Agda CheatSheet

Administrivia

Agda is based on Martin-Löf's intuitionistic type theory.

Agda ≈ Haskell + Harmonious Support for Dependent Types

In particular, *types* ≈ *terms* and so, for example, `ℕ : Set = Set₀` and $\texttt{Set}_i$: $\texttt{Set}_{i+1}$. One says *universe* $\texttt{Set}_n$ has *level* n.

- It is a programming language and a proof assistant.
  A proposition is proved by writing a program of the corresponding type.
- Its Emacs interface allows programming by gradual refinement of incomplete type-correct terms. One uses the "hole" marker `?` as a placeholder that is used to stepwise write a program.
- Agda allows arbitrary mixfix Unicode lexemes, identifiers.
  - Underscores are used to indicate where positional arguments.
  - Almost anything can be a valid name; e.g., `[]` and `_::_` below. Hence it's important to be liberal with whitespace: `e:T` is a valid identifier whereas `e : T` declares `e` to be of type `T`.

```
module CheatSheet where

open import Level using (Level)
open import Data.Nat
open import Data.Bool hiding (_<?_)
```

```
open import Data.List using (List; []; _::_; length)
```

Every Agda file contains at most one top-level module whose name corresponds to the name of the file. This document is generated from a `.lagda` file.

Dependent Functions

A *dependent function type* has those functions whose result *type* depends on the *value* of the argument. If `B` is a type depending on a type `A`, then `(a : A) → B a` is the type of functions `f` mapping arguments `a : A` to values `f a : B a`. Vectors, matrices, sorted lists, and trees of a particular height are all examples of dependent types.

For example, *the* generic identity function takes as *input* a type `X` and returns as *output* a function `X → X`. Here are a number of ways to write it in Agda.

All these functions explicitly require the type `X` when we use them, which is silly since it can be inferred from the element `x`.

```
id₀ : (X : Set) → X → X
id₀ X x = x

id₁ id₂ id₃ : (X : Set) → X → X

id₁ X = λ x → x
id₂   = λ X x → x
id₃   = λ (X : Set) (x : X) → x
```

Curly braces make an argument *implicitly inferred* and so it may be omitted. E.g., the `{X : Set} → ⋯` below lets us make a polymorphic function since `X` can be inferred by inspecting the given arguments. This is akin to informally writing id_X versus `id`.

```
id : {X : Set} → X → X
id x = x

sad : ℕ
sad = id₀ ℕ 3

nice : ℕ
nice = id 3

explicit : ℕ
explicit = id {ℕ} 3

explicit' : ℕ
explicit' = id₀ _ 3
```

Notice that we may provide an implicit argument *explicitly* by enclosing the value in braces in its expected position. Values can also be inferred when the `_` pattern is supplied in a value position.

Essentially wherever the typechecker can figure out a value —or a type—, we may use `_`. In type declarations, we have a contracted form via ∀—which is **not** recommended since it slows down typechecking and, more importantly, types *document* our understanding and it's useful to have them explicitly.

In a type, `(a : A)` is called a *telescope* and they can be combined for convenience.

```
  {x : _} {y : _} (z : _) → ⋯
≈  ∀ {x y} z → ⋯
```

```
  (a₁ : A) → (a₂ : A) → (b : B) → ⋯
≈  (a₁ a₂ : A) (b : B) → ⋯
```

Reads

- Dependently Typed Programming in Agda
  - Aimed at functional programmers
- Agda Meta-Tutorial and The Agda Wiki
- Agda by Example: Sorting
  - One of the best introductions to Agda

- ◇ Programming Language Foundations in Agda
  - ∘ Online, well-organised, and accessible book
- ◇ Graphs are to categories as lists are to monoids
  - ∘ A brutal second tutorial
- ◇ Brutal {Meta}Introduction to Dependent Types in Agda
  - ∘ A terse but accessible tutorial
- ◇ Learn You An Agda (and achieve enlightenment)
  - ∘ Enjoyable graphics
- ◇ The Agda Github Umbrella
  - ∘ Some Agda libraries
- ◇ The Power of Pi
  - ∘ Design patterns for dependently-typed languages, namely Agda
- ◇ Making Modules with Meta-Programmed Meta-Primitives
  - ∘ An Emacs editor extension for Agda
- ◇ A gentle introduction to reflection in Agda —Tactics!
- ◇ Epigram: Practical Programming with Dependent Type
  - ∘ "If it typechecks, ship it!" ...
  - ∘ Maybe not; e.g., `if null xs then tail xs else xs`
  - ∘ *We need a static language capable of expressing the significance of particular values in legitimizing some computations rather than others.*

Dependent Datatypes

Algebraic datatypes are introduced with a `data` declaration, giving the name, arguments, and type of the datatype as well as the constructors and their types. Below we define the datatype of lists of a particular length. The Unicode below is entered with `\McN`, `\::`, and `\to`.

```
data Vec {ℓ : Level} (A : Set ℓ) : ℕ → Set ℓ where
  []  : Vec A 0
  _::_ : {n : ℕ} → A → Vec A n → Vec A (1 + n)
```

Notice that, for a given type `A`, the type of `Vec A` is `ℕ → Set`. This means that `Vec A` is a family of types indexed by natural numbers: For each number `n`, we have a type `Vec A n`.

One says `Vec` is *parametrised* by `A` (and ℓ), and *indexed* by `n`.

They have different roles: `A` is the type of elements in the vectors, whereas `n` determines the 'shape' —length— of the vectors and so needs to be more 'flexible' than a parameter.

Notice that the indices say that the only way to make an element of `Vec A 0` is to use `[]` and the only way to make an element of `Vec A (1 + n)` is to use `_::_`. Whence, we can write the following safe function since `Vec A (1 + n)` denotes non-empty lists and so the pattern `[]` is impossible.

```
head : {A : Set} {n : ℕ} → Vec A (1 + n) → A
head (x :: xs) = x
```

The ℓ argument means the `Vec` type operator is *universe polymorphic*: We can make vectors of, say, numbers but also vectors of types. Levels are essentially natural numbers: We have `lzero` and `lsuc` for making them, and `_⊔_` for taking the maximum of two levels. *There is no universe of all universes:* $\texttt{Set}_n$ has type $\texttt{Set}_{n+1}$ *for any* n, however the *type* `(n : Level) → Set n` is *not* itself typeable —i.e., is not in $\texttt{Set}_l$ for any `l`— and Agda errors saying it is a value of `Set`ω.

Functions are defined by pattern matching, and must cover all possible cases. Moreover, they must be terminating and so recursive calls must be made on structurally smaller arguments; e.g., `xs` is a sub-term of `x :: xs` below and catenation is defined recursively on the first argument. Firstly, we declare a *precedence rule* so we may omit parenthesis in seemingly ambiguous expressions.

```
infixr 40 _++_

_++_ : {A : Set} {n m : ℕ} → Vec A n → Vec A m → Vec A (n + m)
[]        ++ ys  =  ys
(x :: xs) ++ ys  =  x :: (xs ++ ys)
```

Notice that the **type encodes a useful property**: The length of the catenation is the sum of the lengths of the arguments.

- ◇ Different types can have the same constructor names.
- ◇ Mixifx operators can be written prefix by having all underscores mentioned; e.g., `x :: xs` is the same as `_::_ x xs`.
- ◇ In a function definition, if you don't care about an argument and don't want to bother naming it, use `_` with whitespace around it. This is the "wildcard pattern".
- ◇ Exercise: Define the Booleans then define the *control flow construct* `if_then_else_`.

The Curry-Howard Correspondence —"Propositions as Types"

Programming and proving are two sides of the same coin.

Logic	**Programming**		Example Use in Programming
proof / proposition	element / type		"p is a proof of P" ≈ "p is of type P"
true	singleton type		return type of side-effect only methods
false	empty type		return type for non-terminating methods
$\Rightarrow$	function type	$\to$	methods with an input and output type
$\wedge$	product type	$\times$	simple records of data and methods
$\vee$	sum type	$+$	enumerations or tagged unions
$\forall$	dependent function type	Π	return type varies according to input *value*
$\exists$	dependent product type	Σ	record fields depend on each other's *values*
natural deduction	type system		ensuring only "meaningful" programs
hypothesis	free variable		global variables, closures
modus ponens	function application		executing methods on arguments
$\Rightarrow$-introduction	λ-abstraction		parameters acting as local variables to method definitions
induction; elimination rules	Structural recursion		`for`-loops are precisely ℕ-induction

Let's augment the table a bit:

Logic	**Programming**
Signature, term	Syntax; interface, record type, `class`
Algebra, Interpretation	Semantics; implementation, instance, object
Free Theory	Data structure
Inference rule	Algebraic datatype constructor
Monoid	Untyped programming / composition
Category	Typed programming / composition

Equality

An example of propositions-as-types is a definition of the identity relation —the least reflexive relation.

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

This states that `refl {x}` is a proof of `l ≡ r` whenever `l` and `r` simplify, by definition chasing only, to `x`.

This definition makes it easy to prove Leibniz's substitutivity rule, "equals for equals":

```
subst : {A : Set} {P : A → Set} {l r : A}
      → l ≡ r → P l → P r
subst refl it = it
```

Why does this work? An element of `l ≡ r` must be of the form `refl {x}` for some canonical form `x`; but if `l` and `r` are both `x`, then `P l` and `P r` are the *same type*. Pattern matching on a proof of `l ≡ r` gave us information about the rest of the program's type!

Modules —Namespace Management

Modules are not a first-class construct, yet.

- Within a module, we may have nested module declarations.
- All names in a module are public, unless declared `private`.

A Simple Module

```
module M where

  𝒩 : Set
  𝒩 = ℕ

  private
    x : ℕ
    x = 3

  y : 𝒩
  y = x + 1
```

Using It

```
use₀ : M.𝒩
use₀ = M.y

use₁ : ℕ
use₁ = y
  where open M

open M

use₂ : ℕ
use₂ = y
```

Parameterised Modules

```
module M' (x : ℕ)
  where
    y : ℕ
    y = x + 1
```

Names are Functions

```
exposed : (x : ℕ)
        → ℕ
exposed = M'.y
```

Using Them

```
use'₀ : ℕ
use'₀ = M'.y 3

module M'' = M' 3

use'' : ℕ
use'' = M''.y

use'₁ : ℕ
use'₁ = y
  where
    open M' 3
```

- Public names may be accessed by qualification or by opening them locally or globally.
- Modules may be parameterised by arbitrarily many values and types —but not by other modules.

Modules are essentially implemented as syntactic sugar: Their declarations are treated as top-level functions that takes the parameters of the module as extra arguments. In particular, it may appear that module arguments are 'shared' among their declarations, but this is not so.

"Using Them":

- This explains how names in parameterised modules are used: They are treated as functions.
- We may prefer to instantiate some parameters and name the resulting module.
- However, we can still `open` them as usual.

Anonymous modules correspond to named-then-immediately-opened modules, and serve to approximate the informal phrase "for any `A : Set` and `a : A`, we have ⋯". This is so common that the `variable` keyword was introduced and it's clever: Names in ⋯ are functions of *only* those `variable`-s they actually mention.

```
   module _ {A : Set} {a : A} ⋯
≈
   module T {A : Set} {a : A} ⋯
   open T
```

```
variable
  A : Set
  a : A
⋯
```

When opening a module, we can control which names are brought into scope with the `using`, `hiding`, and `renaming` keywords.

`open M hiding` $(n_0;\ \ldots;\ n_k)$	Essentially treat n_i as private
`open M using` $(n_0;\ \ldots;\ n_k)$	Essentially treat *only* n_i as public
`open M renaming` $(n_0$ `to` $m_0;\ \ldots;\ n_k$ `to` $m_k)$	Use names m_i instead of n_i

Splitting a program over several files will improve type checking performance, since when you are making changes the type checker only has to check the files that are influenced by the change.

- `import X.Y.Z`: Use the definitions of module `Z` which lives in file `./X/Y/Z.agda`.
- `open M public`: Treat the contents of `M` as if they were public contents of the current module.

Records

A record type is declared much like a datatype where the fields are indicated by the `field` keyword.

`record` ≈ `module` + `data` with one constructor

```
record PointedSet : Set₁ where
  constructor MkIt  {- Optional -}
  field
    Carrier : Set
    point   : Carrier

  {- It's like a module,
  we can add derived definitions -}
  blind : {A : Set} → A → Carrier
  blind = λ a → point
ex₀ : PointedSet
```

```
ex₀ = record {Carrier = ℕ; point = 3}

ex₁ : PointedSet
ex₁ = MkIt ℕ 3

open PointedSet

ex₂ : PointedSet
Carrier ex₂ = ℕ
point   ex₂ = 3
```

Start with `ex₂ = ?`, then in the hole enter `C-c C-c RET` to obtain the *co-pattern* setup. Two tuples are the same when they have the same components, likewise a record is defined by its projections, whence *co-patterns*. If you're using many local definitions, you likely want to use co-patterns!

To allow projection of the fields from a record, each record type comes with a module of the same name. This module is parameterised by an element of the record type and contains projection functions for the fields.

```
use⁰ : ℕ
use⁰ = PointedSet.point ex₀

use¹ : ℕ
use¹ = point where open PointedSet ex₀

open PointedSet

use² : ℕ
use² = blind ex₀ true
```

You can even pattern match on records —they're just `data` after all!

```
use³ : (P : PointedSet) → Carrier P
use³ record {Carrier = C; point = x}
  = x

use⁴ : (P : PointedSet) → Carrier P
use⁴ (MkIt C x)
  = x
```

Interacting with the real world —Compilation, Haskell, and IO

Let's demonstrate how we can reach into Haskell, thereby subverting Agda!

An Agda program module containing a `main` function is compiled into a standalone executable with `agda --compile myfile.agda`. If the module has no main file, use the flag `--no-main`. If you only want the resulting Haskell, not necessarily an executable program, then use the flag `--ghc-dont-call-ghc`.

The type of `main` should be `Agda.Builtin.IO.IO A`, for some `A`; this is just a proxy to Haskell's `IO`. We may `open import IO.Primitive` to get *this* `IO`, but this one works with costrings, which are a bit awkward. Instead, we use the standard library's wrapper type, also named `IO`. Then we use `run` to move from `IO` to `Primitive.IO`; conversely one uses `lift`.

```
open import Data.Nat                  using (ℕ; suc)
open import Data.Nat.Show             using (show)
open import Data.Char                 using (Char)
open import Data.List as L            using (map; sum; upTo)
open import Function                  using (_$_; const; _∘_)
open import Data.String as S          using (String; _++_; fromList)
open import Agda.Builtin.Unit         using (⊤)
open import Codata.Musical.Colist     using (take)
open import Codata.Musical.Costring   using (Costring)
open import Data.BoundedVec.Inefficient as B using (toList)
open import Agda.Builtin.Coinduction using (♯_)
open import IO as IO                  using (run ; putStrLn ; IO)
import IO.Primitive as Primitive
```

*Agda has **no** primitives for side-effects, instead it allows arbitrary Haskell functions to be imported as axioms, whose definitions are only used at run-time.*

Agda lets us use "do"-notation as in Haskell. To do so, methods named `_»_` and `_»=_` need to be in scope —that is all. The type of `IO._»_` takes two "lazy" IO actions and yield a non-lazy IO action. The one below is a homogeneously typed version.

```
infixr 1 _>>=_ _>>_

_>>=_ : ∀ {ℓ} {α β : Set ℓ} → IO α → (α → IO β) → IO β
this >>= f = ♯ this IO.>>= λ x → ♯ f x

_>>_ : ∀{ℓ} {α β : Set ℓ} → IO α → IO β → IO β
x >> y = x >>= const y
```

Oddly, Agda's standard library comes with `readFile` and `writeFile`, but the symmetry ends there since it provides `putStrLn` but not `getLine`. Mimicking the `IO.Primitive` module, we define *two* versions ourselves as proxies for Haskell's `getLine` —the second one below is bounded by 100 characters, whereas the first is not.

```
postulate
  getLine∞ : Primitive.IO Costring

{-# FOREIGN GHC
  toColist :: [a] -> MAlonzo.Code.Codata.Musical.Colist.AgdaColist a
  toColist []       = MAlonzo.Code.Codata.Musical.Colist.Nil
  toColist (x : xs) =
    MAlonzo.Code.Codata.Musical.Colist.Cons x (MAlonzo.RTE.Sharp (toColist xs))
#-}

{- Haskell's prelude is implicitly available; this is for demonstration. -}
{-# FOREIGN GHC import Prelude as Haskell #-}
{-# COMPILE GHC getLine∞  = fmap toColist Haskell.getLine #-}

-- (1)
-- getLine : IO Costring
-- getLine = IO.lift getLine∞

getLine : IO String
getLine = IO.lift
  $ getLine∞ Primitive.>>= (Primitive.return ∘ S.fromList ∘ B.toList ∘ take 100)
```

We obtain `MAlonzo` strings, then convert those to colists, then eventually lift those to the wrapper `IO` type.

Let's also give ourselves Haskell's `read` method.

```
postulate readInt  : L.List Char → ℕ
{-# COMPILE GHC readInt = \x -> read x :: Integer  #-}
```

Now we write our `main` method.

```
main : Primitive.IO ⊤
main = run do putStrLn "Hello, world! I'm a compiled Agda program!"

              putStrLn "What is your name?"
              name ← getLine

              putStrLn "Please enter a number."
              num ← getLine
              let tri = show $ sum $ upTo $ suc $ readInt $ S.toList num
              putStrLn $ "The triangle number of " ++ num ++ " is " ++ tri

              putStrLn "Bye, "
              -- IO.putStrLn∞ name  {- If we use approach (1) above. -}
              putStrLn $ "\t" ++ name
```

For example, the 12^{th} triangle number is $\sum_{i=0}^{12} i = 78$. Interestingly, when an integer parse fails, the program just crashes! Super cool dangerous stuff!

Calling this file `CompilingAgda.agda`, we may compile then run it with:

```
NAME=CompilingAgda; time agda --compile $NAME.agda; ./$NAME
```

The very first time you compile may take ~80 seconds since some prerequisites need to be compiled, but future compilations are within ~10 seconds.

The generated Haskell source lives under the newly created MAlonzo directory; namely `./MAlonzo/Code/CompilingAgda.hs`. Here's some fun: Write a parameterised module with multiple declarations, then use those in your `main`; inspect the generated Haskell to see that the module is thrown away in-preference to top-level functions —as mentioned earlier.

- ◇ When compiling you may see an error `Could not find module 'Numeric.IEEE'`.
- ◇ Simply open a terminal and install the necessary Haskell library:

  ```
  cabal install ieee754
  ```

Absurd Patterns

When there are no possible constructor patterns, we may match on the pattern `()` and provide no right hand side —since there is no way anyone could provide an argument to the function.

For example, here we define the datatype family of numbers smaller than a given natural number: `fzero` is smaller than `suc n` for any `n`, and if `i` is smaller than `n` then `fsuc i` is smaller than `suc n`.

```
{- Fin n  ≅  numbers i with i < n -}
data Fin : ℕ → Set where
  fzero : {n : ℕ} → Fin (suc n)
  fsuc  : {n : ℕ} → Fin n → Fin (suc n)
```

For each n, the type `Fin n` contains n elements; e.g., `Fin 2` has elements `fsuc fzero` and `fzero`, whereas `Fin 0` has no elements at all.

Using this type, we can write a safe indexing function that never "goes out of bounds".

```
_‼_ : {A : Set} {n : ℕ} → Vec A n → Fin n → A
[] ‼ ()
(x :: xs) ‼ fzero  = x
(x :: xs) ‼ fsuc i = xs ‼ i
```

When we are given the empty list, `[]`, then `n` is necessarily `0`, but there is no way to make an element of type `Fin 0` and so we have the absurd pattern. That is, since the empty type `Fin 0` has no elements there is nothing to define —we have a definition by *no cases*.

Logically "anything follows from false" becomes the following program:

```
data False : Set where

magic : {Anything-you-want : Set} → False → Anything-you-want
magic ()
```

Starting with `magic x = ?` then casing on `x` yields the program above since there is no way to make an element of `False` —we needn't bother with a result(ing right side), since there's no way to make an element of an empty type.

Sometimes it is not easy to capture a desired precondition in the types, and an alternative is to use the following `isTrue`-approach of passing around explicit proof objects.

```
{- An empty record has only
   one value: record {} -}
record True : Set where

isTrue : Bool → Set
isTrue true  = True
isTrue false = False
_<₀_ : ℕ → ℕ → Bool
_ <₀ zero      = false
zero <₀ suc y  = true
suc x <₀ suc y = x <₀ y
```

```
find : {A : Set} (xs : List A) (i : ℕ) → isTrue (i <₀ length xs) → A
find [] i ()
find (x :: xs) zero pf    = x
find (x :: xs) (suc i) pf = find xs i pf

head' : {A : Set} (xs : List A) → isTrue (0 <₀ length xs) → A
head' [] ()
head' (x :: xs) _ = x
```

Unlike the `_‼_` definition, rather than there being no index into the empty list, there is no proof that a natural number `i` is smaller than 0.

Mechanically Moving from `Bool` to `Set` —Avoiding "Boolean Blindness"

In Agda we can represent a proposition as a type whose elements denote proofs of that proposition. Why would you want this? Recall how awkward it was to request an index be "in bounds" in the `find` method, but it's much easier to encode this using `Fin` —likewise, `head'` obtains a more elegant type when the non-empty precondition is part of the datatype definition, as in `head`.

Here is a simple recipe to go from Boolean functions to inductive datatype families.

1. Write the Boolean function.
2. Throw away all the cases with right side `false`.
3. Every case that has right side `true` corresponds to a new nullary constructor.
4. Every case that has n recursive calls corresponds to an `n`-ary constructor.

Following these steps for `_<₀_`, from the left side of the page, gives us:

```
data _<₁_ : ℕ → ℕ → Set where
  z< : {y : ℕ} → zero <₁ y
  s< : {x y : ℕ} → x <₁ y → suc x <₁ suc y
```

To convince yourself you did this correctly, you can prove "soundness" —constructed values correspond to Boolean-true statements— and "completeness" —true things correspond to terms formed from constructors. The former is ensured by the second step in our recipe!

```
completeness : {x y : ℕ} → isTrue (x <₀ y) → x <₁ y
completeness {x}     {zero}  ()
completeness {zero}  {suc y} p = z<
completeness {suc x} {suc y} p = s< (completeness p)
```

We began with `completeness {x} {y} p = ?`, then we wanted to case on `p` but that requires evaluating `x <₀ y` which requires we know the shapes of `x` and `y`. *The shape of proofs usually mimics the shape of definitions they use*; e.g., `_<₀_` here.