Agda Documentation

Release 2.5.2

Ulf Norell, Andreas Abel, Nils Anders Danielsson, Makoto Takeyar

Sep 07, 2017

Contents

CHAPTER 1

Overview

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

This is the manual for the Agda programming language, its type checking, compilation and editing system and related 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*.

CHAPTER 2

Getting Started

Installation

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
```

Fedora

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

```
yum install Agda
```

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

NixOS

Agda is part of the Nixpkgs collection that is used by http://nixos.org/nixos. To install Agda, type:

```
nix-env -iA haskellPackages.Agda
```

If you're just interested in the library, you can also install the library without the executable. Neither the emacs mode nor the Agda standard library are currently installed automatically, though.

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.

CHAPTER 3

Language Reference

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.)

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.

Examples

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

```
module Integer where

  abstract

     = Nat × Nat

    0 :
    0 = 0 , 0

    1 :
    1 = 1 , 0

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

    -_ :  →
    - (p , n) = (n , p)

    __ : (x y : ) → Set
    (p , n)  (p' , n') = (p + n')  (p' + n)

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

    inv :  x → (x + (- x))  0
    inv (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 `0`, `1`, `_+_`, and `-`, but only reason about equality with the provided lemma `inv`.

The following property `shape-of-0` of the integer zero exposes the representation of integers as pairs. As such, it is rejected by Agda: when checking its type signature, `proj₁ x` fails to type check since `x` is of abstract type . Remember that the abstract definition of does not unfold in type signatures, even when in an abstract block! However, if we make `shape-of-` private, unfolding of abstract definitions like is enabled, and we succeed:

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

  abstract
    private
      shape-of-0 :  (x : ) (is0 : x  0) → proj₁ x  proj₂ x
      shape-of-0 (p , n) refl rewrite +comm p 0 = refl
```

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

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.

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!

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
      xy : x  0
      xy = 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`.

Built-ins

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

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

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.

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.

Booleans

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

Built-in booleans are bound using the `BOOLEAN`, `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 only effect of binding the boolean type is that you can then use primitive functions returning booleans, such as built-in `NATEQUALS`.

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
```

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
```

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
  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 consisteny, the following laws apply:

- `primFloatEquality NaN NaN` returns `true`.
- `primFloatEquality NaN (primFloatNegate NaN)` returns `true`.
- `primFloatEquality 0.0 -0.0` returns `false`.

For numerical comparisons, use the `primFloatNumericalEquality` and `primFloatNumericalLess` primitives. These are implemented by the corresponding Haskell functions with the following behaviour and exceptions:

- `primFloatNumericalEquality 0.0 -0.0` returns `true`.
- `primFloatNumericalEquality NaN NaN` returns `false`.
- `primFloatNumericalLess NaN NaN` returns `false`.
- `primFloatNumericalLess (primFloatNegate NaN) (primFloatNegate NaN)` returns `false`.
- `primFloatNumericalLess NaN (primFloatNegate NaN)` returns `false`.
- `primFloatNumericalLess (primFloatNegate NaN) NaN` returns `false`.
- `primFloatNumericalLess` sorts `NaN` below everything but negative infinity.
- `primFloatNumericalLess -0.0 0.0` returns `false`.

Warning: Do not use `primFloatNumericalEquality` to establish decidable propositional equality. Doing so makes Agda inconsistent, see Issue #2169.

Lists

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

Built-in lists are bound using the `LIST`, `NIL` and `CONS` built-ins:

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

Even though Agda could easily tell which constructor is `NIL` and which is `CONS` you still have to bind them separately.

As with booleans, the only effect of binding the `LIST` built-in is to let you use primitive functions working with lists, such as `primStringToList` and `primStringFromList`.

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`.

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*.

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 __  #-}
{-# BUILTIN REFL      refl #-}
```

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

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
```

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  _⊔_   #-}
```

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  ω        #-}  --  ω        : Size
{-# BUILTIN SIZEMAX  _⊔ˢ_     #-}  --  _⊔ˢ_     : Size → Size → Size
```

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.

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 #-}
```

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.

Reflection

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

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

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.

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 and UHC *backends*) to `let x = e in seq x (f x)`.

For example, consider the following function:

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

At compile-time this will be exponential, due to call-by-name evaluation, and at run-time there is a space leak caused by unevaluated `a + a` thunks. Both problems can be fixed with `primForce`:

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

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

Coinduction

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 opossed 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)
```

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

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 : Co
inf = suc ( inf)
```

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

```
-- Infinite streams.

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

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

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

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

eat :  {A B} → SP A B → Stream A → Stream B
eat (get f)     (a  as) = eat (f a) ( as)
```

```
eat (put b sp) as       = b   eat ( sp) as

-- Composition of stream processors.

__ : {A B C} → SP B C → SP A B → SP A C
get f₁     put x sp₂ = f₁ x   sp₂
put x sp₁  sp₂       = put x ( ( sp₁  sp₂))
sp₁        get f₂    = get (λ x → sp₁  f₂ x)
```

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

```
data _'_ : Co → Co → Set where
  zero : zero ' zero
  suc  : {m n} → ∞ (m ' n) → suc ( m) ' suc ( n)
```

The recommended definition is the following one:

```
data __ : Co → Co → Set where
  zero : zero  zero
  suc  : {m n} → ∞ ( m   n) → suc m  suc n
```

The current implementation of coinductive types comes with some limitations.

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
```

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
```

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
```

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.
```

Data Types

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 : Set where
  c₁ : A₁
  ...
  c : A
```

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

```
(y₁ : B₁) → ... → (y : B) → D
```

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
```

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) ... (x : P) : (y₁ : Q₁) → ... → (y : Q) → Set  where
  c₁ : A₁
  ...
  c : A
```

where the types A_1, ..., A are function types of the form

```
(z₁ : B₁) → ... → (z : B) → D x₁ ... x t₁ ... t
```

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 : A, Agda checks that each *A* has the form

```
(y₁ : B₁) → ... → (y : B) → 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) → ... → (z : C) → 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*.

Foreign Function Interface

Haskell FFI

Note: This section currently only applies to the GHC backend.

The FFI is controlled by five pragmas:

- IMPORT
- COMPILED_TYPE
- COMPILED_DATA
- COMPILED
- COMPILED_EXPORT

All FFI bindings are only used when executing programs and do not influence the type checking phase.

The IMPORT pragma

```
{-# IMPORT HsModule #-}
```

The `IMPORT` pragma instructs the compiler to generate a Haskell import statement in the compiled code. The pragma above will generate the following Haskell code:

```
import qualified HsModule
```

`IMPORT` pragmas can appear anywhere in a file.

The COMPILED_TYPE pragma

```
postulate D : Set
{-# COMPILED_TYPE D HsType #-}
```

The `COMPILED_TYPE` pragma tells the compiler that the postulated Agda type `D` corresponds to the Haskell type `HsType`. This information is used when checking the types of `COMPILED` functions and constructors.

The COMPILED_DATA pragma

```
{-# COMPILED_DATA D HsD HsC1 .. HsCn #-}
```

The `COMPILED_DATA` pragma tells the compiler that the Agda datatype `D` corresponds to the Haskell datatype `HsD` and that its constructors should be compiled to the Haskell constructors `HsC1 .. HsCn`. The compiler checks that the Haskell constructors have the right types and that all constructors are covered.

Example:

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

{-# COMPILED_DATA List [] [] (:) #-}
```

Built-in Types

The GHC backend compiles certain Agda built-ins to special Haskell types. The mapping between Agda built-in types and Haskell types is as follows:

Agda Built-in	Haskell Type
`STRING`	`Data.Text.Text`
`CHAR`	`Char`
`INTEGER`	`Integer`
`BOOL`	`Boolean`
`FLOAT`	`Double`

> **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 to avoid making Agda inconsistent.

The COMPILED pragma

```
postulate f :  a b → (a → b) → List a → List b
{-# COMPILED f HsCode #-}
```

The `COMPILED` pragma tells the compiler to compile the postulated function `f` to the Haskell Code `HsCode`. `HsCode` can be an arbitrary Haskell term of the right type. This is checked by translating the given Agda type of `f` into a Haskell type (see *Translating Agda types to Haskell*) and checking that this matches the type of `HsCode`.

Example:

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

data Unit : Set where unit : Unit
{-# COMPILED_DATA Unit () () #-}

postulate
  IO       : Set → Set
  putStrLn : String → IO Unit

{-# COMPILED_TYPE IO IO #-}
{-# COMPILED putStrLn putStrLn #-}
```

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
  map : {A B : Set} → (A → B) → List A → List B

{-# COMPILED map (\_ _ → map) #-}
```

In this case compiled calls to map will still have `A` and `B` as arguments, so the compiled definition ignores its two first arguments and then calls the polymorphic Haskell `map` function.

Handling typeclass constraints

The problem here is that Agda's Haskell FFI doesn't understand Haskell's class system. If you look at this error message, GHC complains about a missing class constraint:

```
No instance for (Graphics.UI.Gtk.ObjectClass xA)
  arising from a use of Graphics.UI.Gtk.objectDestroy'
```

A work around to represent Haskell Classes in Agda is to use a Haskell datatype to represent the class constraint in a way Agda understands:

```
{-# LANGUAGE GADTs #-}
data MyObjectClass a = ObjectClass a => Witness
```

We also need to write a small wrapper for the `objectDestroy` function in Haskell:

```
myObjectDestroy :: MyObjectClass a -> Signal a (IO ())
myObjectDestroy Witness = objectDestroy
```

Notice that the class constraint disappeared from the Haskell type signature! The only missing part are the Agda FFI bindings:

```
postulate
  Window : Set
  Signal : Set → Set → Set
  MyObjectClass : Set → Set
  windowInstance : MyObjectClass Window
  myObjectDestroy :  {a} → MyObjectClass a → Signal a Unit
{-# COMPILED_TYPE Window Window #-}
{-# COMPILED_TYPE Signal Signal #-}
{-# COMPILED_TYPE MyObjectClass MyObjectClass #-}
{-# COMPILED windowInstance (Witness :: MyObjectClass Window) #-}
{-# COMPILED myObjectDestroy (\_ → myObjectDestroy) #-}
```

Then you should be able to call this as follows in Agda:

```
p : Signal Window Unit
p = myObjectDestroy windowInstance
```

This is somewhat similar to doing a dictionary-translation of the Haskell class system and generates quite a bit of boilerplate code.

The COMPILED_EXPORT pragma

New in version 2.3.4.

```
g :  {a : Set} → a → a
g x = x

{-# COMPILED_EXPORT g hsNameForG #-}
```

The `COMPILED_EXPORT` pragma tells the compiler that the Agda function `f` should be compiled to a Haskell function called `hsNameForF`. Without this pragma, functions are compiled to Haskell functions with unpredictable names and, as a result, cannot be invoked from Haskell. The type of `hsNameForF` will be the translated type of `f` (see *Translating Agda types to Haskell*). If f is defined in file A/B.agda, then `hsNameForF` should be imported from module `MAlonzo.Code.A.B`.

Example:

```
-- file IdAgda.agda
module IdAgda where

idAgda : {A : Set} → A → A
idAgda x = x

{-# COMPILED_EXPORT idAgda idAgda #-}
```

The compiled and exported function `idAgda` can then be imported and invoked from Haskell like this:

```
-- file UseIdAgda.hs
module UseIdAgda where
```

```
import MAlonzo.Code.IdAgda (idAgda)
-- idAgda :: () -> a -> a

idAgdaApplied :: a -> a
idAgdaApplied = idAgda ()
```

Translating Agda types to Haskell

Note: This section may contain outdated material!

When checking the type of COMPILED function f : A, the Agda type A is translated to a Haskell type TA and the Haskell code Ef is checked against this type. The core of the translation on kinds K[[M]], types T[[M]] and expressions E[[M]] is:

```
K[[ Set A ]] = *
K[[ x As ]] = undef
K[[ fn (x : A) B ]] = undef
K[[ Pi (x : A) B ]] = K[[ A ]] ->  K[[ B ]]
K[[ k As ]] =
  if COMPILED_TYPE k
  then *
  else undef

T[[ Set A ]] = Unit
T[[ x As ]] = x T[[ As ]]
T[[ fn (x : A) B ]] = undef
T[[ Pi (x : A) B ]] =
  if x in fv B
  then forall x . T[[ A ]] -> T[[ B ]]
  else T[[ A ]] -> T[[ B ]]
T[[ k As ]] =
  if COMPILED_TYPE k T
  then T T[[ As ]]
  else if COMPILED k E
  then Unit
  else undef

E[[ Set A ]] = unit
E[[ x As ]] = x E[[ As ]]
E[[ fn (x : A) B ]] = fn x . E[[ B ]]
E[[ Pi (x : A) B ]] = unit
E[[ k As ]] =
  if COMPILED k E
  then E E[[ As ]]
  else runtime-error
```

The T[[Pi (x : A) B]] case is worth mentioning. Since the compiler doesn't erase type arguments we can't translate (a : Set) → B to forall a. B — an argument of type Set will still be passed to a function of this type. Therefore, the translated type is forall a. () → B where the type argument is assumed to have unit type. This is safe since we will never actually look at the argument, and the compiler compiles types to ().

Function Definitions

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))
```

General form

The general form for defining a function is

```
f : (x₁ : A₁) → ... → (x : A) → B
f p₁ ... p = d
...
f q₁ ... q = e
```

where `f` is a new identifier, `p` and `q` are patterns of type `A`, and `d` and `e` are expressions.

The declaration above gives the identifier `f` the type $(x_1 : A_1) \to \ldots \to (x_1 : A_1) \to B$ 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*.

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 i₁ ... i` with a constructor `c : (x₁ : A₁) → ... → (x : A) → D j₁ ... j`, Agda will attempt to unify `i₁ ... i` with `j₁ ... j`. When the unification algorithm instantiates a variable `x` with value `t`, the corresponding argument of the function can be replaced by a dot pattern `.t`. Using a dot pattern is optional, but can help readability. The following are also legal definitions of `root`:

Since Agda 2.4.2.4:

```
root₁ : (n : Nat) → Square n → Nat
root₁ _ (sq m) = m
```

Since Agda 2.5.2:

```
root₂ : (n : Nat) → Square n → Nat
root₂ n (sq m) = m
```

In the case of `root₂`, `n` evaluates to `m * m` in the body of the function and is thus equivalent to

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

Absurd patterns

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

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

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

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

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

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

In general, when matching on an argument of type `D i₁ ... i` with an absurd pattern, Agda will attempt for each constructor `c : (x₁ : A₁) → ... → (x : A) → D j₁ ... j` of the datatype `D` to unify `i₁ ... i` with `j₁ ... j`. 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.

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 equation `max m zero = m` will not hold by definition, but must be proven instead. Since 2.5.1 you can have Agda warn you when a situation like this occurs by adding `{-# OPTIONS --exact-split #-}` at the top of your file.

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]`.

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)
```

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) → A → A`, we can define the identity function with implicit type argument as

```
id' = λ {A} → id A
```

Implicit arguments can also be referred to by name, so if we want to give the expression `e` explicitly for `y` without giving a value for `x` we can write

```
subst C {y = e} eq cx
```

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

```
z1 = λ {A} x → x
z2 = λ x → x
```

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

```
 : ( {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.

Metavariables

Unification

Instance Arguments

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.

Usage

Instance arguments are enclosed in double curly braces `{{ }}`, or their unicode equivalent   (`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 must have the form `{Γ} → C vs`, where `C` is 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). 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

A 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
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.

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.

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

Find candidates In the second stage we compute a set of *candidates*. *Let-bound* variables and top-level definitions in scope are candidates if they are defined in an `instance` block. Lambda-bound variables, i.e. variables bound in lambdas, function types, left-hand sides, or module parameters, are candidates if they are bound as instance arguments using `{{ }}`. Only candidates 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 `{Γ} → C vs`. First we extend the current context by Γ. Then, given a candidate `c : Δ → A` we generate fresh metavariables `αs : Δ` for the arguments of `c`, with ordinary metavariables for implicit arguments, and instance metavariables, solved by a recursive call to instance resolution, for explicit arguments and instance arguments.

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

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

If there are left-over instance problems at the end of type checking, the corresponding metavariables are printed in the Emacs status buffer together with their types and source location. The candidates that gave rise to potential solutions can be printed with the *show constraints command* (`C-c C-=`).

Irrelevance

Note: This is a stub.

Lambda Abstraction

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 }
```

Local Definitions: let and where

There are two ways of declaring local definitions in Agda:

- let-expressions
- where-blocks

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 f : A₁ → ... → A → A
    f x₁ ... x = e
in  e′
```

After type-checking, the meaning of this is simply the substitution `e′[f := λ x₁ ...  x → e]`. Since Agda substitutes away let-bindings, they do not show up in terms Agda prints, nor in the goal display in interactive mode.

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 → ... → A → A
  f p₁₁ ... p₁= e₁
  ...
  ...
  f p₁ ... p= e
```

Here, the `p` are patterns of the corresponding types and `e` 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.

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 []
```

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

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

Same definition using where

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

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

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.

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` `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 `Set₄₂` 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 starts 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: `" \"\"\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.

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 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
```

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 preable 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}
```

Literal Overloading

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:

```
{-# 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 Fin n can then be defined as follows:

```
data IsTrue : Bool → Set where
  itis : IsTrue true

instance
  indeed : IsTrue true
  indeed = itis
```

```
_<?_ : Nat → Nat → Bool
zero <? zero = false
zero <? suc y = true
suc x <? zero = false
suc x <? suc y = x <? y

natToFin : ∀ {n} (m : Nat) {{_ : IsTrue (m <? n)}} → Fin n
natToFin {zero} zero {{()}}
natToFin {zero} (suc m) {{()}}
natToFin {suc n} zero {{itis}} = zero
natToFin {suc n} (suc m) {{t}} = suc (natToFin m)

instance
  NumFin : ∀ {n} → Number (Fin n)
  Number.Constraint (NumFin {n}) k = IsTrue (k <? n)
  Number.fromNat    NumFin       k = natToFin k
```

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 #-}
```

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 #-}
```

Other types

Currently only integer and string literals can be overloaded.

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
```

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
```

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
```

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).

Module System

Module application

Anonymous modules

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
```

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.

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.

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.

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]
```

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.

Datatype 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.

Record update syntax

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.

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.

(Instance arguments are explained below.) 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 = _").

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
```

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.

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   : Set where
  zero :
  suc  :   →

pattern z    = zero
pattern ss x = suc (suc x)
```

```
f :   →
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.

Positivity Checking

Note: This is a stub.

NO_POSITIVITY_CHECK pragma

The pragma switch off the positivity checker for data/record definitions and mutual blocks.

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
```

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.

Postulates

Note: This is a stub.

Pragmas

Note: This is a stub.

- *NO_POSITIVITY_CHECK*
- *POLARITY*

Record Types

Note: This is a stub.

Record declarations

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
```

Record types behaves much like single constructor datatypes (but see *eta-expansion* below), and you can name the constructor using the `constructor` keyword

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

p45 : Pair Nat Nat
p45 = 4 , 5
```

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

Record modules

Along with a new type, a record declaration also defines a module 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
```

It 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 : Set where
  constructor mkR
  field f₁ : A₁
  D₁
  field f₂ : A₂

mkR :  {Γ} (f₁ : A₁) (let D₁) (f₂ : A₂) → R Γ
```

Eta-expansion

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 → Bool
x 3 = (x == fromNat 3) || (x < fromNat 3)
```

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

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

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

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

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

_3 : {A : Set} {{OrdA : Ord A}} {{NumA : Num A}} → A → Bool
x 3 = (x == fromNat 3) || (x < fromNat 3)
```

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

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

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

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

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

Reflection

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
  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 AGDALITFLOAT  float   #-}
{-# BUILTIN AGDALITCHAR   char    #-}
{-# BUILTIN AGDALITSTRING string  #-}
{-# BUILTIN AGDALITQNAME  name    #-}
{-# BUILTIN AGDALITMETA   meta    #-}
```

Arguments

Arguments can be (visible), {hidden}, or {{instance}}:

```
data Visibility : Set where
  visible hidden instance : Visibility

{-# BUILTIN HIDING   Visibility #-}
{-# BUILTIN VISIBLE  visible    #-}
{-# BUILTIN HIDDEN   hidden     #-}
{-# BUILTIN INSTANCE instance  #-}
```

Arguments can be relevant or irrelevant:

```
data Relevance : Set where
  relevant irrelevant : Relevance

{-# BUILTIN RELEVANCE  Relevance  #-}
{-# BUILTIN RELEVANT   relevant   #-}
{-# BUILTIN IRRELEVANT irrelevant #-}
```

Visibility and relevance characterise the behaviour of an argument:

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

data Arg (A : Set) : Set where
  arg : (i : ArgInfo) (x : A) → Arg A

{-# BUILTIN ARGINFO    ArgInfo  #-}
{-# BUILTIN ARGARGINFO arg-info #-}
{-# BUILTIN ARG        Arg      #-}
{-# BUILTIN ARGARG     arg      #-}
```

Patterns

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

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

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

Name abstraction

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

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

Terms

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

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

data Term where
  var       : (x : Nat) (args : List (Arg Term)) → Term
  con       : (c : Name) (args : List (Arg Term)) → Term
```

```
  def       : (f : Name) (args : List (Arg Term)) → Term
  lam       : (v : Visibility) (t : Abs Term) → Term
  pat-lam   : (cs : List Clause) (args : List (Arg Term)) → Term
  pi        : (a : Arg Type) (b : Abs Type) → Term
  agda-sort : (s : Sort) → Term
  lit       : (l : Literal) → Term
  meta      : (x : Meta) → List (Arg Term) → Term
  unknown   : Term -- Treated as '_' when unquoting.

data Sort where
  set     : (t : Term) → Sort -- A Set of a given (possibly neutral) level.
  lit     : (n : Nat) → Sort  -- A Set of a given concrete level.
  unknown : Sort

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

{-# BUILTIN AGDASORT    Sort   #-}
{-# BUILTIN AGDATERM    Term   #-}
{-# BUILTIN AGDACLAUSE  Clause #-}

{-# BUILTIN AGDATERMVAR         var       #-}
{-# BUILTIN AGDATERMCON         con       #-}
{-# BUILTIN AGDATERMDEF         def       #-}
{-# BUILTIN AGDATERMMETA        meta      #-}
{-# BUILTIN AGDATERMLAM         lam       #-}
{-# BUILTIN AGDATERMEXTLAM      pat-lam   #-}
{-# BUILTIN AGDATERMPI          pi        #-}
{-# BUILTIN AGDATERMSORT        agda-sort #-}
{-# BUILTIN AGDATERMLIT         lit       #-}
{-# BUILTIN AGDATERMUNSUPPORTED unknown   #-}

{-# BUILTIN AGDASORTSET         set     #-}
{-# BUILTIN AGDASORTLIT         lit     #-}
{-# BUILTIN AGDASORTUNSUPPORTED unknown #-}

{-# BUILTIN AGDACLAUSECLAUSE clause        #-}
{-# BUILTIN AGDACLAUSEABSURD absurd-clause #-}
```

Absurd lambdas λ () are quoted to extended lambdas with an absurd clause.

The built-in constructors AGDATERMUNSUPPORTED and AGDASORTUNSUPPORTED are translated to meta variables when unquoting.

Declarations

There is a built-in type AGDADEFINITION representing definitions. Values of this type is returned by the AGDATCMGETDEFINITION built-in *described below*.

```
data Definition : Set where
  function    : (cs : List Clause) → Definition
  data-type   : (pars : Nat) (cs : List Name) → Definition   -- parameters and␣
↪constructors
  record-type : (c : Name) → Definition                      -- name of data/record␣
↪type
  data-cons   : (d : Name) → Definition                      -- name of constructor
```

```
  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.
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'.
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 ⊤

-- 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

{-# 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 AGDATCMDEFINEFUN          defineFun          #-}
{-# BUILTIN AGDATCMGETTYPE            getType            #-}
{-# BUILTIN AGDATCMGETDEFINITION      getDefinition      #-}
{-# BUILTIN AGDATCMCOMMIT             commitTC           #-}
{-# BUILTIN AGDATCMISMACRO            isMacro            #-}
{-# BUILTIN AGDATCMWITHNORMALISATION  withNormalisation  #-}
```

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 `t₁ → t₂ → .. → Term → TC ⊤` that are defined in a `macro` block. The last argument is supplied by the type checker and will be the representation of a metavariable that should be instantiated with the result of the macro.

Macro application is guided by the type of the macro, where `Term` and `Name` arguments are quoted before passed to the macro. Arguments of any other type are preserved as-is.

For example, the macro application `f u v w` where `f : Term → Name → Bool → Term → TC ⊤` desugars into:

```
unquote (f (quoteTerm u) (quote v) w)
```

where `quoteTerm u` takes a `u` of arbitrary type and returns its representation in the `Term` data type, and `unquote m` runs a computation in the `TC` monad. Specifically, when checking `unquote m : A` for some type `A` the type checker proceeds as follows:

- Check `m : Term → TC ⊤`.
- Create a fresh metavariable `hole : A`.

- Let `qhole : Term` be the quoted representation of `hole`.
- Execute `m qhole`.
- Return (the now hopefully instantiated) `hole`.

Reflected macro calls are constructed using the `def` constructor, so given a macro `g : Term → TC` 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 .. x = m
unquoteDef  x₁ .. x = 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` whereas `unquoteDef` expects the `x` to be already declared. In other words, `unquoteDecl` brings the `x` into scope, but `unquoteDef` requires them to already be in scope.

In `m` the `x` stand for the names of the functions being defined (i.e. `x : 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.

Rewriting

Note: This is a stub.

Safe Agda

Note: This is a stub.

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*.

Example: Finite languages

See *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 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 `w` $= \alpha\beta$, with α `a` and β `b`.

We have to consider two cases:

1. ϵ `a`. Then, either: * $\alpha = \epsilon$, and `w` $= \beta =$ `x` $\cdot\ \beta'$, where β' `δ b x`. * $\alpha =$ `x`α', with α' `δ a x`.
2. ϵ `a`. Then, only the second case above is possible: * $\alpha =$ `x`α', with α' `δ a x`.

```
_·_ : {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 ω.

```
__ : {A} → List A → Lang ω A → Bool
[]   a = ν a
(x  w)  a = w  δ a x
```

Intuitively, ω 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 containing a single word of length 1:

```
_ : {i} → Bool → Lang i Bool
ν  _      =  false

δ  false  false = ϵ
δ  true  true = ϵ
δ  false  true =
δ  true  false =
```

Now we can define the bip-bop language, consisting of strings of even length starting with "true", where each "true" is followed by "false", and viceversa.

```
bip-bop = ( true  ·  false )*
```

We can now test 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
```

References

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

Telescopes

Note: This is a stub.

Termination Checking

Note: This is a stub.

With-functions

Universe Levels

Note: This is a stub.

With-Abstraction

With abstraction was first introduced by Conor McBride *[McBride2004]* and lets you pattern match on the result of an intermediate computation by effectively adding an extra argument to the left-hand side of your function.

Usage

In the simplest case the `with` construct can be used just to discriminate on the result of an intermediate computation. For instance

```
filter : {A : Set} → (A → Bool) → List A → List A
filter p [] = []
filter p (x  xs) with p x
filter p (x  xs)    | true  = x  filter p xs
filter p (x  xs)    | false = filter p xs
```

The clause containing the with-abstraction has no right-hand side. Instead it is followed by a number of clauses with an extra argument on the left, separated from the original arguments by a vertical bar (|).

When the original arguments are the same in the new clauses you can use the ... syntax:

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

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

- If you want to do further pattern matching on the original arguments.
- When the pattern matching on the intermediate result refines some of the other arguments (see *Dot patterns*).
- To disambiguate the clauses of nested with abstractions (see *Nested with-abstractions* below).

Generalisation

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

```
postulate P :  {A} → List A → Set
postulate p-nil : P []
postulate Q : Set
postulate q-nil : Q
```

```
proof : {A : Set} (p : A → Bool) (xs : List A) → P (filter p xs)
proof p []       = {! P [] !}
proof p (x   xs) = {! P (filter p xs | p x) !}
```

In the cons case we have to prove that `P` holds for `filter p xs | p x`. This is the syntax for a stuck with-abstraction–`filter` cannot reduce since we don't know the value of `p x`. This syntax is used for printing, but is not accepted as valid Agda code. Now if we with-abstract over `p x`, but don't pattern match on the result we get:

```
proof : {A : Set} (p : A → Bool) (xs : List A) → P (filter p xs)
proof p [] = p-nil
proof p (x   xs) with p x
...                 | r   = {! P (filter p xs | r) !}
```

Here the `p x` in the goal type has been replaced by the variable `r` introduced for the result of `p x`. If we pattern match on `r` the with-clauses can reduce, giving us:

```
proof : {A : Set} (p : A → Bool) (xs : List A) → P (filter p xs)
proof p [] = p-nil
proof p (x   xs) with p x
...                 | true  = {! P (x   filter p xs) !}
...                 | false = {! P (filter p xs) !}
```

Both the goal type and the types of the other arguments are generalised, so it works just as well if we have an argument whose type contains `filter p xs`.

```
proof₂ : {A : Set} (p : A → Bool) (xs : List A) → P (filter p xs) → Q
proof₂ p [] _ = q-nil
proof₂ p (x  xs) H with p x
...                    | true  = {! H : P (filter p xs) !}
...                    | false = {! H : P (x  filter p xs) !}
```

The generalisation is not limited to scrutinees in other with-abstractions. All occurrences of the term in the goal type and argument types will be generalised.

Note that this generalisation is not always type correct and may result in a (sometimes cryptic) type error. See *Ill-typed with-abstractions* below for more details.

Nested with-abstractions

With-abstractions can be nested arbitrarily. The only thing to keep in mind in this case is that the `...` syntax applies to the closest with-abstraction. For example, suppose you want to use `...` in the definition below.

```
compare : Nat → Nat → Comparison
compare x y with x < y
compare x y    | false with y < x
compare x y    | false     | false = equal
compare x y    | false     | true  = greater
compare x y    | true = less
```

You might be tempted to replace `compare x y` with `...` in all the with-clauses as follows.

```
compare : Nat → Nat → Comparison
compare x y with x < y
...            | false with y < x
...                        | false = equal
...                        | true  = greater
...            | true = less     -- WRONG
```

This, however, would be wrong. In the last clause the `...` is interpreted as belonging to the inner with-abstraction (the whitespace is not taken into account) and thus expands to `compare x y | false | true`. In this case you have to spell out the left-hand side and write

```
compare : Nat → Nat → Comparison
compare x y with x < y
...            | false with y < x
...                        | false = equal
...                        | true  = greater
compare x y    | true = less
```

Simultaneous abstraction

You can abstract over multiple terms in a single with abstraction. To do this you separate the terms with vertical bars (`|`).

```
compare : Nat → Nat → Comparison
compare x y with x < y | y < x
...            | true  | _     = less
...            | _     | true  = greater
...            | false | false = equal
```

In this example the order of abstracted terms does not matter, but in general it does. Specifically, the types of later terms are generalised over the values of earlier terms. For instance

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

```
thm : (a b : Nat) → P (a + b) → P (b + a)
thm a b t with a + b | plus-commute a b
thm a b t     | ab    | eq = {! t : P ab, eq : ab  b + a !}
```

Note that both the type of `t` and the type of the result `eq` of `plus-commute a b` have been generalised over `a + b`. If the terms in the with-abstraction were flipped around, this would not be the case. If we now pattern match on `eq` we get

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

and can thus fill the hole with `t`. In effect we used the commutativity proof to rewrite `a + b` to `b + a` in the type of `t`. This is such a useful thing to do that there is special syntax for it. See *Rewrite* below. A limitation of generalisation is that only occurrences of the term that are visible at the time of the abstraction are generalised over, but more instances of the term may appear once you start filling in the right-hand side or do further matching on the left. For instance, consider the following contrived example where we need to match on the value of `f n` for the type of `q` to reduce, but we then want to apply `q` to a lemma that talks about `f n`:

```
postulate
  R     : Set
  P     : Nat → Set
  f     : Nat → Nat
  lemma :  n → P (f n) → R

Q : Nat → Set
Q zero    =
Q (suc n) = P (suc n)
```

```
proof : (n : Nat) → Q (f n) → R
proof n q with f n
proof n ()    | zero
proof n q     | suc fn = {! q : P (suc fn) !}
```

Once we have generalised over `f n` we can no longer apply the lemma, which needs an argument of type `P (f n)`. To solve this problem we can add the lemma to the with-abstraction:

```
proof : (n : Nat) → Q (f n) → R
proof n q with f n    | lemma n
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.

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

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

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

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

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

```
postulate
  R     : Set
  P     : Nat → Set
  f     : Nat → Nat
  lemma :  n → P (f n) → R

Q : Nat → Set
Q zero    =
Q (suc n) = P (suc n)

proof : (n : Nat) → Q (f n) → R
proof n q with f n    | inspect f n
proof n ()    | zero   | _
proof n q     | suc fn | ingraph eq = {! q : P (suc fn), eq : f n  suc fn !}
```

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

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

Alternatives to with-abstraction

Although with-abstraction is very powerful there are cases where you cannot or don't want to use it. For instance, you cannot use with-abstraction if you are inside an expression in a right-hand side. In that case there are a couple of alternatives.

Pattern lambdas

Agda does not have a primitive `case` construct, but one can be emulated using *pattern matching lambdas*. First you define a function `case_of_` as follows:

```
case_of_ :  {a b} {A : Set a} {B : Set b} → A → (A → B) → B
case x of f = f x
```

You can then use this function with a pattern matching lambda as the second argument to get a Haskell-style case expression:

```
filter : {A : Set} → (A → Bool) → List A → List A
filter p [] = []
filter p (x  xs) =
  case p x of
  λ { true  → x  filter p xs
    ; false → filter p xs
    }
```

This version of `case_of_` only works for non-dependent functions. For dependent functions the target type will in most cases not be inferrable, but you can use a variant with an explicit `B` for this case:

```
case_return_of_ :  {a b} {A : Set a} (x : A) (B : A → Set b) → ( x → B x) → B x
case x return B of f = f x
```

The dependent version will let you generalise over the scrutinee, just like a with-abstraction, but you have to do it manually. Two things that it will not let you do is

- further pattern matching on arguments on the left-hand side, and
- refine arguments on the left by the patterns in the case expression. For instance if you matched on a `Vec A n` the `n` would be refined by the nil and cons patterns.

Helper functions

Internally with-abstractions are translated to auxiliary functions (see *Technical details* below) and you can always[1] write these functions manually. The downside is that the type signature for the helper function needs to be written out explicitly, but fortunately the *Emacs Mode* has a command (`C-c C-h`) to generate it using the same algorithm that generates the type of a with-function.

Performance considerations

The *generalisation step* of a with-abstraction needs to normalise the scrutinee and the goal and argument types to make sure that all instances of the scrutinee are generalised. The generalisation also needs to be type checked to make sure that it's not *ill-typed*. This makes it expensive to type check a with-abstraction if

- the normalisation is expensive,
- the normalised form of the goal and argument types are big, making finding the instances of the scrutinee expensive,
- type checking the generalisation is expensive, because the types are big, or because checking them involves heavy computation.

In these cases it is worth looking at the *alternatives to with-abstraction* from above.

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

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

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

where $\Delta \vdash ps : \Gamma$ (i.e. Δ types the variables bound in ps), we

- Infer the types of the scrutinees $t_1 : A_1, \ldots, t_m : A_m$.
- Partition the context Δ into Δ_1 and Δ_2 such that Δ_1 is the smallest context where $\Delta_1 \vdash t_i : A_i$ for all i, i.e., where the scrutinees are well-typed. Note that the partitioning is not required to be a split, $\Delta_1\Delta_2$ can be a (well-formed) reordering of Δ.
- Generalise over the t_i s, by computing

  $$C = (w_1 : A_1)(w_1 : A'_2) \ldots (w_m : A'_m) \to \Delta'_2 \to B'$$

  such that the normal form of C does not contain any t_i and

  $$
  \begin{array}{r}
  A'_i[w_1 := t_1 \ldots w_{i-1} := t_{i-1}] \simeq A_i \\
  (\Delta'_2 \to B')[w_1 := t_1 \ldots w_m := t_m] \simeq \Delta_2 \to B
  \end{array}
  $$

  where $X \simeq Y$ is equality of the normal forms of X and Y. The type of the auxiliary function is then $\Delta_1 \to C$.
- Check that $\Delta_1 \to C$ is type correct, which is not guaranteed (see *below*).
- Add a function f_{aux}, mutually recursive with f, with the definition

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

  where $qs_i = q_{i1} \ldots q_{im}$, and $ps_{1i} : \Delta_1$ and $ps_{2i} : \Delta_2$ are the patterns from ps_i corresponding to the variables of ps. Note that due to the possible reordering of the partitioning of Δ into Δ_1 and Δ_2, the patterns ps_{1i} and ps_{2i} can be in a different order from how they appear ps_i.
- Replace the with-abstraction by a call to f_{aux} resulting in the final definition

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

  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.

Without K

Note: This is a stub.

CHAPTER 4

Tools

Automatic Proof Search (Auto)

Note: This is a stub.

Command-line options

Note: This is a stub.

Compilers

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

{-# IMPORT Data.Text.IO #-}

data Unit : Set where
  unit : Unit

{-# COMPILED_DATA Unit () () #-}

postulate
  String : Set

{-# BUILTIN STRING String #-}

postulate
  IO : Set → Set

{-# BUILTIN IO IO #-}
{-# COMPILED_TYPE IO IO #-}

postulate
  putStr : String → IO Unit

{-# COMPILED putStr Data.Text.IO.putStr #-}

main : IO Unit
main = putStr "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.

UHC Backend

New in version 2.5.1.

Note: The Agda Standard Library has been updated to support this new backend. This backend is currently experimental.

The Agda UHC backend targets the Core language of the Utrecht Haskell Compiler (UHC). This backend works on the Mac and Linux platforms and requires GHC >= 7.10.

The backend is disabled by default, as it will pull in some large dependencies. To enable the backend, use the "uhc" cabal flag when installing Agda:

```
cabal install Agda -fuhc
```

The backend also requires UHC to be installed. UHC is not available on Hackage and needs to be installed manually. This version of Agda has been tested with UHC 1.1.9.4, using other UHC versions may cause problems. To install UHC, the following commands can be used:

```
cabal install uhc-util-0.1.6.6 uulib-0.9.22
wget https://github.com/UU-ComputerScience/uhc/archive/v1.1.9.4.tar.gz
tar -xf v1.1.9.4.tar.gz
cd uhc-1.1.9.4/EHC
./configure
make
make install
```

The Agda UHC compiler can be invoked from the command line using the flag `--uhc`:

```
agda --uhc [--compile-dir=<DIR>]
    [--uhc-bin=<UHC>] [--uhc-dont-call-uhc] <FILE>.agda
```

Limitations

The UHC backend currently does not support Unicode strings. See issue 1857 for details.

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
```

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.

Emacs Mode

Note: This is a stub.

Keybindings

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-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-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)

Unicode input

The Agda emacs mode comes with an input method for for easily writing Unicode characters. Most Unicode character can be input by typing their corresponding TeX or LaTeX commands, eg. typing `\lambda` will input λ. To see all characters you can input using the Agda input method see `M-x describe-input-method Agda`.

If you know the Unicode name of a character you can input it using `M-x ucs-insert` or `C-x 8 RET`. Example: `C-x 8 RET not SPACE a SPACE sub TAB RET` to insert “NOT A SUBSET OF” .

To find out how to input a specific character, eg from the standard library, position the cursor over the character and use `M-x describe-char` or `C-u C-x =`.

The Agda input method can be customised via `M-x customize-group agda-input`.

Common 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`	
03bb	λ	`\Gl`	`\lambda`
041f			
0432			
0435			
0438			
043c			
0440			
0442			
1d62		`\_i`	
2032		`\'1`	`\prime`
207f		`\^n`	
2081	$_1$	`\_1`	
2082	$_2$	`\_2`	
2083	$_3$	`\_3`	
2084	$_4$	`\_4`	
2096		`\_k`	
2098		`\_m`	
2099		`\_n`	

Hex code	Character	Short key-binding	TeX command
2113	(PDF TODO)		`\ell`

Hex code	Character	Short key-binding	TeX command
2115		`\bN`	`\Bbb{N}`
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`	
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)	`\}}`	

Hex code	Character	Short key-binding	TeX command
2c7c		`\_j`	

Generating HTML

Note: This is a stub.

Generating LaTeX

Note: This is a stub.

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*.

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!

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.

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.

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).

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.

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. It has format

```
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).

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`.

Upgrading

If you are upgrading from a pre 2.5 version of Agda, be aware that you may have remnants of the previous library management system in your preferences. In particular, if you get warnings about `agda2-include-dirs`, you will need to find where this is defined. This may be buried deep in `.el` files, whose location is both operating system and emacs version dependant.

CHAPTER 5

Contribute

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

Documentation

Note: This is a stub.

Documentation is written in reStructuredText format.

Code examples

You can include code examples in your documentation.

If your give the documentation file the extension `.lagda.rst`, code examples in the can be checked as part of the continuous integration. This way, they will be guaranteed to always work with the latest version of Agda.

Tip: If you edit documentation files in Emacs, you can use Agda's interactive mode to write your code examples. Use `M-x agda2-mode` to switch to Agda mode, and `M-x rst-mode` to switch back to rST mode.

Syntax

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.

Tip: Remember to always leave a blank like after the `::`. Otherwise, the code will be checked by Agda, but it will appear variable-width paragraph text in the documentation.

Visible, unchecked code examples

This is code that the reader will see, but will not be checked by Agda. It is useful for examples of incorrect code, program output, or code in languages different from Agda.

```
.. code-block:: agda

   -- This is not a valid definition

   ω :  a → a
   ω x = x


.. code-block:: haskell

   -- This is haskell code

   data Bool = True | False
```

Result:

```
-- This is not a valid definition

ω :  a → a
ω x = x
```

```
-- This is haskell code

data Bool = True | False
```

Invisible, checked code examples

This is code that is not shown to the reader, but which is used to typecheck the code that is actually displayed.

This might be definitions that are well known enough that do not need to be shown again.

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

::

  add : Nat → Nat → Nat
  add zero y = y
  add (suc x) y = suc (add x y)
```

Result:

```
add : Nat → Nat → Nat
add zero y = y
add (suc x) y = suc (add x y)
```

File structure

Documentation literate files (*.lagda.**) are type-checked as whole Agda files, as if all literate text was replaced by whitespace. Thus, **indentation** is interpreted globally.

Namespacing

In the documentation, files are typechecked starting from the *doc/user-manual/* root. For example, the file *doc/user-manual/language/data-types.lagda.rst* should start with a hidden code block declaring the name of the module as *language.data-types*:

```
..
  ::
  module language.data-types where
```

Scoping

Sometimes you will want to use the same name in different places in the same documentation file. You can do this by using hidden module declarations to isolate the definitions from the rest of the file.

```
..
  ::
  module scoped-1 where

::

    foo : Nat
    foo = 42

..
  ::
  module scoped-2 where

  ::
    foo : Nat
    foo = 66
```

Result:

```
foo : Nat
foo = 42
```

CHAPTER 6

The Agda License

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.

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 IMPLIED 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 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.

CHAPTER 7

Indices and tables

- genindex
- search

Bibliography

[McBride2004] C. McBride and J. McKinna. **The view from the left**. Journal of Functional Programming, 2004. http://strictlypositive.org/vfl.pdf.