Equations: a tool for dependent pattern-matching

Cyprien Mangin
IRIF, Univ. Paris Diderot
Paris, France
cyprien.mangin@m4x.org

Matthieu Sozeau
Inria Paris & IRIF, Univ. Paris Diderot
Paris, France
matthieu.sozeau@inria.fr

Abstract

EQUATIONS[1] (Sozeau 2010) is a toolbox built as a plugin for the COQ proof assistant which adds some capabilities for writing dependently-typed programs. As a main feature, it is capable to compile a high-level specification of a function to a pure COQ term. The specification is given as a list of pattern-matching clauses, similar to what is possible in Agda (Norell 2007) or Lean (de Moura et al. 2015). These clauses can be arbitrarily dependent, and EQUATIONS also enables the use of complex recursion schemes to define recursive functions whose termination is not structural.

The compilation scheme used by EQUATIONS relies on the work of Goguen et al. (Goguen et al. 2006) In essence, dependent pattern-matching is compiled away by generalizing through the use of equality the term being pattern-matched on, and the simplification of the ensuing equalities. In addition to this, EQUATIONS derives the propositional equalities between both sides of a pattern-matching clause, allowing the user to simplify a function without repeating the simplification steps and abstracting from the actual encoding of the function (e.g. in which order pattern-matching is performed, or how simplification of equalities is done).

EQUATIONS also derives automatically a functional elimination scheme, which lets the user reason directly on the final branches of the function, instead of doing the usual induction on its arguments. This allows for much shorter and natural proofs when establishing some fact about a function, since the user does not have to do again the splitting, or any reasoning that intervened during the dependent pattern-matching itself. The corresponding proof-terms are also much smaller.

Finally, EQUATIONS comes with some tools to reason about dependently-typed terms, even if they are not defined with EQUATIONS. For instance, the tactic `depelim` is a more complete replacement for the built-in tactic `dependent destruction` when it comes to eliminating dependent terms. EQUATIONS also allows to derive several utilities about inductive families, for instance: decidable equality, a signature to pack a term in an inductive type with its indices, a well-founded subterm relationship on the elements of the inductive type (for computational inductive families, it mimicks the implicit subterm relationship used by the syntactic guardedness criterion) and a principle of "no confusion" (McBride 2000) which subsumes injectivity and disjointness of its constructors.

Keywords COQ, dependent pattern-matching, proof assistants

The aim of this presentation is to first introduce what EQUATIONS is and underline its main features, as well as give some small examples to give a grasp of the syntax itself, and the use of the additional features such as functional elimination or dependent elimination of a variable.

Then we will talk about the recent improvements that have been brought to EQUATIONS. These are mainly:

- the transition from a translation based on heterogeneous equalities, as it was the case in the original work by Goguen et al., to a translated built on a homogeneous equality between telescopes of terms, as was explored by J. Cockx (Cockx et al. 2014);
- the extension of the syntax of functions defined with EQUATIONS to include a `where` keyword, which essentially acts as a let-in. This allows the definition of nested recursive functions using logical recursion, for instance;
- some work has been done to reduce the use of axioms in general, such as propositional irrelevance or a more precise decidable equality.

Now we will explain in more details these improvements and how they can be used.

From heterogeneous to homogeneous equalities

In a first iteration, EQUATIONS was basically a textbook implementation of the work of Goguen et al., using heterogeneous equality as a way to express identity between indices whose type involve other indices. Heterogeneous equality, also known as John Major equality (McBride 2000), is useful because it is a simple way to convey what we need without worrying about the type of these indices. It has, however, the undesirable drawback of forcing the use of an axiom equvalent to the axiom K when we need to simplify such an equality.

This is fine in some settings, but EQUATIONS also aims to be used in the context of the recent field of homotopy type theory (The Univalent Foundations Program 2013), which often admits the univalence axiom, incompatible with the axiom K. Axiom K is still useful or even necessary to have in some cases, at least on some types. Therefore, we need a finer-grained control of its use, instead of requiring it globally. The use of heterogeneous equalities would indeed simply require K on `Type`, which is exactly what we want to avoid.

One solution is to replace a list of heterogeneous equalities, where the type of the sides of an equality might depend on the previous one, by one single equality between telescopes of the same type, which are simply nested dependent pairs. We can then simplify this homogeneous equality in a similar way, and only require the use of the axiom K on some of the elements of these telescopes, instead of globally.

Currently, this is implemented, and while the heterogeneous equalities are still what is used by default – pattern-matching is more expressive if one does not mind the axiom K – the user can switch easily to the version which uses homogeneous equality and specific instances of K instead.

Using `where`

A less heavy change is the addition of the `where` keyword to the syntax of EQUATIONS. In some cases, we need some kind of nested

[1] `http://github.com/mattam82/Coq-Equations`

dependent pattern-matching. This is exactly what `where` allows to do by providing a let-in which will capture the local context.

A simple example is working with rose trees, which have the following definition:

```
Variable (A : Set).
Inductive rose_tree : Set :=
| leaf : A → rose_tree
| node : list rose_tree → rose_tree.
```

The more natural way to express what happens in the `node` case is to use nested recursion, which the `where` keyword allows to use. For instance, if we want to write a function which returns a list of elements inside the rose tree:

```
Equations elements (t : rose_tree) : list A :=
elements t by rec t (MR lt size) :=
elements (leaf a) := cons a nil;
elements (node l) := aux l _

where aux (x : list rose_tree)
  (H : list_size size x < size (node l)) : list A :=
aux x H by rec x (MR lt (list_size size)) :=
aux nil _ := nil;
aux (cons t x) H := elements t ++ aux x _.
```

Notice that we explicitely use a well-founded relation on rose trees and on lists of rose trees to define this function, through the use of the `by rec` clause. For instance, `elements` is defined by recursion on its argument `t`, which will be decreasing according to the relation `MR lt size`, the relation induced by the size of a tree.

We will need to fulfill some proofs here to justify the recursion, but this is easily done thanks to the obligations mechanism of Coq.

Note that even in this case, Equations is still able to derive the usual lemmas and principles about this function, like the equations between the left- and right-hand sides of the clauses and a functional elimination principle. The extracted code for this function simply does nested recursion and removes the `H` argument of `aux` which is only used to justify recursive calls. It is also easy to use functional elimination to define an additional `aux` definition without this assumption and show its equivalence with the original one.

Towards less axioms

The axiom K is not the only axiom which is useful to reason about dependent pattern-matching combined with functional elimination and other features of Equations. We also work on reducing or controlling better the use of these other axioms.

For example, we need some kind of proof irrelevance to prove the fixpoint lemmas about function defined using a general recursor – and not the structural recursion. Instead of admitting it globally, we just prove it for the inductive type representing accessibility for a relation in Coq, and work from there. This only requires functional extensionality, which is not only unavoidable, but also more widely accepted and used than propositional irrelevance.

Another example is the use of decidable equality to prove uniqueness of identity proofs (UIP/K) on some type. To construct these UIP proofs we actually only rely on pointed decidable equality, which is a slight restriction over decidable equality on the whole type.

A small example

To have another grasp of the syntax of Equations, here is a short example that demonstrates the basic use of Equations and functional elimination.

```
Inductive vect : nat → Set :=
| nil : vect 0
| cons : forall (n : nat), A → vect n → vect (S n).
```

```
Equations dest {n : nat} (v : vect (S n)) :
  (A * vect n) :=
dest (cons x v) := (x, v).

Goal forall (n : nat) (v : vect (S n)),
  cons (fst (dest v)) (snd (dest v)) = v.
Proof.
  intros.
  funelim (dest v).
  reflexivity.
Qed.
```

The use of `funelim` will automatically discard the impossible `nil` case, just like we did not need to provide anything for this branch while defining the function `dest`. Of course, on such a simple example, it would be still very easy to do in pure Coq, but Equations will work for arbitrarily deep and complicated definitions.

Applications and future work

Equations has been successfully used in some developments.

We used it to prove the consistency of Leivant's Predicative System F through hereditary substitution (Mangin and Sozeau 2015). It allowed to write in a very natural way a constructive normalization function for this language. The examples provided with Equations also include a proof of normalization for LF, and an example reflexive tactic to decide equality of polynomials by R. Bocquet. In this formalization, originally a course project, the representation of a polynomial is indexed by its number of free variables and a boolean indicating if it is null. This was the shortest and arguably the cleanest submission for this project.

Equations has of course more room for improvement. First of all, the current features need more polish. At the time of the writing of this abstract, Equations is released as version 0.9 on GitHub and OPAM. We plan to release the first stable version shortly.

References

J. Cockx, D. Devriese, and F. Piessens. Pattern matching without k. *SIGPLAN Not.*, 49(9):257–268, Aug. 2014.

L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer. *The Lean Theorem Prover (System Description)*, pages 378–388. Springer International Publishing, Cham, 2015. ISBN 978-3-319-21401-6.

H. Goguen, C. McBride, and J. McKinna. Eliminating Dependent Pattern Matching. In K. Futatsugi, J.-P. Jouannaud, and J. Meseguer, editors, *Essays Dedicated to Joseph A. Goguen*, volume 4060 of *Lecture Notes in Computer Science*, pages 521–540. Springer, 2006. ISBN 3-540-35462-X.

C. Mangin and M. Sozeau. Equations for Hereditary Substitution in Leivant's Predicative System F: A Case Study. In *Proceedings Tenth International Workshop on Logical Frameworks and Meta Languages: Theory and Practice*, volume 185 of *EPTCS*, May 2015. LFMTP'15.

C. McBride. Elimination with a motive. In *International Workshop on Types for Proofs and Programs*, pages 197–216. Springer, 2000.

U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

M. Sozeau. Equations: A dependent pattern-matching compiler. In *First International Conference on Interactive Theorem Proving*. Springer, July 2010.

The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations for Mathematics*. Institute for Advanced Study, 2013.