Programming with Dependent types (based on a presentation by Matthieu Sozeau)

Danil Annenkov and Bas Spitters

November 15, 2021.

Aarhus University

Equations in Coq

Equations: function definitions by dependent pattern-matching and recursion

- A plugin for the Coq proof assistant.
- Developed in INRIA, France.
- Features powerful (dependent) pattern-matching.
- Derives (generates) useful reasoning principles for definitions.

Equations vs match

Equality of natural numbers in Haskell (Agda, Idris)

```
equal :: Nat → Nat → Bool
equal 0 0 = True
equal (S m') (S n') = equal m' n'
equal _ _ = False
```

Equations vs match

Equality of natural numbers in Haskell (Agda, Idris)

```
equal :: Nat → Nat → Bool
equal O O = True
equal (S m') (S n') = equal m' n'
equal _ _ = False
```

Equality of natural numbers in Coq (basic `match ... with`)

```
Fixpoint fix_equal (m n : nat) : bool :=
  match m with
  | O ⇒ match n with
        | O ⇒ true
        | S n' ⇒ false
        end
  | S m' ⇒ match n with
           | O ⇒ false
           | S n' ⇒ fix_equal m' n'
           end
  end.
```

Equations vs match

Equality of natural numbers in Haskell (Agda, Idris)

```
equal :: Nat → Nat → Bool
equal 0 0 = True
equal (S m') (S n') = equal m' n'
equal _ _ = False
```

Equality of natural numbers in Coq (with EQUATIONS)

```
Equations equal (m n : nat) : bool :=
equal 0 0 := true ;
equal (S m) (S n) := equal m n ;
equal _ _ := false.
```

Equations vs match, cont.

- The "native" pattern-matching in Coq `match ... with` is simple.
  - ⇒ easier to implement
  - ⇒ smaller trusted computing base
  - ⇒ less bugs in the implementation

Equations vs match, cont.

- The "native" pattern-matching in Coq `match` ... `with` is simple.
  - ⇒ easier to implement
  - ⇒ smaller trusted computing base
  - ⇒ less bugs in the implementation
- Issues: hard to use with dependent types.

Equations: features

```
Equations equal (m n : nat) : bool :=
equal O O := true ;
equal (S m) (S n) := equal m n ;
equal _ _ := false.
```

- An equational presentation of functions rather than a computational one: each "case" becomes an equation (a lemma in Coq)

```
equal_equation_1 : equal O O = true
...
equal_equation_4 : forall n m : nat, equal (S n) (S m) = equal n m
```

- Use equations to simplify the goal by automated rewriting with `simp`.
- Rewriting gives better control in the presence of dependent types.
- Equations support convenient definitions by well-founded recursion.
- Computational representation can be recovered through pattern-matching *compilation*.

Equations: pattern-matching and unification

- The Equations plugin builds a *case splitting tree*.
- The tree is built using *unification*.

Equations: pattern-matching and unification

- The Equations plugin builds a *case splitting tree*.
- The tree is built using *unification*.
- `pattern` $\equiv$ `expr` $\rightsquigarrow$ `result`
- Unify a pattern `S x` with `S (S O)`:
  "find out what `x` is if we know that `S x` = `S (S O)`"
- What is the answer?

Equations: pattern-matching and unification

- The Equations plugin builds a *case splitting tree*.
- The tree is built using *unification*.
- `pattern` $\equiv$ `expr` $\rightsquigarrow$ `result`
- Unify a pattern `S x` with `S (S O)`:
  "find out what `x` is if we know that `S x` = `S (S O)`"
- What is the answer?
- `S x` $\equiv$ `S (S O)` $\rightsquigarrow$ `Success [x:=S O]` (with a substitution `[x:=S O]`)

Equations: pattern-matching and unification

- The Equations plugin builds a *case splitting tree*.
- The tree is built using *unification*.
- `pattern` $\equiv$ `expr` $\rightsquigarrow$ `result`
- Unify a pattern `S x` with `S (S O)`:
  "find out what `x` is if we know that `S x` = `S (S O)`"
- What is the answer?
- `S x` $\equiv$ `S (S O)` $\rightsquigarrow$ `Success [x:=S O]` (with a substitution `[x:=S O]`)
- `S x` $\equiv$ `O` $\rightsquigarrow$ `Fail` (impossible to unify)
- `S x` $\equiv$ `m` $\rightsquigarrow$ `Stuck m` (we don't know what the variable `m` is)

Equations: pattern-matching and unification

- The Equations plugin builds a *case splitting tree*.
- The tree is built using *unification*.
- `pattern` ≡ `expr` ⇝ `result`
- Unify a pattern `S x` with `S (S O)`:
  "find out what `x` is if we know that `S x` = `S (S O)`"
- What is the answer?
- `S x` ≡ `S (S O)` ⇝ `Success [x:=S O]` (with a substitution [`x`:=`S O`])
- `S x` ≡ `O` ⇝ `Fail` (impossible to unify)
- `S x` ≡ `m` ⇝ `Stuck m` (we don't know what the variable `m` is)
- The unification algorithm can be formalised as a collection of inference rules.x

Equations: a splitting tree

Covering a signature:

build a splitting tree exhaustively covering all cases for input parameters.

Equations: a splitting tree

Covering a signature:

build a splitting tree exhaustively covering all cases for input parameters.

For our example program

```
Equations equal (m n : nat) : bool :=
equal O O := true ;
equal (S m') (S n') := equal m' n' ;
equal _ _ := false.
```

Signature to cover: `m n : nat`.

`cover(m n : nat ⊢ m n : (m n : nat))`

Equations: a splitting tree

Covering a signature:

build a splitting tree exhaustively covering all cases for input parameters.

For our example program

```
Equations equal (m n : nat) : bool :=
equal 0 0 := true ;
equal (S m') (S n') := equal m' n' ;
equal _ _ := false.
```

Signature to cover: `m n : nat`.

$\texttt{cover}(\texttt{m n} : \texttt{nat} \vdash \texttt{m n}) \to \texttt{0 0} \quad \equiv \quad \texttt{m n} \quad \rightsquigarrow \quad \texttt{Stuck m}$

Equations: a splitting tree

Covering a signature:

build a splitting tree exhaustively covering all cases for input parameters.

For our example program

```
Equations equal (m n : nat) : bool :=
equal O O := true ;
equal (S m') (S n') := equal m' n' ;
equal _ _ := false.
```

Signature to cover: `m n : nat`.

`Split(m n : nat ⊢ m n,` *m*`, [...])`

Equations: a splitting tree

Covering a signature:

build a splitting tree exhaustively covering all cases for input parameters.

For our example program

```
Equations equal (m n : nat) : bool :=
equal O O := true ;
equal (S m') (S n') := equal m' n' ;
equal _ _ := false.
```

Signature to cover: `m n : nat`.

```
Split(m n : nat ⊢ m n, m, [
  cover(n : nat ⊢ O n)
  cover(m' n : nat ⊢ (S m') n)])
```

Equations: a splitting tree

Covering a signature:

build a splitting tree exhaustively covering all cases for input parameters.

For our example program

```
Equations equal (m n : nat) : bool :=
equal 0 0 := true ;
equal (S m') (S n') := equal m' n' ;
equal _ _ := false.
```

Signature to cover: `m n : nat`.

```
Split(m n : nat ⊢ m n, m, [
  Split(n : nat ⊢ 0 n, n, [
    Compute(⊢ 0 0 ⇒ true),
    Compute(n' : nat ⊢ 0 (S n') ⇒ false)]),
  cover(m' n : nat ⊢ (S m') n)])
```

Equations: a splitting tree

Covering a signature:

build a splitting tree exhaustively covering all cases for input parameters.

For our example program

```
Equations equal (m n : nat) : bool :=
equal 0 0 := true ;
equal (S m') (S n') := equal m' n' ;
equal _ _ := false.
```

Signature to cover: `m n : nat`.

```
Split(m n : nat ⊢ m n, m, [
  Split(n : nat ⊢ 0 n, n, [
    Compute(⊢ 0 0 ⇒ true),
    Compute(n' : nat ⊢ 0 (S n') ⇒ false)]),
  Split(m' n : nat ⊢ (S m') n, n, [
    Compute(m' : nat ⊢ (S m') 0 ⇒ false),
    Compute(m' n' : nat ⊢ (S m') (S n') ⇒ equal m' n')])])
```

Equations: a splitting tree

Covering a signature:

build a splitting tree exhaustively covering all cases for input parameters.

For our example program

```
Equations equal (m n : nat) : bool :=
equal 0 0 := true ;
equal (S m') (S n') := equal m' n' ;
equal _ _ := false.
```

Signature to cover: `m n : nat`.

```
Split(m n : nat ⊢ m n, m, [
  Split(n : nat ⊢ 0 n, n, [
    Compute(⊢ 0 0 ⇒ true),
    Compute(n' : nat ⊢ 0 (S n') ⇒ false)]),
  Split(m' n : nat ⊢ (S m') n, n, [
    Compute(m' : nat ⊢ (S m') 0 ⇒ false),
    Compute(m' n' : nat ⊢ (S m') (S n') ⇒ equal m' n')])])
```

- Compile the splitting tree to `match ... with`
- If patterns overlap, the first match takes precedence.

Equations: dependent pattern-matching

The Equations plugin is particularly tailored towards programming with dependent types.

Equations: dependent pattern-matching

The Equations plugin is particularly tailored towards programming with dependent types.

Vectors: lists that keep track of the length in the type.

```
Inductive vector (A : Type) : nat → Type :=
| nil : vector A 0
| cons {n : nat} : A → vector A n → vector A (S n).
```

Equations: dependent pattern-matching

The Equations plugin is particularly tailored towards programming with dependent types.

Vectors: lists that keep track of the length in the type.

```
Inductive vector (A : Type) : nat → Type :=
| nil : vector A 0
| cons {n : nat} : A → vector A n → vector A (S n).
```

Taking a tail of a non-empty vector:

```
Equations tail {A n} (v : vector A (S n)) : vector A n :=
tail (cons _ v ) := v .
```

Why there is only one case in the definition?

Equations: dependent pattern-matching

The Equations plugin is particularly tailored towards programming with dependent types.

Vectors: lists that keep track of the length in the type.

```
Inductive vector (A : Type) : nat → Type :=
| nil : vector A 0
| cons {n : nat} : A → vector A n → vector A (S n).
```

Taking a tail of a non-empty vector:

```
Equations tail {A n} (v : vector A (S n)) : vector A n :=
tail (cons _ v ) := v .
```

Why there is only one case in the definition?

```
cover(A n v : vector A (S n) ⊢ A n v)
```

Equations: dependent pattern-matching

The Equations plugin is particularly tailored towards programming with dependent types.

Vectors: lists that keep track of the length in the type.

```
Inductive vector (A : Type) : nat → Type :=
| nil : vector A 0
| cons {n : nat} : A → vector A n → vector A (S n).
```

Taking a tail of a non-empty vector:

```
Equations tail {A n} (v : vector A (S n)) : vector A n :=
tail (cons _ v ) := v .
```

Why there is only one case in the definition?

```
Split(A n (v : vector A (S n)) ⊢ A n v, v, [
  (* the type of [nil] is not unifiable with the type of [v]: S n ≠ 0 *)
  vector A (S n)  ≡  vector A 0  ⇝  Fail;
  Compute(A n' a (v' : vector A n' ) ⊢ A n' (@cons ?(n') a v')  ⇒ v')])
```

The splitting tree contains only one `Compute` node: unification helps to determine the impossible cases.

DEMO

DEMO