Agda User Manual

Release 2.6.1

The Agda Team

Sep 17, 2019

Contents

CHAPTER 1

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.

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 2

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. Other languages in this tradition are Alf, Alfa, Agda 1, Cayenne. Some other loosely related languages are Coq, Epigram, and Idris.

Because of strong typing and dependent types, Agda can be used as a proof assistant, allowing 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 analyse 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 `Nat` (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 functions taking an argument `a` in a type `A` and 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 argument and produces the `n x n` identity matrix is then a function of type `identity : (n : Nat) -> (Mat n n)`.

Remark: We could of course just specify the `identity` function with the type `Nat -> Nat -> 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 takes arguments 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. Using only typing, it is for example possible to define

- equality on natural numbers
- properties of arithmetical operations
- the type `(n : Nat) -> (PrimRoot n)` consisting of functions computing primitive root in modular arithmetic.

Of course a program of the above type will be more difficult to write than the corresponding program of type `Nat -> Nat` which produces a natural number which is a primitive root. However, the difficulty can be compensated by the fact that the program is guaranteed to work: it cannot produce something which is not a primitive root.

On a more mathematical level, we can express formulas and prove them using an algorithm. For example, a function of type `(n : Nat) -> (PrimRoot n)` is also a proof that every natural number has a primitive root.

2.2 Prerequisites

You need recent versions of the following programs to compile Agda:

- GHC: https://www.haskell.org/ghc/
  - Agda have been tested with GHC 8.0.2, 8.2.2, 8.4.4 and 8.6.5.
- cabal-install: https://www.haskell.org/cabal/
- Alex: https://www.haskell.org/alex/
- Happy: https://www.haskell.org/happy/
- GNU Emacs: http://www.gnu.org/software/emacs/

You should also make sure that programs installed by *cabal-install* are on your shell's search path.

For instructions on installing a suitable version of Emacs under Windows, see *Installing Emacs under Windows*.

Non-Windows users need to ensure that the development files for the C libraries *zlib** and *ncurses** are installed (see http://zlib.net and http://www.gnu.org/software/ncurses/). Your package manager may be able to install these files for you. For instance, on Debian or Ubuntu it should suffice to run

```
apt-get install zlib1g-dev libncurses5-dev
```

as root to get the correct files installed.

Optionally one can also install the ICU library, which is used to implement the `--count-clusters` flag. Under Debian or Ubuntu it may suffice to install *libicu-dev*. Once the ICU library is installed one can hopefully enable the `--count-clusters` flag by giving the `-fenable-cluster-counting` flag to *cabal install*.

2.2.1 Installing Emacs under Windows

A precompiled version of Emacs 26.1, with the necessary mathematical fonts, is available at http://www.cs.uiowa.edu/~astump/agda.

2.3 Installation

There are several ways to install Agda:

- Using a *released source* package from Hackage
- Using a *binary package* prepared for your platform
- Using the *development version* from the Git repository

Agda can be installed using different flags (see *Installation Flags*).

2.3.1 Installation from Hackage

You can install the latest released version of Agda from Hackage. Install the *prerequisites* and then run the following commands:

```
cabal update
cabal install Agda
agda-mode setup
```

The last command tries to set up Emacs for use with Agda via the *Emacs mode*. 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-mode locate")))
```

It is also possible (but not necessary) to compile the Emacs mode's files:

```
agda-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.

2.3.2 Prebuilt Packages and System-Specific Instructions

Arch Linux

The following prebuilt packages are available:

- Agda
- Agda standard library

However, due to significant packaging bugs [such as this](https://bugs.archlinux.org/task/61904?project=5&string=agda), you might want to use alternative installation methods.

Debian / Ubuntu

Prebuilt packages are available for Debian and Ubuntu from Karmic onwards. To install:

```
apt-get install agda-mode
```

This should install Agda and the Emacs mode.

The standard library is available in Debian and Ubuntu from Lucid onwards. To install:

```
apt-get install agda-stdlib
```

More information:

- Agda (Debian)
- Agda standard library (Debian)
- Agda (Ubuntu)
- Agda standard library (Ubuntu)

Reporting bugs:

Please report any bugs to Debian, using:

```
reportbug -B debian agda
reportbug -B debian agda-stdlib
```

Fedora

Agda is packaged in Fedora (since before Fedora 18).

```
yum install Agda
```

will pull in emacs-agda-mode and ghc-Agda-devel.

FreeBSD

Packages are available from FreshPorts for Agda and Agda standard library.

NixOS

Agda is part of the Nixpkgs collection that is used by https://nixos.org/nixos. To install Agda and agda-mode for Emacs, type:

```
nix-env -f "<nixpkgs>" -iA haskellPackages.Agda
```

If you're just interested in the library, you can also install the library without the executable. The Agda standard library is currently not installed automatically.

OS X

Homebrew provides prebuilt packages for OS X. To install:

```
brew install agda
```

This should take less than a minute, and install Agda together with the Emacs mode and the standard library.

By default, the standard library is installed in `/usr/local/lib/agda/`. To use the standard library, it is convenient to add `/usr/local/lib/agda/standard-library.agda-lib` to `~/.agda/libraries`, and specify `standard-library` in `~/.agda/defaults`. Note this is not performed automatically.

It is also possible to install `--without-stdlib`, `--without-ghc`, or from `--HEAD`. Note this will require building Agda from source.

For more information, refer to the Homebrew documentation.

Note: If Emacs cannot find the `agda-mode` executable, it might help to install the exec-path-from-shell package by doing `M-x package-install RET exec-path-from-shell RET`, and adding

```
(exec-path-from-shell-initialize)
```

to your `.emacs` file.

2.3.3 Installation of the Development Version

After getting the development version following the instructions in the Agda wiki:

- Install the *prerequisites*
- In the top-level directory of the Agda source tree

- Follow the *instructions* for installing Agda from Hackage (except run `cabal install` instead of `cabal install Agda`) or
- You can try to install Agda (including a compiled Emacs mode) by running the following command:

```
make install
```

Note that on a Mac, because ICU is installed in a non-standard location, you need to specify this location on the command line:

```
make install-bin CABAL_OPTS='--extra-lib-dirs=/usr/local/opt/icu4c/lib --
↪extra-include-dirs=/usr/local/opt/icu4c/include'
```

2.3.4 Installation Flags

When installing Agda the following flags can be used:

`cpphs` Use cpphs instead of cpp. Default: off.

`debug` Enable debugging features that may slow Agda down. Default: off.

`flag enable-cluster-counting` Enable the `--count-clusters` flag (see *Counting Extended Grapheme Clusters*). Note that if `enable-cluster-counting` is `False`, then the `--count-clusters` flag triggers an error message. Default: off.

2.4 'Hello world' in Agda

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

```
module hello-world where

open import IO

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

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.

A quick line-by-line explanation:

- Agda programs are structured in *modules*. The first module in each file is the *top-level* module whose name matches the filename. The contents of a module are declaration such as *data types* and *function definitions*.
- Other modules can be imported using an `import` statement, for example `open import IO`. This imports the *IO* module from the standard library and brings its contents into scope.
- A module exporting a function `main : IO a` 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.

2.5 Quick Guide to Editing, Type Checking and Compiling Agda Code

2.5.1 Introduction

Agda programs are commonly edited using Emacs or Atom. To edit a module (assuming you have *installed* Agda and its Emacs mode (or Atom's) properly), start the editor and open a file ending in `.agda`. Programs are developed *interactively*, which means that one can type check code which is not yet complete: if a question mark (`?`) is used as a placeholder for an expression, and the buffer is then checked, Agda will replace the question mark with a "hole" which can be filled in later. One can also do various other things in the context of a hole: listing the context, inferring the type of an expression, and even evaluating an open term which mentions variables bound in the surrounding context.

The following commands are the most common (see *Notation for key combinations*):

`C-c C-l` Load. Type-checks the contents of the file.

`C-c C-,` Shows the goal type, i.e. the type expected in the current hole, along with the types of locally defined identifiers.

`C-c C-.` A variant of `C-c C-,` that also tries to infer the type of the current hole's contents.

`C-c C-SPC` Give. Checks whether the term written in the current hole has the right type and, if it does, replaces the hole with that term.

`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-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.

`C-c C-n` Normalise. The system asks for a term which is then evaluated.

`M-.` Go to definition. Goes to the definition site of the identifier under the cursor (if known).

`M-*` Go back (Emacs < 25.1)

`M-,` Go back (Emacs $\geq$ 25.1)

For information related to the Emacs mode (configuration, keybindings, Unicode input, etc.) see *Emacs Mode*.

2.5.2 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.

2.5.3 Writing mathematical symbols in source code

Agda uses Unicode characters in source files (more specifically: the UTF-8 character encoding). Almost any character can be used in an identifier (like $\forall$, α, $\wedge$, or $\spadesuit$, for example). It is therefore necessary to have spaces between most lexical units.

Many mathematical symbols can be typed using the corresponding LaTeX command names. For instance, you type `\forall` to input $\forall$. A more detailed description of how to write various characters is *available*.

(Note that if you try to read Agda code using another program, then you have to make sure that it uses the right character encoding when decoding the source files.)

2.5.4 Errors

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.

If you do not like the way 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.

2.5.5 Compiling Agda programs

To compile a module containing a function `main : IO A` for some `A` (where `IO` can be found in the Primitive.agda), use `C-c C-x C-c`. If the module is named `A.B.C` the resulting binary will be called `C` (located in the project's top-level directory, the one containing the `A` directory).

2.5.6 Batch-mode command

There is also a batch-mode command line tool: `agda`. To find out more about this command, use `agda --help`.

2.6 A List of Tutorials

2.6.1 Introduction to Agda

- **Ulf Norell and James Chapman.**
  - Dependently Typed Programming in Agda. This is aimed at functional programmers.
- **Ana Bove and Peter Dybjer.**
  - Dependent Types at Work. A gentle introduction including logic and proofs of programs.
- **Ana Bove, Peter Dybjer, and Ulf Norell.**
  - A Brief Overview of Agda - A Functional Language with Dependent Types (in TPHOLs 2009) with an example of reflection. Code
- **Anton Setzer.**
  - Lecture notes on Interactive Theorem Proving. Swansea University. These lecture notes are based on Agda and contain an introduction of Agda for students with a very basic background in logic and functional programming.
- **Daniel Peebles.**
  - Introduction to Agda. Video of talk from the January 2011 Boston Haskell session at MIT.
- **Conor McBride.**

  - Introduction to Dependently Typed Programming using Agda. (videos of lectures). Associated source files, with exercises.

- **Andreas Abel.**
  - Agda lecture notes. Lecture notes used in teaching functional programming: basic introduction to Agda, Curry-Howard, equality, and verification of optimizations like fusion.
- **Jan Malakhovski.**
  - Brutal [Meta]Introduction to Dependent Types in Agda
- **Thorsten Altenkirch.**
  - Computer Aided Formal Reasoning - online lecture notes
- **Daniel Licata.**
  - Dependently Typed Programming in Agda (OPLSS 2013).
- **Tesla Ice Zhang.**
  - Some books about Formal Verification in Agda (in Chinese)
  - A blog created with Literate Agda (in Chinese)
- **Phil Wadler.**
  - Programming Languages Foundations in Agda
- **Aaron Stump.**
  - Verified Functional Programming in Agda
- **Diviánszky Péter.**
  - Agda Tutorial
- **Musa Al-hassy.**
  - A slow-paced introduction to reflection in Agda

2.6.2 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. - More info and feedback
- 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.
- Advanced Functional Programming (in German), course at Ludwig-Maximilians-University Munich.
- 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.
- Interactive Theorem proving (CS__336), Anton Setzer, Swansea University, Lent 2008.
- 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

2.6.3 Miscellaneous

- Agda has a Wikipedia page

CHAPTER 3

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).
- Inside `private` type signatures in `abstract` blocks, abstract definitions do reduce. However, there are some problems with this. See Issue #418.
- 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

    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_ : ℤ → ℤ
    -ℤ (p , n) = (n , p)

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

    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! However, if we make `shape-of-ℤ` private, unfolding of abstract definitions like `ℤ` is enabled, and we succeed:

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

  abstract
    private
      shape-of-0ℤ : ∀ (x : ℤ) (is0ℤ : x ≡ℤ 0ℤ) → proj₁ x ≡ proj₂ x
      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
```

(continues on next page)

(continued from previous page)

```
    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 = 0
    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
```

Type signatures in `where` blocks are private, so it is fine to make type abbreviations in `where` blocks of abstract definitions:

```
module WherePrivate where
  abstract
    x : Nat
    x = proj₁ t
      where
      T = Nat × Nat
      t : T
      t = 0 , 1
      p : proj₁ t ≡ 0
      p = refl
```

Note that if `p` was not private, application `proj₁ t` in its type would be ill-formed, due to the abstract definition of `T`.

Named `where`-modules do not make their declarations private, thus this example will fail if you replace `x`'s `where` by `module M where`.

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 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*.

3.2.4 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 _*_ #-}

_==_ : 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
```

3.2.5 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`).

3.2.6 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    #-}
{-# 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
```

3.2.7 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* and *Int*):

```
primitive
  primNatToFloat             : Nat → Float
  primFloatPlus              : Float → Float → Float
  primFloatMinus             : Float → Float → Float
  primFloatTimes             : Float → Float → Float
  primFloatNegate            : Float → Float
  primFloatDiv               : Float → Float → Float
  primFloatEquality          : Float → Float → Bool
  primFloatLess              : Float → Float → Bool
  primFloatNumericalEquality : Float → Float → Bool
  primFloatNumericalLess     : Float → Float → Bool
  primRound                  : Float → Int
  primFloor                  : Float → Int
  primCeiling                : Float → Int
  primExp                    : Float → Float
  primLog                    : Float → Float
  primSin                    : Float → Float
  primCos                    : Float → Float
  primTan                    : Float → Float
  primASin                   : Float → Float
  primACos                   : Float → Float
  primATan                   : Float → Float
  primATan2                  : Float → Float → Float
  primShowFloat              : Float → String
```

The `primFloatEquality` primitive is intended to be used for decidable propositional equality. To enable proof carrying comparisons while preserving consistency, the following laws apply:

```
nan=nan : primFloatEquality NaN NaN ≡ true
nan=nan = refl

nan=-nan : primFloatEquality NaN (primFloatNegate NaN) ≡ true
nan=-nan = refl

neg0≠0 : primFloatEquality 0.0 -0.0 ≡ false
neg0≠0 = refl
```

Correspondingly, the `primFloatLess` can be used to provide a decidable total order, given by the following laws:

```
_[<]_ : Float → Float → Set
x [<] y = primFloatLess x y && not (primFloatLess y x) ≡ true

-inf<nan : -Inf [<]  NaN
nan<neg  :  NaN [<] -1.0
neg<neg0 : -1.0 [<] -0.0
neg0<0   : -0.0 [<]  0.0
0<pos    :  0.0 [<]  1.0
pos<Inf  :  1.0 [<]  Inf

-inf<nan = refl
nan<neg  = refl
neg<neg0 = refl
neg0<0   = refl
0<pos    = refl
pos<Inf  = refl
```

For numerical comparisons, use the `primFloatNumericalEquality` and `primFloatNumericalLess` primitives. These are implemented by the corresponding IEEE functions.

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

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

which normalises all `NaN` to a canonical `NaN` with an injectivity proof:

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

in the `Properties` module. These primitives can be used to define a decidable propositional equality with the `--safe` option.

3.2.8 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.

3.2.9 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`.

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.

3.2.10 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
  primStringToList   : String → List Char
  primStringFromList : List Char → String
```

(continues on next page)

(continued from previous page)

```
  primStringAppend   : String → String → String
  primStringEquality : String → String → Bool
  primShowString     : String → String
```

String literals can be *overloaded*.

Converting to a list is injective, and its proof:

```
primitive
  primStringToListInjective : ∀ a b → primStringToList a ≡ primStringToList b → a ≡
↪b
```

can found in the `Properties` module.

3.2.11 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`.

3.2.12 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 : Set
  lzero : Level
  lsuc  : Level → Level
  _⊔_   : Level → Level → Level
```

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

3.2.13 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
```

3.2.14 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.

3.2.15 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 #-}
```

3.2.16 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.

3.2.17 Reflection

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

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

3.2.18 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`.

3.2.19 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> #-}
```

3.2.20 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 is not necessary to give an explicit constructor to the record type `Stream A`.

We can as well define bisimilarity (equivalence) of a pair of `Stream A` as a coinductive record.

```
record _≈_ {A : Set} (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 `Stream` such that one returns a `Stream` with the elements in the even positions and the other the elements in odd positions.

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

odd : ∀ {A} → Stream A → Stream A
odd x = even (tl x)

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

And merge a pair of `Stream` by interleaving their elements.

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

Finally, we can prove that split is the left inverse of merge.

```
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)
```

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 CoNat : Set where
  zero : CoNat
  suc  : ∞ CoNat → CoNat
```

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
```

The current implementation of coinductive types comes with some limitations.

3.4 Copatterns

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
```

(continues on next page)

(continued from previous page)

```
    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 co-patterns

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

Note: This is a stub

```
data Term = Var Int Elims
          | Def QName Elims                -- ^ @f es@, possibly a delta/iota-redex
          | Con ConHead Args               -- ^ @c vs@
          | Lam ArgInfo (Abs Term)         -- ^ Terms are beta normal. Relevance is␣
↪ignored
          | Lit Literal
          | Pi (Dom Type) (Abs Type)       -- ^ dependent or non-dependent function␣
↪space
          | Sort Sort
          | Level Level
          | MetaV MetaId Elims
          | DontCare Term
            -- ^ Irrelevant stuff in relevant position, but created
            --   in an irrelevant context.
```

3.6 Cubical

The Cubical mode extends Agda with a variety of features from Cubical Type Theory. In particular, computational univalence and higher inductive types which hence gives 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.

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.

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 is a standard `agda/cubical` 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*. The main design choices of the core part of the library are explained in https://homotopytypetheory.org/2018/12/06/cubical-agda/ (lagda rendered version: https://ice1000.org/lagda/CubicalAgdaLiterate.html).

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.Everything
```

For detailed install instructions for `agda/cubical` see: https://github.com/agda/cubical/blob/master/INSTALL.md. 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.

There is also an older version of the library available at https://github.com/Saizan/cubical-demo/. However this is relying on deprecated features and is not recommended to use.

3.6.1 The interval and path types

The key idea of Cubical Type Theory is to add an interval type `I : Setω` (the reason this is in `Setω` is because it doesn't support the `transp` and `hcomp` operations). A variable `i : I` intuitively corresponds to a point 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 (∧), maximum (∨) and negation (~).

```
_∧_ : 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
i  ∨ i1    = i1
i  ∨ j     = j ∨ i
i0 ∧ i     = i0
i1 ∧ i     = i
```

(continues on next page)

(continued from previous page)

```
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, the following is not valid:

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

(continues on next page)

(continued from previous page)

```
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 lets us prove new things are not provable in standard Agda, for example function extensionality (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
```

3.6.2 Transport

While path types are great for reasoning about equality they don't 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. In order to remedy this we also have a built-in (generalized) transport operation and homogeneous composition operations. 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 `transp A r a` to type-check, which is that `A` has to be *constant* on `r`. This means that `A` should be a constant function whenever the constraint `r = i1` is satisfied. This side condition is vacuously true when `r` is `i0`, so there is nothing to check when writing `transp A i0 a`. However when `r` is equal to `i1` the `transp` function will compute as the identity function.

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

This requires `A` to be constant for it to be well-typed.

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.

3.6.3 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`, this 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 and the two elements will still be considered the same). The idea is that `Partial φ A` is the type of cubes in `A` that are only defined when `IsOne φ`. There is also a dependent version of this called `PartialP φ A` which allows `A` to be defined only when `IsOne φ`.

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

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

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 matching lambda*.

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

When the cases overlap they must agree (note that the order of the cases doesn't have to match the interval formula exactly):

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

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) → Setω
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:

```
inc : ∀ {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:

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

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

3.6.4 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 → λ { (i = i0) → x
                                       ; (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 (\ 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 → λ { (φ = i1) → u (i ∧ j) 1=1
                                      ; (i = i0) → ouc u0 })
                             (ouc 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 _ → λ { (i = i0) → x
                                                    ; (i = i1) → y })
                                           (inc (p i))
                                           (~ j)
```

3.6.5 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

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): https://github.com/agda/cubical/blob/master/Cubical/Foundations/Isomorphism.agda.

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 equivalence. 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.

For more results about Glue types and univalence see https://github.com/agda/cubical/blob/master/Cubical/Core/Glue.agda and https://github.com/agda/cubical/blob/master/Cubical/Foundations/Univalence.agda. For some examples of what can be done with this for working with binary and unary numbers see https://github.com/agda/cubical/blob/master/Cubical/Data/BinNat/BinNat.agda.

3.6.6 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
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)
```

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

isProp : ∀ {ℓ} → Set ℓ → Set ℓ
isProp A = (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
```

For many more examples of higher inductive types see: https://github.com/agda/cubical/tree/master/Cubical/HITs.

3.6.7 Cubical identity types and computational HoTT/UF

As mentioned above the computation rule for `J` does not hold definitionally for path types. Cubical Agda solves this by introducing a cubical identity type. The https://github.com/agda/cubical/blob/master/Cubical/Core/Id.agda file exports all of the primitives for this type, including the notation _≡_ and a `J` eliminator that computes definitionally on `refl`.

The cubical identity type and the path type are equivalent, so all of the results for one can be transported to the other one (using univalence). Using this we have implemented an interface to HoTT/UF in https://github.com/agda/cubical/blob/master/Cubical/Core/HoTT-UF.agda which provides the user with the key primitives of Homotopy Type Theory and Univalent Foundations implemented using cubical primitives under the hood. This hence gives an axiom free version of HoTT/UF which computes properly.

```
module Cubical.Core.HoTT-UF where

open import Cubical.Core.Id public
   using ( _≡_             -- The identity type.
         ; refl             -- Unfortunately, pattern matching on refl is not␣
↪available.
         ; J               -- Until it is, you have to use the induction principle J.

         ; transport       -- As in the HoTT Book.
         ; ap
         ; _•_
         ; _⁻¹
         ; _

         ; _≡⟨_⟩_          -- Standard equational reasoning.
         ; _∎

         ; funExt          -- Function extensionality
                           -- (can also be derived from univalence).

         ; Σ               -- Sum type. Needed to define contractible types,␣
↪equivalences
         ; _,_             -- and univalence.
         ; pr₁             -- The eta rule is available.
         ; pr₂

         ; isProp          -- The usual notions of proposition, contractible type, set.
         ; isContr
         ; isSet

         ; isEquiv         -- A map with contractible fibers
                           -- (Voevodsky's version of the notion).
         ; _≃_              -- The type of equivalences between two given types.
         ; EquivContr      -- A formulation of univalence.

         ; ∥_∥              -- Propositional truncation.
         ; ∣_∣             -- Map into the propositional truncation.
         ; ∥∥-isProp        -- A truncated type is a proposition.
         ; ∥∥-recursion     -- Non-dependent elimination.
         ; ∥∥-induction     -- Dependent elimination.
         )
```

In order to get access to only the HoTT/UF primitives start a file as follows:

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

open import Cubical.Core.HoTT-UF
```

However, even though this interface exists it is still recommended that one uses the cubical identity types unless one really need `J` to compute on `refl`. The reason for this is that the syntax for path types does not work for the identity types, making many proofs more involved as the only way to reason about them is using `J`. Furthermore, the path types satisfy many useful definitional equalities that the identity types don't.

3.6.8 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".

3.6.9 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.

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

```
{-# BUILTIN INTERVAL I     #-} -- I : Setω
{-# 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 → Setω

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
```

(continues on next page)

(continued from previous page)

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

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

{-# 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 inc #-}

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

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

```
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) \ 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 → (Partial ψ (fiber (w .fst) a)) → fiber (w .fst) a
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 → λ { (ϕ = i1) → p (u 1=1) i
                                                ; (ϕ = i0) → c }) c

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

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

-- pathToEquiv proves that transport is an equivalence (for details
-- see Agda.Builtin.Cubical.Glue). This is needed internally.
{-# BUILTIN PATHTOEQUIV pathToEquiv #-}
```

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)
```

The `Agda.Builtin.Cubical.Id` exports the cubical identity types:

```
postulate
  Id : ∀ {ℓ} {A : Set ℓ} → A → A → Set ℓ

{-# BUILTIN ID           Id       #-}
{-# BUILTIN CONID        conid    #-}

primitive
  primDepIMin : _
  primIdFace : ∀ {ℓ} {A : Set ℓ} {x y : A} → Id x y → I
  primIdPath : ∀ {ℓ} {A : Set ℓ} {x y : A} → Id x y → x ≡ y

primitive
  primIdJ : ∀ {ℓ ℓ'} {A : Set ℓ} {x : A} (P : ∀ y → Id x y → Set ℓ') →
              P x (conid i1 (λ i → x)) → ∀ {y} (p : Id x y) → P y p


primitive
  primIdElim : ∀ {a c} {A : Set a} {x : A}
                 (C : (y : A) → Id x y → Set c) →
                 ((φ : I) (y : A [ φ ↦ (λ _ → x) ])
                  (w : (x ≡ ouc y) [ φ ↦ (λ { (φ = i1) → \ _ → x}) ]) →
                  C (ouc y) (conid φ (ouc w))) →
                 {y : A} (p : Id x y) → C y p
```

3.7 Data Types

3.7.1 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
  c₁ : A₁
  ...
  cₙ : Aₙ
```

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

```
(y₁ : B₁) → ... → (yₘ : Bₘ) → D
```

3.7.2 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
```

3.7.3 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 (x₁ : P₁) ... (xₖ : Pₖ) : (y₁ : Q₁) → ... → (yₗ : Qₗ) → Set ℓ where
  c₁ : A₁
  ...
  cₙ : Aₙ
```

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

```
(z₁ : B₁) → ... → (zₘ : Bₘ) → D x₁ ... xₖ t₁ ... tₗ
```

3.7.4 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

```
(y₁ : B₁) → ... → (yₘ : Bₘ) → 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

  ```
  (z₁ : C₁) → ... → (zₖ : Cₖ) → 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*.

3.8 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.

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
```

3.8.1 Pattern Matching on `@♭`

Agda allows matching on `@♭` arguments by default. 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.

Pattern matching on `@♭` arguments can be disabled entirely by using the `--no-flat-split` flag

```
{-# OPTIONS --no-flat-split #-}
```

3.9 Foreign Function Interface

3.9.1 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.

3.9.2 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 `COMPILED` 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).

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 an Agda type 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 ⊤
```

(continues on next page)

(continued from previous page)

```
{-# 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 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
```

(continues on next page)

(continued from previous page)

```
{-# 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
```

3.9.3 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));
    }
  } #-}
{-# COMPILE JS []  = Array() #-}
{-# COMPILE JS _::_ = function (x) { return function(y) { return Array(x).concat(y); };␣
↪} #-}
```

3.10 Function Definitions

3.10.1 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))
```

3.10.2 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.
- The function should be terminating on all possible inputs, see *Termination Checking*.

3.10.3 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. 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 $\texttt{root} : (n : \texttt{Nat}) \to \texttt{Square n} \to \texttt{Nat}$ 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 $\texttt{sq} : (m : \texttt{Nat}) \to \texttt{Square}\ (m * m)$, `n` is forced to be equal to `m * m`.

In general, when matching on an argument of type $\texttt{D}\ i_1 \ldots i_n$ with a constructor $\texttt{c} : (x_1 : A_1) \to \ldots \to (x_m : A_m) \to \texttt{D}\ j_1 \ldots j_n$, Agda will attempt to unify $i_1 \ldots i_n$ with $j_1 \ldots j_n$. 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 is optional, but can help readability. 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 `root₂`, `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
```

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 $\texttt{D i}_1 \ldots \texttt{i}_n$ with an absurd pattern, Agda will attempt for each constructor $\texttt{c : (x}_1 \texttt{ : A}_1\texttt{)} \to \ldots \to \texttt{(x}_m \texttt{ : A}_m\texttt{)} \to \texttt{D j}_1 \ldots \texttt{j}_n$ of the datatype `D` to unify $\texttt{i}_1 \ldots \texttt{i}_n$ with $\texttt{j}_1 \ldots \texttt{j}_n$. 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.

3.10.4 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` *command-line and pragma option* causes Agda to raise an error 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 rejected 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` *command-line and pragma option* 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.

3.11 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)`:

```
(A : Set) (n : Nat)(u v : Vec A n) → Vec A n
```

Functions are constructed by lambda abstractions, which can be either typed or untyped. 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
```

You can also use the Unicode symbol λ (type "\lambda" in the Emacs Agda mode) instead of `\\`.

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]`.

3.11.1 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 $\forall$ (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)
```

3.12 Generalization of Declared Variables

3.12.1 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 : Set} (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.

3.12.2 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 : {n : Nat} {xs : Vec Nat (suc n)} → 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.

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 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.

3.12.3 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.
- Variables mentioned eariler 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 indicies 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
```

3.12.4 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.

3.12.5 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.

3.12.6 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.

3.13 Implicit Arguments

It is possible to omit terms that the type checker can figure out for itself, replacing them by _. 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
```

(continues on next page)

(continued from previous page)

```
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.

3.13.1 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
```

3.13.2 Metavariables

3.13.3 Unification

3.14 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.

3.14.1 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 *Instance fields*.

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
```

(continues on next page)

(continued from previous page)

```
      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

  eqNat : Eq Nat
  _==_ {{eqNat}} = natEquals

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
```

(continues on next page)

(continued from previous page)

```
  _ : test₁ ≡ 6
  _ = refl

open M

instance
  defaultNat : Default Nat
  defaultNat .default = 42

test₂ : Nat
test₂ = default

_ : test₂ ≡ 42
_ = refl
```

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.

Overlapping instances

By default, Agda does not allow overlapping instances. Two instances are defined to overlap if they could both solve the instance goal when given appropriate solutions for their recursive (instance) arguments.

For example, in code below, the instances *zero* and *suc* overlap for the goal ex_1, because either one of them can be used to solve the goal when given appropriate arguments, hence instance search fails.

```
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
```

Overlapping instances can be enabled via the `--overlapping-instances` flag. Be aware that enabling this flag might lead to an exponential slowdown in instance resolution and possibly (apparent) looping behaviour.

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 $\texttt{x}_1$ `,` $\texttt{y}_1$, the types of the second components `y` and $\texttt{y}_1$ can be completely different and not admit an equality test. Only when `x` and $\texttt{x}_1$ are *really equal* can we hope to compare `y` and $\texttt{y}_1$. Having the equality function return a proof means that we are guaranteed that when `x` and $\texttt{x}_1$ compare equal, they really are equal, and comparing `y` and $\texttt{y}_1$ 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.

3.14.2 Instance resolution

Given a goal that should be solved using instance resolution we proceed in the following four stages:

Verify the goal First we check that the goal type has the right shape to be solved by instance resolution. It should be of the form `{Γ} → C vs`, where the target type `C` is a variable from the context or the name of a data or record type, and `{Γ}` denotes a telescope of implicit or instance arguments. If this is not the case instance resolution fails with an error message[1].

Find candidates In the second stage we compute a set of *candidates*. *Let-bound* variables and top-level definitions in scope are candidates if they are defined in an `instance` block. Lambda-bound 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 `{{ }}`. 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.

Check the candidates We attempt to use each candidate in turn to build an instance of the goal type `{Γ} → C vs`. First we extend the current context by `{Γ}`. Then, given a candidate `c : {Δ} → A` we generate fresh metavariables `αs : {Δ}` for the arguments of `c`, with ordinary metavariables for implicit arguments, and instance metavariables, solved by a recursive call to instance resolution, for instance arguments.

Next we *unify* `A[Δ := αs]` with `C vs` and apply instance resolution to the instance metavariables in `αs`. Both unification and instance resolution have three possible outcomes: *yes*, *no*, or *maybe*. In case we get a *no* answer from any of them, the current candidate is discarded, otherwise we return the potential solution `λ {Γ} → c αs`.

Compute the result From the previous stage we get a list of potential solutions. If the list is empty we fail with an error saying that no instance for `C vs` could be found (*no*). If there is a single solution we use it to solve the goal (*yes*), and if there are multiple solutions we check if they are all equal. If they are, we solve the goal with one of them (*yes*), but if they are not, we postpone instance resolution (*maybe*), hoping that some of the *maybes* will turn into *nos* once we know more about the involved metavariables.

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-=`).

3.15 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.

3.15.1 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)
```

(continues on next page)

[1] Instance goal verification is buggy at the moment. See issue #1322.

(continued from previous page)

```
          → (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 l_1 and l_2 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 $l_1 \equiv l_2$ by `refl` when p_1 and p_2 are relevant. Instead, we need to reason about proofs of $0 \leq 1$.

```
postulate
  proof-equality : p₁ ≡ p₂
```

Now we can prove $l_1 \equiv l_2$ 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 $\texttt{head} \leq \texttt{bound}$ 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
```

3.15.2 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!

3.15.3 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.

3.15.4 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
    .proof : A

open Squash

.unsquash : ∀ {A} → Squash A → A
unsquash x = proof 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
```

3.15.5 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))`.

3.15.6 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
```

3.16 Lambda Abstraction

3.16.1 Pattern matching lambda

Anonymous pattern matching functions can be defined using the syntax:

```
\ { p11 .. p1n -> e1 ; ... ; pm1 .. pmn -> em }
```

(where, as usual, \ and -> can be replaced by λ and →). Internally this is translated into a function definition of the following form:

```
.extlam p11 .. p1n = e1
...
.extlam pm1 .. pmn = em
```

This means that anonymous pattern matching functions are generative. For instance, `refl` will not be accepted as an inhabitant of the type

```
(λ { true → true ; false → false }) ==
(λ { true → true ; false → false })
```

because this is equivalent to `extlam1 ≡ extlam2` for some distinct fresh names `extlam1` and `extlam2`. Currently the `where` and `with` constructions are not allowed in (the top-level clauses of) anonymous pattern matching functions.

Examples:

```
and : Bool → Bool → Bool
and = λ { true x → x ; false _ → false }

xor : Bool → Bool → Bool
xor = λ { true  true  → false
```

(continues on next page)

(continued from previous page)

```
        ; false false → false
        ; _     _     → true
        }

fst : {A : Set} {B : A → Set} → Σ A B → A
fst = λ { (a , b) → a }

snd : {A : Set} {B : A → Set} (p : Σ A B) → B (fst p)
snd = λ { (a , b) → b }
```

3.17 Local Definitions: let and where

There are two ways of declaring local definitions in Agda:

- let-expressions
- where-blocks

3.17.1 let-expressions

A let-expression defines an abbreviation. In other words, the expression that we define in a let-expression can neither be recursive, nor can let bound functions be defined by pattern matching.

Example:

```
f : Nat
f = let h : Nat → Nat
        h m = suc (suc m)
    in  h zero + h (suc zero)
```

let-expressions have 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.

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`.

3.17.2 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 p_{ij} are patterns of the corresponding types and 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.

3.17.3 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 []
```

3.17.4 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:

```
g : Nat → List Nat
g zero    = [ zero ]
g (suc n) = let sing = [ suc n ]
            in  sing ++ g n
```

Same definition using where:

```
g' : Nat → List Nat
g' zero = [ zero ]
g' (suc n) = sing ++ g' n
   where  sing = [ suc n ]
```

More than one definition in a let:

```
h : Nat → Nat
h n = let add2 : Nat
          add2 = suc (suc n)

          twice : Nat → Nat
          twice m = m * m

      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 (h m) + fibfact m
      in aux (pred n)
  where pred : Nat → Nat
        pred zero = zero
        pred (suc m) = m
```

3.18 Lexical Structure

Agda code is written in UTF-8 encoded plain text files with the extension `.agda`. Most unicode characters can be used in identifiers and whitespace is important, see *Names* and *Layout* below.

3.18.1 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` `codata` *coinductive* `constructor` `data` *do* `eta-equality` `field` *forall* `hiding` `import` `in` `inductive` `infix` `infixl` `infixr` `instance` `let` *macro* `module` `mutual` `no-eta-equality` `open` *overlap* `pattern` `postulate` `primitive` `private` `public` *quote* `quoteContext` `quoteGoal` *quoteTerm* `record` `renaming` `rewrite` `Set` `syntax` `tactic` *unquote* *unquoteDecl* *unquoteDef* `using` *variable* `where` `with`

The `Set` keyword can appear with a number suffix, optionally subscripted (see *Universe Levels*). For instance `Set42` and $\texttt{Set}_{42}$ are both keywords.

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 or hexadecimal (prefixed by `0x`) notation. 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`

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.

3.18.2 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 do field instance let macro mutual postulate primitive private 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.

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
```

3.18.3 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`. The main use case for literate Agda is to generate LaTeX documents from 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}
```

3.19 Literal Overloading

3.19.1 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`. 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)

{-# 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
```

(continues on next page)

(continued from previous page)

```
  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}
```

3.19.2 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 #-}
```

3.19.3 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 #-}
```

3.19.4 Restrictions

Currently only integer and string literals can be overloaded.

Overloading does not work in patterns yet.

3.20 Mixfix Operators

A name containing one or more name parts and one or more _ can be used as an operator where the arguments go in place of the _. For instance, an application of the name `if_then_else_` to arguments `x`, `y`, and `z` can be written either as a normal application `if_then_else_ x y z` or as an operator application `if x then y else z`.

Examples:

```
_and_ : Bool → Bool → Bool
true and x = x
false and _ = false

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

_⇒_   : Bool → Bool → Bool
true  ⇒ b = b
false ⇒ _ = true
```

3.20.1 Precedence

Consider the expression `true and false ⇒ false`. Depending on which of `_and_` and `_⇒_` has more precedence, it can either be read as `(false and true) ⇒ false = true`, or as `false and (true ⇒ false) = true`.

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.

If we give `_and_` more precedence than `_⇒_`, then we will get the first result:

```
infix 30 _and_
-- infix 20 _⇒_ (default)

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
```

3.20.2 Associativity

Consider the expression `true ⇒ false ⇒ false`. Depending on whether _⇒_ is 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
p-left = refl

e-left : false ⇒' true ⇒' false  ≡  false
e-left = refl
```

3.20.3 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).

3.21 Module System

3.21.1 Module application

3.21.2 Anonymous modules

3.21.3 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

  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
```

3.21.4 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.

3.21.5 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` statement with one or more of the modifiers `using`, `hiding`, or `renaming`. You can combine both `using` and `hiding` with `renaming`, but not with each other. 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. The other forms of opening are defined in terms of this one. An omitted `renaming` modifier is equivalent to an empty renaming.

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`.

3.21.6 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`.

3.21.7 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.

It is sometimes useful to be able to work temporarily 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`. In Coq this can be done in two ways: using a functor, which is essentially a function between modules, or using a section. A section allows you to abstract some arguments from several definitions at once. We introduce parameterised modules analogous to sections in Coq. When declaring a module you can give a telescope of module parameters which are abstracted from all the definitions in the module. For instance, 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. It also happens to be the reasonable thing to do.

Something which you cannot do in Coq is to apply a section to its arguments. We allow this through the module application statement. In 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] mods
```

for

```
module M1 Δ = M2 terms mods
open M1 [public]
```

3.21.8 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.

3.21.9 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*.

3.22 Mutual Recursion

Mutual recursive functions can be written by placing the type signatures of all mutually recursive function before their definitions:

```
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
  []   : 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)
```

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
```

3.22.1 Old Syntax: Keyword `mutual`

Note: You are advised to avoid using this old syntax if possible, but the old syntax is still supported.

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.

3.23 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.

3.23.1 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
  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
```

3.23.2 Refolding

For each pattern `pattern lhs = rhs`, Agda declares a `DISPLAY` pragma refolding `rhs` to `lhs` (see *The DISPLAY pragma* for more details).

3.24 Positivity Checking

Note: This is a stub.

3.24.1 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
  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
    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
    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
  field ap : U → U

{-# NO_POSITIVITY_CHECK #-}
data D where
  lam : (D → D) → D
```

3.24.2 *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.

3.25 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>
```

Example:

```
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
```

A preferable way to work 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

  -- ...
```

3.25.1 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.

3.26 Pragmas

Pragmas are comments that are not ignored by Agda but have some special meaning. The general format is:

```
{-# <PRAGMA_NAME> <arguments> #-}
```

3.26.1 Index of pragmas

- *BUILTIN*
- *CATCHALL*
- *COMPILE*
- *FOREIGN*
- *NO_POSITIVITY_CHECK*
- *NO_TERMINATION_CHECK*
- *NON_TERMINATING*
- NON_COVERING
- *POLARITY*

- *STATIC*
- *TERMINATING*
- *INLINE*
- *NOINLINE*
- *WARNING_ON_USAGE*
- *WARNING_ON_IMPORT*

See also *Command-line and pragma options*.

The `DISPLAY` pragma

Users can declare a `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 are restricted to variables, constructors, defined functions or types, and literals. In particular, lambdas are not allowed in left-hand sides.
- Since *DISPLAY* pragmas 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.

The `INLINE` and `NOINLINE` pragmas

A definition marked with an `INLINE` pragma is inlined during compilation. If it is a simple definition that does no pattern matching, it is also inlined in function bodies at type-checking time.

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
```

(continues on next page)

(continued from previous page)

```
-- 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
```

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.

Similarly they can use a `WARNING_ON_IMPORT` pragma to attach to a module a warning to be raised whenever this module is imported.

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`" #-}
```

3.27 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.

3.27.1 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
```

3.27.2 The predicative hierarchy of `Prop`

Just as for `Set`, Agda has a predicative hierarchy of sorts $\texttt{Prop}_0$ (= `Prop`), $\texttt{Prop}_1$, $\texttt{Prop}_2$, ... where $\texttt{Prop}_0$: $\texttt{Set}_1$, $\texttt{Prop}_1$: $\texttt{Set}_2$, $\texttt{Prop}_2$: $\texttt{Set}_3$, 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
```

3.27.3 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.

3.27.4 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 ()
```

3.28 Record Types

Records are types for grouping values together. They generalise the dependent product type by providing named fields and (optional) further components.

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 `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
```

Elements of record types can be defined using a record expression

```
p23 : Pair Nat Nat
p23 = record { fst = 2; snd = 3 }
```

or using *copatterns*

```
p34 : Pair Nat Nat
Pair.fst p34 = 3
Pair.snd p34 = 4
```

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
```

In this sense, record types behave much like single constructor datatypes (but see *Eta-expansion* below).

3.28.1 Declaring, constructing and decomposing records

Declarating 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>
    -- ...
  <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` (see *Eta-expansion*), `inductive` and `co-inductive` (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) }
```

Decomposing record values

With the field name, we can project the corresponding component out of a record value. It is also possible to pattern match against inductive records:

```
sum : Pair Nat Nat → Nat
sum (x , y) = x + y
```

Or, using a *let binding record pattern*:

```
sum' : Pair Nat Nat → Nat
sum' p = let (x , y) = p in x + y
```

Note: Naming the constructor is not required to enable pattern matching against record values. Record expressions can appear as patterns.

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 }
```

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

```
new' : MyRecord
new'  = 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 *instance field*, then it is omitted from `new-fields`.

The reason for treating implicit and instance 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 R : Set where
  field
    {length} : Nat
    vec      : Vec Nat length
    -- More fields ...

xs : R
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 = _`).

3.28.2 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
```

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
```

3.28.3 Eta-expansion

The eta 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 }`. By default, all (inductive) record types enjoy eta-equality if the positivity checker has confirmed it is safe to have it. The keywords `eta-equality`/`no-eta-equality` enable/disable eta rules for the record type being declared.

3.28.4 Recursive records

Recursive records need to be declared as either inductive or coinductive.

```
record Tree (A : Set) : Set where
  inductive
  constructor tree
  field
    elem     : A
    subtrees : List (Tree A)

record Stream (A : Set) : Set where
  coinductive
  constructor _::_
  field
    head : A
    tail : Stream A
```

Inductive records have `eta-equality` on by default, while `no-eta-equality` is the default for coinductive records. In fact, the `eta-equality` and `coinductive` directives are not allowed together, since this can easily make Agda loop. This can be overridden at your own risk by using the pragma `ETA` instead.

It is possible to pattern match on inductive records, but not on coinductive ones.

3.28.5 Instance fields

Instance fields, that is record fields marked with `{{ }}` can be used to model "superclass" dependencies. For example:

```
record Eq (A : Set) : Set where
  field
    _==_ : A → A → Bool

open Eq {{...}}
```

```
record Ord (A : Set) : Set where
  field
    _<_ : A → A → Bool
    {{eqA}} : Eq A
```

(continues on next page)

(continued from previous page)

```
open Ord {{...}} hiding (eqA)
```

Now anytime you have a function taking an `Ord A` argument the `Eq A` instance is also available by virtue of η-expansion. So this works as you would expect:

```
_≤_ : {A : Set} {{OrdA : Ord A}} → A → A → Bool
x ≤ y = (x == y) || (x < y)
```

There is a problem however if you have multiple record arguments with conflicting instance fields. For instance, suppose we also have a `Num` record with an `Eq` field

```
record Num (A : Set) : Set where
  field
    fromNat : Nat → A
    {{eqA}} : Eq A

open Num {{...}} hiding (eqA)

_≤3 : {A : Set} {{OrdA : Ord A}} {{NumA : Num A}} → A → Bool
x ≤3 = (x == fromNat 3) || (x < fromNat 3)
```

Here the `Eq A` argument to `_==_` is not resolved since there are two conflicting candidates: `Ord.eqA OrdA` and `Num.eqA NumA`. To solve this problem you can declare instance fields as *overlappable* using the `overlap` keyword:

```
record Ord (A : Set) : Set where
  field
    _<_ : A → A → Bool
    overlap {{eqA}} : Eq A

open Ord {{...}} hiding (eqA)

record Num (A : Set) : Set where
  field
    fromNat : Nat → A
    overlap {{eqA}} : Eq A

open Num {{...}} hiding (eqA)

_≤3 : {A : Set} {{OrdA : Ord A}} {{NumA : Num A}} → A → Bool
x ≤3 = (x == fromNat 3) || (x < fromNat 3)
```

Whenever there are multiple valid candidates for an instance goal, if **all** candidates are overlappable, the goal is solved by the left-most candidate. In the example above that means that the `Eq A` goal is solved by the instance from the `Ord` argument.

Clauses for instance fields can be omitted when defining values of record types. For instance we can define `Nat` instances for `Eq`, `Ord` and `Num` as follows, leaving out cases for the `eqA` fields:

```
instance
  EqNat : Eq Nat
  _==_ {{EqNat}} = Agda.Builtin.Nat._==_

  OrdNat : Ord Nat
  _<_ {{OrdNat}} = Agda.Builtin.Nat._<_
```

(continues on next page)

(continued from previous page)

```
  NumNat : Num Nat
  fromNat {{NumNat}} n = n
```

3.29 Reflection

3.29.1 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 #-}
```

Visibility and relevance characterise the behaviour of an argument:

```
data ArgInfo : Set where
  arg-info : (v : Visibility) (r : Relevance) → 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      #-}
```

Patterns

Reflected patterns are bound to the `AGDAPATTERN` built-in using the following data type.

```
data Pattern : Set where
  con    : (c : Name) (ps : List (Arg Pattern)) → Pattern
  dot    : Pattern
  var    : (s : String)  → Pattern
  lit    : (l : Literal) → Pattern
  proj   : (f : Name)    → Pattern
  absurd : Pattern

{-# BUILTIN AGDAPATTERN   Pattern #-}
{-# BUILTIN AGDAPATCON    con     #-}
{-# BUILTIN AGDAPATDOT    dot     #-}
{-# BUILTIN AGDAPATVAR    var     #-}
{-# BUILTIN AGDAPATLIT    lit     #-}
{-# BUILTIN AGDAPATPROJ   proj    #-}
{-# BUILTIN AGDAPATABSURD absurd  #-}
```

Name abstraction

```
data Abs (A : Set) : Set where
  abs : (s : String) (x : A) → Abs A

{-# BUILTIN ABS    Abs #-}
{-# BUILTIN ABSABS abs #-}
```

Terms

Terms, sorts and clauses are mutually recursive and mapped to the `AGDATERM`, `AGDASORT` and `AGDACLAUSE` built-ins respectively. Types are simply terms. Terms use de Bruijn indices to represent variables.

```
data Term : Set
data Sort : Set
data Clause : Set
Type = Term

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.
  unknown : Sort

data Clause where
  clause        : (ps : List (Arg Pattern)) (t : Term) → Clause
  absurd-clause : (ps : List (Arg Pattern)) → Clause

{-# BUILTIN AGDASORT    Sort   #-}
{-# BUILTIN AGDATERM    Term   #-}
{-# 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 AGDASORTUNSUPPORTED unknown #-}

{-# 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) → Definition                     -- d: name of data type
  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 `ErrorPart`s. 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
  nameErr : Name → ErrorPart

{-# BUILTIN AGDAERRORPART       ErrorPart #-}
{-# BUILTIN AGDAERRORPARTSTRING strErr    #-}
{-# BUILTIN AGDAERRORPARTTERM   termErr   #-}
{-# BUILTIN AGDAERRORPARTNAME   nameErr   #-}
```

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 ⊤
```

(continues on next page)

(continued from previous page)

```
  -- 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 metavariable. This will abort
  -- the computation and restart it (from the beginning) when the
  -- metavariable is solved.
  blockOnMeta : ∀ {a} {A : Set a} → Meta → 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

  -- 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 (List (Arg Type))

  -- Extend the current context with a variable of the given type.
  extendContext : ∀ {a} {A : Set a} → Arg Type → TC A → TC A

  -- Set the current context. Takes a context telescope with the outer-most
  -- entry first, in contrast to 'getContext'. Each type should be valid in the
  -- context formed by the preceding elements in the list.
  inContext : ∀ {a} {A : Set a} → List (Arg Type) → 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

  -- 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 ⊤
```

(continues on next page)

(continued from previous page)

```
  -- 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 ⊤

  -- Define a declared function. The function may have been declared using
  -- 'declareDef' or with an explicit type signature in the program.
  defineFun : Name → List Clause → TC ⊤

  -- Get the type of a defined name. Replaces 'primNameType'.
  getType : Name → TC Type

  -- Get the definition of a defined name. Replaces 'primNameDefinition'.
  getDefinition : Name → TC Definition

  -- Check if a name refers to a macro
  isMacro : Name → TC Bool

  -- 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

  -- 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 ⊤

  -- Fail if the given computation gives rise to new, unsolved
  -- "blocking" constraints.
  noConstraints : ∀ {a} {A : Set a} → TC A → TC A

  -- Tries to solve all constraints.
  solveConstraints : TC ⊤

  -- Wakes up all constraints mentioning the given meta-variables,
  -- and then tries to solve all awake constraints.
  solveConstraintsMentioning : List Meta → TC ⊤

  -- 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

{-# BUILTIN AGDATCMUNIFY                    unify                    #-}
{-# BUILTIN AGDATCMTYPEERROR                typeError                #-}
{-# BUILTIN AGDATCMBLOCKONMETA              blockOnMeta              #-}
{-# BUILTIN AGDATCMCATCHERROR               catchTC                  #-}
{-# BUILTIN AGDATCMINFERTYPE                inferType                #-}
{-# BUILTIN AGDATCMCHECKTYPE                checkType                #-}
{-# BUILTIN AGDATCMNORMALISE                normalise                #-}
{-# BUILTIN AGDATCMREDUCE                   reduce                   #-}
{-# BUILTIN AGDATCMGETCONTEXT               getContext               #-}
```

(continues on next page)

(continued from previous page)

```
{-# BUILTIN AGDATCMEXTENDCONTEXT              extendContext              #-}
{-# BUILTIN AGDATCMINCONTEXT                  inContext                  #-}
{-# BUILTIN AGDATCMQUOTETERM                  quoteTC                    #-}
{-# BUILTIN AGDATCMUNQUOTETERM                unquoteTC                  #-}
{-# BUILTIN AGDATCMFRESHNAME                  freshName                  #-}
{-# BUILTIN AGDATCMDECLAREDEF                 declareDef                 #-}
{-# BUILTIN AGDATCMDECLAREPOSTULATE           declarePostulate           #-}
{-# BUILTIN AGDATCMDEFINEFUN                  defineFun                  #-}
{-# BUILTIN AGDATCMGETTYPE                    getType                    #-}
{-# BUILTIN AGDATCMGETDEFINITION              getDefinition              #-}
{-# BUILTIN AGDATCMCOMMIT                     commitTC                   #-}
{-# BUILTIN AGDATCMISMACRO                    isMacro                    #-}
{-# BUILTIN AGDATCMWITHNORMALISATION          withNormalisation          #-}
{-# BUILTIN AGDATCMDEBUGPRINT                 debugPrint                 #-}
{-# BUILTIN AGDATCMNOCONSTRAINTS              noConstraints              #-}
{-# BUILTIN AGDATCMSOLVECONSTRAINTS           solveConstraints           #-}
{-# BUILTIN AGDATCMSOLVECONSTRAINTSMENTIONING solveConstraintsMentioning #-}
{-# BUILTIN AGDATCMRUNSPECULATIVE             runSpeculative             #-}
```

3.29.2 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 $t_1 \to t_2 \to .. \to$ `Term` $\to$ `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 `f : Term → Name → Bool → Term → 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 `unquote m : A` for some type `A` the type checker proceeds as follows:

- Check `m : Term → TC` $\top$.
- Create a fresh metavariable `hole : A`.
- Let `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 `g : Term → 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.

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
```

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` and `defineFun` 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:

```
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:

```
-- Defining: id-name {A} x = x
defId : (id-name : Name) → TC ⊤
defId id-name = do
  defineFun id-name
    [ clause
      ( arg (arg-info hidden relevant) (var "A")
      ∷ arg (arg-info visible relevant) (var "x")
      ∷ [] )
      (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 (arg-info visible relevant) id-name) ty
  defId id-name

unquoteDecl id´ = mkId id´
```

3.30 Rewriting

This is the stub for `--rewrite` and the associated *REWRITE* builtin. You might also be looking for the documentation on the *rewrite constuct*.

Note: This is a stub.

3.31 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`; forces Agda to accept unfinished proofs.
- `--no-positivity-check`; makes it possible to write non-terminating programs by structural "induction" on non strictly positive datatypes.
- `--no-termination-check`; gives loopy programs any type.
- `--type-in-type` and `--omega-in-omega`; allow the user to encode the Girard-Hurken paradox.
- `--injective-type-constructors`; together with excluded middle leads to an inconsistency via Chung-Kil Hur's construction.

- `--guardedness` together with `--sized-types`; currently can be used to define a type which is both inductive and coinductive, which leads to an inconsistency. This might be fixed in the future.
- `--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` together with `--with-K`; the univalence axiom is provable using cubical constructions, which falsifies the K axiom.
- The `primEraseEquality` primitive together with `--without-K`; using `primEraseEquality`, one can derive the K axiom.

The option `--safe` is coinfective (see *Consistency checking of options used*); if a module is declared safe, then all its imported modules must also be declared safe.

Note: The `--guardedness` and `--sized-types` options are both on by default. However, unless they have been set explicitly by the user, setting the `--safe` option will turn them both off. That is to say that

```
{-# OPTIONS --safe #-}
```

will correspond to `--safe`, `--no-guardedness`, and `--no-sized-types`. When both

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

and

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

will turn on `--safe`, `--guardedness`, and `--no-sized-types`.

Setting both `--sized-types` and `--guardedness` whilst demanding that the module is `--safe` will lead to an error as combining these options currently is inconsistent.

3.32 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*.

3.32.1 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 ≡ 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 ∈ δ a x` iff `xw ∈ 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 ∈ δ (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:
   - $\alpha = \epsilon$, and $\beta =$ `xw`, where `w ∈ δ b x`.
   - $\alpha =$ `x`α', with $\alpha' \in$ `δ a x`, and `w` $= \alpha'\beta \in$ `δ a x · b`.
2. $\epsilon \notin$ `a`. Then, only the second case above is possible:
   - $\alpha =$ `x`α', with $\alpha' \in$ `δ a x`, and `w` $= \alpha'\beta \in$ `δ 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
```

3.32.2 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).

3.33 Syntactic Sugar

3.33.1 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 _>>_, whatever those happen to be bound in the context of the do-block. 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
```

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
```

(continues on next page)

(continued from previous page)

```
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)
  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 mechaincally 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: nat ≠ _ => _"
(_ => _) =?= nat      = typeError "type mismatch: _ => _ ≠ nat"
(A => B) =?= (A₁ => B₁) = do
```

(continues on next page)

(continued from previous page)

```
  refl ← A =?= A₁
  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.

3.33.2 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
```

As do-notation, idiom brackets desugar before scope checking, so whatever the names `pure` and `_<*>_` are bound to gets used when desugaring the idiom brackets.

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
  ```

3.34 Syntax Declarations

Note: This is a stub

It is now possible to declare user-defined syntax that binds identifiers. Example:

```
postulate
  State  : Set → Set → Set
  put    : ∀ {S} → S → State S ⊤
  get    : ∀ {S} → State S S
  return : ∀ {A S} → A → State S A
  bind   : ∀ {A B S} → State S B → (B → State S A) → State S A

syntax bind e₁ (λ x → e₂) = x ← e₁ , e₂

increment : State ℕ ⊤
increment = x ← get ,
            put (suc x)
```

The syntax declaration for `bind` implies that `x` is in scope in e_2, but not in e_1.

You can give fixity declarations along with syntax declarations:

```
infixr 40 bind
syntax bind e₁ (λ x → e₂) = x ← e₁ , e₂
```

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
```

3.35 Telescopes

Note: This is a stub.

3.35.1 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
```

3.36 Termination Checking

Not all recursive functions are permitted - Agda accepts only these recursive schemas that it can mechanically prove terminating.

3.36.1 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.

3.36.2 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.

3.36.3 With-functions

3.36.4 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. They do reduce at run-time and when invoking `C-c C-n` at top-level (but not in a hole). 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
  ```

(continues on next page)

(continued from previous page)

```
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
{-# TERMINATING #-}
h = g
```

3.36.5 References

Andreas Abel, Foetus – termination checker for simple functional programs

3.37 Universe Levels

3.37.1 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 ⊆ U` of all sets that do not contain themselves. Then we would have `A ∈ A` if and only if `A ∉ A`, a contradiction.

For similar reasons, 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 : Set₁`, 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 **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 $\texttt{Set}_n$.

A note on syntax: 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 `Set₁ != 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 takes 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
```

3.37.2 Universe polymorphism

Although we were able to give a type to the `Prod` operator by defining a special notion of large list, this quickly gets tiresome. Sooner or later, we find that we require yet another list type $\texttt{List}_2$, and it doesn't stop there. 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
```

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
```

`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
```

Expressions of kind `Set`ω

In a sense, universes were introduced to ensure that every Agda expression has a type, including expressions such as `Set`, `Set₁`, 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. Indeed, the expression denotes a function mapping levels to universes, so if it had a type, it should be something like `Level → Universe`, where `Universe` is the collection of all universes. But since the elements of `Universe` are types, `Universe` is itself a universe, so we have `Universe : Universe`. This leads to circularity and inconsistency. In other words, just as we cannot have a set of all sets, we also can't have a universe of all universes.

As a consequence, although the expression `(n : Level) → Set n` **is** a type, it does not **have** a type. It does, however, have a "kind", which Agda calls `Setω`. The expression `Setω` itself is a valid Agda type but cannot appear as part of an Agda term. For example, the following definition is valid:

```
largeType : Setω
largeType = (n : Level) → Set n
```

As a counterexample which attempts to use `Setω` as part of a term, consider trying to form the singleton list `[ Unit ]`:

```
badList : List ((n : Level) → Set n)
badList = Unit :: []
```

This generates an error message stating that `Setω` is not of the form `Set _`. The problem is that `List` can only be applied to types that are part of `Set n` for some `n : Level`, but `(n : Level) → Set n` belongs to `Setω` which is not of this form. The only type constructor that can be applied to expressions of kind `Setω` is →.

3.38 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.

3.38.1 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
...                   | 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 : P []
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 xs | p x) !}
```

In the cons case we have to prove that `P` holds for `filter p 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 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 (filter p xs) !}
...                 | false = {! H : P (x ∷ 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
```

(continues on next page)

(continued from previous page)

```
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 *The inspect idiom* below for an alternative approach.

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
```

(continues on next page)

(continued from previous page)

```
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 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
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.

Rewrite

Remember example of *simultaneous abstraction* from above.

```
postulate plus-commute : (a b : Nat) → a + b ≡ b + a
```

(continues on next page)

(continued from previous page)

```
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.

The inspect idiom

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 use the *inspect idiom*, which retains a proof that the original term is equal to its abstraction.

In the simplest form, the inspect idiom uses a singleton type:

```
data Singleton {a} {A : Set a} (x : A) : Set a where
  _with≡_ : (y : A) → x ≡ y → Singleton x

inspect : ∀ {a} {A : Set a} (x : A) → Singleton x
inspect x = x with≡ refl
```

Now instead of with-abstracting `t`, you can abstract over `inspect t`. For instance,

```
filter : {A : Set} → (A → Bool) → List A → List A
filter p [] = []
filter p (x ∷ xs) with inspect (p x)
...                  | true  with≡ eq = {! eq : p x ≡ true !}
...                  | false with≡ eq = {! eq : p x ≡ false !}
```

Here we get proofs that `p x ≡ true` and `p x ≡ false` in the respective branches that we can use on the right. Note that since the with-abstraction is over `inspect (p x)` rather than `p x`, the goal and argument types are no longer generalised over `p x`. To fix that we can replace the singleton type by a function graph type as follows (see *Anonymous modules* to learn about the use of a module to bind the type arguments to `Graph` and `inspect`):

```
module _ {a b} {A : Set a} {B : A → Set b} where

  data Graph (f : ∀ x → B x) (x : A) (y : B x) : Set b where
    ingraph : f x ≡ y → Graph f x y

  inspect : (f : ∀ x → B x) (x : A) → Graph f x (f x)
  inspect _ _ = ingraph refl
```

To use this on a term `g v` you with-abstract over both `g v` and `inspect g v`. For instance, applying this to the example from above we get

```
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    | inspect f n
proof n ()    | zero   | _
proof n q     | suc fn | ingraph eq = {! q : P (suc fn), eq : f n ≡ suc fn !}
```

We could then use the proof that `f n ≡ suc fn` to apply `lemma` to `q`.

This version of the inspect idiom is defined (using slightly different names) in the standard library in the module `Relation.Binary.PropositionalEquality` and in the agda-prelude in `Prelude.Equality.Inspect` (reexported by `Prelude`).

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 matching 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 matching 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_return_of_ : ∀ {a b} {A : Set a} (x : A) (B : A → Set b) → (∀ x → B x) → B x
case x return 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[1] 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.

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.

[1] The termination checker has *special treatment for with-functions*, so replacing a *with* by the equivalent helper function might fail termination.

3.38.2 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)
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 = {!!}
```

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
```

(continues on next page)

(continued from previous page)

```
  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.

3.39 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.

3.39.1 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)].

3.39.2 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.

3.39.3 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 4

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. For instance, giving the type signature of the map function over vectors, you will get the desired function as the first solution.

The tool is based on a term search implementation independent of Agda called Agsy. Agsy is a general purpose search algorithm for a dependently typed language similar to Agda. 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.

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`. Agsy'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
`-c`	Allow Agsy to use case-split
`-d`	Attempt to disprove the goal
ID	Use definition *ID* as a hint
`-m`	Use the definitions in the current module as hints
`-r`	Use the unqualified definitions in scope as hints
`-l`	List up to ten solutions, does not commit to any
`-s` *N*	Commit to the *N* th solution

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.

Case split

Auto normally only tries to find a term that could replace the current hole. However, if the hole constitutes the entire RHS of the clause (same as for the make-case command), you can instruct Auto to try case splitting by writing (since version 2.2.8) `-c`.

Note that if a solution is found the whole file will be reloaded (as for make-case) resulting in a delay. Case splitting cannot yet be combined with `-l` or `-s <n>`.

Equality reasoning

If the constants `_≡_ begin_ _≡⟨_⟩_ _∎ sym cong` from the standard library are in scope, then Auto will do equality reasoning using these constructs. However, it will not do anything more clever than things like not nesting several `sym` or `cong`. Hence long chains of equalities should not be expected and required arithmetical rules have to be given as hints.

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> ...`.

The other way is to write `-m`. This includes all constants in scope which are defined or postulated in the innermost module surrounding the current hole. It is also possible to combine `-m` with a list of named constants (not included by `-m`).

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.

Disproving

If you are uncertain about the validity of what you are trying to prove, you can use Auto to try to find a counterproof. The flag `-d` makes Auto negate the current goal and search for a term disproving it. If such a term is found, it will be displayed in the info buffer. The flag `-d` can be combined with `-l` and `-l -s <n>`.

Auto refine / suggest

There is a special mode for searching (part of) the scope of constants for possible refinement candidates. The flag `-r` chooses this mode. By default all constants which are in scope unqualified are included.

The constants that matches the current goal are sorted in order of how many constructs their result type contains. This means that the constants which in some sense match the goal most specifically will appear first and the most general ones last. By default, Auto will present a list of candidates, rather than refining using the topmost constant. To select one of them for refinement, combine `-r` with `-s <n>`. In order to list constants other than the ten first ones, write `-r -l -s <n>`.

The auto refine feature has little to do with the rest of the Auto tool. If it turns out to be useful it could be improved and made into a separate Emacs mode command.

Dependencies between meta variables

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

- Irrelevance is not yet respected. Agsy will happily use a parameter marked as irrelevant and does not disregard irrelevant arguments when comparing terms.
- Records that lack a constructor name are still deconstructed when case splitting, but the name of the record type is used instead of a constructor name in the resulting pattern.
- Literals representing natural numbers are supported (but any generated natural number will be given in constructor form). Apart from this, literals are not supported.
- Primitive functions are not supported.
- Copatterns are not supported.
- Termination checking for recursive calls is done locally, so a non-terminating set of clauses might be sent back to Agda.
- Agsy currently does not automatically pick a datatype when instantiating types. A frequently occurring situation is when you try to disprove a generic property. Then Agsy must come up with a particular type as part of the disproof. You can either fix your generic type to e.g. `Nat` or `Fin n` (for an arbitrary `n` if you wish), or you give `Nat` or `Fin` as a hint to the search.
- Case split mode currently does not do case on expressions (`with`).

- Case split mode sometimes gives a unnecessarily complex RHS for some clause when the solution consists of several clauses.
- The special constraints that apply to `codata` are not respected by Agsy. Agsy treats `codata` just like `data`.
- Agsy has universe subtyping, so sometimes suggests solutions not accepted by Agda.
- Universe polymorphism is only partially supported. Agsy may fail when trying to construct universe polymorphic definitions, but will probably succeed (with respect to this) when constructing terms which refer to, or whose type is defined in terms of, universe polymorphic definitions.
- In case split and disproving modes, the current goal may not depend on any other meta variables. For disproving mode this means that there may be implicitly universally quantified but not existentially quantified stuff.
- Searching for simultaneous solutions of several holes does not combine well with parameterised modules and recursive calls.

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.

General options

`--version -V` Show version number

`--help[=`*`TOPIC`*`] -?[`*`TOPIC`*`]` Show basically this help, or more help about *`TOPIC`*. Current topics available: `warning`.

`--interactive -I` Start in interactive mode (no longer supported)

`--interaction` For use with the Emacs mode (no need to invoke yourself)

`--interaction-json` For use with other editors such as Atom (no need to invoke yourself)

`--only-scope-checking` Only scope-check the top-level module, do not type-check it

Compilation

See *Compilers* for backend-specific options.

`--no-main` Do not treat the requested module as the main module of a program when compiling

`--compile-dir=`*`DIR`* Set *`DIR`* as directory for compiler output (default: the project root)

`--no-forcing` Disable the forcing optimisation

`--with-compiler=`*`PATH`* Set `PATH` as the executable to call to compile the backend's output (default: ghc for the GHC backend).

Generating highlighted source code

--vim Generate Vim highlighting files

--latex Generate LaTeX with highlighted source code (see *Generating LaTeX*)

--latex-dir=*DIR* Set directory in which LaTeX files are placed to *DIR* (default: latex)

--count-clusters Count extended grapheme clusters when generating LaTeX code (see *Counting Extended Grapheme Clusters*)

--html Generate HTML files with highlighted source code (see *Generating HTML*)

--html-dir=*DIR* Set directory in which HTML files are placed to *DIR* (default: html)

--css=*URL* Set URL of the CSS file used by the HTML files to *URL* (can be relative)

--html-highlight=[code,all,auto] Whether to highlight non-Agda code as comments in generated HTML files (default: all; see :ref: *generating-html*)

--dependency-graph=*FILE* Generate a Dot file *FILE* with a module dependency graph

Imports and libraries

(see *Library Management*)

--ignore-interfaces Ignore interface files (re-type check everything, except for builtin and primitive modules)

--ignore-all-interfaces Ignore *all* interface files, including builtin and primitive modules; only use this if you know what you are doing!

--local-interfaces Read and write interface files next to the Agda files they correspond to (i.e. do not attempt to regroup them in a `_build/` directory at the project's root).

--include-path=*DIR* -i=*DIR* Look for imports in *DIR*

--library=*DIR* -l=*LIB* Use library *LIB*

--library-file=*FILE* Use *FILE* instead of the standard libraries file

--no-libraries Don't use any library files

--no-default-libraries Don't use default library files

4.2.2 Command-line and pragma options

The following options can also be given in .agda files in the `{-# OPTIONS --{opt1} --{opt2} ... #-}` form at the top of the file.

Caching

--caching Enable caching of typechecking (default)

--no-caching Disable caching of typechecking

Printing and debugging

--show-implicit Show implicit arguments when printing

--show-irrelevant Show irrelevant arguments when printing

--no-unicode Don't use unicode characters to print terms

--verbose=*N* -v=*N* Set verbosity level to *N*

Copatterns and projections

--copatterns Enable definitions by copattern matching (default; see *Copatterns*)

--no-copatterns Disable definitions by copattern matching

--postfix-projections Make postfix projection notation the default

Experimental features

--injective-type-constructors Enable injective type constructors (makes Agda anti-classical and possibly inconsistent)

--experimental-irrelevance Enable potentially unsound irrelevance features (irrelevant levels, irrelevant data matching) (see *Irrelevance*)

--rewriting Enable declaration and use of REWRITE rules (see *Rewriting*)

--cubical Enable cubical features. Turns on `--without-K` (see *Cubical*)

Errors and warnings

--allow-unsolved-metas Succeed and create interface file regardless of unsolved meta variables (see *Metavariables*)

--allow-incomplete-matches Succeed and create interface file regardless of incomplete pattern-matching definitions

--no-positivity-check Do not warn about not strictly positive data types (see *Positivity Checking*)

--no-termination-check Do not warn about possibly nonterminating code (see *Termination Checking*)

--warning=*GROUP|FLAG* -W *GROUP|FLAG* Set warning group or flag (see *Warnings*)

Pattern matching and equality

--without-K Disables definitions using Streicher's K axiom (see *Without K*)

--with-K Overrides a global `--without-K` in a file (see *Without K*)

--no-pattern-matching Disable pattern matching completely

--exact-split Require all clauses in a definition to hold as definitional equalities unless marked `CATCHALL` (see *Case trees*)

--no-exact-split Do not require all clauses in a definition to hold as definitional equalities (default)

--no-eta-equality Default records to no-eta-equality (see *Eta-expansion*)

Search depth and instances

--termination-depth=*N* Allow termination checker to count decrease/increase upto `N` (default: 1; see *Termination Checking*)

--instance-search-depth=*N* Set instance search depth to `N` (default: 500; see *Instance Arguments*)

--inversion-max-depth=*N* Set maximum depth for pattern match inversion to `N` (default: 50). Should only be needed in pathological cases.

--no-overlapping-instances Don't consider recursive instance arguments during pruning of instance candidates (default)

--overlapping-instances Consider recursive instance arguments during pruning of instance candidates

Other features

--safe Disable postulates, unsafe `OPTION` pragmas and `primTrustMe`. Turns off `--sized-types` and `--guardedness` (at most one can be turned back on again) (see *Safe Agda*)

--type-in-type Ignore universe levels (this makes Agda inconsistent; see *Universe Levels*)

--omega-in-omega Enable typing rule *Setω : Setω* (this makes Agda inconsistent).

--sized-types Enable sized types (default, inconsistent with constructor-based guarded corecursion; see *Sized Types*). Turned off by `--safe` (but can be turned on again, as long as not also `--guardedness` is on).

--no-sized-types Disable sized types (see *Sized Types*)

--guardedness Enable constructor-based guarded corecursion (default, inconsistent with sized types; see *Coinduction*). Turned off by `--safe` (but can be turned on again, as long as not also `--sized-types` is on).

--no-guardedness Disable constructor-based guarded corecursion (see *Coinduction*)

--universe-polymorphism Enable universe polymorphism (default; see *Universe Levels*)

--no-universe-polymorphism Disable universe polymorphism (see *Universe Levels*)

--no-irrelevant-projections Disable projection of irrelevant record fields (see *Irrelevance*)

--no-auto-inline Disable automatic compile-time inlining. Only definitions marked INLINE will be inlined.

--no-print-pattern-synonyms 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.

--double-check Enable double-checking of all terms using the internal typechecker

--no-syntactic-equality Disable the syntactic equality shortcut in the conversion checker

--no-fast-reduce Disable reduction using the Agda Abstract Machine

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

Individual warnings can be turned on and off by `-W {Name}` and `-W {noName}` respectively. The flags available are:

AbsurdPatternRequiresNoRHS RHS given despite an absurd pattern in the LHS.

CantGeneralizeOverSorts Attempt to generalize over sort metas in 'variable' declaration.

CoverageIssue Failed coverage checks.

CoverageNoExactSplit Failed exact split checks.

DeprecationWarning Feature deprecation.

EmptyAbstract Empty `abstract` blocks.

EmptyInstance Empty `instance` blocks.

EmptyMacro Empty `macro` blocks.

EmptyMutual Empty `mutual` blocks.

EmptyPostulate Empty `postulate` blocks.

EmptyPrimitive Empty `primitive` blocks.

EmptyPrivate Empty `private` blocks.

EmptyRewritePragma Empty `REWRITE` pragmas.

IllformedAsClause Illformed `as`-clauses in `import` statements.

InstanceNoOutputTypeName Instance arguments whose type does not end in a named or variable type are never considered by instance search.

InstanceArgWithExplicitArg Instance arguments with explicit arguments are never considered by instance search.

InstanceWithExplicitArg Instance declarations with explicit arguments are never considered by instance search.

InvalidCatchallPragma `CATCHALL` pragmas before a non-function clause.

InvalidNoPositivityCheckPragma No positivity checking pragmas before non-*data*', `record` or `mutual` blocks.

InvalidTerminationCheckPragma Termination checking pragmas before non-function or `mutual` blocks.

InversionDepthReached Inversions of pattern-matching failed due to exhausted inversion depth.

LibUnknownField Unknown field in library file.

MissingDefinitions Names declared without an accompanying definition.

ModuleDoesntExport Names mentioned in an import statement which are not exported by the module in question.

NotAllowedInMutual Declarations not allowed in a mutual block.

NotStrictlyPositive Failed strict positivity checks.

OldBuiltin Deprecated `BUILTIN` pragmas.

OverlappingTokensWarning Multi-line comments spanning one or more literate text blocks.

PolarityPragmasButNotPostulates Polarity pragmas for non-postulates.

PragmaCompiled `COMPILE` pragmas not allowed in safe mode.

PragmaCompileErased `COMPILE` pragma targeting an erased symbol.

PragmaNoTerminationCheck `NO_TERMINATION_CHECK` pragmas are deprecated.

RewriteMaybeNonConfluent Failed confluence checks while computing overlap.

RewriteNonConfluent Failed confluence checks while joining critical pairs.

SafeFlagNonTerminating `NON_TERMINATING` pragmas with the safe flag.

SafeFlagNoPositivityCheck `NO_POSITIVITY_CHECK` pragmas with the safe flag.

SafeFlagNoUniverseCheck `NO_UNIVERSE_CHECK` pragmas with the safe flag.

SafeFlagPolarity `POLARITY` pragmas with the safe flag.

SafeFlagPostulate `postulate` blocks with the safe flag

SafeFlagPragma Unsafe `OPTIONS` pragmas with the safe flag.

SafeFlagTerminating `TERMINATING` pragmas with the safe flag.

SafeFlagWithoutKFlagPrimEraseEquality `primEraseEquality` used with the safe and without-K flags.

ShadowingInTelescope Repeated variable name in telescope.

TerminationIssue Failed termination checks.

UnknownFixityInMixfixDecl Mixfix names without an associated fixity declaration.

UnknownNamesInFixityDecl Names not declared in the same scope as their syntax or fixity declaration.

UnknownNamesInPolarityPragmas Names not declared in the same scope as their polarity pragmas.

UnreachableClauses Unreachable function clauses.

UnsolvedConstraints Unsolved constraints.

UnsolvedInteractionMetas Unsolved interaction meta variables.

UnsolvedMetaVariables Unsolved meta variables.

UselessAbstract `abstract` blocks where they have no effect.

UselessInline `INLINE` pragmas where they have no effect.

UselessInstance `instance` blocks where they have no effect.

UselessPrivate `private` blocks where they have no effect.

UselessPublic `public` blocks where they have no effect.

WithoutKFlagPrimEraseEquality `primEraseEquality` used with the without-K flags.

WrongInstanceDeclaration Terms marked as eligible for instance search should end with a name.

CoInfectiveImport Importing a file not using e.g. `--safe` from one which does.

InfectiveImport Importing a file using e.g. `--cubical` into one which doesn't.

For example, the following command runs Agda with all warnings enabled, except for warnings about empty `abstract` blocks:

```
agda -W all --warning=noEmptyAbstract file.agda
```

4.2.3 Consistency checking of options used

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:

- `--cubical`
- `--prop`

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:

- `--safe`
- `--without-K`
- `--no-universe-polymorphism`
- `--no-sized-types`
- `--no-guardedness`

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:

- `--termination-depth`
- `--no-unicode`
- `--allow-unsolved-metas`
- `--allow-incomplete-matches`
- `--no-positivity-check`
- `--no-termination-check`
- `--type-in-type`
- `--omega-in-omega`
- `--no-sized-types`
- `--no-guardedness`
- `--injective-type-constructors`
- `--prop`
- `--no-universe-polymorphism`
- `--irrelevant-projections`
- `--experimental-irrelevance`
- `--without-K`
- `--exact-split`
- `--no-eta-equality`
- `--rewriting`
- `--cubical`
- `--overlapping-instances`
- `--safe`

- `--double-check`
- `--no-syntactic-equality`
- `--no-auto-inline`
- `--no-fast-reduce`
- `--instance-search-depth`
- `--inversion-max-depth`
- `--warning`

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 backend can be invoked from the command line using the flag `--compile`:

```
agda --compile [--compile-dir=<DIR>] [--ghc-flag=<FLAG>] <FILE>.agda
```

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
```

(continues on next page)

(continued from previous page)

```
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!`.

Required libraries for the Built-ins

- `primFloatEquality` requires the ieee754 library.

JavaScript Backend

The JavaScript backend translates Agda code to JavaScript code.

Usage

The backend can be invoked from the command line using the flag `--js`:

```
agda --js [--compile-dir=<DIR>] <FILE>.agda
```

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.

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 Emacs Mode

4.4.1 Introduction

4.4.2 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 ∀≥ℕ→π[[]]) 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.

4.4.3 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.

Commands working with types can be prefixed with `C-u` to compute type without further normalisation and with `C-u C-u` to compute normalised types.

Global commands

C-c C-l **L**oad file

C-c C-x C-c **C**ompile file

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-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-= Show constraints

C-c C-s **S**olve constraints

C-c C-? Show all goals

C-c C-f Move to next goal (**f**orward)

C-c C-b Move to previous goal (**b**ackwards)

C-c C-d Infer (**d**educe) type

C-c C-o M**o**dule c**o**ntents

C-c C-z Search through definitions in scope

C-c C-n Compute **n**ormal form

C-u C-c C-n Compute normal form, ignoring `abstract`

C-u C-u C-c C-n Compute and print normal form of `show <expression>`

C-c C-x M-; Comment/uncomment rest of buffer

C-c C-x C-s Switch to a different Agda version

Commands in context of a goal

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 **R**efine. Partial give: makes new holes for missing arguments

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-a *Automatic Proof Search (Auto)*

C-c C-c **C**ase split

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-d Infer (**d**educe) type

C-c C-, Goal type and context

C-c C-. Goal type, context and inferred type

C-c C-; Goal type, context and checked term

C-c C-o M**o**dule c**o**ntents

C-c C-n Compute **n**ormal form

C-u C-c C-n Compute normal form, ignoring `abstract`

C-u C-u C-c C-n Compute and print normal form of `show <expression>`

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

M-* Go back (Emacs < 25.1)

M-, Go back (Emacs $\geq$ 25.1)

4.4.4 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). Note that not all characters have a subscript or superscript counterpart in Unicode.

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	

4.4.5 Highlight

Clauses which do not hold definitionally (see *Case trees*) are highlighted in white smoke.

4.5 Literate Programming

Agda supports a limited form of literate programming, i.e. code interspersed with prose, if the corresponding filename extension is used.

4.5.1 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.

4.5.2 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.

4.5.3 Literate Markdown

Files ending in `.lagda.md` are interpreted as literate Markdown files. 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
```
````

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.

4.5.4 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 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).

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.

4.6.1 Options

`--html-dir=`*`directory`* Changes the directory where the output is placed to *`directory`*. Default: `html`.

`--css=`*`URL`* The CSS file used by the HTML files (*`URL`* can be relative).

`--html-highlight=[code,all,auto]` Highlight Agda code only or everything in the generated HTML files. Default: `all`.

4.7 Generating LaTeX

An experimental 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`.

4.7.1 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}}
```

- 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.

4.7.2 Options

The following command-line options change the behaviour of the LaTeX backend:

`--latex-dir=`*`directory`* Changes the output directory where `agda.sty` and the output `.tex` file are placed to *`directory`*. Default: `latex`.

`--only-scope-checking` Generates highlighting without typechecking the file. See *Quicker generation without typechecking*.

`--count-clusters` Count extended grapheme clusters when generating LaTeX code. This option can be given in `OPTIONS` pragmas. See *Counting Extended Grapheme Clusters*.

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*.

4.7.3 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.

4.7.4 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:

```
  |
+_|
o¨^|
PDF TODOPDF TODO|
PDF TODOPDF TODOPDF TODO|
PDF TODOPDF TODOPDF TODOPDF TODOPDF TODOPDF TODOPDF TODOPDF TODOPDF TODOPDF TODOPDF TO
  |
```

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}
\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}}.
```

(continues on next page)

(continued from previous page)

```
\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
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.

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 $(dirname $(dirname $(agda-mode locate)))/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
$ 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{
    \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. Here is one way in which this can be accomplished. (Perhaps this technique was 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.

4.7.5 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}

% Support for Agda code.
\usepackage{agda}
```

(continues on next page)

(continued from previous page)

```
% 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}

% 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}
```

(continues on next page)

(continued from previous page)

```
% 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}

% 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}}
```

(continues on next page)

(continued from previous page)

```
% 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.
\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}}
```

(continues on next page)

(continued from previous page)

```
% 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
```

(continues on next page)

(continued from previous page)

```
% 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}

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}

Some code:
\begin{code}
{-# OPTIONS --without-K --count-clusters #-}
```

(continues on next page)

(continued from previous page)

```
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 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*.

4.8.1 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 `~/.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!

4.8.2 Library files

A library consists of

- a name
- a set of dependencies
- a set of include paths

Libraries are defined in `.agda-lib` files with the following syntax:

```
name: LIBRARY-NAME  -- Comment
depend: LIB1 LIB2
  LIB3
  LIB4
include: PATH1
  PATH2
  PATH3
```

Dependencies are library names, not paths to `.agda-lib` files, and include paths are relative to the location of the library-file.

Each of the three 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.

4.8.3 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 `~/.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.

Each line of the libraries file shall be the absolute file system path to the root of a library.

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=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).

4.8.4 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.
- No explicit `--library` flag 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` flag, 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 flag `--no-default-libraries`. To disable using libraries altogether, use the `--no-libraries` flag.

4.8.5 Default libraries

If you want to usually use a variety of libraries, it is simplest to list them all in the `AGDA_DIR/defaults` file.

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).

4.8.6 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` flag, 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`.

4.8.7 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.

CHAPTER 5

Contribute

See also the *HACKING* file in the root of the Agda repository.

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`.
  - LaTeX
  - dvipng

  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 like 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

::

  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 6

The Agda Team and License

Authors:

- Ulf Norell
- Andreas Abel
- Nils Anders Danielsson
- Makoto Takeyama
- Catarina Coquand

Contributors (alphabetically sorted):

- Stevan Andjelkovic
- Marcin Benke
- Jean-Philippe Bernardy
- James Chapman
- Jesper Cockx
- Dominique Devriese
- Péter Diviánszky
- Fredrik Nordvall Forsberg
- Olle Fredriksson
- Daniel Gustafsson
- Philipp Hausmann
- Patrik Jansson
- Alan Jeffrey
- Wolfram Kahl
- John Leo

- Fredrik Lindblad
- Francesco Mazzoli
- Stefan Monnier
- Darin Morrison
- Guilhem Moulin
- Nicolas Pouillard
- Benjamin Price
- Andrés Sicard-Ramírez
- Andrea Vezzosi
- Tesla Ice Zhang
- and many more

The Agda license is here.

CHAPTER 7

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.