Automatic refinements in Coq
Internship report

Damien Rouhling

January - June 2016

Abstract

This report is the result of my internship in the Marelle team at INRIA Méditerranée Sophia Antipolis, in France, for the validation of my second year of masters in computer science. It is about the (re-)development of CoqEAL, a library for effective algebra in the proof-assistant Coq. This library exploits the concept of refinement to reintroduce computation in proofs. During this internship, I implemented in CoqEAL a better use of parametricity in order to automatize the refinement process.

Contents

Introduction

I did my second year of masters' internship under the supervision of Cyril Cohen, in the Marelle team at INRIA Méditerranée Sophia Antipolis, in France. I worked on CoqEAL ([DMS12]), a library for effective algebra in the proof-assistant Coq ([dt16]).

There are two main approaches to computer-aided reasoning: proof search and proof verification. Proof-assistants fall into the second category, even if nowadays most proof-assistants embed automatic procedures that help the user to build proofs, thus making some features of the first category available. They are interactive tools that allow the user to give the steps of a proof and that automatize the verification of their validity.

One of the interests of proof-assistant is thus to make proofs more reliable through systematic checking. In the particular case of mathematics, one goal of the Mathematical Components project is to provide a library of mathematical results that are efficiently reusable in the proof-assistant Coq, with a good trade-off between abstraction and ease of use. This project and the extension of Coq's tactic language, SSReflect ([GMT15]), have shown their success through the proofs of the Four Colour Theorem ([Gon08]) and of the Odd Order Theorem ([GAA+13]).

CoqEAL is built on top of the Mathematical Components library. It contains algebraic results and provides a framework for changing the representation of objects in the middle of proofs. This has the interest of allowing to perform efficient computations to ease some proofs. For this purpose, CoqEAL comes with certified optimized algorithms. To prove the correctness of those algorithms on the data structures defined in the library, three steps are used: write the algorithm with the data structure *as a parameter* (this is called generic programming), prove its correctness on a "proof-oriented" data structure such as those of the Mathematical Components library and finally transport the proof to any other (usually "computation-oriented") data structure satisfying the same interface/properties.

The last step is realised through data refinement. "Refinement" is a term usually used to describe a step by step approach in the verification of a program. The program and its specification are first specified using high-level languages and then the program is refined to more low-level structures, finally reaching an actual implementation. Each step is proven correct with respect to the previous one. What we mean by data refinement here is only a verified step for changing the representation of some data type.

Data refinement has a strong connection with Reynolds' abstraction theorem ([Rey84]). This result allows us to get "free theorems" ([Wad89]) that, in our case, help to automatize the third step. The goal of my internship was to exploit this link in order to (re-)develop CoqEAL [1]. We give further details on the context of this internship in Section 1 before explaining in Section 2 how Reynolds' result can be of practical use in our case. Section 3 gathers technical details on the refinement process. Finally, we show applications of this work in Section 4.

1 Context

CoqEAL is a library which contains efficient data structures and optimized algorithms. One of its use is to perform *efficient* computations in the middle of proofs to simplify the property that has to be proven. For this, there is a mechanism for changing the representation of the manipulated objects, namely refinement. We first discuss the reasons for this mechanism. Then, we present the methodology of refinement and finally we illustrate it on an example.

[1] https://github.com/CoqEAL/CoqEAL/tree/paramcoq-dev

1.1 On proofs and computation

Coq is a proof-assistant based on the Calculus of Inductive Constructions ([CH88], [CPM90]). It exploits the correspondence between lambda calculus and logic, also known as the Curry-Howard correspondence: properties are interpreted as types and proofs as lambda terms. In Coq, to prove a property the user has to build a term inhabiting the corresponding type. The role of Coq is to check that the constructed term is well-formed and has indeed the desired type.

In Coq, the user focuses on the property (the type) (s)he wants to prove. For this, (s)he can use tactics (i.e. routines building part of the structure of the proof term) to transform it, to break it into smaller part, to reduce it to other properties, etc. Thus, it is not necessary to manipulate big unreadable proof terms. On top of this, lambda calculus has a computational model. Hence, terms are valuable, for they also are programs one might want to execute. Coq allows the user to print the terms but has also an extraction mechanism to get code from functions or existence proofs. This makes Coq a powerful tool for program certification: the user can either write a function in Coq and prove it correct by working on a specification theorem, or directly work on a proof that there exists a term which satisfies a given specification. Finally, (s)he can extract a functional program out of this work.

However, proving that a program is correct can be hard, especially if it involves complex optimizations on specialized data structures. Dijkstra ([Dij82]) argued for studying both aspects, correctness and efficiency, separately:

> We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day, so to speak. [...] But nothing is gained – on the contrary! – by tackling these various aspects simultaneously. It is what I sometimes have called "the separation of concerns", [...].

In CoqEAL, we achieve this concretely by proving the correctness of an algorithm on "proof-oriented" data structures and by transporting the proof to more "computation-oriented" data structures. This second step is achieved through refinement ([DMS12]).

This separation between proofs and computation is reinforced in the Mathematical Components library, where the data structures are well-fitted for proofs but not for computation. Indeed, some of them are *locked* in order to block computation and thus to make type-checking more efficient on complex structures. Nevertheless, it is sometimes interesting to perform computation in order to simplify the property to be proven. Even if it is possible to do so by rewriting equalities, it is extremely tedious and can be really inefficient, for instance if you have to unfold a determinant using its definition instead of an optimized algorithm. Theoretically, one could tune the system to remove locks in order to perform computations, but still the data structures and algorithms are not optimized for computation, hence inefficient. Thus, it could be interesting during a proof to alternate between "proof-oriented" and "computation-oriented" representations of the objects (data refinement) and/or functions (program refinement).

1.2 Refinement methodology

Program refinement consists in replacing an algorithm with a different one that computes more efficiently the same thing. For instance, to compute a determinant it is better to use Bareiss' algorithm ([Bar68]) instead of the formula that defines it. No change in the data structures is involved and the correctness of such refinement often amounts to proving that the two algorithms are extensionally equal. Two programs P and P' are extensionally equal if for all input x the outputs $P\,(x)$ and $P'\,(x)$ are equal. In Coq, proving

such a theorem allows you to rewrite every instance of the pattern $P(x)$ as $P'(x)$, and conversely.

Data refinement, however, is more complex. It involves putting in correspondence the base objects of two different data representations (e.g. the null polynomial of the Mathematical Components library with the empty sequence, if you represent polynomials as the sequences of their coefficients) as well as the basic operations on them (e.g. the sum of two polynomials should correspond to the pointwise sum of the two sequences that represent them). It should be possible to compose multiple refinements. For instance, refinements from dense to sparse polynomials and from unary to binary integers should give "for free" a refinement from dense polynomials over unary integers to sparse polynomials over binary integers.

The methodology used in CoqEAL for data refinement is the following ([CDM13]):

1. parametrize an algorithm by the data it manipulates using an abstract type and abstract basic operations,
2. prove the correctness of the algorithm when it is instantiated on a "proof-oriented" representation of the data,
3. use the parametricity of the algorithm to deduce the correctness of the algorithm when it is instantiated on a corresponding "computation-oriented" representation of the data.

Remark that this methodology also applies to data structures, instead of algorithms. This is how one can compose refinements: for instance, in the case of polynomials seen as sequences, we parametrize the base objects/operations by the representation of the coefficients.

This approach is similar to the one that is used in the Isabelle/HOL code generator ([Lam13]). However, in Isabelle/HOL, the goal is to generate executable versions of abstract programs, whereas Coq programs are already executable thanks to the extraction mechanism, so that we only use refinement to perform computations more efficiently. The approach in Isabelle/HOL goes from abstract objects to concrete ones and is called pure data refinement. In CoqEAL, refinement is performed either on *instantiations* of an abstract program or on concrete structures, so that we never leave the concrete world.

1.3 Example

We illustrate this methodology on the example of rational numbers, taken from the paper of Cohen, Dénès and Mörtberg ([CDM13]). The actual refinement of rational numbers in CoqEAL is a bit more complex but we stick to the simpler version for a better understanding. In the Mathematical Components library, rational numbers are represented as a record containing a pair of integers and a proof that they are coprime and that the second one is positive:

```
Record rat : Set := Rat {
  valq : int * int ;
  _ : (0 < valq.2) && coprime `|valq.1| `|valq.2|
}.
```

A first step toward a "computation-oriented" version of rational numbers is to remove the proofs and to keep only the pair of integers. Another possible optimization is to avoid computing the coprime representation of the number at each operation (still, one has to be careful with the size of the involved integers). Furthermore, since the type `int` of integers in the Mathematical Components library is "proof-oriented" we can compose this refinement with a refinement of integers. This leads to the following definition of addition on rational numbers, where the type `Z` is meant to represent integers:

```
Definition addQ Z +Z *Z : Z * Z -> Z * Z -> Z * Z :=
  fun x y => (x.1 *Z y.2 +Z y.1 *Z x.2, x.2 *Z y.2).
```

The next step is to prove that this addition is correct when the argument `Z` is instantiated with `int`. For this, we use a relation

```
Rrat : rat -> int * int -> Type
```

that expresses the fact that an object of type `rat` and a pair of `int` objects represent the same rational number. We want to show that addition *preserves* in some sense this relation. To achieve this, we transport `Rrat` to the level of functions: given two relations

```
R : A -> B -> Type,
R' : A' -> B' -> Type,
```

it is possible to build a functional relation

```
R ==> R' : (A -> A') -> (B -> B') -> Type
```

so that two functions are related if they send related inputs to related outputs. This is a heterogeneous generalization of the `respectful` function defined for generalized rewriting ([Soz09]). Using this construction and the relation `Rrat`, we can express the correctness of `addQ` as follows:

```
Lemma Rrat_addQ : (Rrat ==> Rrat ==> Rrat) +rat (addQ int +int *int).
```

The final step is to compose this result with a refinement of the type `int`, thus proving the correctness of `addQ` on a fully "computation-oriented" representation of rational numbers. Assume given a "computation-oriented" version `C` of `int`, a relation

```
Rint : int -> C -> Type
```

and the correctness theorems for addition and multiplication:

```
Lemma Rint_add : (Rint ==> Rint ==> Rint) +int +C,
Lemma Rint_mul : (Rint ==> Rint ==> Rint) *int *C.
```

We can then define the relation `RratC : rat -> C * C -> Type` as

```
Definition RratC := Rrat o (Rint * Rint),
```

where `o` is relation composition, and `R * R'` denotes the product of the two relations `R` and `R'`. As we will see, this is this composition that allows us to use the parametricity of `addQ` in order to prove its correctness (in fact, it is not specific to this example). The goal is to prove the following theorem:

```
Lemma RratC_add : (RratC ==> RratC ==> RratC) +rat (addQ C +C *C).
```

Because of the composition in the definition of `RratC`, this goal splits into:

```
(Rrat ==> Rrat ==> Rrat) +rat (addQ int +int *int)
```

and

```
(Rint * Rint ==> Rint * Rint ==> Rint * Rint)
  (addQ int +int *int) (addQ C +C *C).
```

The first goal is exactly the statement of the correctness theorem `Rrat_addQ`, and the proof of the second one can be automatized using parametricity and the correctness theorems for addition and multiplication `Rint_add` and `Rint_mul`. A goal of my internship was to realize this automation.

2 The role of parametricity

Parametricity ([Wad89]) is a reformulation of Reynolds' abstraction theorem ([Rey83]). It is based on the idea that all inhabitants of a (closed) type share a property expressed by its relational interpretation. We first describe this interpretation and state the parametricity theorem. Then we show how it can be used in CoqEAL.

2.1 The parametricity theorem

Reynolds introduced a relational interpretation of types in a work about polymorphism ([Rey83]). It is expressed in terms of a set-theoretic model of the polymorphic lambda calculus that in fact does not exist ([Rey84]). However, Wadler transposed it to another context ([Wad89]), where models actually exist. For the sake of simplicity, we stick to the set-theoretic view to give the intuition of the abstraction theorem.

For each type A, we denote by $[\![A]\!]$ its interpretation as a relation. For instance, the constant type for integers `int` can be interpreted as the diagonal $\Delta_{\texttt{int}} = \{(x, x) \mid x \in \texttt{int}\}$. For function types, we use the same idea as in Subsection 1.3 to define a functional relation: two functions are related if they send related inputs to related outputs. Keeping our notations, this gives $[\![A \to B]\!] = [\![A]\!] \texttt{==>} [\![B]\!]$. The most important case is the one of the polymorphic type $\forall X.A$. Since X is a type variable, the interpretation of A will depend on a relation, which corresponds to the interpretation of X. Thus, two functions f and g are related by $[\![\forall X.A]\!]$, denoted by $[\![\forall X.A]\!]\ f\ g$, if *for all relation* R between some sets (types) S and T, f_S and g_T are related by $[\![A]\!]\{R/[\![X]\!]\}$, where f_S is the instantiation of f for the type $A\{S/X\}$. Thus, any closed type A can be seen as a relation $[\![A]\!]$ on itself. Parametricity for A can then be expressed as follows: for all closed term t of type A, $[\![A]\!]\ t\ t$.

In a less naive context, terms also need to be interpreted and parametricity also holds for terms and types that contain free variables, so long as these variables respect the relations corresponding to their types. However, in CoqEAL we use parametricity only on closed terms so we will not express the theorem in its full generality. If we denote by $\vdash t : A$ the typing judgement "the term t is of type A", we can state parametricity as follows:

Theorem 1 (Parametricity)**.** *For all closed type A and closed term t, if $\vdash t : A$, then $\vdash [\![t]\!] : [\![A]\!]\ t\ t$.*

Let us look at an example of "free theorem" derived from parametricity, taken from the paper of Wadler ([Wad89]). Consider *any* polymorphic function f over sequences: the type of f is $\forall X.\texttt{seq}\ X \to \texttt{seq}\ X$. The type variable X represents here the type of the coefficients of the sequence. Since f works on any sequence, no matter what the type of the coefficients is, f cannot use any type-specific operation: its effect must be independent of the values of the coefficients, hence f must rearrange the sequence, possibly deleting some coefficients depending only on their position in the sequence. By parametricity we know that $(f, f) \in [\![\forall X.\texttt{seq}\ X \to \texttt{seq}\ X]\!]$. Unfolding the definitions, we get that for all relation R between some A and A', for all $(l, l') \in [\![\texttt{seq}\ X]\!]\{R/[\![X]\!]\}$, we have $(f_A\ l, f_{A'}\ l') \in [\![\texttt{seq}\ X]\!]\{R/[\![X]\!]\}$. Using *any* function g from A to A' for R, we get that for all sequences l and l', if $\texttt{map}\ g\ l = l'$, then $\texttt{map}\ g\ (f_A\ l) = f_{A'}\ l'$. To sum up, *any* polymorphic function over sequences commutes with the mapping of *any* function. Seeing such functions as rearrangements, this "free theorem" is not quite a revolution. However, it has proven useful in CoqEAL for one of our refinements, as well as many other such "free theorems".

2.2 Data refinements as "free theorems"

Using relations to express the correctness of a data refinement, as in Subsection 1.3, puts us in a framework that is really close to the one of parametricity. Indeed, we have seen in Subsection 2.1 that the interpretations of polymorphic types are quantified over some relations. By instantiating those relations by ours, we can get "for free" parts of the correctness theorems we try to prove.

Reynolds' result is stated for a simple calculus, but it can be extended to more complex systems. To do so, it is sufficient to give an interpretation to the supplementary type constructors, and to prove that the new constants satisfy the interpretation of their types. This has been done for the Calculus of Inductive Constructions by Keller and Lasson ([KL12]), thus making parametricity usable in our context.

Recall the methodology for data refinement we presented in Subsection 1.2 and consider again the example of rational numbers, seen in Subsection 1.3. We first parametrized our data structure by a type representing integers, thus obtaining the type $\forall$`Z. Z * Z`. Then, we expressed the fact that this data structure represents the type `rat` and that `addQ` is correct, when `Z` is instantiated by `int`, using the relation `Rrat`. Finally, we defined the relation `RratC` and expressed the fact that this is also correct to instantiate `Z` by a refinement `C` of `int`. In fact, `RratC` is the composition `Rrat o` $[\![$`Z * Z`$]\!]\{$`Rint`$/[\![$`Z`$]\!]\}$, thus paving the way for the use of parametricity. Indeed, we know that $[\![$`addQ`$]\!]$ is a proof of

$[\![\forall$`Z`$.\,($`Z -> Z -> Z`$) \to ($`Z -> Z -> Z`$) \to$ `Z * Z -> Z * Z -> Z * Z`$]\!]$ `addQ addQ`

which expands to

```
∀Z : Type. ∀Z' : Type. ∀R : Z -> Z' -> Type.
∀addZ : Z -> Z -> Z. ∀addZ' : Z' -> Z' -> Z'.
[[Z → Z → Z]]{R/[[Z]]} addZ addZ' ->
  ∀mulZ : Z -> Z -> Z. ∀mulZ' : Z' -> Z' -> Z'.
  [[Z → Z → Z]]{R/[[Z]]} mulZ mulZ' ->
    [[Z * Z → Z * Z → Z * Z]]{R/[[Z]]} (addQ Z addZ mulZ) (addQ Z' addZ' mulZ').
```

This, with our notations, simplifies into

```
∀Z : Type. ∀Z' : Type. ∀R : Z -> Z' -> Type.
∀addZ : Z -> Z -> Z. ∀addZ' : Z' -> Z' -> Z'.
(R ==> R ==> R) addZ addZ' ->
  ∀mulZ : Z -> Z -> Z. ∀mulZ' : Z' -> Z' -> Z'.
  (R ==> R ==> R) mulZ mulZ' ->
    (R * R ==> R * R ==> R * R)
      (addQ Z addZ mulZ) (addQ Z' addZ' mulZ').
```

Choosing `int` for `Z`, `C` for `Z'` and `Rint` for `R` (as in the definition of `RratC`), this becomes

```
∀addInt : int -> int -> int. ∀addC : C -> C -> C.
(Rint ==> Rint ==> Rint) addInt addC ->
  ∀mulInt : int -> int -> int. ∀mulC : C -> C -> C.
  (Rint ==> Rint ==> Rint) mulInt mulC ->
    (Rint * Rint ==> Rint * Rint ==> Rint * Rint)
      (addQ int addInt mulInt) (addQ C addC mulC).
```

Finally, using the two theorems

```
Lemma Rint_add : (Rint ==> Rint ==> Rint) +int +C,
Lemma Rint_mul : (Rint ==> Rint ==> Rint) *int *C,
```

we have a proof of

```
(Rint * Rint ==> Rint * Rint ==> Rint * Rint)
  (addQ int +int *int) (addQ C +C *C),
```

which is exactly the result that we were missing in Subsection 1.3.

Thus, the automation of data refinement is only a matter of applying a "free theorem" obtained from parametricity. What is more, Keller and Lasson have implemented ParamCoq [2], a plugin for Coq that computes the interpretation of a term t and of its type A, and that lets Coq's type checker verify that indeed $[\![t]\!]$ is a proof of $[\![A]\!]\ t\ t$. This plugin is the starting point of my work. First, I slightly adapted its code [3] to make it compile with the latest version of Coq. Later on, I corrected its naming system with the possibility of giving user-defined names to interpretations to avoid conflicts when declaring the interpretation of two different terms with the same name (but defined in different modules). What was left was proving refinements using theorems generated by this plugin.

2.3 On partial refinements

When several data types are involved, one might want to keep intact those that are efficient for computation and to refine the others. Currently, this is only possible through the use of SSReflect's extended `rewrite` tactic ([GMT15]), by giving patterns for the subterms that need to be refined (see Subsection 3.3 more details on refinement by rewriting).

Sometimes, however, even the data types we want to refine embed other ones that we want to keep intact. For instance, a natural way of representing matrices is by a type depending on two natural numbers, the dimensions: an abstract representation would thus be the type `∀n : N. ∀m : N. M n m`, where `N` is meant to represent natural numbers and `M n m` is the type of the matrices with `n` rows and `m` columns. In practice, one rarely needs to compute on the dimensions. Hence, it is still reasonable to use the type `nat` for `N` and to use refinement only on `M`.

If abstract types (introduced through quantification) are interpreted as quantifications over relations, concrete types have their own specific and more concrete interpretation. For example, we have seen the type constructor for pairs, and its interpretation as the product relation of the interpretations of the types of its components. In the case of the type `nat`, the ParamCoq plugin outputs the following interpretation:

```
Inductive nat_R : nat -> nat -> Set :=
    nat_R_O_R : nat_R 0 0
  | nat_R_S_R : forall n m : nat, nat_R n m -> nat_R n.+1 m.+1.
```

Here the notation `.+1` corresponds to the successor function, and `nat_R` is ParamCoq's notation for $[\![\texttt{nat}]\!]$. This relation appears in the parametricity theorems involving our type for matrices. Indeed, the quantification `∀n : nat. ∀m : nat. ...` is interpreted as

```
∀n1 : nat. ∀n2 : nat. [[nat]] n1 n2 ->
  ∀m1 : nat. ∀m2 : nat. [[nat]] m1 m2 -> ...
```

Consequently, in order to be able to use parametricity in refinements of objects parametrized by a representation of matrices, we need to prove refinement theorems with such quantifications. On the other hand, when using in practice refinement on matrices, it is sufficient to have theorems with the quantification over the dimensions `∀n : nat. ∀m : nat. ...`, since we want to keep them intact. The parametricity theorem ensures that $[\![\texttt{nat}]\!]$ is reflexive, and it is easy to see that in fact `nat_R` corresponds to equality. Hence, the theorems with the simpler quantification are direct consequences of the more

[2] https://github.com/mlasson/paramcoq
[3] https://github.com/drouhling/paramcoq

general ones. But because of the refinement inference mechanism (see Section 3), keeping only the more general theorems would require declaring a refinement of `nat` as `nat` using the relation ⟦`nat`⟧ and reflexivity, which amounts to doing nothing when applying refinement to natural numbers, not mentioning the conflicts it would create with our actual refinement of `nat` and the theorems already involving ⟦`nat`⟧...

Instead, we have two instances of each theorem about matrices: the more general version for further developments involving matrices, and a version instantiated with reflexivity for practical use. This makes development inefficient but it has no consequence on the running time when computing refinement, whereas the other solution, using an abstract type for natural numbers and our refinement of `nat` into binary numbers, lengthens inference. We still need to find an adapted framework for partial refinements, which could solve these issues.

3 Implementation

As mentioned in Subsection 1.1, one of the interests of refinement is to switch representations of objects to perform computation during a proof, thus simplifying the goal. For this to be less tedious than rewriting equalities, it is essential to have a mechanism to automatize the search of refinements. There exist two mechanisms in Coq that make feasible this automation: canonical structures ([MT13]) and type classes ([SO08]). Canonical structures are at the heart of the Mathematical Components library. Type classes were used to design CoqEAL. We first make a short introduction to the type class mechanism and explain the role of type classes in our context. Then, we present the different ways of switching representations of objects using this mechanism. Finally, we describe how to tune inference in order to use the best structures available and we discuss the particular case of the refinement of functions.

3.1 Type classes

Type classes are specified by adding constraints to one or several type variables. For instance, one can declare the class of the types that admit an equality operator as follows:

```
Class eq_of A := eq_op : A -> A -> bool.
```

Here, `eq_of` is the name of the class and `eq_op` is the name of the operator. One can then for each type that satisfies the constraints declare it as an instance of the class by giving the term proving that indeed the constraints are fulfilled. For example, in the case of `eq_of`, one can declare that natural numbers in binary representation admit an equality operator through:

```
Instance eq_N : eq_of N := N.eqb.
```

Each declared instance of a type class is stored in a database of hints which guide type class inference. Thus, whenever `eq_op` appears, Coq will try each entry in the database, solving unification and type constraints, until a correct one is found. For example, in a proper context `eq_op` would be substituted by `N.eqb`.

This mechanism makes overloading of notation possible (here, `eq_op` is the overloaded notation), thus making the proof process closer to the one we use in mathematics. For example, we can assimilate a group structure to its support thanks to a type class where the argument type represents the support of the group, and the constraints are the group axioms. Since CoqEAL was designed for algebra, we make use of overloading and we have a class for each operator/object one could need in this context: addition, multiplication and their neutrals, substraction, division...

Type classes were preferred to design CoqEAL over canonical structures for their modularity. New classes and instances can be defined at any point in a development and

one can select the ones (s)he wants to use, whereas canonical structures are more rigid: they pack data/operators/properties together and adding/deleting one of them requires the definition of a new structure.

3.2 Refinement inference

In CoqEAL, we also use type classes to keep a database of refinement theorems, which allows us to let type class inference do the search of a refinement for us. Indeed, we have seen in Subsection 1.3 how to compare different representations using relations. By using a tag to define a type class, one can feed type class inference with all the theorems involving these relations. Together with particular instances defining rules to guide the inference, this gives us a logical program computing refinements. The class for refinements is the following:

```
Class refines A B (R : A -> B -> Type) (m : A) (n : B) :=
  refines_rel : R m n.
```

It takes two types (understand two representations of the same data type), a relation between them and two objects as arguments and asks for a proof that those two objects are related. The types `A` and `B` can be deduced from the relation `R` so that it is not necessary to write them. Inference is by default guided by the argument `m` (intuitively, we go from the data type `A` to the data type `B` by translating a term of type `A`). From a logical programming point of view, `m` is the input of the program whereas the other arguments are the outputs. In that context, the lemma `RratC_add` of Subsection 1.3 translates to

```
Instance RratC_add : refines (RratC ==> RratC ==> RratC) add_op add_op.
```

The proof of this theorem will be the term stored as instance of the class `refines`. It will be found and used by type class inference whenever we try to refine the addition over the type `rat`.

Once all the refinement instances for the base operators/objects are stored, Coq can combine them using rules defined by particular instances to refine expressions defined with these operators/objects. Among these rules, the one for the application of a function:

```
Instance refines_apply
  A B (R : A -> B -> Type) A' B' (R' : A' -> B' -> Type) :
  forall (f : A -> A') (g : B -> B'), refines (R ==> R') f g ->
    forall (a : A) (b : B), refines R a b -> refines R' (f a) (g b).
```

In short, a function applied to an argument refines to a refinement of the function, applied to a refinement of the argument. This rule is sufficient for dealing with any number of arguments, thanks to currying. The rule `refines_apply` is essential to break expressions into smaller bricks for which refinement instances have been proven. For example, assume given the following refinement instances:

```
refines R x y,
refines R z t,
refines (R ==> R ==> R) + +,
refines (R ==> R ==> R) * *.
```

We used the same notations for both versions of addition (and multiplication), thanks to overloading. Now assume we want to compute a refinement of `x + (x * z)`. We have to deal with the following goal:

```
refines ?R (x + (x * z)) ?y,
```

where `?a` is an existential variable standing for `a` that needs to be instantiated (remark the consequent separation between the inputs and outputs of our logical program). By the rule refines apply, this goal breaks up into

```
  refines (?R' ==> ?R) (fun a => x + a) ?f,
  refines ?R' (x * z) ?y',
```

and `?y` is (partially) instantiated into `?f ?y'`. The rule `refines_apply` can also be applied to the first goal, which splits into

```
refines (?R'' ==> ?R' ==> ?R) + ?f',
refines ?R'' x ?y''.
```

These two goals are respectively solved by the assumptions on addition and `x`, thus refining the instantiation of `?y` into `y + ?y'`. Proceding similarly on the remaining goal, which has been refined into

```
refines R (x * z) ?y',
```

type class inference finally succeeds to prove

```
refines R (x + (x * z)) (y + (y * t)).
```

Another rule is the one for the composition of relations:

```
Lemma refines_trans T U V
  (rTU : T -> U -> Type) (rUV : U -> V -> Type) (rTV : T -> V -> Type)
  (t : T) (u : U) (v : V) :
  composable rTU rUV rTV ->
    refines rTU t u -> refines rUV u v -> refines rTV t v.
```

Here `composable` is another type class meaning that the first two arguments compose into a subrelation of the third one. Remark the presence of the keyword `Lemma` instead of `Instance`. Indeed, we cannot let the program use this lemma during inference for questions of termination. That is not a real issue since this rule is only used to introduce parametricity lemmas, as in Subsection 1.3; the instances that are stored are the ones for the composed relation. When using this rule, the hypothesis `refines rTU t u` has to be treated before `refines rUV u v` since in the latter `u` is an input and has thus to be fully instantiated before triggering the inference of a proof for this hypothesis. We give in Subsection 3.5 an example where this remark takes its importance.

3.3 Switching representations

Now that we have described how refinement inference is done, we need to explain how to trigger it. In CoqEAL, there are two ways of switching representations of an object. The first one is by rewriting and only applies when going from a "proof-oriented" representation to a "computation-oriented" one. In theory it would also be feasible in the other direction but we lack a command in the type class mechanism to do so. Indeed, as mentioned in Subsection 3.2, the structure of the first representation that occurs in the arguments of the class `refines` guides the inference of the second one. We lack a command to switch temporarily inputs and outputs of the logical program.

To perform refinement by rewriting we use the following lemma:

```
Lemma refines_eq T (x y : T) : refines eq x y -> x = y.
```

This lemma simply removes the tag `refines` to get an equality that one can rewrite. The use of a type class instance as hypothesis triggers type class inference, thus computing the refinement. The next step is to find a theorem with a `refines eq` instance as conclusion. Since most of the relations defining a refinement in CoqEAL are expressed using an equality, it is easy to produce such theorems.

The other way of going from a representation to another is by computation. It should be used only when one want to go back from a "computation-oriented" representation to a "proof-oriented" one. This method is inherently bound to one direction of translation

whereas there is only a technicality to make the first method avaible for both directions. We use a function `spec` that computes the goal representation from the starting one.

We can use this function combined with the first refinement method. Indeed, one can prove that `spec` is a refinement of the identity function through the following theorem:

```
Instance refine_spec : refines (R ==> eq) id spec,
```

where `R` is the relation used to define the refinement of the considered data type. Thus, it is possible to use refinement by rewriting to transform any "proof-oriented" expression into `spec` applied to a "computation-oriented" version of the expression (it is necessary to explicitely introduce the identity function for inference to succeed), and then to compute the result of this application, hence going back to the "proof-oriented" representation.

Remark that the conclusion relation in `refine_spec` is equality. In fact, if we used parametricity it would instead depend on the data type. For instance, for a refinement of natural numbers, `R` would be `Rnat` and `eq` would be replaced with ⟦`nat`⟧. I decided to prove this kind of theorems "by hand" instead of using parametricity because it would have required interpreting many objects of the Mathematical Components library; for `nat` it is not an issue, but it could one be for more complex structures that contain proofs, which are not relevant for parametricity. On the contrary, without parametricity for these functions, the theorems to prove "by hand" depend only on the structure used as a refinement, which most often does not contain any proof and is much simpler.

3.4 Optimization

The type class mechanism makes the design of a refinement very systematic. The user only has to prove refinement instances for each base object/operator (s)he wants to refine. Then, to refine an expression, the logical program defined by a few rules such as `refines_apply` decomposes this expression and uses the small bricks at its disposition. Nevertheless, it is possible to tune the system to use the best structures available as refinements.

An interesting remark is that type class inference is not bound to use `refines_apply` each time it works on a function applied to an argument. It is indeed one instance stored in the database among others. One can provide instances for specific input patterns in order to refine such expressions into more efficient representations than when using the rule `refines_apply`.

For example, I worked on a refinement of Mathematical Components polynomials as sequences of coefficients. Using `refines_apply` and refinements of multiplication and of the indeterminate, it is possible to refine the multiplication of a polynomial by the indeterminate. However, on sequences, multiplying by the indeterminate is really easy and efficient: it suffices to put a 0 at the beginning of the sequence. To avoid going through the refinement of multiplication, and to use this more efficient operator (called `shift_op` below), I introduced specific instances to recognize this pattern:

```
Instance RseqpolyC_mulX p sp :
  refines RseqpolyC p sp -> refines RseqpolyC (p * 'X) (shift_op sp),
Instance RseqpolyC_Xmul p sp :
  refines RseqpolyC p sp -> refines RseqpolyC ('X * p) (shift_op sp).
```

I could have written them as

```
Instance RseqpolyC_mulX p sp :
  refines (RseqpolyC ==> RseqpolyC) (fun p => p * 'X) shift_op,
Instance RseqpolyC_Xmul p sp :
  refines (RseqpolyC ==> RseqpolyC) (fun p => 'X * p) shift_op,
```

but because of the form of `refines_apply` only the second instance would have been found by type class inference. Indeed, when f takes two arguments, applying `refines_apply` in order to refine $f\ x\ y$ means trying to refine $f\ x$ on one hand, and y on the other hand. Hence, type class inference fails to recognize `(fun p => p * 'X) q` in the pattern `q * 'X`. That is why I expanded the definition of `RseqpolyC ==> RseqpolyC`.

Moreover, to make sure that these instances will be used during inference rather than `refines_apply`, it is possible to give them priorities. Priorities define the order that is used during inference to try the instances stored in the database. The type class mechanism already computes priorities when instances are declared, but one can override the system by explicitely specifying a priority. In our case, giving a very low priority to the rule `refines_apply` will be sufficient to make sure that all the particular cases are tested before going through this generic step.

3.5 Refinement of functions

Our logical program heavily relies on the rule `refines_apply` to compute refinements. Indeed, thanks to this rule it is sufficient to provide refinements for a few basic operators and constants, and it is the rule `refines_apply` that decomposes the expressions to isolate these operators. However, this does not work for the refinement of a function which is not one of these operators: it is necessary to introduce variables for its arguments before trying to refine its body. This can be done through the following rule, complementary to `refines_apply`:

```
Lemma refines_abstr
  A B A' B' (R : A -> B -> Type) (R' : A' -> B' -> Type)
  (f : A -> A') (g : B -> B') :
  (forall (a : A) (b : B), refines R a b -> refines R' (f a) (g b)) ->
    refines (R ==> R') f g.
```

For reasons of termination we cannot define it as an instance. Consequently, when a refinement of a function is needed one has to use the hypothesis of `refines_abstr` instead of its conclusion. When working on a refinement of Mathematical Components matrices, I came across a case where such a manipulation was necessary. In the Mathematical Components library, it is possible to define a matrix from a function f taking two arguments: the coefficient in position (i, j) is $f\ i\ j$. To refine expressions involving this construct, we need a refinement of the operator defining the matrix and a refinement of the function involved in the expression. This led to the following first attempt (remark the quantification over the dimensions of the matrix, as mentioned in Subsection 2.3):

```
Instance RseqmxC_seqmx_of_fun m1 m2 (rm : nat_R m1 m2) n1 n2
  (rn : nat_R n1 n2) f g
  `{forall x y, refines (rI rm) x y ->
     forall z t, refines (rI rn) z t ->
       refines rAC (f x z) (g y t)} :
  refines (RseqmxC rm rn) (\matrix_(i, j) f i j) (seqmx_of_fun g).
```

Here, `rI` is a relation which defines a refinement of Mathematical Components type `ordinal`, which takes a natural number n and returns the type of natural numbers smaller than n (it represents the ring $\mathbb{Z}/n\mathbb{Z}$), and `rAC` is a relation which defines a refinement of the coefficients of the matrices. What happens during type class inference with this instance is that the program tries to refine the (simplified) application `f x z` as the (not yet instantiated) application `?g y t`. In some cases, depending on the form of the body `f x z`, the use of `refines_apply` would lead the inference to fail. For example, with the body `x + (x * z)`, the program splits the goal into

```
refines (?R ==> rAC) (fun a => x + a) (?g y),
```

```
refines ?R (x * z) t,
```

thus failing. This issue would not occur if, instead of `?g y t`, the program generated a single existential variable `?e`. Indeed, the different instances used by type class inference would progressively instantiate this variable, giving it the right shape, and it would be sufficient afterwards to unify the result with the application `?g y t`.

To force the program to infer a refinement of `f x z` with an existential variable `?e`, and only then to unify the result with the application `?g y t`, I composed the refinement relation `rAC` with a new type class having a single instance (reflexivity) forcing unification:

```
Class unify A (x y : A) := unify_rel : x = y.
Instance unifyxx A (x : A) : unify x x := erefl.
```

Now, by using the rule for relation composition (`refines_trans` in Subsection 3.2) on the goal

```
refines (rAC o unify) (f x z) (?g y t),
```

the program is left with the two goals

```
refines rAC (f x z) ?e,
refines unify ?e (?g y t).
```

The first goal is the one that we could solve with type class inference. With a "proof" that `unify x y` implies `refines unify x y` (recall that `refines` is only a tag), unification finishes the proof for the second goal. Here, `refines_trans` can safely be used through an instance since with the type class mechanism we can tell the program to use it only in the specific case where the relation is of the form `R o unify`.

4 Applications

I present here two applications of my work. The first one, already mentioned several times before, is the use of computation inside of proofs. We give here an example that involves almost all the refinements I proved during this internship. The other application is a reimplementation of the tactic `ring` ([MG05]) using refinement.

4.1 Proofs by computation

One of the ideas behind the Mathematical Components library is to let the user specify the most significant steps in a proof and to use small steps of computation to eliminate low-level details. This, together with overloading of notations, makes the proof process closer to the one we use in mathematics. However, for bigger steps, the data structures of the library do not really fit because they are "proof-oriented": they embed proofs and information that make mathematical developments easier, and they use algorithms that are simple enough to be easily proven correct. With refinements, it is possible to replace these algorithms with more efficient ones, and to use lighter data structures, in the sense that they do not carry any irrelevant information for computation.

Thus, when a simplification of the goal seems obvious because this goal should compute to something simpler, it is possible to use refinement before triggering the computation. For example, in the Mathematical Components library there is a development about real closed fields [4] in which a determinant has to be computed. The function `\det` in the Mathematical Components library is defined using the Leibniz formula:

$$det\,(M) = \sum_{\sigma \in \mathfrak{S}_n} \varepsilon\,(\sigma) \prod_{i=1}^{n} M_{\sigma(i),i}.$$

[4] https://github.com/math-comp/math-comp/blob/master/mathcomp/real_closed/qe_rcf_th.v

It would be more efficient to use a refinement of this function instead of this formula, even if we could remove locks on Mathematical Components structures. In the case of this development, the determinant of the following matrix had to be computed:

```
Definition ctmat1 := \matrix_(i < 3, j < 3)
  (nth [::] [:: [:: 1%:Z  ; 1 ; 1 ]
              ; [::  -1   ; 1 ; 1 ]
              ; [::   0   ; 0 ; 1 ] ] i)`_j.
```

Here `nth` is a function that takes a default value, a sequence and a natural number n and that returns the n-th element of the sequence if n is smaller than its size and the default value otherwise. The notation `s`_j` stands for `nth 0 s j` and the notation `%:Z` is an explicit coercion from the type `nat` to the type `int`. In short, `ctmat1` is the matrix of integers that you would represent on paper as the array from which it is defined:

$$\begin{pmatrix} 1 & 1 & 1 \\ -1 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}.$$

The goal is to prove the following lemma:

```
Lemma det_ctmat1 : \det ctmat1 = 2.
```

The proof of this lemma is obvious since the determinant of this matrix indeed computes to 2. However, because of locks, in the Mathematical Components library it is necessary to rewrite several equalities in order to prove it. Let us have a look at the proof:

```
by do ?[rewrite (expand_det_row _ ord0) //=;
  rewrite ?(big_ord_recl,big_ord0) //= ?mxE //=;
    rewrite /cofactor /= ?(addn0, add0n, expr0, exprS);
      rewrite ?(mul1r,mulr1,mulN1r,mul0r,mul1r,addr0) /=;
        do ?rewrite [row' _ _]mx11_scalar det_scalar1 !mxE /=].
```

The first theorem that is used, as suggested by its name, is a theorem that expands the determinant by development along a row. The theorems `addn0` and `add0n` state that `0` is a neutral on the right (resp. on the left) for the addition on natural numbers, and similarly for the theorems `mul1r` and `mulr1` with `1` and the multiplication in a ring. The theorem `expr0` states that the exponentiation of any ring element by `0` is equal to `1`. We will not explain the remaining of the proof, since our purpose is not to make this proof understandable but to show how heavy it can be to do computations in this context. This results in a fairly complicated proof script for the computation of the determinant of a 3 by 3 matrix. There is a similar example in a formalisation of sign determination for real algebraic numbers [5], involving finite sets (for which we do not have a refinement yet), and giving a script of more than 40 lines.

With refinement, this proof is reduced to the following:

```
apply/eqP.
rewrite [_ == _]refines_eq.
by vm_compute.
```

The first line is typical in SSReflect proof scripts: it is a reflection that replaces Coq equality with the equality operator of `int`, called `eq_op` (similarly as in Subsection 3.1 thanks to the overloading provided by canonical structures), and denoted by `==`. The second line triggers refinement on the boolean `ctmat1 == 2`. The last line concludes the proof by computation on the refined structures.

[5] https://github.com/Barbichu/signdet/blob/master/signdet.v

This proof involved many different refinements that I proved during this internship. First, there is a program refinement that replaces the function `\det` with Bareiss' algorithm for the computation of the determinant. The data structures for which a refinement was required are matrices and integers, the latter requiring refinements of natural numbers and positive natural numbers, but also polynomials since Bareiss' algorithm uses the characteristic polynomial of a matrix. For most of them, my work was to patch the existing proofs that were not working anymore. However I redevelopped the refinement of polynomials from scratch using a different relation (that left more theorems of the Mathematical Components library at my disposition for the proofs) and also did almost all the proofs for matrices (all except those for the null matrix and the operator $M \mapsto -M$), adding some new refinements such as diagonal and scalar matrices or the trace function.

4.2 The tactic `ring`

`ring` [MG05] is a tactic that proves equations modulo the axioms of rings. It works on both sides of the equation in three steps:

1. interprete the ring expression as a polynomial expression,
2. put the polynomial in normal form,
3. go back to the world of ring expressions.

The first step is done by observing the head operator in the expression and interpreting it as an operator over polynomials, and then recursively interpreting its arguments. The tactic has an interpretation in terms of polynomial expressions for the terms generated by the following grammar:

$$e \quad ::= \quad 0 \mid 1 \mid e_1 + e_2 \mid e_1 * e_2 \mid e_1 - e_2 \mid -e \mid e^n \mid c,$$

where n ranges over natural numbers and c over ring-specific constants for which the user has provided an interpretation. For the remaining operations that have no interpretation, the subexpression that cannot be translated is interpreted as an indeterminate. For instance ([dt16]), the expression

```
((f(5) + x) * x) + ((if b then 4 else f(3)) * 2)
```

is interpreted as the polynomial

```
((Y + Z) * Z) + (X * 2),
```

where the variable map `{X -> if b then 4 else f(3), Y -> f(5), Z -> x}` is stored to be able to go back to ring expressions in the last step. Here, `f` and `x` are variables and consequently have no interpretation as polynomial operations/objects, and the `if` construct is not a standard ring operation, even if it could be encoded as follows:

```
if b then x else y = x * b + y * (1 - b).
```

The normal form considered for the second step is an optimized version of Horner's representation of polynomials. The non-optimized version of this representation, to which I have proven a refinement of Mathematical Components polynomials, is the following: a polynomial is either a constant, or $X^n * P + c$ where c is a constant, n is positive and P is a polynomial in Horner's representation. For normalization, multivariate polynomials (with indeterminates $X_1, \ldots, X_n$) are considered as monovariate polynomials (with indeterminate X_1) over multivariate polynomials (with indeterminate $X_2, \ldots, X_n$). On our example the normal form is

```
(X * 2) + (Y * Z) + Z^2,
```

giving as a result the expression

```
((if b then 4 else f(3)) * 2) + (f(5) * x) + x^2.
```

My reimplementation of the tactic `ring` makes a clear distinction between these three steps. It is composed of three tactics: `translateEq`, which performs the translation of expressions as polynomials, `simplPoly`, which normalizes the polynomials, and `evalHorner`, which evaluates the polynomials to go back to the world of ring expressions.

For the first step, instead of a user defined type for polynomials, I used Mathematical Components type `polynomial`. Note that since every specific operator is replaced with a variable, the translation as a polynomial of an expression does not depend on the ring. I used the type `int` for the ring of coefficients, since it is a type for which there exists a refinement in CoqEAL; this is not a requirement, but I knew I would use refinement for the normalization step. The variable map is computed as a sequence but does not appear to the user when applying `translateEq`. In fact, this tactic proves that each side of the goal equality is equal to the evaluation of its translation as a polynomial at the values provided by the variable map. This proof is not difficult since the expressions are exactly their corresponding polynomial where the indeterminates are replaced with their assignment in the variable map. Thus, this tactic changes the goal from the equality between two expressions in a ring to the equality between two evaluations of multivariate polynomials over integers. For instance, the goal

```
a + b - (1 * b + c * 0) = a
```

becomes

```
let P := X + Y - (1 * Y + Z * 0) in
let Q := X in
P[a][b][c] = Q[a][b][c].
```

The second step, normalization, is done through refinement. The polynomials, isolated thanks to the `let ... in` construct, are refined to "computation-oriented" structures and normalized through computation using the `spec` operator. Remark that in this case I implemented a `spec` operator computing polynomials in a specific form instead of a less specific but more easily provable function. On our example, the tactic `simplPoly` transforms the goal into

```
X[a][b][c] = X[a][b][c].
```

Finally, the tactic `evalHorner` evaluates both polynomials by rewriting equations provided by the Mathematical Components library, giving here the trivial goal

```
a = a.
```

Presently, this implementation is incomplete and not very efficient. First, the power function is not interpreted as an operator on polynomials during the translation step because we do not have a refinement for it yet. Then, we do not provide the possibility to declare interpretations for ring-specific constants. Moreover, evaluation, both for the proofs in the first step and for the third one, is done by rewriting equations about Mathematical Components polynomials. As we have seen on the example of the determinant, these theorems are very specific. Since we want this evaluation to be automatic, we cannot choose the right theorem to rewrite at each small step in the evaluation. All we can do is to provide a rule with all the theorems that could be needed, and Coq will try them, making progress when it finds one that is usable. Rewriting involves unification and, in spite of the optimizations on the `rewrite` tactic in SSReflect in order to lower the number of considered non-solvable unification goals, this method is still expensive in terms of running time. Finally, the time needed to perform the refinement mainly depends on the number of indeterminates but does not scale very well, for some reasons we still need to investigate.

Switching back to user defined polynomial expressions should solve the running time issue in the translation step but we need to provide some way of seeing them as Mathematical Components polynomials for the refinement step. A possibility is to use a function to translate expressions into Mathematical Components polynomials, together with a proof of its correctness.

This reimplementation should allow the tactic to deal with more equations than the tactic `ring`. The modularity brought by the clear distinction between the three steps makes it possible to change the tactic used for one of them so long as the interface between the three steps stays the same. In particular, an improvement over the tactic `ring` would be a more powerful tactic that handles morphisms for the first step. For instance, we would be able to simplify the expression

```
f(x + y) - f(y)
```

into `f(x)` if the corresponding polynomial for this expression was

```
X + Y - Y
```

with the variable map `{X -> f(x), Y -> f(y)}`. The Mathematical Components library already embeds the necessary theorems to "push" in such a way the morphisms to the leaves of an expression.

Conclusion

The goal of this work was to automatize the use of parametricity ([Wad89]) in CoqEAL ([DMS12]), in order to make the development of certified efficient algorithms/data structures more systematic.

We used Keller and Lasson's plugin for parametricity ([KL12]) to generate the needed instantiations of the parametricity theorem. This required some modifications of its code, to make it work with the new version of Coq, and to solve an issue raised by the overloading of some names in the Mathematical Components library.

We used these theorems to prove refinements for several structures of Mathematical Components: natural numbers, positive natural numbers, integers, polynomials (two different refinements), matrices, rational numbers and the field $\mathbb{F}_2$. For most of them, it was only a matter of fixing the already existing proofs. In the case of polynomials and matrices, however, almost all the proofs were done from scratch. We also adapted the proofs of two program refinements: Karatsuba's algorithm for the multiplication of polynomials and Bareiss' algorithm for the determinant.

This work opens the door to the use of bigger computation steps inside proofs. For instance, a refinement of finite sets should bring the necessary material for the reduction of a 40 lines long script for the computation of 3 determinants to only a few lines. Moreover, we started a reimplementation of the `ring` tactic ([MG05]), using refinement, that should make it possible to apply it to more equations.

This work can be extended in different ways. A first possibility is to finish the reimplementation of the `ring` tactic and to improve it in order to deal with morphisms. Using Gröbner bases, it would also be possible to solve equations modulo other equations given as hypotheses. Another way to continue this work is to develop more refinements on top of those we proved. For data refinements, there is the type of finite sets, which we mentioned before, but also the rings $\mathbb{Z}/n\mathbb{Z}$, which is work in progress, and the refinements of the power function on several data types, needed to complete the reimplementation of the `ring` tactic with refinements. From program refinements, we can mention Strassen's algorithm for matrix multiplication or an algorithm computing the Smith normal form of matrices over Euclidean rings, already present in the previous version of CoqEAL but needing modifications for the use of the parametricity plugin.

References

[Bar68] Erwin H. Bareiss. Sylvester's identity and multistep integer-preserving gaussian elimination. *Mathematics of Computation*, 22(103):565–578, 1968.

[CDM13] Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for free! In Georges Gonthier and Michael Norrish, editors, *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings*, volume 8307 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2013.

[CH88] Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.

[CPM90] Th. Coquand and C. Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88*, volume 417 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[Dij82] Edsger W. Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer-Verlag, 1982.

[DMS12] Maxime Dénès, Anders Mörtberg, and Vincent Siles. A refinement-based approach to computational algebra in coq. In Lennart Beringer and Amy P. Felty, editors, *Interactive Theorem Proving - Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings*, volume 7406 of *Lecture Notes in Computer Science*, pages 83–98. Springer, 2012.

[dt16] The Coq development team. *The Coq proof assistant reference manual*, 2016. Version 8.5.

[GAA+13] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A Machine-Checked Proof of the Odd Order Theorem. In Sandrine Blazy, Christine Paulin, and David Pichardie, editors, *ITP 2013, 4th Conference on Interactive Theorem Proving*, volume 7998 of *LNCS*, pages 163–179, Rennes, France, July 2013. Springer.

[GMT15] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. A Small Scale Reflection Extension for the Coq system. Research Report RR-6455, Inria Saclay Ile de France, 2015.

[Gon08] Georges Gonthier. Formal proof–the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.

[KL12] Chantal Keller and Marc Lasson. Parametricity in an impredicative sort. In Patrick Cégielski and Arnaud Durand, editors, *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France*, volume 16 of *LIPIcs*, pages 381–395. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.

[Lam13] Peter Lammich. Automatic data refinement. *Archive of Formal Proofs*, 2013, 2013.

[MG05] Assia Mahboubi and Benjamin Gregoire. Proving Equalities in a Commutative Ring Done Right in Coq. In Joe Hurd and Tom Melham, editors, *TPHOLs 2005*, volume 3603, pages 98–113, Oxford, United Kingdom, August 2005. Springer.

[MT13] Assia Mahboubi and Enrico Tassi. Canonical structures for the working coq user. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving - 4th International Conference, ITP 2013,*

Rennes, France, July 22-26, 2013. Proceedings, volume 7998 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2013.

[Rey83] John C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.

[Rey84] John C. Reynolds. Polymorphism is not set-theoretic. In Gilles Kahn, David B. MacQueen, and Gordon D. Plotkin, editors, *Semantics of Data Types, International Symposium, Sophia-Antipolis, France, June 27-29, 1984, Proceedings*, volume 173 of *Lecture Notes in Computer Science*, pages 145–156. Springer, 1984.

[SO08] Matthieu Sozeau and Nicolas Oury. First-class type classes. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*, pages 278–293. Springer, 2008.

[Soz09] Matthieu Sozeau. A new look at generalized rewriting in type theory. *J. Formalized Reasoning*, 2(1):41–62, 2009.

[Wad89] Philip Wadler. Theorems for free! In Joseph E. Stoy, editor, *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*, pages 347–359. ACM, 1989.