Agda User Manual

Release 2.5.4.1

The Agda Team

Jul 18, 2018

CONTENTS

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 tools.

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*.

GETTING STARTED

2.1 Prerequisites

You need recent versions of the following programs:

- GHC: https://www.haskell.org/ghc/

- cabal-install: https://www.haskell.org/cabal/

- Alex: https://www.haskell.org/alex/

- Happy: https://www.haskell.org/happy/

- cpphs: https://hackage.haskell.org/package/cpphs

- 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*.

2.1.1 Installing Emacs under Windows

A precompiled version of Emacs 24.3, with the necessary mathematical fonts, is available at http://homepage.cs.uiowa.edu/~astump/agda/ .

2.2 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*).

2.2.1 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.

2.2.2 Prebuilt Packages and System-Specific Instructions

Arch Linux

The following prebuilt packages are available:

- Agda
- Agda standard library

Debian / Ubuntu

Prebuilt packages are available for Debian testing/unstable 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 testing/unstable 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.

FreBSD

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](#).

2.2.3 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 or
    - You can try to install Agda (including a compiled Emacs mode) by running the following command:

```
make install
```

2.2.4 Installation Flags

When installing Agda the following flags can be used:

cpphs Use cpphs instead of cpp. Default: on.

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.

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

2.3.1 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 25.1)

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

2.3.2 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.

2.3.3 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 ∀, α, ∧, or ♠, 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 ∀. 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.)

2.3.4 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.

2.3.5 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).

2.3.6 Batch-mode command

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

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, $proj_1$ 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
    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

- *Using the built-in types*
- *The unit type*
- *Booleans*
- *Natural numbers*
- *Machine words*
- *Integers*
- *Floats*
- *Lists*
- *Characters*
- *Strings*
- *Equality*
- *Universe levels*
- *Sized types*
- *Coinduction*
- *IO*
- *Literal overloading*
- *Reflection*
- *Rewriting*
- *Static values*
- *Strictness*

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
```

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 proofs of these theorems are not primitive, but can be defined in a library using *primTrustMe*.

Basic arithmetic operations can be defined on `Word64` by converting to natural numbers, peforming 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
```

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.

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
```

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`.

3.2.10 Strings

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

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*):

```
postulate primStringToList   : String → List Char
postulate primStringFromList : List Char → String
postulate primStringAppend   : String → String → String
postulate primStringEquality : String → String → Bool
postulate primShowString     : String → String
```

String literals can be *overloaded*.

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 `primTrustMe` has to match the flavor of identity type.

primTrustMe

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

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

```
primitive
  primTrustMe : ∀ {a} {A : Set a} {x y : A} → 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`. Another use case is to erase computationally expensive equality proofs and replace them by `primTrustMe`:

```
eraseEquality : ∀ {a} {A : Set a} {x y : A} → x ≡ y → x ≡ y
eraseEquality _ = primTrustMe
```

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 _↦_ #-}
```

There is no `Agda.Builtin` module for the rewrite relation since different rewriting experiments typically want different relations.

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

3.3 Coinduction

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 neccessary 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.

Note: The type constructor ∞ can be used to prove absurdity!

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

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

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

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

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

```
inf : Coℕ
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.
```

(continues on next page)

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

The recommended definition is the following one:

```
data _≈_ : CoℕΝ → CoℕΝ → 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 where
  constructor enumeration
  field
    start    : A
    forward  : A → A
    backward : A → A
```

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

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

```
open Enumeration

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

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

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

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

```
open Enumeration

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

test₁ : 3rd enum-Nat ≡ 3
test₁ = refl

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

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

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

```
open Enumeration

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

The resulting behaviour is the same in both cases:

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

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

3.4.1 Copatterns in function definitions

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

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

```
open Enumeration

enum-Nat : Nat → Enumeration Nat
enum-Nat initial = record {
    start   = initial
  ; forward = suc
  ; backward = pred
```

```
  }
  where
    pred : Nat → Nat
    pred zero    = zero
    pred (suc x) = x


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

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

```
open Enumeration

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

3.4.2 Mixing patterns and 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 Data Types

3.6.1 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 represent 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 : Setᵢ where
  c₁ : A₁
  ...
  cₙ : Aₙ
```

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

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

3.6.2 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
```

3.6.3 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, ..., A_n are function types of the form

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

3.6.4 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, ..., c_n : A_n, Agda checks that each A has the form

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

where an argument types B 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.

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*.

3.7 Foreign Function Interface

- *Compiler Pragmas*
- *Haskell FFI*
    - *The `FOREIGN` pragma*
    - *The `COMPILE` pragma*
    - *Using Haskell Types from Agda*
    - *Using Haskell functions from Agda*
    - *Using Agda functions from Haskell*
    - *Polymorphic functions*
    - *Level-polymorphic types*
    - *Handling typeclass constraints*

- *JavaScript FFI*

3.7.1 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.

3.7.2 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 ⊤
{-# 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)
```

(continues on next page)

```
-- 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
{-# 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
```

3.7.3 JavaScript FFI

The *JavaScript backend* recognizes `COMPILE` pragmas of the following form:

```
{-# COMPILE JS <Name> = <JsCode> #-}
```

where `<Name>` is a postulate, constructor, or data type. The code for a data type is used to compile pattern matching and should be a function taking a value of the data type and a table of functions (corresponding to case branches) indexed by the constructor names. For instance, this is the compiled code for the `List` type, compiling lists to JavaScript arrays:

```
data List {a} (A : Set a) : Set a where
  []  : List A
  _::_ : (x : A) (xs : List A) → List A

{-# COMPILE JS List = function(x,v) {
    if (x.length < 1) {
      return v["[]"]();
    } else {
      return v["_::_"](x[0], x.slice(1));
```

(continues on next page)

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

3.8 Function Definitions

3.8.1 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))
```

3.8.2 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*.

3.8.3 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 `root : (n : Nat) → Square n → Nat` that takes as its arguments a number `n` and a proof that it is a square, and returns the square root of that number. We can do so as follows:

```
root : (n : Nat) → Square n → Nat
root .(m * m) (sq m) = m
```

Notice that by matching on the argument of type `Square n` with the constructor `sq : (m : Nat) → Square (m * m)`, `n` is forced to be equal to `m * m`.

In general, when matching on an argument of type $D\ i_1\ …\ i_n$ with a constructor $c\ :\ (x_1\ :\ A_1)\ →\ …\ →\ (x_m\ :\ A_m)\ →\ D\ j_1\ …\ j_n$, Agda will attempt to unify $i_1\ …\ i_n$ with $j_1\ …\ j_n$. When the unification algorithm instantiates a variable x with value `t`, the corresponding argument of the function can be replaced by a dot pattern `.t`. Using a dot pattern 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 $root_2$, `n` evaluates to `m * m` in the body of the function and is thus equivalent to

```
root₃ : (n : Nat) → Square n → Nat
root₃ _ (sq m) = let n = m * m in m
```

Absurd patterns

Absurd patterns can be used when none of the constructors for a particular argument would be valid. The syntax for an absurd pattern is `()`.

As an example, if we have a datatype `Even` defined as follows

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

then we can define a function `one-not-even : Even 1 → ⊥` by using an absurd pattern:

```
one-not-even : Even 1 → ⊥
one-not-even ()
```

Note that if the left-hand side of a clause contains an absurd pattern, its right-hand side must be omitted.

In general, when matching on an argument of type $D\ i_1\ \dots\ i_n$ with an absurd pattern, Agda will attempt for each constructor $c\ :\ (x_1\ :\ A_1)\ \to\ \dots\ \to\ (x_m\ :\ A_m)\ \to\ D\ j_1\ \dots\ j_n$ of the datatype D to unify $i_1\ \dots\ i_n$ with $j_1 \dots\ j_n$. 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.

3.8.4 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.

3.9 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]`.

3.9.1 Notational conventions

Function types:

```
prop₁ : ((x : A) (y : B) → C) is-the-same-as   ((x : A) → (y : B) → C)
prop₂ : ((x y : A) → C)       is-the-same-as   ((x : A)(y : A) → C)
prop₃ : (forall (x : A) → C)  is-the-same-as   ((x : A) → C)
prop₄ : (forall x → C)        is-the-same-as   ((x : _) → C)
prop₅ : (forall x y → C)      is-the-same-as   (forall x → forall y → C)
```

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

Functional abstraction:

```
(\x y → e)                    is-the-same-as    (\x → (\y → e))
```

Functional application:

```
(f a b)                       is-the-same-as    ((f a) b)
```

3.10 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 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` there 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)` $\rightarrow$ `A` $\rightarrow$ `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
```

When constructing implicit function spaces the implicit argument can be omitted, so both expressions below are valid expressions of type `{A : Set}` $\rightarrow$ `A` $\rightarrow$ `A`:

```
z1 = λ {A} x → x
z2 = λ x → x
```

The ∀ (or `forall`) syntax for function types also has implicit variants:

```
☐ : (∀ {x : A} → B)    is-the-same-as  ({x : A} → B)
☐ : (∀ {x} → B)        is-the-same-as  ({x : _} → B)
☐ : (∀ {x y} → B)      is-the-same-as  (∀ {x} → ∀ {y} → B)
```

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.

3.10.1 Metavariables

3.10.2 Unification

3.11 Instance Arguments

- *Usage*
  - *Defining type classes*
  - *Declaring instances*
  - *Examples*
- *Instance resolution*

Instance arguments are the Agda equivalent of Haskell type class constraints and can be used for many of the same purposes. In Agda terms, they are *implicit arguments* that get solved by a special *instance resolution* algorithm, rather than by the unification algorithm used for normal implicit arguments. In principle, an instance argument is resolved, if a unique *instance* of the required type can be built from *declared instances* and the current context.

3.11.1 Usage

Instance arguments are enclosed in double curly braces `{{ }}`, e.g. `{{x : T}}`. Alternatively they can be enclosed, with proper spacing, e.g. ⦃ x : T ⦄, in the unicode braces ⦃ ⦄ (`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
```

Note that instance arguments in types are always named, but the name can be `_`:

```
_==_ : {A : Set} → {{Eq A}} → A → A → Bool    -- INVALID
```

```
_==_ : {A : Set} {{_ : Eq A}} → A → A → Bool   -- VALID
```

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 (ordinary) implicit arguments (see *Dependent instances* below for an example where Γ is non-empty). Instance arguments that do not have this form are currently accepted, but instance resolution may or may not work as described below for such arguments.

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
        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.

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. Typically arguments to constructors are not instance arguments, so during instance resolution explicit arguments are treated as 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
```

<div align="right">(continues on next page)</div>

```
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.

Examples

Proof search

Instance arguments are useful not only for Haskell-style type classes, but they can also be used to get some limited form of proof search (which, to be fair, is also true for Haskell type classes). Consider the following type, which models a proof that a particular element is present in a list as the index at which the element appears:

```
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
```

Here we have declared the constructors of `_∈_` to be instances, which allows instance resolution to find proofs for concrete cases. For example,

```
ex₁ : 1 + 2 ∈ 1 ∷ 2 ∷ 3 ∷ 4 ∷ []
ex₁ = it  -- computes to suc (suc zero)

ex₂ : {A : Set} (x y : A) (xs : List A) → x ∈ y ∷ y ∷ x ∷ xs
ex₂ x y xs = it  -- suc (suc zero)

ex₃ : {A : Set} (x y : A) (xs : List A) {{i : x ∈ xs}} → x ∈ y ∷ y ∷ xs
ex₃ x y xs = it  -- suc (suc i)
```

It will fail, however, if there are more than one solution, since instance arguments must be unique. For example,

```
fail₁ : 1 ∈ 1 ∷ 2 ∷ 1 ∷ []
fail₁ = it  -- ambiguous: zero or suc (suc zero)

fail₂ : {A : Set} (x y : A) (xs : List A) {{i : x ∈ xs}} → x ∈ y ∷ x ∷ xs
fail₂ x y xs = it -- suc zero or suc (suc i)
```

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 x_1 , y_1, the types of the second components y and y_1 can be completely different and not admit an equality test. Only when x and x_1 are *really equal* can we hope to compare y and y_1. Having the equality function return a proof means that we are guaranteed that when x and x_1 compare equal, they really are equal, and comparing y and 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.

3.11.2 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 is not already solved. This can happen if there are *unification constraints* determining the value, or if it is of singleton record type and thus solved by *eta-expansion*.

Next 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 arguments. If this is not the case instance resolution fails with an error message[1].

Finally we have to check that there are no *unconstrained metavariables* in vs. A metavariable α is considered constrained if it appears in an argument that is determined by the type of some later argument, or if there is an existing constraint of the form α us = C vs, where C inert (i.e. a data or type constructor). For example, α is constrained in T α xs if T : (n : Nat) → Vec A n → Set, since the type of the second argument of T determines the value of the first argument. The reason for this restriction is that instance resolution risks looping in the presence of unconstrained metavariables. For example, suppose the goal is Eq α for some metavariable α. Instance resolution would decide that the eqList instance was applicable if setting α := List β for a fresh metavariable β, and then proceed to search for an instance of Eq β.

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

---

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

if they are bound as instance arguments using `{{ }}`. Only candidates that compute something of type `C us`, where `C` is the target type computed in the previous stage, are considered.

Check the candidates We attempt to use each candidate in turn to build an instance of the goal type $\{\Gamma\}$ $\rightarrow$ `C vs`. First we extend the current context by Γ. Then, given a candidate `c` : $\Delta \rightarrow$ `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 explicit arguments and instance arguments.

Next we *unify* `A[`Δ` := `α`s]` 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 $\{\Gamma\}$ $\rightarrow$ `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-=`).

3.12 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.

3.12.1 Motivating example

One intended use case of irrelevance is data structures with embedded proofs, like sorted lists.

```
data _≤_ : Nat → Nat → Set where
  zero≤   : {n : Nat} → zero ≤ n
  suc≤suc : {m n : Nat} → m ≤ n → suc m ≤ suc n

postulate
  p₁ : 0 ≤ 1
  p₂ : 0 ≤ 1

module No-Irrelevance where
  data SList (bound : Nat) : Set where
    []    : SList bound
    scons : (head : Nat)
          → (head ≤ bound)
          → (tail : SList head)
          → SList bound
```

Usually, when we define datatypes with embedded proofs we are forced to reason about the values of these proofs. For example, suppose we have two lists l_1 and l_2 with the same elements but different proofs:

```
l₁ : SList 1
l₁ = scons 0 p₁ []

l₂ : SList 1
l₂ = scons 0 p₂ []
```

Now suppose we want to prove that 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 `head ≤ bound` exists, i.e. any proof suffices. We can use irrelevance annotations to tell Agda we don't care about the values of the proofs:

```
data SList (bound : Nat) : Set where
  []    : SList bound
  scons : (head : Nat)
        → .(head ≤ bound)        -- note the dot!
        → (tail : SList head)
        → SList bound
```

The effect of the irrelevant type in the signature of scons is that scons's second argument is never inspected after Agda has ensured that it has the right type. The type-checker ignores irrelevant arguments when checking equality, so two lists can be equal even if they contain different proofs:

```
l₁ : SList 1
l₁ = scons 0 p₁ []

l₂ : SList 1
l₂ = scons 0 p₂ []

l₁≡l₂ : l₁ ≡ l₂
l₁≡l₂ = refl
```

3.12.2 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 ()
```

Example 4. We can match on an irrelevant record (see *Record Types*) as long as we only use the fields irrelevantly.

```
record _×_ (A B : Set) : Set where
  constructor _,_
  field
    fst : A
    snd : B

irrElim : {A B C : Set} → .(A × B) → (.A → .B → C) → C
irrElim (a , b) f = f a b

lemma : {A B C : Set} {a a' : A} {b b' : B}
      → (f : .A -> .B -> C) -> irrElim (a , b) f ≡ f a' b'
lemma f = refl
```

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
```

3.12.3 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.

3.12.4 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.

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
```

3.12.5 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))`.

3.12.6 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
```

3.13 Lambda Abstraction

3.13.1 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
        ; 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 }
```

3.14 Local Definitions: let and where

There are two ways of declaring local definitions in Agda:

- let-expressions
- where-blocks

3.14.1 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 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 $e'[f_1 := \lambda\ x_1\ …\ x_n \to e;$ $…;\ f_m := \lambda\ x_1\ …\ x_k \to e_m]$. Since Agda substitutes away let-bindings, they do not show up in terms Agda prints, nor in the goal display in interactive mode.

3.14.2 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.

3.14.3 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 []
```

3.14.4 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
```

3.15 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.

3.15.1 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` `where` `with`

> The `Set` keyword can appear with a number suffix, optionally subscripted (see *Universe Levels*). For instance `Set42` and 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: `"Привет \"мир\"\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.

3.15.2 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
```

3.15.3 Literate Agda

Agda supports literate programming where everything in a file is a comment unless enclosed in `\begin{code}`, `\end{code}`. Literate Agda files have the extension `.lagda` instead of `.agda`. The main use case for literate Agda is to generate LaTeX documents from Agda code. See *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}
```

3.16 Literal Overloading

3.16.1 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     = T
suc m ≤ zero  = ⊥
suc m ≤ suc n = m ≤ n

fromN≤ : ∀ m n → m ≤ n → Fin (suc n)
fromN≤ zero     _       _ = zero
fromN≤ (suc _) zero    ()
fromN≤ (suc m) (suc n) p = suc (fromN≤ m n p)

instance
  NumFin : ∀ {n} → Number (Fin (suc n))
  NumFin {n} .Number.Constraint m        = m ≤ n
  NumFin {n} .Number.fromNat    m {{m≤n}} = fromN≤ m n m≤n

test : Fin 5
test = 3
```

It is important that the constraint for literals is trivial. Here, `3 ≤ 5` evaluates to `T` 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}
```

3.16.2 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 #-}
```

3.16.3 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 #-}
```

3.16.4 Restrictions

Currently only integer and string literals can be overloaded.

Overloading does not work in patterns yet.

3.17 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
```

3.17.1 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 an integer (can be negative!). 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
```

3.17.2 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
```

3.17.3 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).

3.18 Module System

3.18.1 Module application

3.18.2 Anonymous modules

3.18.3 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
```

3.18.4 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 .Main.A.IsZero' n. The leading dot 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.

3.18.5 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. Note that if xs and ys overlap there will be two names introduced for the same definition. We do not permit xs and zs to overlap. The other forms of opening are defined in terms of this one. Let A denote all the (public) names in A. Then

```
open A renaming (ys to zs)
== open A hiding () renaming (ys to zs)

open A hiding (xs) renaming (ys to zs)
== open A using (A ; xs ; ys) renaming (ys to zs)
```

An omitted renaming modifier is equivalent to an empty renaming.

3.18.6 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.

3.18.7 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]
```

3.18.8 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 is 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.

3.18.9 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 Nat_2.zero, Nat_2.suc, Fin.zero, and Fin.suc (Nat_2 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*.

3.19 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
```

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
```

3.19.1 Old Syntax

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]
```

This alternative syntax desugars into the new syntax.

3.20 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.

3.20.1 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
```

3.21 Positivity Checking

Note: This is a stub.

3.21.1 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
```

3.21.2 *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.

3.22 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

  -- ...
```

3.22.1 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.

3.23 Pragmas

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

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

3.23.1 Index of pragmas

- *BUILTIN*
- *CATCHALL*
- *COMPILE*
- *FOREIGN*
- *NO_POSITIVITY_CHECK*
- *NO_TERMINATION_CHECK*
- *NON_TERMINATING*
- *POLARITY*
- *STATIC*
- *TERMINATING*
- *INLINE*
- *NOINLINE*
- *WARNING_ON_USAGE*

See also *Command-line and pragma options*.

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

-- 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_USAGE` pragma

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.

This would typically be used to declare a name 'DEPRECATED' and advise the end-user to port their code before the feature is dropped.

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`" #-}
```

3.24 Record Types

- *Record modules*
- *Eta-expansion*
- *Recursive records*
- *Instance fields*

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).

3.24.1 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 matches 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 accept 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
```

Internally, this is translated to

```
sum' : Pair Nat Nat → Nat
sum' p = (Pair.fst p) + (Pair.snd p)
```

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 = _`).

3.24.2 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 D_1 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
```

3.24.3 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.

3.24.4 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 `inductive` 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.

3.24.5 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

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 → 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
```

(continues on next page)

```
    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._<_

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

3.25 Reflection

3.25.1 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
```

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 and show:

```
postulate Meta : Set
{-# BUILTIN AGDAMETA Meta #-}

primitive
  primMetaEquality : Meta → Meta → Bool
  primMetaLess     : Meta → Meta → Bool
  primShowMeta     : Meta → String
```

Builtin metavariables show up in reflected terms.

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
```

(continues on next page)

```
{-# 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 ⊤

  -- 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 ⊤
```

```
-- 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 if the corresponding verbosity level is turned
-- on (with the -v flag to Agda).
debugPrint : String → Nat → List ErrorPart → TC ⊤

{-# 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         #-}
{-# 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         #-}
```

3.25.2 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 $t_1 \rightarrow t_2 \rightarrow .. \rightarrow$ Term $\rightarrow$ 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 $\rightarrow$ Name $\rightarrow$ Bool $\rightarrow$ Term $\rightarrow$ 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 $\rightarrow$ 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 $\rightarrow$ 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.

3.26 Rewriting

Note: This is a stub.

3.27 Safe Agda

By using the command-line option `--safe`, 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.

- `--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` allows the user to encode the Girard-Hurken paradox.

- `--injective-type-constructors` together with excluded middle leads to an inconsistency via Chnug-Kil Hur's construction.

- `guardedness-preserving-type-constructors` is based on a rather operational understanding of $\infty/\sharp\_$; it's not yet clear if this extension is consistent.

- `--experimental-irrelevance` enables potentially unsound irrelevance features (irrelevant levels, irrelevant data matching).

- `--rewriting` turns any equation into one that holds definitionally. It can at the very least break convergence.

3.27.1 Known Issues

Pragma Option

It is possible to specify `{-# OPTIONS --safe #-}` at the top of a file. Unfortunately a known bug (see #2487) means that the option choice is not repercuted in the imported file. Therefore only the command-line option can be trusted.

Standard Library

The standard library uses a lot of unsafe features (e.g. `postulate` in the Foreign Function Interface) and these are not isolated in separate modules. As a consequence virtually any project relying on the standard library will not be successfully typechecked with the `--safe` option. There is work in progress to fix this issue.

3.28 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*.

3.28.1 Example for coinduction: finite languages

See *Abel 2017* and *Traytel 2016*.

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 ε ∈ 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: ε ∈ a · b iff ε ∈ a and ε ∈ b.

For the derivative (δ), assume that we have a word w, w ∈ δ (a · b) x. This means that xw = $\alpha\beta$, with α ∈ a and β ∈ b.

We have to consider two cases:

1. ε ∈ a. Then, either:

   - α = ε, and β = xw, where w ∈ δ b x.

   - α = xα', with α' ∈ δ a x, and w = $\alpha'\beta$ ∈ δ a x · b.

2. ε ∉ a. Then, only the second case above is possible:

   - α = xα', with α' ∈ δ a x, and w = $\alpha'\beta$ ∈ 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  = ø
δ ⟦ true  ⟧ false = ø
```

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
```

3.28.2 References

Equational Reasoning about Formal Languages in Coalgebraic Style, Andreas Abel.

Formal Languages, Formally and Coinductively, Dmitriy Traytel, FSCD (2016).

3.29 Syntactic Sugar

- *Do-notation*
  - *Desugaring*
  - *Example*
- *Idiom brackets*

3.29.1 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 pᵢ → mᵢ`
	`   m'`
		`m >>= λ where`
		`        p  → m'`
		`        pᵢ → mᵢ`

Non-binding statement	`do m` `   m'`	`m >>` `m'`
Let	`do let ds` `   m'`	`let ds in` `m'`

If the pattern in the bind is exhaustive, the where-clause can be omitted.

Example

Do-notation becomes quite powerful together with pattern matching on indexed data. As an example, let us write a correct-by-construction type checker for simply typed λ-calculus.

First we define the raw terms, using de Bruijn indices for variables and explicit type annotations on the lambda:

```
infixr 6 _=>_
data Type : Set where
  nat  : Type
  _=>_ : (A B : Type) → Type

data Raw : Set where
  var : (x : Nat) → Raw
  lit : (n : Nat) → Raw
  suc : Raw
  app : (s t : Raw) → Raw
  lam : (A : Type) (t : Raw) → Raw
```

Next up, well-typed terms:

```
Context = List Type

-- A proof of x ∈ xs is the index into xs where x is located.
infix 2 _∈_
data _∈_ {A : Set} (x : A) : List A → Set where
  zero : ∀ {xs} → x ∈ x :: xs
  suc  : ∀ {y xs} → x ∈ xs → x ∈ y :: xs

data Term (Γ : Context) : Type → Set where
  var : ∀ {A} (x : A ∈ Γ) → Term Γ A
  lit : (n : Nat) → Term Γ nat
  suc : Term Γ (nat => nat)
  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
  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.

3.29.2 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
```

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
  ```

3.30 Telescopes

Note: This is a stub.

3.31 Termination Checking

Note: This is a stub.

3.31.1 With-functions

3.31.2 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

   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
  ```

3.32 Universe Levels

Note: This is a stub.

3.33 With-Abstraction

- *Usage*
  - *Generalisation*
  - *Nested with-abstractions*
  - *Simultaneous abstraction*
  - *Using underscores and variables in pattern repetition*
  - *Rewrite*
  - *The inspect idiom*
  - *Alternatives to with-abstraction*
  - *Performance considerations*
- *Technical details*
  - *Examples*
  - *Ill-typed with-abstractions*

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.

3.33.1 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
```

(continued from previous page)

```
...                      | true  = x :: filter p xs
...                      | false = filter p xs
```

In this case `...` expands to `filter p (x :: xs)`. There are three cases where you have to spell out the left-hand side:

- If you want to do further pattern matching on the original arguments.

- When the pattern matching on the intermediate result refines some of the other arguments (see *Dot patterns*).

- To disambiguate the clauses of nested with-abstractions (see *Nested with-abstractions* below).

Generalisation

The power of with-abstraction comes from the fact that the goal type and the type of the original arguments are generalised over the value of the scrutinee. See *Technical details* below for the details. This generalisation is important when you have to prove properties about functions defined using `with`. For instance, suppose we want to prove that the `filter` function above satisfies some property `P`. Starting out by pattern matching of the list we get the following (with the goal types shown in the holes)

```
postulate P : ∀ {A} → List A → Set
postulate p-nil : 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
```

(continues on next page)

```
...                      | 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
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.

Rewrite

Remember example of *simultaneous abstraction* from above.

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

thm : (a b : Nat) → P (a + b) → P (b + a)
thm a b t with   a + b  | plus-commute a b
thm a b t    | .(b + a) | refl = t
```

This pattern of rewriting by an equation by with-abstracting over it and its left-hand side is common enough that there is special syntax for it:

```
thm : (a b : Nat) → P (a + b) → P (b + a)
thm a b t rewrite plus-commute a b = t
```

The `rewrite` construction takes a term `eq` of type `lhs ≡ rhs`, where `_≡_` is the *built-in equality type*, and expands to a with-abstraction of `lhs` and `eq` followed by a match of the result of `eq` against `refl`:

```
f ps rewrite eq = v

  -->

f ps with lhs | eq
...     | .rhs | refl = v
```

One limitation of the `rewrite` construction is that you cannot do further pattern matching on the arguments *after* the rewrite, since everything happens in a single clause. You can however do with-abstractions after the rewrite. For instance,

```
postulate T : Nat → Set

isEven : Nat → Bool
isEven zero = true
isEven (suc zero) = false
isEven (suc (suc n)) = isEven n

thm₁ : (a b : Nat) → T (a + b) → T (b + a)
thm₁ a b t rewrite plus-commute a b with isEven a
thm₁ a b t | true  = t
thm₁ a b t | false = t
```

Note that the with-abstracted arguments introduced by the rewrite (`lhs` and `eq`) are not visible in the code.

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 (re-exported 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.

3.33.2 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

$$
\begin{aligned}
&f : \Gamma \to B \\
&f\ ps \quad \textbf{with}\ t_1 \ \mid \ldots \mid t_m \\
&f\ ps_1 \qquad \mid q_{11} \mid \ldots \mid q_{1m} = v_1 \\
&\vdots \\
&f\ ps_n \qquad \mid q_{n1} \mid \ldots \mid q_{nm} = v_n
\end{aligned}
$$

---

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

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

$$A_i'[w_1 := t_1 \ldots w_{i-1} := t_{i-1}] \simeq A_i$$
$$(\Delta_2' \to B')[w_1 := t_1 \ldots w_m := t_m] \simeq \Delta_2 \to B$$

  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

$$f_{aux} : \Delta_1 \to C$$
$$f_{aux} \ ps_{11} \ qs_1 \ ps_{21} = v_1$$
$$\vdots$$
$$f_{aux} \ ps_{1n} \ qs_n \ ps_{2n} = v_n$$

  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

$$f : \Gamma \to B$$
$$f \ ps = f_{aux} \ xs_1 \ ts \ xs_2$$

  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
  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.

3.34 Without K

The option `--without-K` makes pattern matching more restricted. If the option is activated, then Agda only accepts certain case splits. This option was added in Agda 2.2.10.

Since Agda 2.4.0 when the option `--without-K` is enabled, then 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 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
```

On the other hand, 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.

For more details, see the paper *Eliminating dependent pattern matching without K* [Cockx, Devriese, and Piessens (2016)].

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 was added in Agda 2.4.2 and it is on by default.

Since Agda 2.4.2 termination checking `--without-K` 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.

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)

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

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)

--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)

--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

Sharing and caching

--sharing Enable sharing and call-by-need evaluation (experimental) (default: OFF)

--no-sharing Disable sharing and call-by-need evaluation

--caching Enable caching of typechecking (experimental) (default: OFF)

--no-caching Disable caching of typechecking

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

4.2.2 Command-line and pragma options

The following options can also be given in .agda files in the `{-# OPTIONS --{opt`$_1$`} --{opt`$_2$`} ... #-}` form at the top of the file.

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)

--guardedness-preserving-type-constructors Treat type constructors as inductive constructors when checking productivity

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

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

Errors and warnings

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

--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

--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.

Other features

--safe Disable postulates, unsafe `OPTION` pragmas and `primTrustMe` (see *Safe Agda*)

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

--sized-types Use sized types (default, inconsistent with "musical" coinduction; see *Sized Types*)

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

--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.

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:

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

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

UnknownFixityInMixfixDecl Mixfix names without an associated fixity declaration.

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

PolarityPragmasButNotPostulates Polarity pragmas for non-postulates.

UselessPrivate `private` blocks where they have no effect.

UselessAbstract `abstract` blocks where they have no effect.

UselessInstance `instance` blocks where they have no effect.

EmptyMutual Empty `mutual` blocks.

EmptyAbstract Empty `abstract` blocks.

EmptyPrivate Empty `private` blocks.

EmptyInstance Empty `instance` blocks.

EmptyMacro Empty `macro` blocks.

EmptyPostulate Empty `postulate` blocks.

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

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

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

OldBuiltin Deprecated `BUILTIN` pragmas.

EmptyRewritePragma Empty `REWRITE` pragmas.

UselessPublic `public` blocks where they have no effect.

UnreachableClauses Unreachable function clauses.

UselessInline `INLINE` pragmas where they have no effect.

DeprecationWarning Feature deprecation.

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

TerminationIssue Failed termination checks.

CoverageIssue Failed coverage checks.

CoverageNoExactSplit Failed exact split checks.

NotStrictlyPositive Failed strict positivity checks.

UnsolvedMetaVariables Unsolved meta variables.

UnsolvedInteractionMetas Unsolved interaction meta variables.

UnsolvedConstraints Unsolved constraints.

SafeFlagPostulate `postulate` blocks with the safe flag

SafeFlagPragma Unsafe `OPTIONS` pragmas with the safe flag.

SafeFlagNonTerminating `NON_TERMINATING` pragmas with the safe flag.

SafeFlagTerminating `TERMINATING` pragmas with the safe flag.

SafeFlagPrimTrustMe `primTrustMe` usages with the safe flag.

SafeFlagNoPositivityCheck `NO_POSITIVITY_CHECK` pragmas with the safe flag.

SafeFlagPolarity `POLARITY` pragmas with the safe flag.

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
```

4.3 Compilers

- *Backends*
  - *GHC Backend*
  - *JavaScript Backend*
- *Optimizations*
  - *Builtin natural numbers*
  - *Erasable types*

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
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 →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-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   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" (⊄).

(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	П		
0432	в		
0435	е		
0438	и		
043c	м		
0440	р		
0442	т		
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	⛾ (PDF TODO)		`\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. 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 tex code from literate Agda files. See *Unicode and LaTeX* 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.

4.6 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).

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).

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*. The source file must import the agda latex package by including the line \usepackage{agda}. Only the top-most module is processed, like with lhs2tex but unlike with the *HTML-backend*. If you want to process imported modules you have to call `agda --latex` manually on each of those modules.

The latex-backend checks if `agda.sty` is found by the latex environment. If it isn't, a default `agda.sty` is copied from Agda's data directory into the working directory (and thus made available to the latex environment). Colors, fonts, spacing etc can be modified by editing `agda.sty` and putting it somewhere where the latex environment can find it.

4.7.1 Unicode and LaTeX

LaTeX does not by default understand the UTF-8 character encoding. You can tell LaTeX to treat the input as UTF-8 using the ucs package by inserting the following code in the preamble of your source file:

```
\usepackage{ucs}
\usepackage[utf8x]{inputenc}
```

Unicode characters are translated to LaTeX commands, so e.g. the following packages might be needed. You may need more packages if you use more unicode characters:

```
\usepackage{amssymb}
\usepackage{bbm}
\usepackage[greek,english]{babel}
```

The ucs package does not contain definitions for all Unicode characters. If LaTeX complains about a missing definition for some character U+xxxx, then you can define it yourself:

```
\DeclareUnicodeCharacter{"xxxx}{<definition>}
```

It may also be necessary to instruct LaTeX to use a specific font encoding. The autofe package (from the ucs bundle) tries to select the font encoding automatically:

```
\usepackage{autofe}
```

A *complete LaTeX template* can be found below.

Note: LaTeX was never written with unicode in mind. Hacks like the ucs package makes it possible to use them, but for the best possible output consider using **xelatex** or **lualatex** instead. If you do, `agda.sty` is using the more complete XITS font by default.

4.7.2 Features

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}
```

Alignment

Two or more spaces can be used to make the backend align code, as in the following example:

```
\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 more detail, the 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).

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) lead to empty lines in the output. The height of such empty lines can be controlled by the length `\AgdaEmptySkip`, which by default is `\baselineskip`.

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`.

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.

Hyperlinks (experimental)

If the hyperref latex package is loaded before the agda package and the `links` option is passed to the agda package, then the agda package provides a function called `\AgdaTarget`. Identifiers which have been declared targets, by the user, will become clickable hyperlinks in the rest of the document. Here is a small example:

```
\documentclass{article}
\usepackage{hyperref}
\usepackage[links]{agda}
\begin{document}

\AgdaTarget{ℕ}
\AgdaTarget{zero}
\begin{code}
data ℕ : Set where
  zero  : ℕ
  suc   : ℕ → ℕ
\end{code}

See next page for how to define \AgdaFunction{two} (doesn't turn into a
link because the target hasn't been defined yet). We could do it
manually though; \hyperlink{two}{\AgdaDatatype{two}}.

\newpage

\AgdaTarget{two}
\hypertarget{two}{}
\begin{code}
two : ℕ
two = suc (suc zero)
\end{code}

\AgdaInductiveConstructor{zero} is of type
\AgdaDatatype{ℕ}. \AgdaInductiveConstructor{suc} has not been defined to
```

<div align="right">(continues on next page)</div>

```
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.

Typesetting inline code

The backend only typesets code inside code blocks; inline code has to be typeset manually, e.g.:

```
Below we postulate a set called \AgdaDatatype{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 `latex/agda.sty` file. If you are doing a lot of manual typesetting, then you might want to introduce shorter command names, e.g.:

```
\newcommand{\D}{\AgdaDatatype}
\newcommand{\F}{\AgdaFunction}
```

etc. Long names were chosen by default to avoid name clashes.

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.

Emulating %format rules

The LaTeX-backend has no feature analogue to **lhs2TeX**'s %format rules, however most systems come with **sed** which can be used to achieve similar goals. Given a file called, for example, `replace.sed`, containing:

```
# Super- and subscripts.
#s/\\textasciicircum\([^\}]*\)_\([^\}]*\)/\$\^\\AgdaFontStyle\{\\scriptscriptstyle \1\}\_\\AgdaFontStyle\
↪{\\scriptscriptstyle \2\}\$/g

s/\\textasciicircum\([^\}]*\)/\{\^\\AgdaFontStyle\{\\scriptscriptstyle \1\}\}/g

#s/_\([^\}]*\)/\$_\\AgdaFontStyle\{\\scriptscriptstyle \1\}\$/g

# Σ[ x ∈ X ] into (x : X) ×
s/\\AgdaRecord{Σ\[} \(.*\) \\AgdaRecord{∈} \(.*\) \\AgdaRecord{]}/\\AgdaSymbol\{(\}\1 \\AgdaSymbol\{:\}
↪\2\\AgdaSymbol\{)\} \\AgdaFunction\{×\}/g

# Bind, Kleisli extension and fmap.
#s/>>=/\$\\mathbin\{>\\!\\!\\!>\\mkern-6.7mu=\}\$/g
s/>>=/\\mathbin\{>\\!\\!\\!>\\mkern-6.7mu=\}/g
#s/>>/\$\\mathbin\{>\\!\\!\\!>}\$/g
#s/=<</\$\\mathbin\{=\\mkern-6.7mu<\\!\\!\\!<\}\$/g
#s/<\\$>/\$\\mathop\{<\\!\\!\\!\\$\\!\\!\\!>\}\$/g
s/<\\$>/\\mathop\{<\\!\\!\\!\\$\\!\\!\\!>\}/g

# Append.
s/++/+\\!+/g

# Comments.
#s/AgdaComment{\-\-/AgdaComment\{\-\\!\-/g
```

We can postprocess the tex output as follows:

```
$ sed -f replace.sed {file}.tex > {file}.sedded
$ mv {file}.sedded {file}.tex
```

Note that the above sed file assumes the use of **{xe|lua}latex** where code is in math mode rather than text mode (as is the case when using the **pdflatex** compiler). The commented out variants of the substitution rules are their pdflatex equivalents.

The substitution rules dealing with super- and subscripts lets us write `apa^bepa` in the code for things we want superscripted in the output (`\undertie` does the same for subscripts).

Including Agda code into a larger LaTeX document

Sometimes you might want to include a bit of code without necessarily making the whole document a literate Agda file. Here is how to do this in the context of a beamer presentation, but the method should work similarly also for other documents. Given two files `Presentation.tex` and `Code.lagda` as follows:

Listing 3: Presentation.tex

```latex
\documentclass{beamer}

% When using XeLaTeX, the following should be used instead:
% \documentclass[xetex]{beamer}
% \usefonttheme[onlymath]{serif}

\usepackage{latex/agda}
\usepackage{catchfilebetweentags}

\begin{document}
\begin{frame}\frametitle{Some Agda code}

  \begin{itemize}
    \item The natural numbers
  \end{itemize}

  \ExecuteMetaData[latex/Code.tex]{nat}

  \begin{itemize}
    \item Addition (\AgdaFunction{\_+\_})
  \end{itemize}

  \ExecuteMetaData[latex/Code.tex]{plus}

\end{frame}
\end{document}
```

Listing 4: Code.lagda

```latex
%<*nat>
\begin{code}
data ℕ : Set where
  zero  : ℕ
  suc   : (n : ℕ) → ℕ
\end{code}
%</nat>

%<*plus>
\begin{code}
_+_ : ℕ → ℕ → ℕ
zero   + n = n
suc m  + n = suc (m + n)
\end{code}
%</plus>
```

we can use **pdflatex** to compile a presentation as follows:

```
$ agda --latex Code.lagda
$ latexmk -pdf -use-make Presentation.tex
```

If you are using a lot of unicode it might be more convenient to use **xelatex** instead. See comments about **xelatex** in Presentation.tex and compile as follows:

```
$ agda --latex Code.lagda
$ latexmk -xelatex -pdf -use-make Presentation.tex
```

The \ExecuteMetaData command is part of the catchfilebetweentags package. Also see the following thread on the mailing list for other methods of including Agda code into a presentation.

4.7.3 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 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 Epigram1.

nofontsetup Instructs the package to not select any fonts, and to not change the font encoding.

noinputencodingsetup Instructs the package to not change the input encoding, and to not load the *ucs* package.

references Enables *inline typesetting* of referenced code.

links Enables *hyperlink support*.

4.7.4 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:

```
| |
|+_|
|ö↑|
|□□|
|□□□|
|□□□□□□□□□□□□|
| |
```

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`.

4.7.5 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 Quick-LaTeX behaves like the regular LaTeX backend.

4.7.6 Known issues

The unicode-math package and older versions of the polytable package (still in the Debian stable) are incompatible, which can result in errors in generated latex code. The workaround is to download a more up to date version of polytable and either put it with the generated files, or install it globally.

4.7.7 Complete LaTeX Template for Literate Agda with Unicode

This template has been tested under Debian and Ubuntu with TexLive, but should also work in other distributions. For **xelatex** or **lualatex**, only `\usepackage{agda}` should be needed.

```
\documentclass{article}

\usepackage{agda}

% The following packages are needed because unicode
% is translated (using the next set of packages) to
% latex commands. You may need more packages if you
% use more unicode characters:

\usepackage{amssymb}
\usepackage{bbm}
\usepackage[greek,english]{babel}

% This handles the translation of unicode to latex:

\usepackage{ucs}
\usepackage[utf8x]{inputenc}
\usepackage{autofe}

% Some characters that are not automatically defined
% (you figure out by the latex compilation errors you get),
% and you need to define:

\DeclareUnicodeCharacter{8988}{\ensuremath{\ulcorner}}
\DeclareUnicodeCharacter{8989}{\ensuremath{\urcorner}}
\DeclareUnicodeCharacter{8803}{\ensuremath{\overline{\equiv}}}

% Add more as you need them (shouldn't happen often).
```

```
\begin{document}

\begin{code}
 -- your Agda code goes here
\end{code}

\end{document}
```

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 pathes 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.

CONTRIBUTE

See also the *HACKING* file in the root of the agda repo.

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 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
```

THE AGDA LICENSE

Copyright (c) 2005-2017 remains with the authors. Agda 2 was originally written by Ulf Norell, partially based on code from Agda 1 by Catarina Coquand and Makoto Takeyama, and from Agdalight by Ulf Norell and Andreas Abel.

Agda 2 is currently actively developed mainly by Andreas Abel, Guillaume Allais, Jesper Cockx, Nils Anders Danielsson, Philipp Hausmann, Fredrik Nordvall Forsberg, Ulf Norell, Víctor López Juan, Andrés Sicard-Ramírez, and Andrea Vezzosi.

Further, Agda 2 has received contributions by, amongst others, Stevan Andjelkovic, Marcin Benke, Jean-Philippe Bernardy, Guillaume Brunerie, James Chapman, Dominique Devriese, Péter Diviánszki, Olle Fredriksson, Adam Gundry, Daniel Gustafsson, Kuen-Bang Hou (favonia), Patrik Jansson, Alan Jeffrey, Wolfram Kahl, Wen Kokke, Fredrik Lindblad, Francesco Mazzoli, Stefan Monnier, Darin Morrison, Guilhem Moulin, Nicolas Pouillard, Nobuo Yamashita, Christian Sattler, and Makoto Takeyama. The full list of contributors is available at https://github.com/agda/agda/graphs/contributors

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

The file src/full/Agda/Utils/Parser/ReadP.hs is Copyright (c) The University of Glasgow 2002 and is licensed under a BSD-like license as follows:

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

- Neither name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE UNIVERSITY COURT OF THE UNIVERSITY OF GLAS-GOW AND THE CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, IN-CLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE UNIVER-SITY COURT OF THE UNIVERSITY OF GLASGOW OR THE CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAM-AGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SER-VICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The file src/full/Agda/Utils/Maybe/Strict.hs (and the following license text?) uses the following license:

Copyright (c) Roman Leshchinskiy 2006-2007

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. Neither the name of the author nor the names of his contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IM-PLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHORS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIA-BILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF AD-VISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE AGDA TEAM

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
- Peter Divanski
- Fredrik Nordvall Forsberg
- Olle Fredriksson
- Daniel Gustafsson
- Philipp Hausmann
- Patrik Jansson
- Alan Jeffrey
- Wolfram Kahl
- Fredrik Lindblad
- Francesco Mazzoli
- Stefan Monnier
- Darin Morrison
- Guilhem Moulin
- Nicolas Pouillard

- Andrés Sicard-Ramírez
- Andrea Vezzosi
- and many more

EIGHT

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.