Equations: a tool for dependent pattern-matching

Cyprien Mangin
cyprien.mangin@m4x.org

Matthieu Sozeau
matthieu.sozeau@inria.fr

Inria Paris & IRIF, Université Paris-Diderot

May 15, 2017

Outline

Outline

Setting

- Calculus of Inductive Constructions.
- Type families.
- Coq provides a simple and direct way to do pattern-matching.
- Not easy to program with dependent types.

Type families

```
Inductive vect (A : Type) : nat → Type :=
| vnil : vect A O
| vcons (n : nat) : A → vect A n → vect A (S n).
```

How do you write this?

```
Definition tail {A n} (v : vect A (S n)) : vect A n :=
match v with
| vcons _ v' ⇒ v'
| _ ⇒ ???
end.
```

Type families

```
Inductive vect (A : Type) : nat → Type :=
| vnil : vect A 0
| vcons (n : nat) : A → vect A n → vect A (S n).
```

```
Definition tail {A n} (v : vect A (S n)) : vect A n :=
match v with
| vcons _ v' ⇒ v'
end.
```

works.

Type families

```
Inductive vect (A : Type) : nat → Type :=
| vnil : vect A 0
| vcons (n : nat) : A → vect A n → vect A (S n).
```

But not

```
Definition dtail {A n} (v : vect A (S (S n))) : vect A n :=
match v with
| vcons _ (vcons _ v') ⇒ v'
end.
```

Main features:

- Function definition through a list of clauses.
- Generation of equations.
- Principle of functional elimination.
- Support for refinement, well-founded recursion.

A few more:

- Tactics for EQUATIONS support.
- Replacement for `dependent destruction`.
- Automatic derivation of various classes about inductive types.

Outline

Splitting tree

Most basic version of the splitting tree:

```
type context_map = context * pattern list * context

type splitting =
| Compute of context_map * term
| Split of context_map * int * splitting option array
```

- `Split` refers to the elimination of one variable.
- `Compute` refers to the right-hand side of a clause.

Build a splitting tree by eliminating variables until it covers every clause provided by the user.

Eliminating a variable

Consider a variable $(x : I\ \vec{u})$ in the context.

1. Generalize the variable.

To prove P, it is enough to prove:

$$\forall\ \vec{v}\ (y : I\ \vec{v}), (\vec{u}; x) = (\vec{v}; y) \to P$$

Eliminating a variable

Consider a variable $(x : I\ \vec{u})$ in the context.

1. Generalize the variable.

To prove P, it is enough to prove:

$$\forall\ \vec{v}\ (y : I\ \vec{v}), (\vec{u}; x) = (\vec{v}; y) \to P$$

2. Eliminate the fresh variable y.

This is easy because the variable and all its indices are fresh.

Eliminating a variable

Consider a variable $(x : I\ \vec{u})$ in the context.

1. Generalize the variable.

To prove P, it is enough to prove:

$$\forall\ \vec{v}\ (y : I\ \vec{v}), (\vec{u}; x) = (\vec{v}; y) \to P$$

2. Eliminate the fresh variable y.

This is easy because the variable and all its indices are fresh.

3. In each branch of the inductive type, simplify the generated equalities.

For instance, in `tail` we would simplify equalities like:

$$(\texttt{S n};\ \texttt{v}) = (\texttt{O};\ \texttt{vnil})$$
$$(\texttt{S n};\ \texttt{v}) = (\texttt{S n'};\ \texttt{vcons x v'})$$

Simplification steps

At each step, there is an equality $t = u$ at the head of the goal. We want to unify t and u. Five possible steps:

Simplification steps

At each step, there is an equality $t = u$ at the head of the goal. We want to unify t and u. Five possible steps:

1. Deletion: $\boxed{t = t}$

Remove the equality if possible, otherwise use K.

Simplification steps

At each step, there is an equality $t = u$ at the head of the goal. We want to unify t and u. Five possible steps:

1. Deletion: $\boxed{t = t}$

Remove the equality if possible, otherwise use K.

2. Solution: $\boxed{x = t}$ where x is a variable

Substitute t for x, strengthening variables as needed.

Simplification steps

At each step, there is an equality $t = u$ at the head of the goal. We want to unify t and u. Five possible steps:

1. Deletion: $\boxed{t = t}$

Remove the equality if possible, otherwise use K.

2. Solution: $\boxed{x = t}$ where x is a variable

Substitute t for x, strengthening variables as needed.

3. Injectivity: $\boxed{C\ \vec{u} = C\ \vec{v}}$

Deduce that $\vec{u} = \vec{v}$.

Simplification steps

At each step, there is an equality $t = u$ at the head of the goal. We want to unify t and u. Five possible steps:

1. Deletion: $\boxed{t = t}$

Remove the equality if possible, otherwise use K.

2. Solution: $\boxed{x = t}$ where x is a variable

Substitute t for x, strengthening variables as needed.

3. Injectivity: $\boxed{C\ \vec{u} = C\ \vec{v}}$

Deduce that $\vec{u} = \vec{v}$.

4. Conflict: $\boxed{C\ \vec{u} = D\ \vec{v}}$
5. No cycle: $\boxed{t = C\ \vec{u}[t]}$

Solve the goal immediately.

Simplification steps

At each step, there is an equality $t = u$ at the head of the goal. We want to unify t and u. Five possible steps:

1. Deletion: $\boxed{t = t}$

Remove the equality if possible, otherwise use K.

2. Solution: $\boxed{x = t}$ where x is a variable

Substitute t for x, strengthening variables as needed.

3. Injectivity: $\boxed{C\ \vec{u} = C\ \vec{v}}$

Deduce that $\vec{u} = \vec{v}$.

4. Conflict: $\boxed{C\ \vec{u} = D\ \vec{v}}$
5. No cycle: $\boxed{t = C\ \vec{u}[t]}$ Not yet implemented in EQUATIONS

Solve the goal immediately.

Compilation of the splitting tree

Since we cannot modify pattern-matching in itself, we need to produce a term that will be accepted by COQ to witness the context and goal changes related to these simplification steps.

- Using tactics: easy enough to implement, but risk of coherence problems.
- Writing "manually" the terms: more work to implement, but precise.

A word about refinement

```
Equations unzip {A B n} (v : vect (A * B) n) : vect A n * vect B n :=
unzip vnil := (vnil, vnil) ;
unzip (vcons (pair a b) v) ⇐ unzip v ⇒ {
  | pair v w ⇒ (vcons a v, vcons b w)
}.
```

In this case, the right-hand side is not a `Compute` node. Instead we:

- typecheck the term `unzip v` under the current (and possibly refined) context;
- add a pattern in the current `context_map`;
- process the rest of this node to produce an auxiliary definition;
- apply this auxiliary definition to the current variables and the term which is refined.

A word about refinement

```
Equations unzip {A B n} (v : vect (A * B) n) : vect A n * vect B n :=
unzip vnil := (vnil, vnil) ;
unzip (vcons (pair a b) v) <= unzip v => {
  unzip (vcons (pair a b) _) (pair v w) => (vcons a v, vcons b w)
}.
```

In this case, the right-hand side is not a `Compute` node. Instead we:

- typecheck the term `unzip v` under the current (and possibly refined) context;
- add a pattern in the current `context_map`;
- process the rest of this node to produce an auxiliary definition;
- apply this auxiliary definition to the current variables and the term which is refined.

Outline

Local definition (`where` keyword)

- Similar to a let-in.
- Provide a definition through a splitting tree, as usual.
- Possible to combine it with well-founded recursion to obtain nested or mutual recursion.

Less axioms

Proof irrelevance was used to prove the fixpoint lemmas about well-founded recursion. We avoid it by proving it directly for the accessibility relation.
Additionally, a lot of work about the axiom K...

From heterogeneous to homogeneous equalities

When we generalize a variable $(x : I\ \vec{u})$, we introduce equalities. Before, we used heterogeneous equalities where needed.

- Easy to manipulate (less dependency between equalities).
- Entails the use of the axiom K.

Now we use homogeneous equalities between telescopes.

- Have to be careful because each equality depends on the previous ones.
- The use of the rule K is targeted to a specific type.

Making terms nicer

Pattern-matching in Coq can do part of our work to make terms look nicer.

```
match x as x' in I u' return P u' x' with
| C y ⇒ ...
| D z ⇒ ...
end
```

In each branch, `u'` and `x'` are instantiated with the actuel indices and constructor. This corresponds to a solution step.

- Try to solve as many solution steps as possible through this mechanism.
- Might need to introduce cuts to compensate.

Making terms nicer

Pattern-matching in Coq can do part of our work to make terms look nicer.

```
match x as x' in I u' return forall (a : T u'), P u' x' with
| C y ⇒ ...
| D z ⇒ ...
end a
```

In each branch, u' and x' are instantiated with the actuel indices and constructor. This corresponds to a solution step.

- Try to solve as many solution steps as possible through this mechanism.
- Might need to introduce cuts to compensate.

Replacing Function?

- Proving the correctness of the functional graph.
  For now we only need `forall x, f_ind x (f x)` to derive the functional elimination principle. Function also proves `forall x y, f_ind x y → y = f x`.
- Tracking default cases.
  Equations fully expands its splitting tree, and therefore loses track of clauses that would cover several constructors at once.
- Keeping the same surface syntax.
  Ideally, the user would not see any change of any code written with Function, only the underlying code would branch out to Equations.

Conclusion

EQUATIONS was already used succesfully for a few applications:

- Normalization of LF.
- Consistency of predicative System F.
- Reflexive tactic to decide equality of polynomials.

EQUATIONS is available on GitHub [1] and OPAM.
It is still in an experimental state, and not all features discussed here are available in 8.6.

[1] `https://github.com/mattam82/coq-equations`