Overloading in Agda

Catarina Coquand

December 15, 2004

– Typeset by FoilTEX –

What is Agda?

Not important for this talk, we will talk about a dependent type theory with

1. signature and records
2. meta-variables
3. hidden arguments

Example : Expressions

```
data Exp = EVar (s :: String)
         | ENum (n::Nat)
         | EPlus (e1,e2::Exp)

eval :: Env -> Exp -> Maybe Nat
eval = \env ->  \e ->
       case e of
         (EVar s) -> lookup env s
         (ENum n) -> return n
         (EPlus e1 e2) -> do i1 <- eval env e1
                             i2 <- eval env e2
                             return (nat_plus i1 i2)
```

Example : Exp with Integer

```
data Exp = EVar (s :: String)
         | ENum (n :: Integer)
         | EPlus (e1,e2 :: Exp)

eval :: Env -> Exp -> Maybe Integer
eval = \env ->  \e ->
       case e of
         (EVar s) -> lookup env s
         (ENum n) -> return n
         (EPlus e1 e2) -> do i1 <- eval env e1
                             i2 <- eval env e2
                             return (integer_plus i1 i2)
```

How would we like to write eval

```
eval = \env ->  \e ->
       case e of
         (EVar s) -> lookup env s
         (ENum n) -> return n
         (EPlus e1 e2) -> do i1 <- eval env e1
                             i2 <- eval env e2
                             return (i1 + i2)
```

Why Overloading?

- Readability
- Reusability (of both code and proofs)

What Kind of Overloading?

- Type directed overloading
- User-defined
- Similar to the Classes in Haskell
- Similar to Axiomatic Type Classes in Isabelle

Example: Plus

```
class Plus (a::Set) :: Set exports
    (+) :: a -> a -> a

instance plusNat :: Plus Nat where
    (+) = \n n' -> case n of
                     (Zero) -> n'
                     (Succ m) -> Succ (m + n')

instance plusInteger :: Plus Integer where
    (+) = ....
```

Example: Use of Plus

```
twiceNat :: Nat -> Nat
twiceNat = \x -> x + x

-- twiceNat = \x -> (+) Nat plusNat x x

twiceInt :: Integer -> Integer
twiceInt = \x -> x + x

-- twiceInt = \x -> (+) Integer plusInteger x x
```

Example ctd: Use of Plus

```
twice (|a::Set)(|pa::Plus a) :: a -> a
twice = \x -> x + x

-- twice = \x -> (+) a pa x x

twiceInt :: Integer -> Integer
twiceInt = \x -> twice x
-- twiceInt = \x -> twice Integer plusInteger x
```

Example: Properties

```
class PlusProp(a::Set) :: Set extends (plusa::Plus a) exports
     assoc_plus :: (x,y,z::a) -> (x+y)+z ==  x+(y+z)
     comm_plus  :: (x,y::a) -> x+y == y+x
     ....
instance natPlusProp :: PlusProp Nat where
     assoc_plus = ?
     comm_plus  = ?
     ...

lemma (|a::Set)(ppa::PlusProp a)
      ::  (x,y,z::a) -> (x+y)+z ==  z+(y+x)
lemma = ?
```

Example: Monads

```
class  Monad (m ::Set -> Set)  :: Type exports
    (>>=)  :: (a,b::Set) |-> m a -> (a -> m b) -> m b
    (>>)   :: (a,b::Set) |-> m a -> m b -> m b
    return :: (a::Set) |-> a -> m a
```

Typechecking Classes and Overloaded Functions

- Classes are translated to signatures
- Instances are translated to records
- Overloaded functions are implemented as projections
- Overloaded functions calls are saturated with meta-variables, i.e. $5 + 7$ is replaced by $(+)\ ?_1\ ?_2\ 5\ 7$

Typechecking then yields constraints, such as for example: $?_2 :: Plus\ Integer$ these are solved by a special constraint-solver for classes.

Possible Extensions

- Multi-parameter classes seems easy
- Default definitions
- Since we can name instances, we could have overlapping instances that is not used in constraint solving

Conclusions

- A "light-way" to add a rather powerful technique for overloading
- Works nicely for the intended application: to be able to overload standard functions.