Agda Documentation

Release 2.5

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

December 16, 2015

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

Note: This is a stub.

CHAPTER 3

Language Reference

3.1 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. The primitive functions associated with each built-in type are given below.

3.1.1 Natural numbers

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 _<_ #-}

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

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

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)  divAux 0 m n m
mod n (suc m)  modAux 0 m n m
```

3.1.2 Integers

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

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

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

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

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

3.1.3 Floats

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 Haskell Doubles. 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
  primFloatDiv      : Float → Float → Float
  primFloatEquality : Float → Float → Bool
  primFloatLess     : Float → Float → Bool
  primRound         : Float → Int
  primFloor         : Float → Int
  primCeiling       : Float → Int
  primExp           : Float → Float
  primLog           : Float → Float
  primSin           : Float → Float
  primShowFloat     : Float → String
```

These are implemented by the corresponding Haskell functions with a few exceptions:

- `primFloatEquality NaN NaN` returns `true`.
- `primFloatLess` sorts `NaN` below everything but negative infinity.
- `primShowFloat` returns `"0.0"` on negative zero.

This is to allow decidable equality and proof carrying comparisons on floating point numbers.

3.1.4 Booleans

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`.

3.1.5 Lists

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 __  #-}
```

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`.

3.1.6 Characters

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

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

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

```
primitive
  primIsLower    : Char → Bool
  primIsDigit    : Char → Bool
  primIsAlpha    : Char → Bool
  primIsSpace    : Char → Bool
  primIsAscii    : Char → Bool
```

```
  primIsLatin1    : Char → Bool
  primIsPrint     : Char → Bool
  primIsHexDigit  : Char → Bool
  primToUpper     : Char → Char
  primToLower     : Char → Char
  primCharToNat   : Char → Nat
  primNatToChar   : Nat → Char
  primShowChar    : Char → String
```

These functions are implemented by the corresponding Haskell functions from Data.Char (`ord` and `chr` for `primCharToNat` and `primNatToChar`). To make `primNatToChar` total `chr` is applied to the natural number modulo `0x110000`.

3.1.7 Strings

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*):

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

String literals can be *overloaded*.

3.1.8 Equality

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

```
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

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

```
primitive
  primTrustMe :  {a} {A : Set a} {x y : A} → x  y
```

As can be seen from the type, `primTrustMe` must be used with the utmost care to avoid inconsistencies. What makes it different from a postulate is that if `x` and `y` are actually definitionally equal, `primTrustMe` reduces to `refl`. One use of `primTrustMe` is to lift the primitive boolean equality on built-in types like *String* to something that returns a proof object:

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

With this definition `eqString "foo" "foo"` computes to `just refl`. Another use case is to erase computationally expensive equality proofs and replace them by `primTrustMe`:

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

3.1.9 Universe levels

Universe levels are also declared using `BUILTIN` pragmas. This is done in the auto-imported `Agda.Primitive` module, however, so it need never be done by a library. For reference these are the bindings:

```
    postulate
      Level : Set
      lzero : Level
      lsuc  : Level → Level
      __    : Level → Level → Level
    {-# BUILTIN LEVEL     Level #-}
    {-# BUILTIN LEVELZERO lzero #-}
    {-# BUILTIN LEVELSUC  lsuc  #-}
    {-# BUILTIN LEVELMAX  __    #-}
```

3.1.10 Sized types

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  __     #-}  --  __     : Size → Size → Size
```

3.1.11 Coinduction

The following built-ins are used for coinductive definitions:

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

See *Coinduction* for more information.

3.1.12 IO

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

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

3.1.13 Reflection

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

3.1.14 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 __ #-}
```

3.1.15 Strictness

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
```

3.2 Coinduction

Note: This is a stub.

3.3 Copatterns

Note: This is a stub.

3.4 Core language

Note: This is a stub

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

3.5 Data Types

3.5.1 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
```

Then the unit set:

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

The True set represents the trivially true proposition.:

```
data False : Set where
```

The False set has no constructor and hence no elements. It represent the trivially false proposition.

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
```

3.5.2 General form

The general form of the definition of a simple data type D is the following:

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

The name D of the data type and the names c1, ..., c of the constructors must be new wrt the current signature and context.

Agda checks that A1, ..., A : Set wrt the current signature and context. Moreover, 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.

3.5.3 Strict positivity

The condition that D must not occur in any C can be also phrased as D must only occur **strictly positively** in B.

Agda can check this positivity condition.

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 one can write a non-terminating function using them.

If the positivity check is disabled so that the above declaration of Bad is allowed, it is possible to construct a term of the empty type.:

```
{-# OPTIONS --no-positivity-check #-}
data  : Set where

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

incon :
incon = loop (bad (λ b → b))
    where
        loop : (b : Bad) →
        loop (bad f) = loop (f (bad f))
```

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

3.6 Foreign Function Interface

Note: This is a stub.

3.6.1 Haskell FFI

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

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

3.7 Function Definitions

Note: This is a stub.

3.7.1 Pattern matching

Dot patterns

Absurd patterns

3.8 Function Types

Note: This is a stub.

3.9 Implicit Arguments

Note: This is a stub.

3.9.1 Metavariables

3.9.2 Unification

3.10 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.

3.10.1 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
```

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[1], 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

[1] At the moment any variable in the context is considered for instance resolution, but this may change in the future. See issue #1716 for some discussion.

```
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.

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}}`.

Warning: At the moment there is no termination check on instances, so it is possible to make instance resolution loop by defining non-sensical instances like `loop : {a} {A : Set a} {{_ : Eq A}} → Eq A`.

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

badfive : Fin 5
badfive = mkFin 5 -- Error: No instance of type 1  0 was found in scope.
```

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.

Currently you cannot declare record fields to be instances, but this will likely be possible in the future. See issue #1273.

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)

ex : {A : Set} (x y : A) (xs : List A) {{i : x  xs}} → x  y  y  xs
ex 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 `x1 , y1`, the types of the second components `y` and `y1` can be completely different and not admit an equality test. Only when `x` and `x1` are *really equal* can we hope to compare `y` and `y1`. Having the equality function return a proof means that we are guaranteed that when `x` and `x1` compare equal, they really are equal, and comparing `y` and `y1` makes sense.

An `Eq` instance for Σ can be defined as follows:

```
instance
  eqΣ :  {a b} {A : Set a} {B : A → Set b} {{_ : Eq A}} {{_ :  {x} → Eq (B x)}} → Eq (Σ A B)
  _==_ {{eqΣ}} (x , y) (x1 , y1) with x == x1
  _==_ {{eqΣ}} (x , y) (x1 , y1)    | nothing = nothing
  _==_ {{eqΣ}} (x , y) (.x , y1)    | just refl with y == y1
  _==_ {{eqΣ}} (x , y) (.x , y1)    | just refl    | nothing   = nothing
  _==_ {{eqΣ}} (x , y) (.x , .y)    | just refl    | just refl = just refl
```

Note that the instance argument for `B` states that there should be an `Eq` instance for `B x`, for any `x : A`. The argument `x` must be implicit, indicating that it needs to be inferred by unification whenever the `B` instance is used. See *instance resolution* below for more details.

3.10.2 Instance resolution

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

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

Next we check that the goal type has the right shape to be solved by instance resolution. It should be of the form `{Γ} → C vs`, where the target type `C` is a variable from the context or the name of a data or record type, and `{Γ}` denotes a telescope of implicit arguments. If this is not the case instance resolution fails with an error message[2].

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`.

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

Find candidates In the second stage we compute a set of *candidates*. These are the variables in the context (both lambda-bound and *let-bound*)[1] and the names of top-level instances that compute something of type `C us`, where `C` is the target type computed in the previous stage. If `C` is a variable from the context there will be no top-level instances.

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-=`).

3.11 Irrelevance

Note: This is a stub.

3.12 Lambda Abstraction

Note: This is a stub.

3.12.1 Pattern matching lambda

3.13 Let Expressions

Note: This is a stub.

3.14 Lexical Structure

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

3.14.1 Tokens

Keywords and special symbols

Most non-whitespace unicode can be used as part of an Agda name, but there are two kinds of exceptions:

special symbols Characters with special meaning that cannot appear at all in a name. These are `.;{}()@"`.

keywords Reserved words that cannot appear as a *name part*, but can appear in a name together with other characters.

```
= | -> → : ? \ λ  .. ...
abstract codata coinductive constructor data eta-equality field forall hiding
import in inductive infix infixl infixr instance let macro module mutual
no-eta-equality open 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 `Set2` 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.

3.14.2 Layout

Agda is layout sensitive using similar rules as Haskell, with the exception that layout is mandatory: you cannot use explicit `{`, `}` and `;` to avoid it.

A layout block contains a sequence of *statements* and is started by one of the layout keywords:

```
abstract field instance let macro mutual postulate primitive private where
```

The first token after the layout keyword decides the indentation of the block. Any token indented more than this is part of the previous statement, a token at the same level starts a new statement, and a token indented less lies outside the block.

```
data Nat : Set where -- starts a layout block
    -- comments are not tokens
  zero : Nat      -- statement 1
  suc  : Nat →    -- statement 2
         Nat      -- also statement 2

one : Nat -- outside the layout block
one = suc zero
```

Note that the indentation of the layout keyword does not matter.

An Agda file contains one top-level layout block, with the special rule that the contents of the top-level module need not be indented.

```
module Example where
NotIndented : Set1
NotIndented = Set
```

3.14.3 Literate Agda

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

```
\documentclass{article}
% some 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}
```

3.15 Literal Overloading

3.15.1 Natural numbers

By default *natural number literals* are mapped to the *built-in natural number type*. This can be changed with the `FROMNAT` built-in, which binds to a function accepting a natural number:

```
{-# BUILTIN FROMNAT fromNat #-}
```

This causes natural number literals `n` to be desugared to `fromNat n`. Note that the desugaring happens before *implicit argument* are inserted so `fromNat` can have any number of implicit or *instance arguments*. This can be exploited to support overloaded literals by defining a *type class* containing `fromNat`:

```
record Number {a} (A : Set a) : Set a where
  field fromNat : Nat → A

open Number {{...}} public

{-# BUILTIN FROMNAT fromNat #-}
```

This definition requires that any natural number can be mapped into the given type, so it won't work for types like `Fin n`. This can be solved by refining the `Number` class with an additional constraint:

```
record Number {a} (A : Set a) : Set (lsuc a) where
  field
    Constraint : Nat → Set a
    fromNat : (n : Nat) {{_ : Constraint n}} → A

open Number {{...}} public using (fromNat)

{-# BUILTIN FROMNAT fromNat #-}
```

This is the definition used in the agda-prelude library. A `Number` instance for `Fin n` can then be defined as follows:

```
natToFin :  {n} (m : Nat) {{_ : IsTrue (m <? n)}} → Fin n

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

3.15.2 Negative numbers

Negative integer literals have no default mapping and can only be used through the `FROMNEG` built-in. Binding this to a function `fromNeg` causes negative integer literals `-n` to be desugared to `fromNeg n`, where `n` is a *built-in natural number*. From the agda-prelude:

```
record Negative {a} (A : Set a) : Set (lsuc a) where
  field
    Constraint : Nat → Set a
    fromNeg : (n : Nat) {{_ : Constraint n}} → A

open Negative {{...}} public using (fromNeg)
{-# BUILTIN FROMNEG fromNeg #-}
```

3.15.3 Strings

String literals are overloaded with the `FROMSTRING` built-in, which works just like `FROMNAT`. If it is not bound string literals map to *built-in strings*. From the agda-prelude:

```
record IsString {a} (A : Set a) : Set (lsuc a) where
  field
    Constraint : String → Set a
```

```
    fromString : (s : String) {{_ : Constraint s}} → A

open IsString {{...}} public using (fromString)
{-# BUILTIN FROMSTRING fromString #-}
```

3.15.4 Other types

Currently only integer and string literals can be overloaded.

3.16 Mixfix Operators

Note: This is a stub.

3.17 Module System

Note: This is a stub.

3.17.1 Module application

3.17.2 Anonymous modules

3.18 Mutual Recursion

Note: This is a stub.

3.19 Pattern Synonyms

Note: This is a stub.

3.20 Postulates

Note: This is a stub.

3.21 Pragmas

Note: This is a stub.

3.22 Record Types

Note: This is a stub.

3.22.1 Record modules

3.22.2 Eta-expansion

3.23 Reflection

Note: This section is incomplete.

3.23.1 Builtin types

Literals

Literals are mapped to the builtin `AGDALITERAL` datatype. Given the appropriate builtin binding for the types `Nat`, `Float`, etc, the `AGDALITERAL` datatype has the following shape:

```
data Literal : Set where
  nat    : Nat    → Literal
  float  : Float  → Literal
  char   : Char   → Literal
  string : String → Literal
  qname  : QName  → Literal

{-# BUILTIN AGDALITERAL   Literal #-}
{-# BUILTIN AGDALITNAT    nat     #-}
{-# BUILTIN AGDALITFLOAT  float   #-}
{-# BUILTIN AGDALITCHAR   char    #-}
{-# BUILTIN AGDALITSTRING string  #-}
{-# BUILTIN AGDALITQNAME  qname   #-}
```

Terms

Terms, types and sorts are mapped to the `AGDATERM`, `AGDATYPE` and `AGDASORT` respectively. Terms use a locally-nameless representation using de Bruijn indices.

```
mutual
  data Term : Set where
    -- Variable applied to arguments.
    var     : (x : ) (args : List (Arg Term)) → Term
    -- Constructor applied to arguments.
    con     : (c : Name) (args : List (Arg Term)) → Term
    -- Identifier applied to arguments.
    def     : (f : Name) (args : List (Arg Term)) → Term
    -- Different kinds of λ-abstraction.
    lam     : (v : Visibility) (t : Abs Term) → Term
    -- Pattern matching λ-abstraction.
    pat-lam : (cs : List Clause) (args : List (Arg Term)) → Term
```

```
    -- Pi-type.
    pi       : (t1 : Arg Type) (t2 : Abs Type) → Term
    -- A sort.
    sort     : (s : Sort) → Term
    -- A literal.
    lit      : (l : Literal) → Term
    -- Reflection constructions.
    quote-goal : (t : Abs Term) → Term
    quote-term : (t : Term) → Term
    quote-context : Term
    unquote-term : (t : Term) (args : List (Arg Term)) → Term
    -- Anything else.
    unknown : Term

  data Type : Set where
    el : (s : Sort) (t : Term) → Type

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

  data Clause : Set where
    clause        : (pats : List (Arg Pattern))(body : Term) → Clause
    absurd-clause : (pats : List (Arg Pattern)) → Clause

{-# BUILTIN AGDASORT    Sort    #-}
{-# BUILTIN AGDATYPE    Type    #-}
{-# BUILTIN AGDATERM    Term    #-}

{-# BUILTIN AGDATERMVAR         var     #-}
{-# BUILTIN AGDATERMCON         con     #-}
{-# BUILTIN AGDATERMDEF         def     #-}
{-# BUILTIN AGDATERMLAM         lam     #-}
{-# BUILTIN AGDATERMEXTLAM      pat-lam #-}
{-# BUILTIN AGDATERMPI          pi      #-}
{-# BUILTIN AGDATERMSORT        sort    #-}
{-# BUILTIN AGDATERMLIT         lit     #-}
{-# BUILTIN AGDATERMQUOTETERM    quote-term    #-}
{-# BUILTIN AGDATERMQUOTEGOAL    quote-goal    #-}
{-# BUILTIN AGDATERMQUOTECONTEXT quote-context #-}
{-# BUILTIN AGDATERMUNQUOTE      unquote-term  #-}
{-# BUILTIN AGDATERMUNSUPPORTED unknown #-}
{-# BUILTIN AGDATYPEEL          el      #-}
{-# BUILTIN AGDASORTSET         set     #-}
{-# BUILTIN AGDASORTLIT         lit     #-}
{-# BUILTIN AGDASORTUNSUPPORTED unknown #-}
```

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

The builtin constructors AGDATERMUNSUPPORTED and AGDASORTUNSUPPORTED are translated to meta variables when unquoting. The sort Setω is translated to `AGDASORTUNSUPPORTED`.

Function Definitions

Functions definitions are mapped to the AGDAFUNDEF builtin:

```
-- Function definition.
data FunctionDef : Set where
  fun-def : Type → Clauses → FunctionDef

{-# BUILTIN AGDAFUNDEF    FunctionDef #-}
{-# BUILTIN AGDAFUNDEFCON fun-def     #-}
```

3.23.2 Quoting and Unquoting

Unquoting Terms

The construction "unquote t" converts a representation of an Agda term to actual Agda code in the following way:

1. The argument t must have type Term (see the reflection API above).
2. The argument is normalised.
3. The entire construction is replaced by the normal form, which is treated as syntax written by the user and type-checked in the usual way.

Examples:

```
test : unquote (def (quote ) [])
test = refl

id : (A : Set) → A → A
id = unquote (lam visible (lam visible (var 0 [])))

id-ok : id  (λ A (x : A) → x)
id-ok = refl
```

Unquoting Declarations

You can define (recursive) functions by reflection using the new unquoteDecl declaration:

```
unquoteDecl x = e
```

Here e should have type AGDAFUNDEF and evaluate to a closed value. This value is then spliced in as the definition of x. In the body e, x has type QNAME which lets you splice in recursive definitions.

Standard modifiers, such as fixity declarations, can be applied to x as expected.

Quoting Terms

The construction "quoteTerm t" evaluates to the AGDATERM representation of the term t. This is done in the following way:

1. The type of t is inferred. The term t must be type-correct.
2. The term t is normalised.
3. The construction is replaced by the Term representation (see the reflection API above) of the normal form. Any unsolved metavariables in the term are represented by the "unknown" term constructor.

Examples:

```
test1 : quoteTerm (λ {A : Set} (x : A) → x)
        lam hidden (lam visible (var 0 []))
test1 = refl

-- Local variables are represented as de Bruijn indices.
test2 : (λ {A : Set} (x : A) → quoteTerm x)  (λ x → var 0 [])
test2 = refl

-- Terms are normalised before being quoted.
test : quoteTerm (0 + 0)  con (quote zero) []
```

Quoting Names

The "quote x" expression returns the builtin QNAME representation of the given name.

```
test : Name
test = quote
```

Quoting Goals

The "quoteGoal x in e" construct allows inspecting the current goal type (the type expected of the whole expression):

```
example :
example = quoteGoal x in {! at this point x = def (quote ) [] !}
```

Quote Patterns

Quote patterns allow pattern matching on quoted names. For instance, here is a function that unquotes a (closed) natural number term:

```
unquoteNat : Term → Maybe Nat
unquoteNat (con (quote Nat.zero) [])            = just zero
unquoteNat (con (quote Nat.suc) (arg _ n  [])) = fmap suc (unquoteNat n)
unquoteNat _                                    = nothing
```

3.23.3 Tactics

Tactis are syntactic sugar which allow using reflection in a syntactically lightweigt manner. It desugars as follows:

```
tactic e                --> quoteGoal g in unquote (e g)
tactic e | e1 | .. | en --> quoteGoal g in unquote (e g) e1 .. en
```

Note that in the second form the tactic function should generate a function from a number of new subgoals to the original goal. The type of e should be Term -> Term in both cases.

3.23.4 Macros

Macros are functions of type t1 → t2 → .. → Term that are defined in a 'macro' block. Macro application is guided by the type of the macro, where Term arguments desugar into the 'quoteTerm' syntax and Name arguments into the 'quote' syntax. Arguments of any other type are preserved as-is.

For example, the macro application 'f u v w' where the macro f has the type 'Term → Name → Bool → Term' desugars into 'unquote (f (quoteTerm u) (quote v) w)'

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
  plus-to-times (def (quote _+_) (a  b  [])) = def (quote _*_) (a  b  [])
  plus-to-times v = v

thm : (a b : Nat) → plus-to-times (a + b)  a * b
thm a b = refl
```

Macros are most useful when writing tactics, since they let you hide the reflection machinery. For instance, suppose you have a solver

```
magic : Term → Term
```

that takes a reflected goal and outputs a proof (when successful). You can then use the tactic function from above to define

```
macro
  by-magic : Term
  by-magic = `tactic (quote magic)
```

This lets you apply the magic tactic without any syntactic noise at all:

```
thm : ¬ P  NP
thm = by-magic
```

3.24 Rewriting

Note: This is a stub.

3.25 Safe Agda

Note: This is a stub.

3.26 Sized Types

Note: This is a stub.

3.27 Telescopes

Note: This is a stub.

3.28 Termination Checking

Note: This is a stub.

3.28.1 With-functions

3.29 Universe Levels

Note: This is a stub.

3.30 With-Abstraction

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

3.30.1 Usage

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

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

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

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

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

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

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

Generalisation

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

```
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

```
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    | 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`:

```
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.

```
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,

```
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

```
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[3] write these functions manually. The downside is that the type signature for the helper function needs to be written out explicitly, but fortunately the *Emacs Mode* has a command (`C-c C-h`) to generate it using the same algorithm that generates the type of a with-function.

Performance considerations

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

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

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

3.30.2 Technical details

Internally with-abstractions are translated to auxiliary functions–there are no with-abstractions in the *Core language*. This translation proceeds as follows. Given a with-abstraction

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

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 $\Delta_1 \vdash A_i$ for all i, i.e., where the types of the scrutinees are well-formed. Note that the partitioning is not required to be a split, $\Delta_1\Delta_2$ can be a (well-formed) reordering of Δ.
- Generalise over the t_i s, by computing

  $$C = (w_1 : A_1)(w_1 : A_2') \ldots (w_m : A_m') \to \Delta_2' \to B'$$

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

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

  where $X \simeq Y$ is equality of the normal forms of X and Y. The type of the auxiliary function is then $\Delta_1 \to C$.
- Check that $\Delta_1 \to C$ is type correct, which is not guaranteed (see *below*).

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

- Add a function f_{aux}, mutually recursive with f, with the definition

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

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

- Replace the with-abstraction by a call to f_{aux} resulting in the final definition

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

  where $ts = t_1 \ldots t_m$ and xs_1 and xs_2 are the variables from Δ corresponding to Δ_1 and Δ_2 respectively.

Examples

Below are some examples of with-abstractions and their translations.

```
postulate
  A      : Set
  _+_    : A → A → A
  T      : A → Set
  mkT    :  x → T x
  P      :  x → T x → Set

-- the type A of the with argument has no free variables, so the with
-- argument will come first
f1 : (x y : A) (t : T (x + y)) → T (x + y)
f1 x y t with x + y
f1 x y t    | w = {!!}

-- Generated with function
f-aux1 : (w : A) (x y : A) (t : T w) → T w
f-aux1 w x y t = {!!}

-- x and p are not needed to type the with argument, so the context
-- is reordered with only y before the with argument
f2 : (x y : A) (p : P y (mkT y)) → P y (mkT y)
f2 x y p with mkT y
f2 x y p    | w = {!!}

f-aux2 : (y : A) (w : T y) (x : A) (p : P y w) → P y w
f-aux2 y w x p = {!!}

postulate
  H :  x y → T (x + y) → Set

-- Multiple with arguments are always inserted together, so in this case
-- t ends up on the left since it's needed to type h and thus x + y isn't
-- abstracted from the type of t
f : (x y : A) (t : T (x + y)) (h : H x y t) → T (x + y)
f x y t h with x + y | h
f x y t h    | w1    | w2 = {! t : T (x + y), goal : T w1 !}
```

```
f-aux : (x y : A) (t : T (x + y)) (h : H x y t) (w1 : A) (w2 : H x y t) → T w1
f-aux x y t h w1 w2 = {!!}

-- But earlier with arguments are abstracted from the types of later ones
f : (x y : A) (t : T (x + y)) → T (x + y)
f x y t with x + y | t
f x y t    | w1    | w2 = {! t : T (x + y), w2 : T w1, goal : T w1 !}

f-aux : (x y : A) (t : T (x + y)) (w1 : A) (w2 : T w1) → T w1
f-aux 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,

```
A : Set
B : A → Set
H : (x : A) → B x → Set

bad-with : (p : Σ A B) → H (fst p) (snd p)
bad-with p with fst p
...        | _ = {!!}
```

Here, generalising over `fst p` results in an ill-typed application `H w (snd p)` and you get the following type error:

```
fst p != w of type A
when checking that the type (p : Σ A B) (w : A) → H w (snd p) of
the generated with function is well-formed
```

This message can be a little difficult to interpret since it only prints the immediate problem (`fst p != w`) and the full type of the with-function. To get a more informative error, pointing to the location in the type where the error is, you can copy and paste the with-function type from the error message and try to type check it separately.

3.31 Without K

Note: This is a stub.

CHAPTER 4

Tools

4.1 Automatic Proof Search (Auto)

Note: This is a stub.

4.2 Command-line options

Note: This is a stub.

4.3 Compilers

4.3.1 Backends

GHC Backend

Note: This is a stub.

The Agda GHC compiler can be invoked from the command line using the flag `--compile`:

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

UHC Backend

Note: This backend is available from Agda 2.5.1 on. 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 1.1.9.2, using other UHC versions may cause problems. To install UHC, the following commands can be used:

```
cabal install uhc-util-0.1.6.3 uulib-0.9.21
wget https://github.com/UU-ComputerScience/uhc/archive/v1.1.9.2.tar.gz
tar -xf v1.1.9.2.tar.gz
cd uhc-1.1.9.2/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
```

JavaScript Backend

Note: This is a stub.

4.3.2 Optimizations

Builtin natural numbers

Note: GHC/UHC backend only.

Builtin natural numbers are now properly represented as Haskell Integers, and the builtin functions on natural numbers are compiled to their corresponding Haskell functions.

Note that pattern matching on an Integer is slower than on an unary natural number. Code that does a lot of unary manipulations and doesn't use builtin arithmetic likely becomes slower due to this optimization. If you find that this is the case, it is recommended to use a different, but isomorphic type to the builtin natural numbers.

Erasable types

A data type is considered *erasable* if it has a single constructor whose arguments are all erasable types, or functions into erasable types. The compilers will erase

- calls to functions into erasable types
- pattern matches on values of erasable type

At the moment the compilers only have enough type information to erase calls of top-level functions that can be seen to return a value of erasable type without looking at the arguments of the call. In other words, a function call will not be erased if it calls a lambda bound variable, or the result is erasable for the given arguments, but not for others.

Typical examples of erasable types are the equality type and the accessibility predicate used for well-founded recursion:

```
data __ {a} {A : Set a} (x : A) : A → Set a where
  refl : x  x

data Acc {a} {A : Set a} (_<_ : A → A → Set a) (x : A) : Set a where
  acc : ( y → y < x → Acc _<_ y) → Acc _<_ x
```

The erasure means that equality proofs will (mostly) be erased, and never looked at, and functions defined by well-founded recursion will ignore the accessibility proof.

4.4 Emacs Mode

Note: This is a stub.

4.4.1 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**odule **c**ontents
`C-c C-n`	Compute **n**ormal form
`C-u C-c C-n`	Compute normal form, ignoring `abstract`
`C-c C-x M-;`	Comment/uncomment rest of buffer

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**odule c**o**ntents
C-c C-n	Compute **n**ormal form
C-u C-c C-n	Compute normal form, ignoring `abstract`

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

4.4.2 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 $\rightarrow$. You can replace `r` with another direction: `u`, `d`, `l`. Eg. `\d-` for $\downarrow$. 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 are commonly used in the standard library:

Character	Short key-binding	TeX command
→	\r-	\to
₁	_1	
₂	_2	
	\~~	\approx
	\all	\forall
	\<	
	\>	
		\ell
	\==	\equiv
	\~	\sim
	\<=	\le
	\'1	\prime
	\::	
λ	\Gl	\lambda
¬		\neg
	\o	\circ
	\bn	\Bbb{N}
×	\x	\times

4.5 Generating HTML

Note: This is a stub.

4.6 Generating LaTeX

Note: This is a stub.

4.7 Library Management

Agda has a simple package management system to support working with multiple libraries in different locations. The central concept is that of a *library*.

4.7.1 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.

4.7.2 Installing libraries

To be found by Agda a library file has to be listed (with its full path) in the file `AGDA_DIR/libraries` (where `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, although this is not expected to be very useful.

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).

4.7.3 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 `--librarary=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`.

4.7.4 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`.

CHAPTER 5

The Agda License

Copyright (c) 2005-2015 Ulf Norell, Andreas Abel, Nils Anders Danielsson, Andrés Sicard-Ramírez, Dominique Devriese, Péter Divianszki, Francesco Mazzoli, Stevan Andjelkovic, Daniel Gustafsson, Alan Jeffrey, Makoto Takeyama, Andrea Vezzosi, Nicolas Pouillard, James Chapman, Jean-Philippe Bernardy, Fredrik Lindblad, Nobuo Yamashita, Fredrik Nordvall Forsberg, Patrik Jansson, Guilhem Moulin, Stefan Monnier, Marcin Benke, Olle Fredriksson, Darin Morrison, Jesper Cockx, Wolfram Kahl, Catarina Coquand

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

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

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

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

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

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE UNIVERSITY COURT OF THE UNIVERSITY OF GLASGOW AND 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 UNIVERSITY COURT OF THE UNIVERSITY OF GLASGOW OR THE 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.

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 6

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.