfamily{\AgdaSansSerifFont}{Linux Biolinum O}
\newfontfamily{\AgdaTypewriterFont}{inconsolata}
\renewcommand{\AgdaFontStyle}[1]{{\AgdaSansSerifFont{}#1}}
\renewcommand{\AgdaKeywordFontStyle}[1]{{\AgdaSansSerifFont{}#1}}
\renewcommand{\AgdaStringFontStyle}[1]{{\AgdaTypewriterFont{}#1}}
\renewcommand{\AgdaCommentFontStyle}[1]{{\AgdaTypewriterFont{}#1}}
\renewcommand{\AgdaBoundFontStyle}[1]{\textit{\AgdaSerifFont{}#1}}

\begin{document}
\acmConference{Some conference}
\maketile

Some code:
\begin{code}
{-# OPTIONS --without-K --count-clusters #-}

open import Agda.Builtin.String

-- A comment with some TeX ligatures:
-- --, ---, ?`, !`, `, ``, ', '', <<, >>.

Θ₁ : Set → Set
Θ₁ = λ A → A

a-name-with--hyphens : ∀ {A : Set} → A → A
a-name-with--hyphens ff--fl = ff--fl

ffi : String
ffi = "--"
\end{code}
Note that the code is indented.

\end{document}
```

Note that these examples might not satisfy all your requirements, and might not work in all settings (in particular, for LuaLaTeX or XeLaTeX it might be necessary to install one or more fonts). If you have to follow a particular house style, then you may want to make sure that the Agda code follows this style, and that you do not inadvertently change the style of other text when customising the style of the Agda code.

4.8.6 Generating lagda files directly from Agda using agdaLatex

A tool for for creating lagda files and corresponding LaTeX files directly from Agda code has been created. See <https://github.com/csetzer/agdaLatex>

4.9 Interface files

> **Note**
>
> This is a stub. Contributions, additions and corrections are greatly appreciated.

When an `.agda` file is saved, another file with the same name and extension `.agdai` is automatically created. The latter file is what we call an **interface file**.

Interface files store the results from the type-checking process. These results include:

- A translation of pattern-matching definitions to case trees (this translation speeds up computation).
- The resolution of all implicit arguments. (Note: under the option `--allow-unsolved-metas` not **all** implicit arguments need to be resolved to create an interface file.)

4.9.1 Storage

If an Agda file has one or more *associated* `.agda-lib` files, then the *project root* is the directory containing these files. In that case the Agda file's interface file is stored in the directory `_build/VERSION` under the project root. Different directories are used for different versions of Agda so that one can switch between versions without losing the interface files.

If an Agda file does not have any associated `.agda-lib` file, then its `.agdai` file is stored in the same directory as the Agda file. (With at least one exception, see Agda bug #6134.)

> **Note**
>
> When an `.agda` file is renamed, the old `.agdai` file is kept, and a new `.agdai` file is created. This is the intended behavior, and the orphan files can be safely deleted from the user's file system if needed.

The compression run to create `.agdai` files introduces sharing. Sharing improves the memory efficiency of the code loaded from interface files.

The syntax represented in `.agdai` files differs significantly from the syntax of source files.

4.9.2 Compilation

An external module is loaded by loading its interface file. Interface files are also intermediate points when compiling through a backend to e.g. Haskell.

4.10 Library Management

Agda has a simple package management system to support working with multiple libraries in different locations. The central concept is that of a *library*.

4.10.1 Example: Using the standard library

Before we go into details, here is some quick information for the impatient on how to tell Agda about the location of the standard library, using the library management system.

Let's assume you have downloaded the standard library into a directory which we will refer to by `AGDA_STDLIB` (as an absolute path). A library file `standard-library.agda-lib` should exist in this directory, with the following content:

```
name: standard-library
include: src
```

To use the standard library by default in your Agda projects, you have to do two things:

1. Create a file `AGDA_DIR/libraries` with the following content:

   ```
   AGDA_STDLIB/standard-library.agda-lib
   ```

   (Of course, replace `AGDA_STDLIB` by the actual path.)

   The `AGDA_DIR` defaults to `~/.config/agda` on unix-like systems and `C:\Users\USERNAME\AppData\Roaming\agda` or similar on Windows. (More on `AGDA_DIR` below.)

   Remark: The `libraries` file informs Agda about the libraries you want it to know about.

2. Create a file `AGDA_DIR/defaults` with the following content:

   ```
   standard-library
   ```

   Remark: The `defaults` file informs Agda which of the libraries pointed to by `libraries` should be used by default (i.e. in the default include path).

That's the short version, if you want to know more, read on!

4.10.2 Library files

A library consists of

- a name
- a set of dependencies
- a set of include paths
- a set of default flags

Libraries are defined in `.agda-lib` files with the following syntax:

```
name: LIBRARY-NAME  -- Comment
depend: LIB1 LIB2
  LIB3
  LIB4
include: PATH1
  PATH2
  PATH3
flags: OPTION1 OPTION2
  OPTION3
```

Dependencies are library names, not paths to `.agda-lib` files, and include paths are relative to the location of the library-file.

Default flags can be any valid pragma options (see *Command-line and pragma options*).

Each of the four fields is optional. Naturally, unnamed libraries cannot be depended upon. But dropping the `name` is possible if the library file only serves to list include paths and/or dependencies of the current project.

4.10.3 The `.agda-lib` files associated to a given Agda file

When a given file is type-checked Agda uses the options from the `flags` fields of its library file (if there is such). If the command-line option *`--no-libraries`* is used, then no library file is used. Otherwise the library file is found in the following way:

- First the file's root directory is found. If the top-level module in the file is called `A.B.C`, then it has to be in the directory `root/A/B` or `root\A\B`. The root directory is the directory `root`.
- If `root` contains any `.agda-lib` files, then the search stops. If there is exactly one such file, it is used, otherwise an error is raised.
- If `root` contains no `.agda-lib` files, a search is made upwards in the directory hierarchy, and the search stops once one or more `.agda-lib` files are found in a directory. If no `.agda-lib` files are found all the way to the top of the directory hierarchy, then none are used.

Note also that there must not be any `.agda-lib` files below the root, on the path to the Agda file. For instance, if the top-level module in the Agda file is called `A.B.C`, and it is in the directory `root/A/B`, then there must not be any `.agda-lib` files in `root/A` or `root/A/B`.

4.10.4 Installing libraries

To be found by Agda a library file has to be listed (with its full path) in a `libraries` file

- `AGDA_DIR/libraries-VERSION`, or if that doesn't exist
- `AGDA_DIR/libraries`

where `VERSION` is the Agda version (for instance `2.5.1`). The `AGDA_DIR` defaults to `~/.config/agda` on unix-like systems and `C:\Users\USERNAME\AppData\Roaming\agda` or similar on Windows, and can be overridden by setting the `AGDA_DIR` environment variable.

The `AGDA_DIR` will fall-back to `~/.agda`, if it exists, for backward compatibility reasons. You can find the precise location of `AGDA_DIR` by running `agda --print-agda-app-dir`.

Each line of the libraries file shall be the absolute file system path to the root of a library, or a comment line starting with `--` followed by a space character.

Environment variables in the paths (of the form `$VAR` or `${VAR}`) are expanded. The location of the `libraries` file used can be overridden using the *`--library-file`* command line option.

You can find out the precise location of the `libraries` file by calling `agda -l fjdsk Dummy.agda` at the command line and looking at the error message (assuming you don't have a library called `fjdsk` installed).

Note that if you want to install a library so that it is used by default, it must also be listed in the `defaults` file (details below).

4.10.5 Using a library

There are three ways a library gets used:

- You supply the `--library=LIB` (or `-l LIB`) option to Agda. This is equivalent to adding a `-iPATH` for each of the include paths of `LIB` and its (transitive) dependencies. In this case the current directory is *not* implicitly added to the include paths.

- No explicit *--library* option is given, and the current project root (of the Agda file that is being loaded) or one of its parent directories contains an `.agda-lib` file defining a library `LIB`. This library is used as if a `--library=LIB` option had been given, except that it is not necessary for the library to be listed in the `AGDA_DIR/libraries` file.
- No explicit *--library* option, and no `.agda-lib` file in the project root. In this case the file `AGDA_DIR/defaults` is read and all libraries listed are added to the path. The `defaults` file should contain a list of library names, each on a separate line. In this case the current directory is *also* added to the path.

  To disable default libraries, you can give the option *--no-default-libraries*. To disable using libraries altogether, use the *--no-libraries* option.

4.10.6 Default libraries

If you want to usually use a variety of libraries, it is simplest to list them all in a `defaults` file

- `AGDA_DIR/defaults-VERSION`, or if that does not exist
- `AGDA_DIR/defaults`

where `VERSION` is the Agda version (for instance `2.5.1`). `default-VERSION` has the benefit that you only need to have installed the mentioned library for the one `VERSION` of the compiler you target.

Each line of the defaults file shall be the name of a library resolvable using the paths listed in the libraries file. For example,

```
standard-library
library2
library3
```

where of course `library2` and `library3` are the libraries you commonly use. While it is safe to list all your libraries in `library`, be aware that listing libraries with name clashes in `defaults` can lead to difficulties, and should be done with care (i.e. avoid it unless you really must).

4.10.7 Version numbers

Library names can end with a version number (for instance, `mylib-1.2.3`). When resolving a library name (given in a *--library* option, or listed as a default library or library dependency) the following rules are followed:

- If you don't give a version number, any version will do.
- If you give a version number an exact match is required.
- When there are multiple matches an exact match is preferred, and otherwise the latest matching version is chosen.

For example, suppose you have the following libraries installed: `mylib`, `mylib-1.0`, `otherlib-2.1`, and `otherlib-2.3`. In this case, aside from the exact matches you can also say `--library=otherlib` to get `otherlib-2.3`.

4.10.8 Upgrading

If you are upgrading from a pre 2.5 version of Agda, be aware that you may have remnants of the previous library management system in your preferences. In particular, if you get warnings about `agda2-include-dirs`, you will need to find where this is defined. This may be buried deep in `.el` files, whose location is both operating system and emacs version dependant.

4.11 Performance debugging

Sometimes your Agda program doesn't type check or run as fast as you expected. This section describes some tools available to figure out why not.

> **Note**
>
> This is a stub

4.11.1 Measuring typechecking performance

The Agda Emacs mode has an interactive highlighting feature, which highlights the term that is currently being type checked. This can often reveal which pieces of a definition slow down type checking. To enable interactive highlighting, use `M-x customize-group agda2-highlight` and set `Agda2 Highlight Level` to `Interactive`.

Agda can do some internal book-keeping of how time is spent, which can be turned on using the `--profile` flag:

`--profile``=definitions`
: Break down by time spent checking each top-level definition.

`--profile``=modules`
: Break down by time spent checking each top-level module.

`--profile``=internal`
: Break down by activity (such as parsing, type checking, termination checking, etc).

The Haskell runtime system can also tell you something about how Agda spends its time:

`+RTS` `-s -RTS`
: Show memory usage and time spent on garbage collection.

External tools

- agda-bench is a tool for benchmarking compile-time evaluation and type checking performance of Agda programs.

4.11.2 Measuring run-time performance

Agda programs are compiled (by default) via Haskell (see *Compilers*), so the GHC profiling tools can be applied to Agda programs. For instance,

```
> agda -c Test.agda --ghc-flag=-prof --ghc-flag=-fprof-auto
> ./Test +RTS -p
```

A complication is that the GHC backend generates names like `d76`, so making sense of the profiling output can require a little bit of work.

External tools

- agda-criterion has bindings for a small part of the criterion Haskell library for performance measurement.
- agda-ghc-names can translate the names in generated Haskell code back to Agda names.

4.12 Search Definitions in Scope

Since version 2.5.1 Agda supports the command `Search About` that searches the objects in scope, looking for definitions matching a set of constraints given by the user.

4.12.1 Usage

The tool is invoked by choosing `Search About` in the goal menu or pressing `C-c C-z`. It opens a prompt and users can then input a list of space-separated identifiers and string literals. The search returns the definitions in scope whose type contains *all* of the mentioned identifiers and whose name contains *all* of the string literals as substrings.

For instance, in the following module:

```
open import Agda.Builtin.Char
open import Agda.Builtin.Char.Properties
open import Agda.Builtin.String
open import Agda.Builtin.String.Properties
```

running `Search About` on `Char String` returns:

Definitions about Char, String

primShowChar
: Char → String

primStringFromList
: Agda.Builtin.List.List Char → String

primStringToList
: String → Agda.Builtin.List.List Char

primStringToListInjective

: (a b
[String) →] primStringToList a Agda.Builtin.Equality.≡ primStringToList b → a Agda.Builtin.Equality.≡ b

and running `Search About` on `String "Injective"` returns:

Definitions about String, "Injective"

primStringToListInjective

: (a b
[String) →] primStringToList a Agda.Builtin.Equality.≡ primStringToList b → a Agda.Builtin.Equality.≡ b

CHAPTER

FIVE

CONTRIBUTE

Agda and its related libraries are hosted at Github. To contribute, you will need to fork a repository, make the changes and then send a pull request (PR).

A code of conduct and other considerations are described in the HACKING.md file in the root of the Agda repository.

You can also take a look at the current Agda issues to help us solve them. You can start with the label difficulty: easy and help wanted. You can also explore all the labels.

Note

The Agda User Manual is a work-in-progress and is still incomplete. Contributions, additions, and corrections to the Agda manual are greatly appreciated.

5.1 Documentation

Documentation is written in reStructuredText format.

The Agda documentation is shipped together with the main Agda repository in the `doc/user-manual` subdirectory. The content of this directory is automatically published to https://agda.readthedocs.io.

5.1.1 Rendering documentation locally

- To build the user manual locally, you need to install the following dependencies:
  - Python $\geq$3.3
  - `Sphinx` and `sphinx-rtd-theme`

    ```
    pip install --user -r doc/user-manual/requirements.txt
    ```

    Note that the `--user` option puts the Sphinx binaries in `$HOME/.local/bin`.
  - ImageMagick with SVG and PNG support; check output of

    ```
    convert -list format
    ```

  - LaTeX
  - PyDvi

  To see the list of available targets, execute `make help` in `doc/user-manual`. E.g., call `make html` to build the documentation in html format.

5.1.2 Type-checking code examples

You can include code examples in your documentation.

If your give the documentation file the extension `.lagda.rst`, Agda will recognise it as an Agda file and type-check it.

> **Tip**
>
> If you edit `.lagda.rst` documentation files in Emacs, you can use Agda's interactive mode to write your code examples. Run `M-x agda2-mode` to switch to Agda mode, and `M-x rst-mode` to switch back to rST mode.

You can check that all the examples in the manual are type-correct by running `make user-manual-test` from the root directory. This check will be run as part of the continuous integration build.

> **Warning**
>
> Remember to run `fix-agda-whitespace` to remove trailing whitespace before submitting the documentation to the repository.

5.1.3 Syntax for code examples

The syntax for embedding code examples depends on:

1. Whether the code example should be *visible* to the reader of the documentation.
2. Whether the code example contains *valid* Agda code (which should be type-checked).

Visible, checked code examples

This is code that the user will see, and that will be also checked for correctness by Agda. Ideally, all code in the documentation should be of this form: both *visible* and *valid*.

```
It can appear stand-alone:

::

   data Bool : Set where
     true false : Bool


Or at the end of a paragraph::

   data Bool : Set where
     true false : Bool

Here ends the code fragment.
```

Result:

It can appear stand-alone:

```
data Bool : Set where
  true false : Bool
```

Or at the end of a paragraph:

```
data Bool : Set where
  true false : Bool
```

Here ends the code fragment.

Warning

Remember to always leave a blank line after the `::`. Otherwise, the code will be checked by Agda, but it will appear as regular paragraph text in the documentation.

Visible, unchecked code examples

This is code that the reader will see, but will not be checked by Agda. It is useful for examples of incorrect code, program output, or code in languages different from Agda.

```
.. code-block:: agda

   -- This is not a valid definition

   ω : ∀ a → a
   ω x = x


.. code-block:: haskell

   -- This is haskell code

   data Bool = True | False
```

Result:

```
-- This is not a valid definition

ω : ∀ a → a
ω x = x
```

```
-- This is haskell code

data Bool = True | False
```

Invisible, checked code examples

This is code that is not shown to the reader, but which is used to typecheck the code that is actually displayed.

This might be definitions that are well known enough that do not need to be shown again.

```
..
  ::
  data Nat : Set where
    zero : Nat
    suc  : Nat → Nat
```

(continues on next page)

(continued from previous page)

```
::

  add : Nat → Nat → Nat
  add zero y = y
  add (suc x) y = suc (add x y)
```

Result:

```
add : Nat → Nat → Nat
add zero y = y
add (suc x) y = suc (add x y)
```

File structure

Documentation literate files (`.lagda.*`) are type-checked as whole Agda files, as if all literate text was replaced by whitespace. Thus, **indentation** is interpreted globally.

Namespacing

In the documentation, files are typechecked starting from the `doc/user-manual/` root. For example, the file `doc/user-manual/language/data-types.lagda.rst` should start with a hidden code block declaring the name of the module as `language.data-types`:

```
..
  ::
  module language.data-types where
```

Scoping

Sometimes you will want to use the same name in different places in the same documentation file. You can do this by using hidden module declarations to isolate the definitions from the rest of the file.

```
..
  ::
  module scoped-1 where

::

    foo : Nat
    foo = 42

..
  ::
  module scoped-2 where

  ::
    foo : Nat
    foo = 66
```

Result:

```
foo : Nat
foo = 42
```

CHAPTER

SIX

THE AGDA TEAM AND LICENSE

Agda 2 was originally written by Ulf Norell, partially based on code from Agda 1 by Catarina Coquand and Makoto Takeyama, and from Agdalight by Ulf Norell and Andreas Abel. Cubical Agda was originally contributed by Andrea Vezzosi.

Agda 2 is currently actively developed mainly by (in alphabetical order):

- Andreas Abel
- Nathaniel Burke
- Lawrence Chonavel
- Jesper Cockx
- Nils Anders Danielsson
- András Kovács

Agda 2 has received major contributions by the following developers, amongst others. Some contributors have pioneered a feature which shall be mentioned here. But many have worked on these features for improvements and maintenance.

- Andreas Abel: *termination checker, sized types, irrelevance, copatterns, erasure, github workflows, stackage*
- Arthur Adjedj: `LevelUniv`
- Guillaume Allais: *warnings, pattern guards, interleaved mutual blocks, standard library 1.0 and above*
- Malin Altenmüller: `--polarity`
- Stevan Andjelkovic: *LaTeX backend*
- Miëtek Bak: *Agda logo*
- Marcin Benke: *original "Alonzo" compiler to Haskell*
- Jean-Philippe Bernardy: `syntax` *declarations*
- Guillaume Brunerie
- Joris Ceulemans: `--polarity`
- James Chapman
- Liang-Ting Chen: *github workflows*
- Lawrence Chonavel
- Jonathan Coates: *performance*
- Jesper Cockx: *rewriting, unification* `--without-K`, *recursive instance search*, *reflection*, `Prop`, *cumulativity*
- Catarina Coquand: *Agda 1*

- Jonathan Coates: *performance*
- Matthew Daggitt: *standard library 1.0 and above*
- Nils Anders Danielsson: *efficient positivity checker, HTML backend, highlighting, standard library*, `--cubical=erased`, *erasure*, *performance improvements*
- Dominique Devriese: `instance` *arguments*
- Péter Diviánszky: *web frontent,* `variable` *declarations*
- Lucas Escot: `--polarity`
- Robert Estelle: *refactoring of backends, main driver*
- Naïm Favier
- Olle Fredriksson: *Epic compiler backend*
- Paolo Giarrusso
- Adam Gundry: *pattern synonyms*
- Daniel Gustafsson: *Epic compiler backend*
- Alex Haršáni: *GenericError refactorings*
- Philipp Hausmann: *treeless compiler, UHC compiler backend, testsuite runner, Travis CI*
- Kuen-Bang Hou "favonia"
- Patrik Jansson
- Alan Jeffrey: *JavaScript compiler backend*
- Phil de Joux: some *hlinting*
- Wolfram Kahl
- Philip Kaludercic: *emacs mode refactoring*
- Andre Knispel: *reflection*, `INJECTIVE_FOR_INFERENCE`
- Wen Kokke
- András Kovács: *performance, serialization*
- John Leo
- Fredrik Lindblad: *Agsy proof search "Auto"*
- Víctor López Juan: *"tog" prototype, markdown frontend, documentation*
- Amélia Liao: *maintenance and enhancements of Cubical Agda and instance search*, `opaque` *definitions, parallel type-checking* `-j`, *colored output*
- Ting-Gan Lua
- Francesco Mazzoli: *"tog" prototype*
- James McKinna: *standard library 1.7 and above*
- Stefan Monnier
- Guilhem Moulin: *highlighting*
- Reed Mullanix: *performance, emacs mode fixes*
- Fredrik Nordvall Forsberg: *pattern lambdas, warnings*
- Konstantin Nisht

- Ulf Norell: *Agda 2*
- Andreas Nuyts: `--polarity`
- Josselin Poiret: `--polarity`
- Nicolas Pouillard: *module record expressions*
- Jonathan Prieto: *Agda package manager*
- Christian Sattler
- Michael Shulman: *some Agda input key bindings*
- Andrés Sicard-Ramírez: *Agda releases, stackage, Travis CI*
- Lukas Skystedt: *Mimer proof search "Auto"*
- Nate Soares: *improvements of termination checking for copatterns*
- Makoto Takeyama: *Agda 1, Emacs mode, "MAlonzo" compiler to Haskell, serialization*
- Andrea Vezzosi: *Cubical Agda, Agda-flat, Agda-parametric, Guarded Cubical Agda*
- Szumi Xie: *some bug fixes*
- Noam Zeilberger: *pattern lambdas*
- Tesla Ice Zhang

The full list of code and documentation contributors (over 200) is available at https://github.com/agda/agda/graphs/contributors or from the git repository via `git shortlog -sne`. Numerous further individuals have contributed to Agda by reporting issues, building backends and editor support, packaging Agda etc.

The Agda license is here.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- search

BIBLIOGRAPHY

[McBride2004] C. McBride and J. McKinna. **The view from the left**. Journal of Functional Programming, 2004. http://strictlypositive.org/vfl.pdf.

INDEX

Symbols

A

B

C

L

M

N

O

P

R

S

T

U

W