Practical introduction to Agda

Gergő Érdi
http://gergo.erdi.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

$$\mathit{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta]$$

Haskell tracks side-effects, so by looking at *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

$$map :: (\alpha \to \beta) \to [\alpha] \to [\beta]$$

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

More involved example: ST

$$\begin{aligned} newSTRef &:: \alpha &&\to ST\ \sigma\ (STRef\ \sigma\ \alpha) \\ readSTRef &:: STRef\ \sigma\ \alpha &&\to ST\ \sigma\ \alpha \\ runST &:: (\forall\ \sigma.\ ST\ \sigma\ \alpha) &&\to \alpha \end{aligned}$$

The parametricity of the computation passed to *runST* ensures that references don't leak

Types as static guarantees

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

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}$$

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash e : A}{\Gamma \vdash f\ e : B}$$

$$\frac{\Gamma, x{:}A \vdash e : B}{\Gamma \vdash \lambda x.\ e : A \to B}$$

The Curry-Howard Isomorphism

Propostional logic

$$\frac{A \in \Gamma}{\Gamma \vdash A}$$

$$\frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}$$

The Curry-Howard Isomorphism

The simply typed lambda calculus $\simeq$ Propostional logic

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}$$

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash e : A}{\Gamma \vdash f\ e : B}$$

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x.\ e : A \to B}$$

The type inference rules of the STLC directly parallel the deduction rules of ZOL. Hence, *types* $\simeq$ *propositions*.

The Curry-Howard Isomorphism

The simply typed lambda calculus $\simeq$ Propostional logic

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}$$

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash e : A}{\Gamma \vdash f\ e : B}$$

$$\frac{\Gamma, x{:}A \vdash e : B}{\Gamma \vdash \lambda x.\ e : A \to B}$$

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

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$$

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}$$

Simple extensions to ZOL and STLC

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$$

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}$$

$$\frac{\Gamma \vdash x : A \qquad \Gamma \vdash y : B}{\Gamma \vdash Pair\ x\ y : Pair}$$

$$\frac{\Gamma \vdash xy : Pair}{\Gamma \vdash fst\ x : A}$$

$$\frac{\Gamma \vdash x : A}{\Gamma \vdash Left\ x : Either}$$

Simple extensions to ZOL and STLC

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \qquad \frac{\Gamma \vdash x : A \qquad \Gamma \vdash y : B}{\Gamma \vdash Pair\ x\ y : Pair}$$

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \qquad \frac{\Gamma \vdash xy : Pair}{\Gamma \vdash fst\ x : A}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \qquad \frac{\Gamma \vdash x : A}{\Gamma \vdash Left\ x : Either}$$

We can introduce these *axioms as datatypes*:

$$\begin{aligned} &\mathbf{data}\ Pair = Pair\ A\ B \\ &fst\ (Pair\ x\ y) = x \\ &snd\ (Pair\ x\ y) = y \\ &\mathbf{data}\ Either = Left\ A \mid Right\ B \end{aligned}$$

The Hindley-Milner Type System

HM is already more expressive than these simple extensions because it offers polymorphism. We can *abstract over propositions*:

$$\textit{const} :: \alpha \to \beta \to \alpha$$

The Hindley-Milner Type System

HM is already more expressive than these simple extensions because it offers polymorphism. We can *abstract over propositions*:

$$\mathit{const} :: \alpha \to \beta \to \alpha$$

or with parametric datatypes, introduce whole new *axiom schemes*:

$$\begin{aligned}
\textbf{data}\ \mathit{Pair}\ \alpha\ \beta\ &=\ \mathit{Pair}\ \alpha\ \beta \\
\textbf{data}\ \mathit{Either}\ \alpha\ \beta\ &=\ \mathit{Left}\ \alpha\ \mid\ \mathit{Right}\ \beta
\end{aligned}$$

Haskell as a proof assistant?

We could regard the Haskell type checker as a proof assitant: 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.

Haskell as a proof assistant?

We could regard the Haskell type checker as a proof assitant: 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

Haskell as a proof assistant?

We could regard the Haskell type checker as a proof assitant: 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 assitant: 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
- *undefined* :: α

Haskell as a proof assistant?

We could regard the Haskell type checker as a proof assitant: 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
- *undefined* $:: \alpha$
  This is a limitation of the computational model

Haskell as a proof assistant?

We could regard the Haskell type checker as a proof assitant: 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
- *undefined* $:: \alpha$
  This is a limitation of the computational model
  In Agda, definitions are total

Other type systems

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 $Maybe : \star \to \star$
- *Terms can depend on types:* polymorphism (parametric/typeclasses)

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 $Maybe : \star \to \star$
- *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.

Dependent types: Π

In a dependently-typed setting, the type construction schema Π generalizes the notion of function types, so that the *type of the result depends on the value of the argument*:

$$\frac{\Gamma \vdash A : \star \qquad \Gamma, x : A \vdash B : \star}{\Gamma \vdash \Pi x : A.B : \star}$$

$$\frac{\Gamma \vdash A : \star \qquad \Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x : A.e : \Pi x : A.B}$$

$$\frac{\Gamma \vdash f : \Pi x : A.B \qquad \Gamma \vdash e : A}{\Gamma \vdash f\ e : B[e/x]}$$

Dependent types: Σ

The type construction schema Σ generalizes the notion of product types, so that the *type of the second coordinate depends on the value of the first coordinate*:

$$\frac{\Gamma \vdash A : \star \qquad \Gamma, x : A \vdash B : \star}{\Gamma \vdash \Sigma x : A.B : \star}$$

$$\frac{\Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : B[e_1/x]}{\Gamma \vdash (e_1, e_2) : \Sigma x : A.B}$$

$$\frac{\Gamma \vdash e : \Sigma x : A.B}{\Gamma \vdash proj_1 \ e : A}$$

$$\frac{\Gamma \vdash e : \Sigma x : A.B}{\Gamma \vdash proj_2 \ e : B[proj_1 \ e/x]}$$

Part II

A taste of Agda

Agda syntax

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

$$map : \{A\ B : Set\} \to (A \to B) \to List\ A \to List\ B$$

- Arguments with inferred types: prefixed with ∀

$$map : \forall \{A\ B\} \to (A \to B) \to List\ A \to List\ B$$

- 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. . .

$$
\begin{array}{l}
\textbf{data}\ \mathit{Nat} : \mathit{Set}\ \textbf{where} \\
\quad \mathit{zero} : \mathit{Nat} \\
\quad \mathit{suc} \ \ : \mathit{Nat} \to \mathit{Nat} \\
\textbf{data}\ \mathit{Vec}\ (A : \mathit{Set}) : \mathit{Nat} \to \mathit{Set}\ \textbf{where} \\
\quad \mathit{nil} : \mathit{Vec}\ A\ \mathit{zero} \\
\quad \mathit{cons} : (n : \mathit{Nat}) \to A \to \mathit{Vec}\ A\ n \to \mathit{Vec}\ A\ (\mathit{suc}\ n)
\end{array}
$$

The type of a vector contains its length (a value of type *Nat*)

Defining datatypes

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

$$
\begin{array}{l}
\textbf{data}\ \mathbb{N}\ :\ \mathit{Set}\ \textbf{where} \\
\quad \mathit{zero}\ :\ \mathbb{N} \\
\quad \mathit{suc}\ \ \ :\ \mathbb{N} \to \mathbb{N} \\
\textbf{data}\ \mathit{Vec}\ (A\ :\ \mathit{Set})\ :\ \mathbb{N} \to \mathit{Set}\ \textbf{where} \\
\quad [\,]\ :\ \mathit{Vec}\ A\ \mathit{zero} \\
\quad \_ :: \_\ :\ \forall\ \{n\} \to A \to \mathit{Vec}\ A\ n \to \mathit{Vec}\ A\ (\mathit{suc}\ n)
\end{array}
$$

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.

$$
\begin{aligned}
&map : \forall \{A\ B\ n\} \to (A \to B) \to Vec\ A\ n \to Vec\ B\ n \\
&map\ f\ [\,] = [\,] \\
&map\ f\ (x :: xs) = f\ x :: map\ f\ xs
\end{aligned}
$$

If instead, we wrote

$$map\ f\ \_ = [\,]$$

we would get a type error:

```
zero != .n of type ℕ
when checking that the expression [] has type Vec .B .n
```

But couldn't you do the same with GADTs[1]?

```
{-# LANGUAGE GADTs, DataKinds #-}
data Nat = Z | S Nat

data Vec a n where
  Nil :: Vec a Z
  Cons :: a -> Vec a n -> Vec a (S n)

vmap :: (a -> b) -> Vec a n -> 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:

$$
\begin{array}{lcl}
\_+\_ : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} & & \\
\mathit{zero} \quad + m & = & m \\
(\mathit{suc}\ n) + m & = & \mathit{suc}\ (n + m)
\end{array}
$$

then we can also write:

$$
\begin{array}{lcl}
\_ +\!\!+ \_ : \forall \{A\ n\ m\} \rightarrow \mathit{Vec}\ A\ n \rightarrow \mathit{Vec}\ A\ m \rightarrow \mathit{Vec}\ A\ (n + m) & & \\
[\,] \qquad\quad +\!\!+\ ys & = & ys \\
(x :: xs)\ +\!\!+\ ys & = & x :: (xs\ +\!\!+\ ys)
\end{array}
$$

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

$$\_ +\!\!+ \_ \;:\; \forall \{A\ m\} \to \mathit{Vec}\ A\ \mathit{zero} \to \mathit{Vec}\ A\ m \to \mathit{Vec}\ A\ (\mathit{zero} + m)$$

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

$$\mathit{ys} \;:\; \mathit{Vec}\ A\ m$$

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

$$\_ +\!\!+ \_ : \forall \{A\ m\} \to Vec\ A\ zero \to Vec\ A\ m \to Vec\ A\ (zero + m)$$

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 $zero + m$ to m *at compile type*. That's the magic sauce.

Equality by normalization

If we, instead, wrote

$$\_ +\!\!+ \_ \; : \; \forall \; \{A \; n \; m\} \rightarrow Vec \; A \; n \rightarrow Vec \; A \; m \rightarrow Vec \; A \; (m + n),$$

then the typechecker would reject the same definition, because e.g. for the first branch of *append* setting n to $zero$, $zero + m$ and $m + zero$ 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:

$$\textbf{data}\ \_\equiv\_\ \{A : Set\} : A \to A \to Set\ \textbf{where}$$
$$\quad refl : \forall\ \{x\} \to x \equiv x$$

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

$$cong : \forall\ \{A\ B\ x\ y\} \to (f : A \to B) \to x \equiv y \to f\ x \equiv f\ y$$
$$cong\ f\ refl = refl$$

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

$$cong : \forall\ \{A\ B\ x\ .x\} \to (f : A \to B) \to x \equiv .x \to f\ x \equiv f\ x$$

Proving equalities

Proofs about equalities simpy encode the needed equality in their types. So let's try to prove something:

$$
\begin{aligned}
\_+0 &: \forall\ n \to n \equiv (n + zero) \\
n+0 &= refl
\end{aligned}
$$

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

$$
\begin{aligned}
&\_+0 : \forall\ n \to n \equiv (n + zero) \\
&zero\ +0 = refl \\
&suc\ n\ +0 = cong\ suc\ (n\ +0)
\end{aligned}
$$

Proving equalities: $+$ is commutative

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

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

Let's consider each of the four cases separately:

- $0 + 0 \equiv 0 + 0$: Both sides reduce to 0

$$+-comm\ zero\ zero\ =\ refl$$

Proving equalities: $+$ 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:

  $$+-comm\ zero\ (suc\ m) \ = \ cong\ suc\ (+-comm\ zero\ m)$$

Proving equalities: $+$ 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:

$$+-comm\ zero\ (suc\ m) \;=\; cong\ suc\ (+-comm\ zero\ m)$$

- $Sn + 0 \equiv 0 + Sn$: Analogous to the previous one:

$$+-comm\ (suc\ n)\ zero \;=\; cong\ suc\ (+-comm\ n\ zero)$$

Proving equalities: $+$ is commutative (cont.)

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 $+$.

$$
\begin{array}{l}
\textbf{infixl}\ \mathit{10}\ \_\langle \mathit{trans} \rangle\_ \\
\_\langle \mathit{trans} \rangle\_ \ : \ \forall \{A\}\ \{x\ y\ z\ :\ A\} \to x \equiv y \to y \equiv z \to x \equiv z \\
\mathit{refl}\ \langle \mathit{trans} \rangle\ \mathit{refl}\ = \ \mathit{refl}
\end{array}
$$

$$
\begin{array}{l}
+ - \mathit{comm}\ (\mathit{suc}\ n)\ (\mathit{suc}\ m)\ =\ \mathit{cong}\ \mathit{suc}\ (\\
\quad + - \mathit{comm}\ n\ (\mathit{suc}\ m) \\
\qquad \langle \mathit{trans} \rangle \\
\quad \mathit{cong}\ \mathit{suc}\ (+ - \mathit{comm}\ m\ n) \\
\qquad \langle \mathit{trans} \rangle \\
\quad + - \mathit{comm}\ (\mathit{suc}\ n)\ m \\
\quad)
\end{array}
$$

Proving equalities: the nicer way

The previous proof is basically unreadable. . .
Fortunately, the standard library has a couple of combinators to make equality proofs read like informal ones:

$$
\begin{array}{l}
+-comm\ (suc\ n)\ (suc\ m) \;=\; cong\ suc\ \$ \\
\quad begin \\
\qquad n + suc\ m \equiv\langle\ +-comm\ n\ (suc\ m)\ \rangle \\
\qquad suc\ m + n \equiv\langle\ cong\ suc\ (+-comm\ m\ n)\ \rangle \\
\qquad suc\ n + m \equiv\langle\ +-comm\ (suc\ n)\ m\ \rangle \\
\qquad m + suc\ n \\
\qquad \square
\end{array}
$$

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:

$$\begin{aligned} &\mathit{subst} : \{A : \mathit{Set}\} \to (P : A \to \mathit{Set}) \\ &\qquad \to \forall \{x\ y\} \to x \equiv y \\ &\qquad \to P\ x \to P\ y \\ &\mathit{subst}\ P\ \mathit{refl}\ \mathit{prf} = \mathit{prf} \end{aligned}$$

Which allows us to write:

$$\begin{aligned} &\_ +\!\!+'\_ : \forall \{A\ n\ m\} \to \mathit{Vec}\ A\ n \to \mathit{Vec}\ A\ m \to \mathit{Vec}\ A\ (m + n) \\ &\_ +\!\!+'\_ \{n = n\}\ \{m = m\}\ \mathit{xs}\ \mathit{ys} = \\ &\quad \mathit{subst}\ (\mathit{Vec}\ \_)\ (+ - \mathit{comm}\ n\ m)\ (\mathit{xs}\ +\!\!+\ \mathit{ys}) \end{aligned}$$

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 occurances of *UU* can be removed

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

```
module MU where
```

Words of *MU*

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

```
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.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ M : \mathit{Word} \to \mathit{Set}\ \textbf{where}} \\
\quad \mathit{MI} & : & M\,[\,I\,] \\
\quad \mathit{MxI} \to \mathit{MxIU} & : & \forall\,\{x\} \to M\,(x +\!\!+\ I :: [\,]) \to \\
 & & M\,(x +\!\!+\ I :: U :: [\,]) \\
\quad \mathit{Mx} \to \mathit{Mxx} & : & \forall\,\{x\} \to M\,x \to \\
 & & M\,(x +\!\!+\ x) \\
\quad \mathit{III} \to U & : & \forall\,\{x\ y\} \to M\,(x +\!\!+\ I :: I :: I :: y) \to \\
 & & M\,(x +\!\!+\ U :: y) \\
\quad \mathit{UU} \to \varepsilon & : & \forall\,\{x\ y\} \to M\,(x +\!\!+\ U :: U :: y) \to \\
 & & M\,(x +\!\!+\ y)
\end{array}
$$

Rules of MU, examples

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

$$MIUIU \; : \; M \, (I \; :: \; U \; :: \; I \; :: \; U \; :: \; [\,])$$
$$MIUIU \; = \; Mx \rightarrow Mxx \, (MxI \rightarrow MxIU \, \{[\,]\} \; MI)$$

Note that we had to help Agda a bit when applying $MxI \rightarrow MxIU$, 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?

Negation

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?

Negation

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*:

$$\textbf{data} \perp \ : \ \mathit{Set} \ \textbf{where}$$

$$\neg\_ \ : \ \mathit{Set} \to \mathit{Set}$$
$$\neg\, A \ = \ A \to \perp$$

$$\perp - \mathit{elim} \ : \ \forall \, \{ P \ : \ \mathit{Set} \} \to \perp \to P$$
$$\perp - \mathit{elim} \ ()$$

This works because Agda knows there is no pattern that can match $\perp$. It also means we can't introduce values of type $\perp$ without matching on some other absurd pattern.

Aside: No *law of excluded middle*

In some logic systems, the following is true:

$$excluded - middle \; : \; \{A \; B \; : \; Set\} \rightarrow \neg \neg A \rightarrow A$$

However, the proof scheme this encodes is necessarily non-constructive.

Aside: No *law of excluded middle*

In some logic systems, the following is true:

$$excluded - middle \ : \ \{A\ B \ : \ Set\} \rightarrow \neg\ \neg\ A \rightarrow A$$

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*

$$\_\nmid\_ \ : \ \mathbb{N} \to \mathbb{N} \to Set$$
$$q \nmid n \ = \ \neg \ q \mid n$$

$$Invariant \ : \ Word \to Set$$
$$Invariant \ x \ = \ 3 \nmid \#I \ x$$

$$invariant \ : \ \forall \ \{x\} \to M \ x \to Invariant \ x$$

Divisibility

The type $\_ \mid \_$ we use in the declaration of *inv* comes from the standard library, and is defined as the following:

$$\textbf{data}\ \_ \mid \_ \ :\ \mathbb{N} \to \mathbb{N} \to \mathit{Set}\ \textbf{where}$$
$$\quad \mathit{divides}\ :\ \{m\ n\ :\ \mathbb{N}\}\ (q\ :\ \mathbb{N})\ (\mathit{eq}\ :\ n \equiv q * m) \to m \mid n$$

$$\mathit{keep}\ :\ \forall\ \{x\ y\} \to x \equiv y \to 3 \mid x \to 3 \mid y$$
$$\mathit{keep}\ =\ \mathit{subst}\ (\_ \mid \_\ 3)$$

A couple of proofs about $\#I$

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

$$\#I-{+}{+}\;:\;\forall\; x\; y \rightarrow$$
$$\quad \#I\;(x\; +\!\!+\; y) \equiv \#I\; x + \#I\; y$$

$$\#I-xIU\;:\;\forall\; x \rightarrow$$
$$\quad \#I\;(x\; +\!\!+\; I\; ::\; U\; ::\; [\,]) \equiv \#I\;(x\; +\!\!+\; I\; ::\; [\,])$$

$$\#I-xUy\;:\;\forall\; x\; y \rightarrow$$
$$\quad \#I\;(x\; +\!\!+\; y) \equiv \#I\;(x\; +\!\!+\; U\; ::\; y)$$

$$\#I-xIIIy\;:\;\forall\; x\; y \rightarrow$$
$$\quad 3 + \#I\;(x\; +\!\!+\; y) \equiv \#I\;(x\; +\!\!+\; I\; ::\; I\; ::\; I\; ::\; y)$$

Proving the invariant: Base case

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 *divides*, and realizing it cannot hold:

$$
\begin{array}{l}
\mathit{invariant} \;:\; \forall \{x\} \to M\; x \to \mathit{Invariant}\; x \\
\mathit{invariant}\; \mathit{MI} \;=\; 3 \nmid 1 \\
\quad \textbf{where} \\
\quad 3 \nmid 1 \;:\; 3 \nmid 1 \\
\quad 3 \nmid 1\; (\mathit{divides}\; \mathit{zero}\; ()) \\
\quad 3 \nmid 1\; (\mathit{divides}\; (\mathit{suc}\; q)\; ())
\end{array}
$$

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:

$$\begin{aligned}
&\mathit{invariant}\ (\mathit{MxI} \to \mathit{MxIU}\ \{x\}\ \mathit{MxI})\\
&\quad = \mathit{invariant}\ \mathit{MxI} \circ \mathit{keep}\ (\#I\text{-}xIU\ x)\\
&\mathit{invariant}\ (\mathit{UU} \to \varepsilon\ \{x\}\ \{y\}\ \mathit{MxUUy})\\
&\quad = \mathit{invariant}\ \mathit{MxUUy} \circ \mathit{keep}\ \mathit{lemma}\\
&\quad\ \textbf{where}\\
&\quad\ \mathit{lemma} : \#I\ (x +\!\!+\ y) \equiv \#I\ (x +\!\!+\ U :: U :: y)\\
&\quad\ \mathit{lemma} = \#I\text{-}xUy\ x\ y\ \langle\ \mathit{trans}\ \rangle\ \#I\text{-}xUy\ x\ (U :: y)
\end{aligned}$$

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:

$$
\begin{aligned}
&\mathit{invariant}\ (\mathit{MxI} \rightarrow \mathit{MxIU}\ \{x\}\ \mathit{MxI})\\
&\quad = \ \mathit{invariant}\ \mathit{MxI} \circ \mathit{keep}\ (\#I - \mathit{xIU}\ x)\\
&\mathit{invariant}\ (\mathit{UU} \rightarrow \varepsilon\ \{x\}\ \{y\}\ \mathit{MxUUy})\\
&\quad = \ \mathit{invariant}\ \mathit{MxUUy} \circ \mathit{keep}\ \mathit{lemma}\\
&\quad\ \ \textbf{where}\\
&\quad\ \ \mathit{lemma}\ :\ \#I\ (x \mathbin{+\!\!+} y) \equiv \#I\ (x \mathbin{+\!\!+} U :: U :: y)\\
&\quad\ \ \mathit{lemma}\ =\ \#I - \mathit{xUy}\ x\ y\ \langle\ \mathit{trans}\ \rangle\ \#I - \mathit{xUy}\ x\ (U :: y)
\end{aligned}
$$

So the tricky ones that remain are:

$$
\begin{aligned}
&\mathit{invariant}\ (\mathit{III} \rightarrow U\ \{x\}\ \{y\}\ \mathit{MxIIIy})\\
&\quad = \ \mathit{invariant}\ \mathit{MxIIIy} \circ ?\\
&\qquad\quad \text{-- Need a proof that if } 3 \mid \#I\ \mathit{xUy}\text{, then } 3 \mid \#I\ \mathit{xIIIy}\\
&\mathit{invariant}\ (\mathit{Mx} \rightarrow \mathit{Mxx}\ \{x\}\ \mathit{Mx})\\
&\quad = \ \mathit{invariant}\ \mathit{Mx} \circ \mathit{keep}\ ?\\
&\qquad\quad \text{-- Need a proof that if } 3 \mid x\text{, then } 3 \mid x + x
\end{aligned}
$$

$3 \mid \#I\ xUy \to 3 \mid \#I\ xIIIy$

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 $3 \mid x \to 3 \mid 3 + x$:

$$\begin{aligned} & \mathit{invariant}\ (\mathit{III} \to U\ \{x\}\ \{y\}\ \mathit{MxIIIy}) \\ & \quad = \mathit{invariant}\ \mathit{MxIIIy} \circ \mathit{proof} \\ & \quad \textbf{where} \\ & \quad \mathit{lemma}_1\ :\ \forall\ n \to 3 \mid n \to 3 \mid 3 + n \end{aligned}$$

$3 \mid \#I\ xUy \rightarrow 3 \mid \#I\ xIIIy$

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

$$lemma_1\ :\ \forall\ n \rightarrow 3 \mid n \rightarrow 3 \mid 3 + n$$

$$lemma_1\ n\ (divides\ q\ n \equiv q * 3)$$
$$=\ divides\ (suc\ q)\ (cong\ (\_ + \_\ 3)\ n \equiv q * 3)$$

$3 \mid \#I\ xUy \rightarrow 3 \mid \#I\ xIIIy$

$$
\begin{aligned}
&invariant\ (III \rightarrow U\ \{x\}\ \{y\}\ MxIIIy) \\
&\quad = \ invariant\ MxIIIy \circ proof \\
&\quad \textbf{where} \\
&\quad lemma_2\ :\ 3 + \#I\ (x\ +\!\!+\ U\ ::\ y) \equiv \\
&\qquad \#I\ (x\ +\!\!+\ I\ ::\ I\ ::\ I\ ::\ y)
\end{aligned}
$$

$3 \mid \#I\ xUy \to 3 \mid \#I\ xIIIy$

$$invariant\ (III \to U\ \{x\}\ \{y\}\ MxIIIy)$$
$$=\ invariant\ MxIIIy \circ proof$$
where

$$lemma_2\ :\ 3 + \#I\ (x\ +\!\!+\ U\ ::\ y) \equiv$$
$$\#I\ (x\ +\!\!+\ I\ ::\ I\ ::\ I\ ::\ y)$$

$$lemma_2$$
$$=\ cong\ (\_ + \_\ 3)\ (sym\ (\#I - xUy\ x\ y))$$
$$\langle\ trans\ \rangle$$
$$\#I - xIIIy\ x\ y$$

$3 \mid \#I\ xUy \rightarrow 3 \mid \#I\ xIIIy$

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

where

$$lemma_1\ :\ \forall\ n \rightarrow 3 \mid n \rightarrow 3 \mid 3 + n$$

$$lemma_2\ :\ 3 + \#I\ (x\ +\!\!+\ U\ ::\ y) \equiv \#I\ (x\ +\!\!+\ I\ ::\ I\ ::\ I\ ::\ y)$$

$$proof\ :\ 3 \mid \#I\ (x\ +\!\!+\ U\ ::\ y) \rightarrow 3 \mid \#I\ (x\ +\!\!+\ I\ ::\ I\ ::\ I\ ::\ y)$$

$$proof$$
$$=\ keep\ lemma_2 \circ lemma_1\ (\#I\ (x\ +\!\!+\ U\ ::\ y))$$

$3 \mid \#I\ xx \to 3 \mid \#I\ x$

For the indirect proof here, the crucial lemma is that if $3 \mid 2 * n$, then $3 \mid n$ would also hold, which is in contradiction with our inductive assumption.

$$
\begin{aligned}
& \mathit{invariant}\ (Mx \to Mxx\ \{x\}\ Mx) \\
& \quad = \ \mathit{invariant}\ Mx \circ \mathit{lemma} \circ \mathit{keep}\ (\#I - \mathit{dup}\ x) \\
& \quad \textbf{where} \\
& \quad \mathit{dup}\ : \forall\ n \to n + n \equiv 2 * n \\
& \quad \mathit{dup}\ n\ = \ \mathit{cong}\ (\_ + \_\ n)\ (n + 0) \\
& \quad \#I - \mathit{dup}\ : \forall\ x \to \#I\ (x\ +\!\!+\ x) \equiv 2 * \#I\ x \\
& \quad \#I - \mathit{dup}\ x\ = \ \#I - {+\!\!+}\ x\ x\ \langle\ \mathit{trans}\ \rangle\ \mathit{dup}\ (\#I\ x) \\
& \quad \mathit{lemma}\ : \forall\ \{n\} \to 3 \mid 2 * n \to 3 \mid n
\end{aligned}
$$

$3 \mid 2 * n \to 3 \mid n$

The standard library contains definitions and proofs of some pretty high-level stuff, so we can prove $3 \mid 2 * n \to 3 \mid n$ by observing that 2 and 3 are co-primes...

$$lemma : \forall \{n\} \to 3 \mid 2 * n \to 3 \mid n$$

$lemma = coprime-divisor\ 3-coprime-2$
  where
  open import *Data.Nat.Coprimality*

  $3-coprime-2 : Coprime\ 3\ 2$
  $3-coprime-2 = prime \Rightarrow coprime\ \_\ 3-prime\ 2$
    $(from-yes\ (1 \leqslant?\ 2))$
    $(from-yes\ (3 \leqslant?\ 3))$
    where
    open import *Data.Nat.Primality*
    open import *Relation.Nullary.Decidable*

    $3-prime : Prime\ 3$
    $3-prime = from-yes\ (prime?\ 3)$

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...

Questions?