Agda

Programming with Dependent Types

Johannes Niederhauser

June 13, 2022

VU Interactive Theorem Proving

Introduction

Agda is

- a dependently typed functional programming language
- an interactive system for writing and checking proofs
- based on Martin-Löf's intuitionistic type theory
- a total language
- implemented in Haskell
- primarily designed to be a programming language

It has an *Emacs* interface which assists the programmer in writing the program.

Programming with Dependent Types

Syntax for Dependent Types

For the product type

$$\Pi x : A.B$$

we write

$$(x : A) \to B.$$

Instead of $\star$ and $\square$, a sort system is used

$$\mathtt{Set}_0 \in \mathtt{Set}_1 \in \mathtt{Set}_2 \in \cdots$$

where the sort `Set` is an abbreviation of $\mathtt{Set}_0$ and contains the so-called small types.

Polymorphism

How to emulate polymorphism with dependent types?

- no type variables in Agda
- no full polymorphism in the sense of $\lambda 2$
- no $\star$, but sorts $\texttt{Set}_i$

Instead of

```
id : 'a -> 'a
```

we define

```
id : (A : Set) -> A -> A
id A x = x
```

Implicit Arguments

- in Agda, type inference is undecidable
- types have to be stated explicitly
- for easy cases, it is possible to leave out information
- implicit arguments do not have to be supplied

```
id : {A : set} -> A -> A
id x = x
```

In simple enough cases, we can therefore use polymorphism as in Hindley-Milner type systems.

Vectors of Length n

With a type of vectors of a given length

```
data Vec (A : Set) : Nat -> Set where
  []   : Vec A zero
  _::_ : {n : Nat} -> A -> Vec A n -> Vec A (succ n)
```

we can define safe versions of `head` and `tail`:

```
head : {A : Set} {n : Nat} -> Vec A (succ n) -> A
head (x :: _) = x

tail : {A : Set} {n : Nat} ->
       Vec A (succ n) -> Vec A n
tail (_ :: xs) = xs
```

Vectors of Length n (cont'd)

With a data type for pairs

```
data _X_ (A B : Set) : Set where
  <_,_> : A -> B -> A X B
```

we can also define a version of zip which type checks iff both arguments have the same length:

```
zip : {A B : Set} {n : Nat} ->
      Vec A n -> Vec B n -> Vec (A X B) n
zip [] [] = []
zip (x :: xs) (y :: ys) = < x , y > :: zip n xs ys
```

Propositions as Types in Agda

Short recap

- in intuitionistic logic, a proposition is interpreted as the set of its proofs
- furthermore, a proposition is true iff a proof exists
- according to the Curry-Howard correspondence, a (functional) program is just a proof of its type
- and computation corresponds to proof normalization

In the following, we show that untyped intuitionistic predicate logic with equality can be realized in Agda.

The connective $\wedge$

```
data _/\_ (A B : Set) : Set where
  <_,_> : A -> B -> A /\ B
```

The constructor is the introduction rule and the two elimination rules are defined as follows:

```
fst : {A B : Set} -> A /\ B -> A
fst < a , b > = a

snd : {A B : Set} -> A /\ B -> B
snd < a , b > = b
```

The connective $\vee$

```
data _\/_ (A B : Set) : Set where
  inl : A -> A \/ B
  inr : B -> A \/ B
```

The two constructors are the introduction rules, the elimination rule is defined as follows:

```
case : {A B C : Set} -> A \/ B ->
       (A -> C) -> (B -> C) -> C
case (inl a) d e = d a
case (inr b) d e = e b
```

The constants $\top$ and $\bot$

$\top$ is always provable (by the unit element `<>`) and $\bot$ has no proof, therefore the corresponding set of programs is empty.

```
data True : Set where
  <> : True

data False : Set where
```

$\bot$-elimination states that if we derived $\bot$, everything follows. Since there is no way to construct an element of type $\bot$, there is nothing to define.

```
nocase : {A : Set} -> False -> A
nocase ()
```

The connectives $\to$ and $\wedge$

As in the λ-calculus, we obtain the properties of $\to$ simply by function abstraction and application.

Hence, we use Agda's built-in `->` for $\to$.

Furthermore, we can define $\neg\phi \equiv \phi \to \bot$, so

```
Not : Set -> Set
Not A = A -> False
```

Universal quantification

Since Agda uses dependent types, ∀-introduction and ∀-elimination are just dependent function abstraction/application:

```
Forall : (A : Set) -> (B : A -> Set) -> Set
Forall A B = (x : A) -> B x
```

Existential quantification

The BHK interpretation states that a proof of $\exists x : A.B$ consist of an element $a : A$ together with a proof $B[x := a]$:

```
data Exists (A : Set) (B : A -> Set) : Set where
  [_,_] : (a : A) -> B a -> Exists A B
```

From the elimination rules, the witness and the corresponding proof can be obtained:

```
dfst : {A : Set} {B : A -> Set} -> Exists A B -> A
dfst [ a , b ] = a

dsnd : {A : Set} {B : A -> Set} ->
       (p : Exists A B) -> B (dfst p)
dsnd [ a , b ] = b
```

Equality

Equality introduction is quite simple:

```
data _==_ {A : Set} : A -> A -> Set where
refl : (a : A) -> a == a
```

The elimination rule allows to substitute equals for equals:

```
subst : {A : Set} -> {C : A -> Set} -> {a' a'' : A} ->
        a' == a'' -> C a' -> C a''
subst (refl a) c = c
```

An Example

The `with` Construct

We write

```
f p with d
... | q1 = e1
    :
... | qn = en
```

to perform an exhaustive pattern match on `d` where `q1` – `qn` are the patterns.

This construct is not a basic type-theoretic construct and we do not look into the implementation details.

Sorted Binary Trees with Correctness Proof

See live demo

Further Reading

This presentation is largely based on Dependent Types at Work by Ana Bove and Peter Dybjer.

More detailed information can be found here:

- The Agda Wiki
- Ulf Norell's PhD thesis

Questions?