Agda

CS242

Lecture 15

Reprise: Program Verification

- Proving properties of programs

- But not just that programs are well-typed
  - Much deeper, almost arbitrary properties
  - And often verifying full functional correctness

- Components
  - A specification: What property the program is supposed to have
  - A proof: Written mostly manually
  - A proof assistant: Supports defining the concepts, managing the proof, checking the proof, some automation of easy parts of the proof

- Proof assistants are based on *type theory*

Today

- An overview of Agda

- One of many systems that implement dependent type theory
  - And provide a basis for developing mechanically checked proofs

- Agda is fairly close to Haskell
  - In syntax and semantics

Data

```
data Bool : Set where
  false : Bool
  true : Bool
```

```
not : Bool -> Bool
not true = false
not false = true
```

“Set” is the equivalent of “Type” in Lecture 14

“Data” declares algebraic data types

Not is defined using multiple clauses with a *pattern* describing which argument(s) match each clause

White Space in Agda

- Agda allows most special characters to appear in identifiers (names)
  - And to start identifiers
- Examples of valid names:
  - fish+chips
  - +10
  - i:Int
- *Thus whitespace is important in Agda*
  - i:Int is very different from i : Int

Data (Cont.)

```
data Nat : Set where
  zero : Nat
  succ : Nat -> Nat

plus : Nat -> Nat -> Nat
plus zero m = m
plus (succ n) m =    succ (plus n m)
```

- The definition of plus also uses pattern matching
- Plus is defined with multiple clauses, each clause handles a certain pattern on the lhs
- The first clause handles the case where the first argument is zero

Data (Cont.)

```
data Nat : Set where
  zero : Nat
  succ : Nat -> Nat

plus : Nat -> Nat -> Nat
plus zero m = m
plus (succ n) m =    succ (plus n m)
```

- The definition of plus uses pattern matching
- Plus is defined with multiple clauses, each clause handles a certain pattern on the lhs
- The second clause handles when the first argument has the form (succ n) and *binds the variable n*

Pattern Matching

- In general a function can be defined by cases:

f pattern1 = rhs1

f pattern2 = rhs2

…

- There can be any number of clauses
- Patterns can be nested
  - succ(succ(x))
  - cons(x,cons(y,z))
- Variable names in patterns are
  - Bound to matching subterms
  - In scope on the rhs of the clause

Pattern Matching in Agda

- In general a function can be defined by cases:

```
f pattern1 = rhs1
f pattern2 = rhs2
…
```

- Patterns must be exhaustive
  - Every possible case must be covered
- Patterns must be disjoint
  - The patterns in different clauses cannot overlap in what they can match
  - Other languages allow overlapping patterns in different clauses, requires specifying which pattern will take priority if more than one matches

Infix Operators

```
_+_ : Nat -> Nat -> Nat
zero + m = m
succ n + m = succ (n + m)

data List (A : Set) : Set where
 [] : List A
 _::_ : A -> List A -> List A

zlist = zero :: []
```

- An infix operator *op* is declared using the name _*op*_
- An "_" indicates where the argument will go
- More general than infix! If-then-else can be define as an operator
  - if_then_else_

Polymorphic Functions

identity : (A : Set) -> A -> A

identity A x = x

zero' : Nat

zero' = identity Nat zero

- Polymorphic functions are examples of *dependent types*
  - The function takes a Set argument and the rest of the function signature depends on the *value* of that argument
  - Values of Set are ordinary types
- Note that identity is first applied to Nat to produce an identity function of type Nat -> Nat
  - Instantiation of the polymorphic type is explicit

Implicit Arguments

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

zero'' : Nat
zero'' = id zero
```

- Types wrapped in "{…}" are *implicit arguments*
- The user is asserting that the type checker should be able to infer the type with no argument being actually supplied
  - From the types of subsequent arguments
- In this version, instantiation of the polymorphic type is implicit
  - There is no type argument passed to id

Records

```
record Position : Set  where
  field
    xc : Nat
    yc : Nat

pos : Position
pos = record { xc = zero; yc = succ(zero) }

myxc : Position -> Nat
myxc p = Position.xc p

myyc : Position -> Nat
myyc p = Position.yc p
```

- Records are like records in functional programming languages or structs in C
- A collection of named, typed fields
- There is a selector (destructor) for each field

Polymorphic Records

```
record Position2 (A : Set) (B : Set) : Set
where
  field
    xc : A
    yc : B

pos2 : Position2 Nat Nat
pos2 = record { xc = zero; yc = succ(zero) }
```

- Record constructors can take arguments
- This example is a record type polymorphic in the types of each of its two fields

Next Steps

- So far we've looked at Agda mostly as a mostly standard functional language

- Next we give a non-trivial example using dependent types

Representing True and False

data False : Set where

record True : Set where

A data type with no constructors has no elements – there are no values of type False

A record with no fields has exactly one value, the empty record, which we use for True

Using True and False

```
trivial : True
trivial = _

isTrue : Bool -> Set
isTrue true = True
isTrue false = False
```

- Trivial is equal to the single value in True
  - Which Agda type inference can automatically deduce for the *wildcard*
- isTrue maps the elements of Bool to the corresponding type
  - Even though False has no elements, it is still a type

Less Than

_<_ : Nat -> Nat -> Bool

_ < zero = false

zero < succ n = true

succ m < succ n = m < n

- Note the use of a wildcard pattern in the first clause

Length of a List

```
length : {A : Set} -> List A -> Nat

length [] = zero
length (x :: xs) = succ (length xs)
```

- Note the use of an implicit type parameter in the type of length

A Digression: Holes

- What if we didn't know how to write length?

- We could have Agda's type checker help us by using a *hole*

length l = ?

A Safe List Lookup Function

```
lookup : {A : Set}(xs : List A)(n : Nat) -> isTrue (n < length xs) -> A
lookup [] n ()
lookup (x :: xs) zero p = x
lookup (x :: xs) (succ n) p = lookup xs n p
```

Lookup takes a list xs and a natural number n and returns the nth element of xs.

This lookup function is *safe* – it only type checks if the list has at least n elements.

A Safe List Lookup Function

```
lookup : {A : Set}(xs : List A)(n : Nat) -> isTrue (n < length xs) -> A
lookup [] n ()
lookup (x :: xs) zero p = x
lookup (x :: xs) (succ n) p = lookup xs n p
```

The third argument is a *proof object*:

- it is True only if the length of xs is less than n.
- If the third argument is equal to False, type checking fails
  - since False has no elements, the argument could never be supplied

A Safe List Lookup Function

```
lookup : {A : Set}(xs : List A)(n : Nat) -> isTrue (n < length xs) -> A
lookup [] n ()
lookup (x :: xs) zero p = x
lookup (x :: xs) (succ n) p = lookup xs n p
```

The first clause uses the *absurd pattern* ()

- Lookup on the empty list for any n has no proof object
- This case can never happen!
- Note that there is no right-hand side

A Safe List Lookup Function

```
lookup : {A : Set}(xs : List A)(n : Nat) -> isTrue (n < length xs) -> A
lookup [] n ()
lookup (x :: xs) zero p = x
lookup (x :: xs) (succ n) p = lookup xs n p
```

Notice the proof object in the recursive call (last clause) is p

- We would expect a different proof object here, one for n and xs instead of (succ n) and (x :: xs)
- But Agda type checking is smart enough to discover that p implies that n < length xs

A Useful Datatype

```
data Eq {A : Set} (x : A) : A -> Set where
    refl : Eq x x
```

- An example of an *indexed type*
  - Eq x returns a function from A to a type in Set
  - A different type for every element of A
  - We say that the type is *indexed by A*

- For each x, there is a type Eq x x
  - i.e., Eq x x is a different type for each distinct x
  - Captures that a value is reflexively equal only to itself

A Proof: Congruence of Function Application

```
cong : {A : Set} {B : Set} {m : A} {n : A} (f : A -> B) -> Eq m n -> Eq (f m) (f n)
cong f refl = refl
```

- "If m = n, then f m = f n"
- Refl is a constructor of no arguments
- Pattern matching gives the first occurrence of refl type Eq m n
- Type inference deduces the second occurrence of refl must have type Eq (f m) (f n), which is valid because if m and n are the same term, then f m and f n are also the same term

Other Features: Where

```
sum : List Nat -> Nat
sum xs = helper xs zero
  where
    helper : List Nat -> Nat -> Nat
    helper [] acc = acc
    helper (x :: xs) acc = helper xs (acc + x)
```

- Just as in normal programming, local definitions help to organize the structure of the code and use local names that are not visible outside of the scope

Other Features: Let

```
trip : Nat -> Nat
trip n =
  let double = n + n
      triple = n + double
  in triple
```

- A more common way (in functional languages) to organize code and manage names

Other Features: Lambda

```
addZero : Nat -> Nat
addZero n = (λ x → x + zero) n
```

- While we have not mentioned it to this point, Agda supports unicode, so this is actual Agda code.

Summary

- Dependent type theory includes functional programming
  - But it has a lot more!
- Very expressive types with non-trivial computational content allow us to state complex propositions as types
  - And the programs of that type are then the proofs