Practical introduction to Agda

Gergő Érdi
http://gergo.erdhi.hu/

Singapore Functional Meetup, June/July 2012.

module SGMeetup where
Introduction

"Agda is a proof assistant [...] for developing constructive proofs based on the Curry-Howard correspondence [...]. It can also be seen as a functional programming language with dependent types."

Wikipedia on Agda

My goal here is to explain what these key concepts mean and how they combine to form Agda.
Introduction

“Agda is a proof assistant [...] for developing constructive proofs based on the Curry-Howard correspondence [...]. It can also be seen as a functional programming language with dependent types.”

Wikipedia on Agda

My goal here is to explain what these key concepts mean and how they combine to form Agda.
I am assuming familiarity with Haskell, or other mainstream functional programming languages.
Part I

A crash course on the Curry-Howard correspondence
Types as static guarantees

Types matter because they enable automated checking of certain properties.

Trivial example: map

\[ \text{map} :: (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta] \]

Haskell tracks side-effects, so by looking at \textit{map}'s type, we already know that it does no IO.
Types as static guarantees

Types matter because they enable automated checking of certain properties.

Trivial example: map

\[
map :: (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta]
\]

Haskell tracks side-effects, so by looking at \(map\)'s type, we already know that it does no IO.

More involved example: ST

\[
\begin{align*}
\text{newSTRef} & :: \alpha \rightarrow ST \sigma (STRef \sigma \alpha) \\
\text{readSTRef} & :: STRef \sigma \alpha \rightarrow ST \sigma \alpha \\
\text{runST} & :: (\forall \sigma. ST \sigma \alpha) \rightarrow \alpha
\end{align*}
\]

The parametricity of the computation passed to \text{runST} ensures that references don’t leak.
What properties can we express in types? Is this all just a collection of ad-hoc kludges exploiting lucky coincidences?
The Curry-Howard Isomorphism

The simply typed lambda calculus

\[
\frac{x : A \in \Gamma}{\Gamma \vdash x : A}
\]

\[
\frac{\Gamma \vdash f : A \rightarrow B \quad \Gamma \vdash e : A}{\Gamma \vdash f \; e : B}
\]

\[
\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x. \; e : A \rightarrow B}
\]

The type inference rules of the STLC directly parallel the deduction rules of ZOL. Hence, types \(\equiv\) propositions. Terms \(\equiv\) proofs, with functions corresponding to proofs that assume other properties.
The Curry-Howard Isomorphism

Propositional logic

\[
\frac{A \in \Gamma}{\Gamma \vdash A}
\]

\[
\frac{\Gamma \vdash A \rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}
\]

\[
\frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B}
\]
The Curry-Howard Isomorphism

The simply typed lambda calculus \( \simeq \) Propositional logic

\[
\begin{align*}
\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \\
\frac{\Gamma \vdash f : A \rightarrow B \quad \Gamma \vdash e : A}{\Gamma \vdash f \; e : B} \\
\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x. \; e : A \rightarrow B}
\end{align*}
\]

The type inference rules of the STLC directly parallel the deduction rules of ZOL. Hence, \textit{types} \( \simeq \) \textit{propositions}. 
The Curry-Howard Isomorphism

The simply typed lambda calculus $\simeq$ Propositional logic

The type inference rules of the STLC directly parallel the deduction rules of ZOL. Hence, types $\simeq$ propositions. Terms $\simeq$ proofs, with functions corresponding to proofs that assume other properties.
Simple extensions to ZOL

\[ \Gamma \vdash A \quad \Gamma \vdash B \]
\[ \Gamma \vdash A \land B \]

\[ \Gamma \vdash A \land B \]
\[ \Gamma \vdash A \]

\[ \Gamma \vdash A \]
\[ \Gamma \vdash A \lor B \]

We can introduce these axioms as datatypes:

```
data Pair = Pair A B
```

```
data Either = Left A | Right B
```
Simple extensions to ZOL and STLC

\[
\begin{align*}
\Gamma \vdash A & \quad \Gamma \vdash B \\
\hline
\Gamma \vdash A \land B
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash A \land B \\
\hline
\Gamma \vdash A
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash A \\
\hline
\Gamma \vdash A \lor B
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash x : A & \quad \Gamma \vdash y : B \\
\hline
\Gamma \vdash \text{Pair } x \; y : \text{Pair}
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash xy : \text{Pair} \\
\hline
\Gamma \vdash \text{fst } x : A
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash x : A \\
\hline
\Gamma \vdash \text{Left } x : \text{Either}
\end{align*}
\]
Simple extensions to ZOL and STLC

\[
\begin{align*}
\Gamma \vdash A & \quad \Gamma \vdash B \\
\hline
\Gamma \vdash A \land B \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash A \land B \\
\hline
\Gamma \vdash A \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash A \\
\hline
\Gamma \vdash A \lor B \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash x : A & \quad \Gamma \vdash y : B \\
\hline
\Gamma \vdash \text{Pair } x y : \text{Pair} \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash xy : \text{Pair} \\
\hline
\Gamma \vdash \text{fst } x : A \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash x : A \\
\hline
\Gamma \vdash \text{Left } x : \text{Either} \\
\end{align*}
\]

We can introduce these axioms as datatypes:

```haskell
data Pair = Pair A B
fst (Pair x y) = x
snd (Pair x y) = y

data Either = Left A | Right B
```
HM is already more expressive than these simple extensions because it offers polymorphism. We can *abstract over propositions*:

\[
\text{const} :: \alpha \to \beta \to \alpha
\]
HM is already more expressive than these simple extensions because it offers polymorphism. We can abstract over propositions:

\[
\text{const} :: \alpha \to \beta \to \alpha
\]

or with parametric datatypes, introduce whole new axiom schemes:

\[
\text{data} \Pair \alpha \beta = \Pair \alpha \beta \\
\text{data} \Either \alpha \beta = \Left \alpha \mid \Right \beta
\]
Haskell as a proof assistant?

We could regard the Haskell type checker as a proof assistant: using C-H, we can encode our propositions as types, and if the type checker accepts our definition \( x :: A \), then we can regard \( A \) as proven.

Two problems with this approach:

▶ We can't express predicates
    This is a limitation of the type system

▶ \( \text{undefined} :: \alpha \)
    This is a limitation of the computational model

In Agda, definitions are total
Haskell as a proof assistant?

We could regard the Haskell type checker as a proof assistant: using C-H, we can encode our propositions as types, and if the type checker accepts our definition $x :: A$, then we can regard $A$ as proven.

Two problems with this approach:

- We can’t express predicates
We could regard the Haskell type checker as a proof assistant: using C-H, we can encode our propositions as types, and if the type checker accepts our definition \( x :: A \), then we can regard \( A \) as proven.

Two problems with this approach:

- We can’t express predicates
  - This is a limitation of the type system
Haskell as a proof assistant?

We could regard the Haskell type checker as a proof assistant: using C-H, we can encode our propositions as types, and if the type checker accepts our definition \( x :: A \), then we can regard \( A \) as proven.

Two problems with this approach:

- We can’t express predicates
  - This is a limitation of the type system

- \textit{undefined} :: \( \alpha \)
Haskell as a proof assistant?

We could regard the Haskell type checker as a proof assistant: using C-H, we can encode our propositions as types, and if the type checker accepts our definition \( x :: A \), then we can regard \( A \) as proven.

Two problems with this approach:

- We can’t express predicates
  This is a limitation of the type system

- \textit{undefined} :: \( \alpha \)
  This is a limitation of the computational model
We could regard the Haskell type checker as a proof assistant: using C-H, we can encode our propositions as types, and if the type checker accepts our definition \( x :: A \), then we can regard \( A \) as proven.

Two problems with this approach:

- We can’t express predicates
  - This is a limitation of the type system
  Agda uses a dependent type system
- \textit{undefined} :: \( \alpha \)
  - This is a limitation of the computational model
  In Agda, definitions are total
The C-H correspondence generalizes to other type systems and other logic systems.
To give more precise specifications to our definitions, we need something that corresponds (via the C-H isomorphism) to a more expressive logic.
Where does this “dependency” thing come from?

In Haskell...

- **Terms can depend on terms**: regular function definitions
- **Types can depend on types**: type constructors like `Maybe : * → *`
- **Terms can depend on types**: polymorphism (parametric/typeclasses)
In Haskell.

- **Terms can depend on terms**: regular function definitions
- **Types can depend on types**: type constructors like `Maybe : ∗ → ∗`
- **Terms can depend on types**: polymorphism (parametric/typeclasses)

So what about **types depending on terms**? This would correspond, via C-H, to predicates. A **dependent type system** is one where types can depend on terms.
In a dependently-typed setting, the type construction schema $\Pi$ generalizes the notion of function types, so that the type of the result depends on the value of the argument:

\[
\begin{align*}
\Gamma \vdash A : \star & \quad \Gamma, x : A \vdash B : \star \\
\Gamma \vdash \Pi x : A.B : \star
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash A : \star & \quad \Gamma, x : A \vdash e : B \\
\Gamma \vdash \lambda x : A.e : \Pi x : A.B
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash f : \Pi x : A.B & \quad \Gamma \vdash e : A \\
\Gamma \vdash f \ e : B[e/x]
\end{align*}
\]
The type construction schema $\Sigma$ generalizes the notion of product types, so that the type of the second coordinate depends on the value of the first coordinate:

$$
\Gamma \vdash A : \star \quad \Gamma, x : A \vdash B : \star \\
\Gamma \vdash \Sigma x : A.B : \star
$$

$$
\Gamma \vdash e_1 : A \quad \Gamma \vdash e_2 : B[e_1/x] \\
\Gamma \vdash (e_1, e_2) : \Sigma x : A.B
$$

$$
\Gamma \vdash e : \Sigma x : A.B \\
\Gamma \vdash \text{proj} \_1 e : A
$$

$$
\Gamma \vdash e : \Sigma x : A.B \\
\Gamma \vdash \text{proj} \_2 e : B[\text{proj} \_1 e/x]
$$
Part II

A taste of Agda
To understand the following slides, we need to know about a couple of important syntactic distinctions between Haskell and Agda:

- **Implicit arguments**: enclosed between \{ \} symbols

  \[
  \text{map} : \{ A \ B : \text{Set} \} \to (A \to B) \to \text{List} \ A \to \text{List} \ B
  \]

- **Arguments with inferred types**: prefixed with \(\forall\)

  \[
  \text{map} : \forall \{ A \ B \} \to (A \to B) \to \text{List} \ A \to \text{List} \ B
  \]

- **Mixfix notation & unicode characters**:

  \[
  _ + _ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}
  \]
Defining datatypes

It seems every introduction to Agda aimed at programmers has to start with vectors...

\[
\textbf{data} \ Nat : \ Set \ where
\]
\[
\quad \text{zero} : Nat
\]
\[
\quad \text{suc} : Nat \to Nat
\]

\[
\textbf{data} \ Vec (A : Set) : Nat \to Set \ where
\]
\[
\quad \text{nil} : Vec A \text{ zero}
\]
\[
\quad \text{cons} : (n : Nat) \to A \to Vec A n \to Vec A \text{ (suc n)}
\]

The type of a vector contains its length (a value of type \(Nat\))
It seems every introduction to Agda aimed at programmers has to start with vectors...

\[
\textbf{data } \mathbb{N} : \text{Set where}
\]
\[
\begin{align*}
\text{zero} & : \mathbb{N} \\
\text{suc} & : \mathbb{N} \to \mathbb{N}
\end{align*}
\]

\[
\textbf{data } \text{Vec } (A : \text{Set}) : \mathbb{N} \to \text{Set where}
\]
\[
\begin{align*}
[] & : \text{Vec } A \text{ zero} \\
_::_ & : \forall \{n\} \to A \to \text{Vec } A n \to \text{Vec } A \text{ (suc } n)\end{align*}
\]

The type of a vector contains its length (a value of type \( \mathbb{N} \))
map for vectors

With just these definitions, we can already give a richer specification of map: one that records the fact that it preserves length.

\[
\text{map} : \forall \{A, B, n\} \rightarrow (A \rightarrow B) \rightarrow \text{Vec } A \ n \rightarrow \text{Vec } B \ n
\]

\[
\text{map } f \ [\ ] = [\ ]
\]

\[
\text{map } f \ (x :: xs) = f \ x :: \text{map } f \ xs
\]

If instead, we wrote

\[
\text{map } f \ _ = [\ ]
\]

we would get a type error:

\[
\text{zero } \neq \ .n \text{ of type } \mathbb{N}
\]

when checking that the expression \([\ ]\) has type \text{Vec } .B \ .n
But couldn’t you do the same with GADTs\(^1\)?

```haskell
{-# LANGUAGE GADTs, DataKinds #-}
data Nat = Z | S Nat
data Vec a n
  where
    Nil :: Vec a Z
    Cons :: a \rightarrow Vec a n \rightarrow Vec a (S n)
vmap :: (a \rightarrow b) \rightarrow Vec a n \rightarrow Vec b n
vmap f Nil = Nil
vmap f (Cons x xs) = Cons (f x) \$\ vmap f xs
```

\(^1\)and data kinds
So what couldn’t we have done with GADTs?

The power of Π types is that you can lift arbitrary terms into your types, not just (types representing lifted) constructors. E.g. if we have:

\[- + \_ : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}\]

\[\text{zero} + m = m\]

\[(\text{suc } n) + m = \text{suc } (n + m)\]

then we can also write:

\[- \# + \_ : \forall \{A \ n \ m\} \rightarrow \text{Vec } A \ n \rightarrow \text{Vec } A \ m \rightarrow \text{Vec } A \ (n + m)\]

\[[\text{[]} \# ys = ys\]

\[(x :: xs) \# ys = x :: (xs \# ys)\]
Just like with GADTs, when you pattern match on e.g. [], locally (for the right-hand side) the type of \_ \_ \_ \_ is specialized to

\[
\_ \_ \_ \_ : \forall \{ A m \} \rightarrow \text{Vec} \ A \text{ zero} \rightarrow \text{Vec} \ A \text{ m} \rightarrow \text{Vec} \ A \ (\text{zero} + m)
\]

On the other hand, the type of the right-hand side is:

\[
\text{ys} : \text{Vec} \ A \text{ m}
\]
So what couldn’t we have done with GADTs? (cont.)

Just like with GADTs, when you pattern match on e.g. [], locally (for the right-hand side) the type of _ ++ _ is specialized to

\[ _ ++ _ : \forall \{ A \ m \} \rightarrow Vec A \ zero \rightarrow Vec A \ m \rightarrow Vec A (zero + m) \]

On the other hand, the type of the right-hand side is:

\[ ys : Vec A \ m \]

When this right-hand side is typechecked, it has to reduce the function application \( \text{zero} + m \) to \( m \) at compile time. That’s the magic sauce.
Equality by normalization

If we, instead, wrote

\[
_\bot + _\bot : \forall \{A\ n\ m\} \rightarrow \text{Vec}\ A\ n \rightarrow \text{Vec}\ A\ m \rightarrow \text{Vec}\ A\ (m + n),
\]

then the typechecker would reject the same definition, because e.g. for the first branch of \textit{append} setting \(n\) to \textit{zero}, \(\text{zero} + m\) and \(m + \text{zero}\) are not the same terms: the first one reduces to \(m\), whereas the second one cannot be reduced further without knowing anything about \(m\).
Propositional equality

We can define our own equality relation by reflexivity:

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

When we pattern match on `refl`, we learn about other arguments as well. That’s why we can prove the following congruence:

```
cong : ∀{A B x y} → (f : A → B) → x ≡ y → f x ≡ f y
cong f refl = refl
```

since by matching `refl`, the type for that branch becomes

```
cong : ∀{A B x .x} → (f : A → B) → x ≡ .x → f x ≡ f x
```
Proofs about equalities simply encode the needed equality in their types. So let’s try to prove something:

\[- + 0 : \forall n \rightarrow n \equiv (n + \text{zero})\]
\[n + 0 = \text{refl}\]

Of course, this will be rejected by the type checker, since \(n + \text{zero}\) and \(n\) are not the same terms, and neither can be reduced further. To reduce \(n + \text{zero}\), we need to know about \(n\)'s constructor:

\[- + 0 : \forall n \rightarrow n \equiv (n + \text{zero})\]
\[\text{zero} + 0 = \text{refl}\]
\[\text{suc } n + 0 = \text{cong suc } (n + 0)\]
To get a better feel of these proofs, let’s prove that $+$ is commutative:

$\forall n \ m \rightarrow (n + m) \equiv (m + n)$

Let’s consider each of the four cases separately:

- $0 + 0 \equiv 0 + 0$: Both sides reduce to 0
  
  $\vdash \text{refl}$

- $0 + S m \equiv S m + 0$: Since $0 + S m \Rightarrow S m$ and $S n + m \Rightarrow S(\text{suc}(n + m))$, we can recurse by taking the suc of both sides:
  
  $\vdash \text{cong suc} (\text{refl})$
Proving equalities: $+\text{ is commutative}$

To get a better feel of these proofs, let’s prove that $+$ is commutative:

$+ - comm : \forall n m \to (n + m) \equiv (m + n)$

Let’s consider each of the four cases separately:

- **$0 + 0 \equiv 0 + 0$:** Both sides reduce to 0
  
  $+ - comm\ zero\ zero\ =\ refl$

- **$0 + Sm \equiv Sm + 0$:** Since $0 + Sm \rightsquigarrow Sm$ and $Sn + m \rightsquigarrow S\ (n + m)$, we can recurse by taking the $suc$ of both sides:
  
  $+ - comm\ zero\ (suc\ m)\ =\ cong\ suc\ (+ - comm\ zero\ m)$
To get a better feel of these proofs, let’s prove that \(+\) is commutative:

\[ + - \text{comm} : \forall n \ m \rightarrow (n + m) \equiv (m + n) \]

Let’s consider each of the four cases separately:

▶ 0 + 0 \equiv 0 + 0: Both sides reduce to 0

\[ + - \text{comm zero zero} = \text{refl} \]

▶ 0 + Sm \equiv Sm + 0: Since 0 + Sm \leadsto Sm and Sn + m \leadsto S (n + m), we can recurse by taking the suc of both sides:

\[ + - \text{comm zero} (\text{suc} m) = \text{cong suc} (+ - \text{comm zero} m) \]

▶ Sn + 0 \equiv 0 + Sn: Analogous to the previous one:

\[ + - \text{comm} (\text{suc} n) \text{ zero} = \text{cong suc} (+ - \text{comm n zero}) \]
We are left with the fourth case: \( Sn + Sm \equiv Sm + Sn \). To prove that, we will need a property of equality (transitivity) and a lemma about \( + \).

\[
\text{infixl} \ 10 \ _{\langle \text{trans} \rangle} \\
_{\langle \text{trans} \rangle} : \forall \{ A \} \{ x \ y \ z : A \} \to x \equiv y \rightarrow y \equiv z \rightarrow x \equiv z \\
\text{refl} \ _{\langle \text{trans} \rangle} \text{refl} = \text{refl}
\]

\[
+ \ _{-\langle \text{trans} \rangle} \ comm \ suc \ n \ suc \ m = cong \ suc \ ( \\
+ \ comm \ n \ suc \ m) \\
_{\langle \text{trans} \rangle} \\
cong \ suc \ (+ \ comm \ m \ n) \\
_{\langle \text{trans} \rangle} \\
+ \ comm \ suc \ n \ m
\]
The previous proof is basically unreadable... Fortunately, the standard library has a couple of combinators to make equality proofs read like informal ones:

\[
\begin{align*}
+ - \text{comm} (\text{suc } n) (\text{suc } m) &= \text{cong suc}$ \\
\begin{align*}
\text{begin} \hfill \\
\text{n + suc m} &\equiv + - \text{comm } n (\text{suc } m) \\
\text{suc m + n} &\equiv \text{cong suc } (+ - \text{comm } m n) \\
\text{suc n + m} &\equiv + - \text{comm (suc n) m} \\
\text{m + suc n} &\equiv
\end{align*}
\]
Using equalities à la Leibniz

Now that we have proven that \( n + m \equiv m + n \), we can use the equality to substitute one for the other in types:

\[
\text{subst} : \{ A : \text{Set} \} \to (P : A \to \text{Set}) \\
\quad \to \forall \{ x y \} \to x \equiv y \\
\quad \to P x \to P y
\]

\[
\text{subst} P \text{ refl prf} = \text{prf}
\]

Which allows us to write:

\[
_+\!+\!_ : \forall \{ A n m \} \to \text{Vec} A n \to \text{Vec} A m \to \text{Vec} A (m + n)
\]

\[
_+\!+\!_ \{ n = n \} \{ m = m \} \text{xs ys} =
\]

\[
\text{subst} (\text{Vec } _) (\text{+ } \text{comm n m}) (\text{xs } +\!+\!\text{ ys})
\]
Part III

The MU Puzzle
The MU Puzzle

In *Gödel, Escher, Bach*, Hofstadter describes a very simple string rewriting system with the following rules:

- *MI* is a valid string
- You can append a *U* to any valid string ending with *I*
- You can double the string after the initial *M*
- Any *III* can be replaced with a single *U*
- Any occurrences of *UU* can be removed

Hofstadter then asks whether it’s possible to derive *MU* from *MI* using these rules.

```haskell
module MU where
```
Since strings of the MU system always start with an $M$ and contain only $I$ and $U$ afterwards, we can represent words as such:

```haskell
data Symbol : Set where
  I : Symbol
  U : Symbol

open import Data.List

Word : Set
Word = List Symbol
```
Rules of $MU$

We can transliterate the rules into a datatype, where each constructor corresponds to one of the derivation rules. The type is indexed by the word that results from that particular sequence of derivation steps.

\[
\text{data } M : \text{Word} \to \text{Set where }
\]

\[
MI : M [ I ]
\]

\[
MxI \to MxIU : \forall \{ x \} \to M ( x + I :: [ ] ) \to M ( x + I :: U :: [ ] )
\]

\[
Mx \to Mxx : \forall \{ x \} \to M x \to M ( x + x )
\]

\[
III \to U : \forall \{ x y \} \to M ( x + I :: I :: I :: y ) \to M ( x + U :: y )
\]

\[
UUU \to \varepsilon : \forall \{ x y \} \to M ( x + U :: U :: y ) \to M ( x + y )
\]
Rules of $MU$, examples

We can use this definition to prove that e.g. $MIUIU$ is a valid string:

$MIUIU : M (I :: U :: I :: U :: [])$

$MIUIU = Mx \rightarrow Mxx \ (Mxl \rightarrow MxlU \{[]\} \ MI)$

Note that we had to help Agda a bit when applying $Mxl \rightarrow MxlU$, since it cannot automatically determine that if $x + I :: [] = I :: [],$ then $x = [].$
Is $MU$ a valid string?

It can be proven that $MU$ is not a valid string, using the invariant that the number of $I$ characters in every valid string is not divisible by 3. Since the number of $I$’s in $MU$ is 0, and 0 is trivially divisible by 3, we can conclude that $MU$ is not a valid string. How can we write such a proof in Agda?
So far, every proposition was a positive one, and every proof has been constructive. How can we encode negation and proof by contradiction into this system?

By using an absurd type to denote false statements, and giving an elimination rule that encodes ex falso quodlibet:

\[
\text{data } \bot : \text{Set} \ \\
\neg : \text{Set} \rightarrow \text{Set} \ \\
\neg A = A \rightarrow \bot \ \\
\bot\text{-elim} : \forall \{P : \text{Set}\} \rightarrow \bot \rightarrow P
\]

This works because Agda knows there is no pattern that can match \(\bot\). It also means we can't introduce values of type \(\bot\) without matching on some other absurd pattern.
So far, every proposition was a positive one, and every proof has been constructive. How can we encode negation and proof by contradiction into this system? By using an *absurd type* to denote false statements, and giving an elimination rule that encodes *ex falso quodlibet*:

\[
\text{data } \bot : \text{Set where}
\]

\[
\neg : \text{Set} \to \text{Set}
\]

\[
\neg A = A \to \bot
\]

\[
\bot - \text{elim} : \forall \{ P : \text{Set} \} \to \bot \to P
\]

\[
\bot - \text{elim} ()
\]

This works because Agda knows there is no pattern that can match $\bot$. It also means we can’t introduce values of type $\bot$ without matching on some other absurd pattern.
Aside: No law of excluded middle

In some logic systems, the following is true:

\[ \text{excluded middle} : \{ A \ B : \text{Set} \} \rightarrow \neg \neg A \rightarrow A \]

However, the proof scheme this encodes is necessarily non-constructive.
Aside: No law of excluded middle

In some logic systems, the following is true:

\[
\text{excluded middle} : \{ A B : \text{Set} \} \rightarrow \neg\neg A \rightarrow A
\]

However, the proof scheme this encodes is necessarily non-constructive.
In Agda, we cannot prove this.
Proving $MI$ is not a valid word

Our goal is to prove the following proposition:

$$\neg MU : \neg M \ [U]$$

and our plan is to do it via the following invariant, which we’ll prove inductively:

**open import** Data.Nat

$$\#I : Word \to \mathbb{N}$$

$$\#I \ [] = 0$$

$$\#I \ (I :: x) = suc \ (\#I \ x)$$

$$\#I \ (U :: x) = \#I \ x$$

**open import** Data.Nat.Divisibility

$$\_ \nmid \_ : \mathbb{N} \to \mathbb{N} \to \text{Set}$$

$$q \nmid n = \neg q \mid n$$

$$Invariant : Word \to \text{Set}$$

$$Invariant \ x = 3 \nmid \#I \ x$$

$$\text{invariant} : \forall \{x\} \to M \ x \to Invariant \ x$$
The type \( \_ \mid \_ \) we use in the declaration of \( inv \) comes from the standard library, and is defined as the following:

\[
\textbf{data} \quad \_ \mid \_ : \mathbb{N} \to \mathbb{N} \to \textbf{Set} \quad \textbf{where}
\]

\[
divides : \{ m n : \mathbb{N} \} (q : \mathbb{N}) (eq : n \equiv q \ast m) \to m \mid n
\]

\[
keep : \forall \{ x y \} \to x \equiv y \to 3 \mid x \to 3 \mid y
\]

\[
keep = \text{subst} \ (\_ \mid \_ \ 3)
\]
A couple of proofs about \$\# I\$

See the full code for the definitions; for now, it’s enough to understand the statements themselves.

\[
\begin{align*}
\# I - ++ & : \forall x y \to \\
\# I (x ++ y) & \equiv \# I \, x + \# I \, y \\
\# I - xIU & : \forall x \to \\
\# I (x ++ I :: U :: []) & \equiv \# I (x ++ I :: []) \\
\# I - xUy & : \forall x y \to \\
\# I (x ++ y) & \equiv \# I (x ++ U :: y) \\
\# I - xIIIy & : \forall x y \to \\
3 + \# I (x ++ y) & \equiv \# I (x ++ I :: I :: I :: y)
\end{align*}
\]
To prove the base case, we only need to prove $3 \nmid 1$, which we can do by trying to pattern-match on the equation inside $\text{divides}$, and realizing it cannot hold:

\[
\text{invariant} : \forall \{x\} \to M x \to \text{Invariant} x
\]

\[
\text{invariant MI} = 3 \nmid 1
\]

where

\[
3 \nmid 1 : 3 \nmid 1
\]

\[
3 \nmid 1 (\text{divides} \text{ zero} ())
\]

\[
3 \nmid 1 (\text{divides} (\text{suc q}) ())
\]
Proving the invariant: Induction

Using the properties of \( \# I \) we proved earlier, it’s easy to use induction to prove some of the other cases:

\[
i \text{invariant } (MxI \rightarrow MxIU \{x\} MxI) \\
= \text{invariant } MxI \circ \text{keep } (#I - xIU x)
\]
\[
i \text{invariant } (UU \rightarrow \varepsilon \{x\} \{y\} MxUUy) \\
= \text{invariant } MxUUy \circ \text{keep lemma}
\]

where
\[
\text{lemma : } \#I (x \uplus y) \equiv \#I (x \uplus U :: U :: y)
\]
\[
\text{lemma } = \#I - xUy x y \langle \text{trans} \rangle \#I - xUy x (U :: y)
\]
Proving the invariant: Induction

Using the properties of \( \#I \) we proved earlier, it’s easy to use induction to prove some of the other cases:

\[
\text{invariant } (MxI \rightarrow MxIU \{x\} MxI) \\
= \text{invariant } MxI \circ \text{keep } (\#I - xIU\ x)
\]
\[
\text{invariant } (UU \rightarrow \varepsilon \{x\} \{y\} MxUUy) \\
= \text{invariant } MxUUy \circ \text{keep lemma}
\]

where

\[
\text{lemma } : \#I (x \mid + \ y) \equiv \#I (x \mid + \ U :: U :: y)
\]
\[
\text{lemma } = \#I - xUy\ x\ y \langle \text{trans} \rangle \#I - xUy\ x\ (U :: y)
\]

So the tricky ones that remain are:

\[
\text{invariant } (III \rightarrow U \{x\} \{y\} MxIll'y) \\
= \text{invariant } MxIll'y \circ ?
\]

\[
\text{-- Need a proof that if } 3 \mid \#I xUy, \text{ then } 3 \mid \#I xIll'y
\]
\[
\text{invariant } (Mx \rightarrow Mxx \{x\} Mx) \\
= \text{invariant } Mx \circ \text{keep } ?
\]

\[
\text{-- Need a proof that if } 3 \mid x, \text{ then } 3 \mid x + x
\]
First of all, we know \( \#I \ xUy \equiv \#I \ xy \), and also that 
\( \#I \ xIIIy \equiv 3 + \#I \ xy \), so the important lemma is that 
\( 3 \mid x \rightarrow 3 \mid 3 + x: \)

\[
invariant (III \rightarrow U \{x\} \{y\} \ MxIIIy) \\
= invariant \ MxIIIy \circ \ proof
\]

where

\[
lemma_1 : \forall \ n \rightarrow 3 \mid n \rightarrow 3 \mid 3 + n
\]
First of all, we know \( \#I \times Uy \equiv \#I \times y \), and also that \( \#I \times IIIy \equiv 3 + \#I \times xy \), so the important lemma is that

\[
3 \mid x \rightarrow 3 \mid 3 + x
\]

\[
\text{invariant} \ (III \rightarrow U \{x\} \{y\} MxIIIy) = \text{invariant} \ MxIIIy \circ \text{proof}
\]

where

\[
\text{lemma}_1 : \forall n \rightarrow 3 \mid n \rightarrow 3 \mid 3 + n
\]

\[
\text{lemma}_1 n (\text{divides} \ qn \equiv q \ast 3) = \text{divides} \ (\text{suc} \ q) \ (\text{cong} \ (\_ + \_ 3) n \equiv q \ast 3)
\]
invariant \((III \rightarrow U \{x\} \{y\} MxIIIy)\)  
\[= \text{invariant } MxIIIy \circ \text{proof}\]

where

\[\text{lemma}_2 : 3 + \#I (x \# U :: y) \equiv \#I (x \# I :: I :: I :: y)\]
invariant \((III \rightarrow U \{x\} \{y\} \ MxIIIy)\)
\[= \text{invariant } MxIIIy \circ \text{proof}\]

where

\(\text{lemma}_2 : \) 3 + \#I (x U : y) \equiv \\
\#I (x ⊕ I :: I :: I :: y)

\(\text{lemma}_2\)
\[= \text{cong } (+ + 3) \ (\text{sym } \ (#I \ - \ xUy \ x \ y))\]
\[\langle \text{trans} \rangle\]
\[#I \ - \ xIIIy \ x \ y\]
\[ 3 \mid \#I \ xUy \rightarrow 3 \mid \#I \ xIIly \]

\[ \text{invariant } (III \rightarrow U \{x\} \{y\} \ MxIIly) = \text{invariant } MxIIly \circ \text{proof} \]

where

\[ \text{lemma}_1 : \forall \ n \rightarrow 3 \mid n \rightarrow 3 \mid 3 + n \]

\[ \text{lemma}_2 : 3 + \#I (x \#U :: y) \equiv \#I (x \#I :: I :: I :: y) \]

\[ \text{proof} : 3 \mid \#I (x \#U :: y) \rightarrow 3 \mid \#I (x \#I :: I :: I :: y) \]

\[ \text{proof} = \text{keep lemma}_2 \circ \text{lemma}_1 (\#I (x \#U :: y)) \]
For the indirect proof here, the crucial lemma is that if $3 \mid 2 \ast n$, then $3 \mid n$ would also hold, which is in contradiction with our inductive assumption.

\[
\text{invariant} \ (Mx \to Mxx \ {x} \ Mx) \\
\quad = \text{invariant} \ Mx \circ \text{lemma} \circ \text{keep} \ (\#I - \text{dup} \ x)
\]

where
\[
dup : \forall \ n \to n + n \equiv 2 \ast n
\]
\[
dup n = \text{cong} \ (_ + _ \ n) \ (n + 0)
\]
\[
\#I - \text{dup} : \forall \ x \to \#I \ (x \ ++ \ x) \equiv 2 \ast \#I \ x
\]
\[
\#I - \text{dup} \ x = \#I - \ + + \ x \ x \ \langle \text{trans} \rangle \ \text{dup} \ (\#I \ x)
\]
\[
\text{lemma} : \forall \ {n} \to 3 \mid 2 \ast n \to 3 \mid n
\]
The standard library contains definitions and proofs of some pretty high-level stuff, so we can prove $3 \mid 2 \ast n \rightarrow 3 \mid n$ by observing that 2 and 3 are co-primes...

\[
\text{lemma} : \forall \{n\} \rightarrow 3 \mid 2 \ast n \rightarrow 3 \mid n
\]

\[
\text{lemma} = \text{coprime} - \text{divisor} 3 - \text{coprime} - 2
\]

where

open import Data.Nat.Coprimality

$3 - \text{coprime} - 2 : \text{Coprime} 3 2$

$3 - \text{coprime} - 2 = \text{prime} \Rightarrow \text{coprime} - 3 - \text{prime} 2$

(from – yes (1 $\leq? 2$))

(from – yes (3 $\leq? 3$))

where

open import Data.Nat.Primality

open import Relation.Nullary.Decidable

$3 - \text{prime} : \text{Prime} 3$

$3 - \text{prime} = \text{from} - \text{yes} (\text{prime?} 3)$
We’re finished... but there’s a lot more to Agda!

Stratified universes
Totality and the termination checker
Coinductive types & corecursive definitions
We’re finished... but there’s a lot more to Agda!

Stratified universes
Totality and the termination checker
Coinductive types & corecursive definitions
And lot more...