PPS - Paris VII Diderot University

Internship Report

Compilation of Dependently Typed Pattern-Matching for Coq

Author:
Meven Bertrand

Supervisor:
Hugo Herbelin

September 19, 2016

ENS DE LYON

INSTITUT DE RECHERCHE EN INFORMATIQUE FONDAMENTALE

Abstract

The aim of this internship was to work on the compilation of pattern-matching in Coq, a tool for interactive theorem-proving developed by the Inria.
For now, the algorithm in place handles most simple cases, but often fails on more tricky instances, and forces the user to add a lot of information in order to see her/his matching compiled. However, other tools for theorem proving, most notably Agda, already implement a way more powerful compilation for pattern-matching.
A compilation algorithm having the same kind of possibilities as the one of Agda had already been roughly designed by my tutor H. Herbelin. My aim was to describe in detail this algorithm through the writing of an article describing the way it worked. Along the way, I also had to learn a lot about Coq and its theoretical background, the CIC (Calculus of Inductive Constructions).

Contents

Introduction

The structure of pattern matching is an important brick in the context of general functional languages, and in particular in the formal proof tool Coq: it is one of the basic tools used to work with inductive types, which are everywhere in Coq. In the current version of Coq, the mechanism used to compile this structure is enough for basics cases, but it often fails on more complicated ones, therefore needing the user to add much superfluous information in order for the compiler to work works. The way this compilation works is not totally documented, as the current implementation is only partial in comparison with the corresponding theory — making it even harder for the user to predict whether or not he has to add a precise piece of information in a given context.

However, other languages used for theorem proving, mostly the language Agda, already provide an effective compilation tool, showing that it is possible to improve the current situation and design a more user-friendly and powerful pattern-matching. It has already been shown in [3] that the same possibility are available in the CIC, the theoretical core of Coq. My tutor H. Herbelin had a rough idea of how to handle it, however at the time I began my internship no precise algorithm was available.

My aim was therefore to detail this algorithm, and possibly to implement it. Since I lacked time, I concentrated on the first part, through the writing of an article presenting the algorithm. In fact, I designed two compilation algorithms: one achieving the original aim, and a lighter version that is useful as it requires less information from the user in order to work.

1 Coq and the Calculus of Inductive Constructions

The elements presented in this section come from diverse sources on Coq, mostly the tutorial book [1], as well as the manuals [2, 5], and older articles, mainly [4]. The last article is one of the firsts ever published on Coq, so some notations and ways of seeing things are a bit old and differ slightly from the most modern ones.

But first, what is Coq? It is a tool for formal proofs, that is a tool which is designed for writing mathematical propositions (in a concrete language called Gallina) and proving them. The aim of Coq is that any proof that is written in Gallina and accepted by Coq at compilation time must be correct, so that no human verification of the proof is needed. To achieve that, the execution by Coq relies on a theoretical basis named the Calculus of Inductive Constructions (CIC), that is responsible for the checking of the proofs. The CIC is quite similar to (typed) lambda-calculus, the biggest differences being the existence of inductive and dependent types that do not exist in lambda-calculus, but are widely used in Coq.

As it is very close to typed lambda-calculus, the CIC is based on the same ideas: the objects considered within the CIC are terms, constructed with a few constructors. Each well-formed term has a type, which in turn is a term.

1.1 Calculus of Constructions: Terms and Types

The Calculus of Constructions (CoC) is the “lambda-term” component of the CIC, that is the part not involving any inductive construction.

Syntax The syntax of the terms of the CoC is based on 5 basic constructions, that is:

- the variables
- the abstraction, noted $\lambda x : T \cdot M$
- the dependent product, noted $\forall x : T, M$ (or $T \to M$ if M does not depend on x)
- the application, noted MN
- the sorts s, elements of a set $\mathcal{S}$

Semantic Each of these construction represents a different thing, namely:

- The abstraction $\lambda x : T \cdot M$ represents the function taking the variable x of type T to the term M.
- The dependent product $\forall x : T, M$ represents all functions taking some term x of type T to a term of type M (which can depend on the variable x). Therefore, the type $T \to M$ is the type of functions taking a term of type T and returning a term of type M.
- The application MN represents the application of the term M to the term N, in the same way as usual function application.
- The sorts are constants used for classification of the terms. In Coq there are 3 sorts, which are Type (representing “everything”), Set (representing all classical sets, for example the integers or the sorted lists of real numbers) and Prop (representing the logical propositions).

Typing In the CoC, each term has a type, and all the above constructions are only allowed under some conditions on the type of their arguments. For example if the term M depending on a variable x is well-formed if x is of type T and has type $N(x)$, then the term $\lambda x : T \cdot M$ is well-formed and has type $\forall x : T, N(x)$.

In order to define properly the rules governing the typing, we first need to define the notion of context: a context Γ is (naively) a list of declarations of variables along with their types. Each of these typing assertion is written $x : T$, so a context is something like $x_1 : T_1, \dots x_n : T_n$.

Having a context, we can define more precisely the typing with two relations, a binary relation $\Gamma \vdash \Delta$ where Δ is a context, saying that under the context Γ the context Δ is valid, and a ternary relation $\Gamma \vdash x : T$ saying that under the context Γ, the term x is well typed, and of type T. These two relations, along with a third one written $\cong$, called the equivalence of types and used to simplify the terms, are theoretically defined with a set of inference rules, that can be found in the annex (section A).

Propositions and Proofs The Coq aims to write propositions and to formally prove these propositions. Propositions are treated as any other terms, however when considering proofs the sense associated with the operators presented above changes.

- A proposition is any term of type Prop.

- A context $x_1 : T_1 \ldots x_n : T_n$ is a list of variables along with hypotheses on these variables, translated into a typing assertion.
- If $\Gamma \vdash p : P$ (with P such that $\Gamma \vdash P : \text{Prop}$), then p is a proof of the proposition P under hypotheses Γ, i.e. proving an assertion is giving any term whose type is the assertion.
- The construction $\forall x : T, M$ represents the universal quantification over x, taking any value in the type T.
- In particular $A \to B$ represents the implication.
- The application $M\ N$ works like the modus ponens, given an implication M of type $A \to B$ and an proof N of A it creates a proof $M\ N$ of B.

1.2 Inductive Constructions

Above the CoC, which is the core of Coq, another construction has been added during the development of Coq and is now the base of almost all definitions in Coq: the inductive objects.

The idea is to create a new type by declaring a certain number of constructors, each of them taking some arguments and giving a new element of the type. Examples are the integers, defined in Coq by:

```
Inductive nat :  Set :=
 | O : nat
 | S : nat -> nat.
```

with the two constructors `O` and `S`, just like the classical Peano definition, or the more complex vector:

```
Inductive vect (A:Type) :  nat -> Type :=
 | nil :  vect A O
 | cons :  forall (h:A) (n:nat), vect A n -> vect A (S n).
```

which takes two arguments, the type `A` of its elements and its length `n`, and is constructed again with two constructors `nil` and `cons`.

Of course, proposition types can be defined this way too, for example this is a possible definition of the parity of a natural number:

```
Inductive even (n:nat) :  Prop :=
 | O_even :  even O
 | S_even :  forall (n:nat), even n -> even (S (S n)).
```

The inductive constructions are a very important feature in Coq, as almost all objects defined in the many libraries of Coq are defined through an inductive definition. The reason for that is that inductive definitions are at the same time quite intuitive and very powerful. They are intuitive since to define an object you give exactly the ways it can be constructed, and nothing more. A lot of mathematical constructions can be naturally translated like this. They are powerful, since given an inductive definition one has a whole range of properties very useful to use: an object within an inductive type is constructed with one and exactly one constructor, so that one can always use case disjunction when working on an inductive object. When an inductive type is defined, an induction principle associated with the type is automatically generated by Coq, allowing easy proofs by induction on the type.

2 Patterns, Pattern-Matching

The idea behind pattern-matching is to decompose an object in an inductive type. As it was mentioned in section 1.2, an object in an inductive type is constructed with exactly one constructor of this type, so the most basic idea behind pattern-matching is to use this in order to perform a case analysis: given an object in an inductive type, we want to do different things (give a different object if we construct a function, or use a different reasoning if we are proving something) depending on the way the object we are considering has been constructed.

However, a simple case analysis is often not enough, so the full pattern-matching will allow more than one variable, and discrimination not only based on the outer most constructor, but on a full combination of constructors — this is what we will call a pattern.

Pattern-matching is therefore a crucial mechanism, as it is one of the only two ways of processing inductive objects, the other one being induction. In particular, being able to process a complex pattern-matching is an important step in order to make proofs clearer to read and simpler to write for the user of Coq. However, the current situation in Coq is not quite satisfying from this perspective, thus the importance of my work.

2.1 Patterns

First we need to precisely define what a pattern is. A pattern p is (in BNF-notation)

$$p ::= v \mid C_1 \underbrace{p \dots p}_{r_1} \mid \dots \mid C_n \underbrace{p \dots p}_{r_n}$$

where v denotes a variable, and C_i are all constructors of inductive types defined before the pattern is used, r_i being the arity of C_i. For any pattern of the form $Cp \dots p$, the constructor C is called the head constructor of the pattern.

We say that a term t matches with the pattern p if one of the two following cases apply:

- p is a variable
- p is of the form $C_i(p_1, \dots p_{r_i})$ and t is of the form $C_i(t_1, \dots t_{r_i})$ and each t_j matches with the corresponding p_j

2.2 Pattern-Matching Problem

A general pattern-matching problem in a certain context Γ consists of four main elements:

- a list of variables $x_1 \dots x_n$ each declared in the context Γ and having an inductive type
- a list of return terms $t_1 \dots t_m$
- a matrix $(p_{i,j})_{\substack{1 \le i \le n \\ 1 \le j \le m}}$ of patterns, which variables may be used in the return term with the same index as their line number
- a return predicate P, the definition of which may rely on the type of the variables $x_1 \dots x_n$

Such a problem will be later displayed under the form:

$$\Gamma \vdash \begin{array}{cc} (x_1 \dots x_n) & [\overrightarrow{y_1}, x'_1, \dots, \overrightarrow{y_n}, x'_n \vdash P] \\ \begin{pmatrix} p_{1,1} \cdots p_{1,n} \\ \vdots \\ p_{m,1} \cdots p_{m,n} \end{pmatrix} & \begin{pmatrix} t_1 \\ \vdots \\ t_m \end{pmatrix} \end{array}$$

Here also appear some $\overrightarrow{y_i}$ which are lists of variables associated with the type of the x_i and are used to make the predicate P depend on the type of the x_i.

The Gallina syntax corresponding to this problem is the following:

$$\Gamma \vdash \begin{array}{l} \texttt{match } x_1 \texttt{ as } x'_1 \texttt{ in } I_1\_\overrightarrow{y_1}, \dots, x_n \texttt{ as } x'_n \texttt{ in } I_n\_\overrightarrow{y_n} \texttt{ return } P \texttt{ with} \\ \begin{array}{lccc} \mid & p_{1,1} \cdots p_{1,n} & \Rightarrow & t_1 \\ \mid & \vdots & \vdots & \vdots \\ \mid & p_{m,1} \cdots p_{m,n} & \Rightarrow & t_m \end{array} \\ . \end{array}$$

with I_i being the inductive type of x_i, and the underscore are used to fill some arguments of the I_i that are not allowed to be used in P.

2.3 Compiling a Pattern-Matching Problem

The idea of a pattern-matching problem is that if j is the minimal index such that every x_i is unifiable with $p_{i,j}$, then we want to return the term t_j, with replacements of the variables of $p_{i,j}$ with the subterm of x_i it was unified against.

This behavior can be seen as a case analysis on the different terms x_i, where the form they have as inductive objects influences the term that is returned — the different cases corresponding to the different lines in the pattern matrix. Therefore the patterns appearing in the matrix must cover all possible cases, that is all possible combinations of constructors for the x_i must appear.

The processing of a pattern-matching problem is decomposed in two phases: what we call a simple pattern-matching, and the compilation. A simple pattern-matching is a pattern-matching with only a variable, and where all patterns appearing in the matrix $(p_{1,j})_{1 \leq j \leq m}$ are of the form $Cv \dots v$, i.e. a constructor applied to variables. This is called a simple pattern-matching as it is the simplest problem possible since it consists only on a case analysis on the head constructor of x_i. In Coq, such a pattern-matching in processed at a lower level, it is in a way a primitive construction included in the core of the tool together with the definition of inductive types. The compilation of a general pattern-matching aims to decompose a complex pattern-matching problem — many variables, arbitrary complex patterns — into a succession of such simple pattern-matchings.

2.4 State of the Art

The compilation of pattern-matching is not a problem specific to Coq, since pattern-matching under one form or another is a very common feature of many programming languages — mostly functional languages, which often heavily rely on it. Therefore, it is a quite old problem, that has already been studied a lot. However, Coq is a very specific setting, namely the one of dependent types. In this setting, where the type of the objects can depend on parameters, the compilation is a lot trickier, compared to a language with non-dependent types, even if these types still have a polymorphic component — as in Caml or Haskell for example. This compilation is the one that was considered in this internship, that is why most of the literature on pattern-matching compilation was irrelevant.

Even though, there has already been some work on the subject in [3], which is the base for the algorithm in place in Agda. This algorithm is powerful, more than the one designed in this internship, as the algorithm described in [3] processes definition in a more correct way. However, its work is based on equality, and in particular it makes an extensive use of an axiom called axiom K. This axiom K is not provable in Coq (nor can be proven wrong), therefore the compilation of pattern-matching must be done in another way if we want to avoid having to admit K. That is why the results of this internship are interesting, even if the compilation is still less effective than the one of Agda.

2.5 Aim of the Dependent Pattern-Matching

Here are two examples that seem quite simple, but that are not so easy to compile, showing the problems associated with pattern-matching and giving an idea of what the algorithm should achieve.

Example 1: Second variable depending on the first

```
Variables (n : nat) (v : vect nat n).
Definition w :=
match n, v in vect _ n' return vect nat (S n') with
        | O, nil nat => cons nat 1 O (nil nat)
        | S n', cons nat k n' v' => cons nat 2 (S n') (cons nat 1 n' v')
end.
```

Here we have an integer and a vector whose size is the natural, and we want to do a simultaneous matching on both objects. It seems obvious that only two cases are needed, as a vector of size 0 can only be constructed with `nil`, and a vector of non-zero size can only be constructed with `cons`. However a too simple algorithm could miss that, and demand the user to precise what to do in the case where n is constructed with `O` and v with `cons`. What is needed here is that the relation between the two objects is preserved throughout the matching process.

Example 2: Pattern in the in clause

```
Variables (n : nat) (v : vect nat (S n)).
Definition w :=
match v in vect _ (S n') return vect nat n' with
        | cons nat k n' v' => v'
end.
```

Here we have a vector which size is non-zero, therefore it must have been constructed with `cons`, and we want this to be understood by the compiler, so that it does not ask for the case where v is constructed with `nil`. What is needed here is that the compiler is able to extract constraints on the constructors thanks to the type of the objects.

3 The Algorithm

This is just a description of the way the algorithm works, for the exact details and the full technical part see the annex D where the article is reproduced.

3.1 Problem Considered

The most general problem the algorithm I designed is capable of handling is — in Gallina syntax — the following:

$$\Gamma \vdash \begin{array}{l} \texttt{match } x_1 \texttt{ as } x'_1 \texttt{ in } I_1\_\overrightarrow{\pi_1}, \ldots, x_n \texttt{ as } x'_n \texttt{ in } I_n\_\overrightarrow{\pi_n} \texttt{ return } P \texttt{ with} \\ \quad \begin{array}{lccc} | & p_{1,1} \cdots p_{1,n} & \Rightarrow & t_1 \\ | & \vdots & \vdots & \vdots \\ | & p_{m,1} \cdots p_{m,n} & \Rightarrow & t_m \end{array} \\ \quad\quad \texttt{end.} \end{array}$$

With x_1 to x_n being variables declared in the context, and not defined — that is, they are generic inhabitants of their type, rather than an alias for another object.

Compared to the syntax that was given in section 2.2, there is now the possibility to specify patterns instead of variables in the `as` clauses (these patterns are here called $\overrightarrow{\pi_i}$). This is for instance useful for the user in example 2, where the `as` clause is used in order to get rid of one constructor.

In the problem, many variables are considered at the same time, and not just one. The reason for this is that a compilation step can create more than one variable on which a matching occurs, so that the general problem must allow more than one variable to enable the recursive call of the compiler on the new problem created.

This problem will be displayed as follows:

$$\Gamma \vdash \begin{array}{cc} (x_1 \ldots x_n) & [\overrightarrow{w_1}, x'_1(\overrightarrow{\pi_1}), \ldots, \overrightarrow{w_n}, x'_n(\overrightarrow{\pi_n}) \vdash P] \\ \begin{pmatrix} p_{1,1} \cdots p_{1,n} \\ \vdots \\ p_{m,1} \cdots p_{m,n} \end{pmatrix} & \begin{pmatrix} t_1 \\ \vdots \\ t_m \end{pmatrix} \end{array}$$

with $\overrightarrow{w_i}$ the variables appearing in $\overrightarrow{\pi_1}$.

3.2 Special Cases

There are two special cases that occur through the compilation.

Terminal Case The first one is the terminal case of the algorithm, it is the case when $n = 0$, i.e. when there are no variables anymore for the matching. In this case, the problem is compiled into the first return term, that is a problem of the form

$$\Gamma \vdash \begin{array}{cc} () & [\vdash P] \\ \begin{pmatrix} \\ \\ \end{pmatrix} & \begin{pmatrix} t_1 \\ \vdots \\ t_m \end{pmatrix} \end{array}$$

is compiled into t_1.

At this point, to ensure that the problem given by the user at the beginning was correct, we may check if the type of t_1 is P. If not, then the predicate was incorrect, and an error can be raised.

Erasure of a variable Another special case is the one when the first column in the matrix of patterns consists only of variables, that is the problem is of the following form:

$$\begin{array}{ccc} & (x_1\ x_2 \dots x_n) & [\overrightarrow{w_1}, x_1'(\overrightarrow{\pi_1}), \dots, \overrightarrow{w_n}, x_n'(\overrightarrow{\pi_n}) \vdash P] \\ \Gamma \vdash & \begin{pmatrix} v\ p_{1,2} \dots p_{1,n} \\ \vdots \\ v\ p_{m,2} \dots p_{m,n} \end{pmatrix} & \begin{pmatrix} t_1 \\ \vdots \\ t_m \end{pmatrix} \end{array}$$

Here we can make the first variable disappear by replacing v in the return terms by x_1.

3.3 Generic Case

In the generic case, the idea is to have a simple match on x_1 — that will be called later the "master match", and in each branch to "remember" the head constructor of x_1.

To do this the algorithm first classes the lines of the pattern matrix (and the corresponding return terms) by the head constructor of the first pattern. In the branch corresponding to the constructor C_i, a list of variables $\overrightarrow{z}$ is introduced (as much as C_i takes arguments). In this branch, only the lines corresponding to the constructor C_i are reproduced, and the $\overrightarrow{z}$ are matched against the arguments of this constructor in the pattern, i.e. if the first pattern of the line is $C_i q_1 \dots q_{r_i}$ then the $\overrightarrow{z}$ are matched against the patterns $\overrightarrow{q}$. In a way, the complex patterns of the first column are classed by head constructor, and then decomposed. The lines with a variable in the first column are reproduced in each branch, as they can unify against every head constructor for x_1. The pattern-matching problem constructed for each branch — what we call a subproblem – is then recursively compiled.

This general mechanism is however not good enough to achieve the goals given in section 2.5. This is why some more mechanisms have been added.

The first one is the generalization. The aim of generalization is that in every branch of the master match the form of x_1 is remembered, in the sense that all types depending on x_1 take into account that x_1 has the head constructor C_i. In order to do this, the return term in each branch does not correspond directly to the compiled subproblem, but to a function taking every element of the context depending on x_1 as argument, and returning the compiled subproblem. The whole master match is then applied to all the elements of the context depending on x_1. Through this, the dependence on the more precise x_1 in the branch is respected. This is the mechanism ensuring that example 1 is compiled well, since it allows that the compilation of the matching on v takes the form of n into account.

The second one is the introduction of an intermediary matching problem within each branch. Its aim is to ensure that the recursive call in the branch is done in a context where the type of x_1 indeed unifies with the pattern that was given for it in the `in` clause, that is the patterns that we called $\overrightarrow{\pi_1}$. The problem looks like this

$$\begin{array}{ccc} & (\overrightarrow{b_1}') & [\overrightarrow{b_1}' \vdash s_T] \\ \Gamma_i \vdash & \begin{pmatrix} \overrightarrow{\pi_1} \\ v \end{pmatrix} & \begin{pmatrix} s_i \\ \mathrm{I} \end{pmatrix} \end{array}$$

where the $\overrightarrow{b_1}'$ are the arguments of the type of x_1 in the branch, that is the ones that should be unified with the $\overrightarrow{\pi_1}$, s_i is the solution to the subproblem created in the branch (and obtained by recursive call of the compilation), I is a filler that is returned in all impossible cases. With this intermediary match, the algorithm is capable of taking into account the presence of some patterns in the `in` clause, and this feature enables the correct compilation of example 2.

Finally, to ensure the well-typing of these intermediary branches, a predicate has to be designed for the master match, the intermediary match and the subproblems. For the subproblems, the predicate is easy, keeping P works just fine. However the addition of an intermediary match in each branch forces the creation of a more complex return predicate, as the type of the filler is not the same as the one of the subproblem. Therefore, the return predicates of the intermediary match and of the master match both use a two-line matching problem very similar to the one of the intermediary matching, that is

$$\Gamma, \overrightarrow{y_1} : \overrightarrow{B_1}, x_1' : I_1 \overrightarrow{a_1}\overrightarrow{y_1}, \overrightarrow{\gamma}'' : \overrightarrow{\Gamma}'' \vdash \begin{array}{cc} \left(\overrightarrow{y_1}\ \overrightarrow{b_2}, \dots \overrightarrow{b_n}\right) & [\vdash S] \\ \begin{pmatrix} \overrightarrow{\pi_1}\ \overrightarrow{\pi_2} \dots\ \overrightarrow{\pi_n} \\ v \end{pmatrix} & \begin{pmatrix} P \\ \text{True} \end{pmatrix} \end{array}$$

The variables on which the matching is done are not exactly the same, but the idea is the same as for the intermediary match: if the arguments of the types unify with the patterns they are supposed to respect, then the type of the returned element corresponds to P, and if not we get a filler (which this time is True and not I).

3.4 Properties of the Algorithm

Termination The termination of the algorithm can be proven by considering the multiset of the maximal depth of the patterns on each column of the pattern matrix. The depth of a pattern is defined to be 0 if the pattern is a variable, and the successor of the maximal depth of its arguments if it is constructed with a constructor.

Indeed this multiset is strictly less for each recursive call of the compilation as it was initially: in the case of the suppression of a variable because there is a strict inclusion of one multiset in the other, and in the generic case because the patterns for the variables x_2 to x_n do not change, and the variables introduced are matched against a pattern with depth at least one unit less than the maximal depth of the patterns x_1 was matched against. Since the set of all multisets of integers is well-founded, the algorithm must terminate.

However, at each steps some more matching problems are created for the return predicates. The depth of these matching problems cannot be easily controlled. This is one of the reasons why I designed another algorithm that cannot make the use of patterns in the `in` clauses, but in return does not create matching problems for the return predicate, as these were introduced precisely in order to handle the patterns in the `in` clauses. This algorithm is just a weaker version of the one exposed above, but is useful here: the matching problems created for the return predicates can be compiled with this algorithm and not the one exposed here, so that even if we do not precisely control the behavior of these matching problems it does not interfere with the proof of termination given above.

Correctness Proving the correctness of the algorithm was not a central point of the internship, and has not been done thoroughly. However, it could be done without too much difficulty by checking that given solutions to all the new problems created by a step of compilation, arranging them together as done by the algorithm indeed gives a solution to the initial compilation problem.

Complexity As for any algorithm, the complexity of the algorithm is a question that appears to be important. Here, as any generic step generates many new subproblems, each of them being recursively compiled, the complexity is at least exponential, and for sure very bad. However, this is not so important, as the entries considered in Coq are entered by a human, so that the size of the matrix of pattern rarely goes beyond 5×5, with patterns with a depth also mostly beyond 5. Therefore, even with an exponential complexity the working time still is reasonable in real-life situations.

Exhaustiveness and Non-Redundancy An other classical thing in the case of pattern-matching is to be able to check that the pattern-matrix used is exhaustive and not redundant. Exhaustiveness means that each possible sets of objects $x_1 \dots x_n$ having the correct types unifies with at least a line in the pattern-matrix, and non-redundancy means that for each line there exists a set of objects $x_1 \dots x_n$ having the correct types that can be matched with this line and no one above, i.e. that all lines are useful.

Checking the non-redundancy can be done during the compilation: each time the terminal case appears, the return term in which the whole matching was compiled can be flagged. At the end of the compilation, unflagged return term correspond to redundant lines, so redundancy can be detected — and an error possibly raised.

Checking the exhaustiveness is however not easy to do here, nor really wanted, as the possibility of giving a pattern in the `in` clause precisely aims to avoid needing to add lines only to preserve exhaustiveness, even if we know there are useless given the type of the objects.

Conclusion

3.5 Results

The main aim of the internship was to design an algorithm for compilation of pattern-matching problems in Coq able to handle in a correct and predictable way examples like the ones presented in section 2.5, in particular without making use of axiom K. This aim was reached with the design of two algorithms handling this task, and with the redaction of an article describing them.

However, there still is work to do around the algorithm I designed, that I lacked time to undertake myself: implementing it in Coq, study the relation of this algorithm with K and its behavior in presence of definition — in order to perhaps modify it and make it better —, prove its correctness more thoroughly than I did, study the possibility of checking exhaustiveness of the initial pattern matrix ...

3.6 Thanks

As for myself, this internship was very pleasing and instructive, as I learned a lot on Coq, type theory and general proof checking. For this I would like to thank a lot my tutor Hugo Herbelin for spending a lot of time working with me and proofreading the many versions of my article, Daniel de Rauglaudre and Matej Kosik for sharing their office with me and the many — productive and not so productive — talks we had, and all the interns, PhD students and postdoctoral students of the team PPS for welcoming me.

Appendices

A Induction Rules for the CoC

Here are the induction rules for the CoC, including the rules associated with the conversion of types. Greek letters denote contexts, capital roman letter denote generic objects, and lower case letters stand for the variables, represented with integers.

The star represents the empty context, $|\Gamma|$ is the length of the context Γ and $[N/x]Q$ is the object Q where all occurrences of the variable x have been replaced by the object N.

These rules are presented more thoroughly in [4].

Construction of a Context

- $\dfrac{}{* \vdash *}$
- $\dfrac{\Gamma \vdash \Delta}{\Gamma, x : \Delta \vdash *}$
- $\dfrac{\Gamma \vdash M : *}{\Gamma, x : M \vdash *}$

Product Formation

- $\dfrac{\Gamma, x : M \vdash \Delta}{\Gamma \vdash \forall x : M, \Delta}$
- $\dfrac{\Gamma, x : M_1 \vdash M_2 : *}{\Gamma \vdash \forall x : M_1, M_2 : *}$

Rules for the Other Constructors

- $\dfrac{\Gamma \vdash *}{\Gamma \vdash l : \Gamma/l}$ if $l \leq |\Gamma|$
- $\dfrac{\Gamma, x : M_1 \vdash M_2 : P}{\Gamma \vdash (\lambda x : M_1 \cdot M_2) : \forall x : M_1, P}$
- $\dfrac{\Gamma \vdash M : \forall x : P, Q \qquad \Gamma \vdash N : P}{\Gamma \vdash (M\ N) : [N/x]Q}$

Rules for Type Conversions

- $\dfrac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta \cong \Delta}$
- $\dfrac{\Gamma \vdash M : N}{\Gamma \vdash M \cong M}$

- $$\frac{\Gamma \vdash M \cong N}{\Gamma \vdash N \cong M}$$

- $$\frac{\Gamma \vdash M \cong N \qquad \Gamma \vdash N \cong P}{\Gamma \vdash M \cong P}$$

- $$\frac{\Gamma \vdash P_1 \cong P_2 \qquad \Gamma, x : P_1 \vdash M_1 \cong M_2}{\Gamma \vdash \forall x : P_1, M_1 \cong \forall x : P_2, M_2}$$

- $$\frac{\Gamma \vdash P_1 \cong P_2 \qquad \Gamma, x : P_1 \vdash M_1 \cong M_2 \qquad \Gamma, x : P_1 \vdash M_1 : N}{\Gamma \vdash \lambda x : P_1 \cdot M_1 \cong \lambda x : P_2 \cdot M_2}$$

- $$\frac{\Gamma \vdash (M_1\ N_1) : P \qquad \Gamma \vdash M_1 \cong M_2 \qquad \Gamma \vdash N_1 \cong N_2}{\Gamma \vdash (M_1\ N_1) \cong (M_2\ N_2)}$$

- $$\frac{\Gamma, x : A \vdash M : P \qquad \Gamma \vdash N \cong A}{\Gamma \vdash ((\lambda x : A \cdot M)\ N) \cong [N/x]M}$$

- $$\frac{\Gamma \vdash M : P \qquad \Gamma \vdash P \cong Q}{\Gamma \vdash M : Q}$$

B Compilation Example

To make things a bit clearer, here is an example of the compilation. The context is the following:

```
Inductive Ind : bool -> Set :=
  | C : forall b : bool, (if b then nat else bool) -> Ind b.

Variable v : Ind true.
```

and we want to compile the following definition

```
Definition w :=
match v in Ind true return nat with
  | C _ O => O
  | C _ (S n) => n
end.
```

The first round of compilation gives

```
Definition w :=
match v in Ind b return (if b then nat else True) with
  | C b t =>
  match b return (if b then nat else True) with
    | true =>
    match b, t return nat with
        | _, O => O
        | _, S n => n
    end
    | false => I
  end
end.
```

The second one processes the second level of matching, and gives, once we δ-reduce the `if-then-else` and simplify the abstraction, (as the Coq compiler automatically does) for more readability:

```
Definition w :=
match v in Ind b return (if b then nat else True) with
  | C b t =>
  match b return ((if b then nat else bool) -> if b then nat else True) with
    | true => fun (t : nat) =>
    match return (forall t : nat, nat) with
      | =>
      match b, t return nat with
        | _, O => O
        | _, S n => n
      end
    end
    | false => fun (t : bool) =>
    match return (forall t : bool, True) with
      | => I
    end
  end t
end.
```

A third step is the application of the first special case for the matching with no variables, after applying it to the two places where it can be done, we get

```
Definition w :=
match v in Ind b return (if b then nat else True) with
  | C b t =>
  match b return ((if b then nat else bool) -> if b then nat else True) with
    | true => fun (t : nat) =>
    match b, t return nat with
      | _, O => O
      | _, S n => n
    end
    | false => fun (t : bool) => I
  end t
end.
```

Now we still have to compile the matching on `b` and `t`, it first gives (second special case)

```
Definition w :=
match v in Ind b return (if b then nat else True) with
  | C b t =>
  match b return ((if b then nat else bool) -> if b then nat else True) with
    | true => fun (t : nat) =>
    match t return nat with
      | O => O
      | S n => n
    end
    | false => fun (t : bool) => I
  end t
end.
```

and then, after two more steps of compilation, the first one expanding the matching on `t` and the second one making it more compact, the algorithm terminates, and gives

```
Definition w :=
match v in Ind b return (if b then nat else True) with
  | C b t =>
  match b return ((if b then nat else bool) -> if b then nat else True) with
    | true => fun (t : nat) =>
    match t return nat with
      | O => O
      | S n => n
    end
    | false => fun (t : bool) => I
  end t
end.
```

which is well-typed in Coq and composed only of simple matchings, as expected.

C Bibliography

References

[1] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development.* Springer, 2015.

[2] The Coq Development Team. *The Coq Proof Assistant: Reference Manual*, 2016.

[3] H. Goguen, C. McBride, and J. McKinna. Eliminating dependent pattern matching. In *Algebra, Meaning, and Computation: Essays dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday*, pages 521–540. Springer, 2006.

[4] G. Huet and T. Coquand. The Calculus of Constructions. *Information and Computation*, 76, 1984.

[5] G. Huet, G. Kahn, and C. Paulin-Mohring. *The Coq Proof Assistant: A Tutorial*, 2016.

D Article

An Algorithm for the Compilation of Dependent Pattern-Matching without Axiom K

Meven BERTRAND, under supervision from Hugo HERBELIN

August 28, 2016

Abstract

The aim of this article is to present an algorithm that was designed in order to compile the pattern-matching in Coq. The main difficulties of this compilation are that the very powerful dependent typing system of Coq is responsible for a lot of dependencies between objects and types, a thing we need to be very careful with when compiling a pattern-matching. The second thing is the non-provability of Streicher's axiom K in Coq, making it more difficult to handle definitions, as compared to the algorithm in place in Agda, that makes an intensive use of it, for instance.

1 General Presentation

The context of the whole article is the type theory of Coq, namely the Calculus of Inductive Constructions. The fragments of code (in true type police) are written in Gallina, the concrete language of the tool Coq.

1.1 Definitions

Inductive Types All objects we will be working for belong to *inductive types*. Inductive types are types defined by giving some constructors, each of them taking arguments and constructing an element of the type. The generic definition of an inductive type in Coq looks like this:

$$\begin{aligned}&\texttt{Inductive } I(\overrightarrow{a} : \overrightarrow{A}) : \forall \overrightarrow{b} : \overrightarrow{B}, S := \\ &|C_1 : \forall \overrightarrow{z_1} : \overrightarrow{Z_1}, I\,\overrightarrow{a}\,\overrightarrow{b_1} \\ &\vdots \\ &|C_n : \forall \overrightarrow{z_n} : \overrightarrow{Z_n}, I\,\overrightarrow{a}\,\overrightarrow{b_n}\end{aligned}$$

here the type defined is I, of sort S. An object in this type can be constructed with any of the constructors C_i, the constructor C_i acting as a function taking some arguments $\overrightarrow{z_i} : \overrightarrow{Z_i}$ and returning an object of type $I\,\overrightarrow{a}\,\overrightarrow{b_i}$, with b_i possibly depending on the z_i and so that $\overrightarrow{b_i}$ is of type $\overrightarrow{B}$.

In fact, the definition of an inductive type is more exactly the definition of a whole family of types, depending on some arguments. Some of these arguments do not vary in the definition (the ones called $\overrightarrow{a}$ in the generic definition), they are called parameters. Others on the contrary may vary (the ones called $\overrightarrow{b_i}$), these are called indexes.

Not every definition of this kind is accepted by Coq: there are some guarding conditions on the $\overrightarrow{z_i}$ in order to allow the use of I in the types $\overrightarrow{Z_i}$ while preserving the fact that the objects constructed do not violate the condition that any reduction in Coq terminates.

Pattern definition A *pattern* p is (in BNF-notation)

$$p ::= \_ \mid v \mid C_1 \underbrace{p \ldots p}_{r_1} \mid C_1 \underbrace{p \ldots p}_{r_1} \texttt{ as } \theta \mid \ldots \mid C_n \underbrace{p \ldots p}_{r_n} \mid C_n \underbrace{p \ldots p}_{r_n} \texttt{ as } \theta$$

where v denotes a variable, and C_i are all constructors of inductive types defined before the pattern is used, r_i being the arity of C_i. For any pattern of the form $Cp \dots p$ or $Cp \dots p$ `as` θ, the constructor C is called the head constructor of the pattern.

In the definition of the pattern here we allow the use of the underscore as a joker symbol, its aim is to enhance readability while programming by allowing not to name certain useless variables. In a theoretical context, we get rid of them and consider that all patterns are of the BNF form

$$p := v \mid C_1 \underbrace{p \dots p}_{r_1} \mid C_1 \underbrace{p \dots p}_{r_1} \texttt{ as } \theta \mid \dots \mid C_n \underbrace{p \dots p}_{r_n} \mid C_n \underbrace{p \dots p}_{r_n} \texttt{ as } \theta$$

Match between a pattern and a term We say that a term t *matches* with the pattern p if one of the three following cases apply:

- p is a variable
- p is of the form $C_i(p_{i,1}, \dots p_{i,r_i})$ and t is of the form $C_i(t_{i,1}, \dots t_{i,r_i})$ and each t_j matches with the corresponding p_j
- p is of the form $C_i(p_{i,1}, \dots p_{i,r_i})$ `as` θ and t is of the form $C_i(t_{i,1}, \dots t_{i,r_i})$ and each t_j matches with the corresponding p_j

Context Finally, a *context* is a succession of two kind of constructions:

- Declaration: declaring a new variable by giving its name and its type. It corresponds to the keyword `Variable` of Gallina.
- Definition: defining a new variable by giving its name and what it is equal to, i.e. the new variable is only an alias for the whole term given. It corresponds to the keyword `Define` of Gallina, or to the construction `let in`.

Each declaration/definition can refer to every object declared or defined before it.

1.2 Basic problem

A *pattern-matching problem* consists of a context Γ, a list of variables $x_1 \dots x_n$ all **declared** in Γ with x_i having the inductive type $I_i \overrightarrow{a_i} \overrightarrow{b_i}$ (with $\overrightarrow{a_i}$ the parameters, of types $\overrightarrow{A_i}$ and $\overrightarrow{b_i}$ the indexes of types $\overrightarrow{B_i}$) and a list of pairs consisting of a list of patterns $p_{i,1} \dots p_{i,n}$ and an associated return term t_i using the variables defined in Γ, and the ones appearing in the $p_{i,j}$ (either as variable pattern or after a `as`). Finally a return predicate P may be given, this predicate is valid in a context $\Gamma, \overrightarrow{y_1} : \overrightarrow{B_1}, x'_1 : I_1 \overrightarrow{a_1} \overrightarrow{y_1}, \dots, \overrightarrow{y_n} : \overrightarrow{B_n}, x'_n : I_n \overrightarrow{a_n} \overrightarrow{y_n}$. We chose to focus on declared variables instead of considering possibly defined variables, as this will make the work easier later (section 3.3), as we will explain it there.

Such a problem will be later displayed under the form:

$$\begin{array}{ccc} & (x_1 \dots x_n) & [\overrightarrow{y_1}, x'_1, \dots, \overrightarrow{y_n}, x'_n \vdash P] \\ \Gamma \vdash & \begin{pmatrix} p_{1,1} \cdots p_{1,n} \\ \vdots \\ p_{m,1} \cdots p_{m,n} \end{pmatrix} & \begin{pmatrix} t_1 \\ \vdots \\ t_m \end{pmatrix} \end{array}$$

The Gallina syntax corresponding to this problem is the following:

$$\Gamma \vdash \begin{array}{l} \texttt{match } x_1 \texttt{ as } x'_1 \texttt{ in } I_1\_\overrightarrow{y_1}, \dots, x_n \texttt{ as } x'_n \texttt{ in } I_n\_\overrightarrow{y_n} \texttt{ return } P \texttt{ with} \\ \qquad\qquad\qquad \mid \; p_{1,1} \cdots p_{1,n} \;\Rightarrow\; t_1 \\ \qquad\qquad\qquad \mid \quad \vdots \qquad\quad \vdots \;\; \vdots \\ \qquad\qquad\qquad \mid \; p_{m,1} \cdots p_{m,n} \;\Rightarrow\; t_m \\ \qquad\qquad\qquad\qquad\quad \texttt{end.} \end{array}$$

where the underscore in I_i stands for as many underscore as I_i has parameters.

We want to *compile* such a pattern-matching, i.e. we want to find a term $t(x_1 \ldots x_n)$ such that $\Gamma \vdash t(x_1 \ldots x_n) : T$ and if v_i are instances of the types $I_i \overrightarrow{a_i}\, \overrightarrow{b_i}$ for $1 \leq i \leq n$, and j is the minimal integer such that for every $1 \leq i \leq n$, v_i matches with the pattern $p_{j,i}$, then $t(v_1 \ldots v_n) = t'_j$ where t'_j is the term t_j where every variable of the patterns has been replaced by the subterm of the x_i it matched against, and the equality is up to β, ι, δ and ζ reduction.

In order to do this with "real" variables x_i we will decompose the complex match (arbitrary complex patterns, many variables) into a series of simple matches, and these simple matches will be processed later on in the compilation process. We call *simple match* a matching problem with only one variable and where all patterns $p_{j,1}$ are of the form $C_i(v_1 \ldots v_{r_i})$ (a constructor applied to variables).

1.3 Development directions

We will follow two directions of development: first, we will try to compile a simpler version of the Coq syntax, namely the one without any `as`, `in` or `return` clauses between the keywords `match` and `with` (there still may be some `as` appearing in the patterns). This will force us to guess the return predicate during the compilation. Then, we will on the contrary introduce more complexity on the return predicate, namely we will introduce patterns in the `in` clauses as well.

2 First algorithm

The Gallina syntax for this first problem is much simpler than the previous since most clauses are absent:

$$\Gamma \vdash \begin{array}{lccc} \multicolumn{4}{c}{\texttt{match } x_1, \ldots, x_n \texttt{ with}} \\ | & p_{1,1} \cdots p_{1,n} & \Rightarrow & t_1 \\ | & \vdots & \vdots & \vdots \\ | & p_{m,1} \cdots p_{m,n} & \Rightarrow & t_m \\ \multicolumn{4}{c}{\texttt{end.}} \end{array}$$

To compile this problem we however introduce again the return predicate and the associated x'_i and $\overrightarrow{y_i}$, only that now the predicate is unknown, and is therefore replaced by an existential variable. To be able to do the compilation, we will need to give a value to this existential variable $?P$, that is why we need the whole term above to have a type, that can be inferred by Coq, and that we name T. The theoretical translation of the syntax of above is thus:

$$\Gamma \vdash \begin{array}{cc} (x_1 \ldots x_n) & [\overrightarrow{y_1}, x'_1, \ldots, \overrightarrow{y_n}, x'_n \vdash ?P] \\ \begin{pmatrix} p_{1,1} \cdots p_{1,n} \\ \vdots \\ p_{m,1} \cdots p_{m,n} \end{pmatrix} & \begin{pmatrix} t_1 \\ \vdots \\ t_m \end{pmatrix} \end{array} : T$$

The idea is to proceed to a double induction: we first decrease the complexity of the patterns corresponding to x_1 by splitting the problem until there are only variables. In this case, we just remove x_1 and replace it in the t_i, getting one variable less to process. In the end we have a matching problem with no variables at all, in which case we can safely return the first return term t_1.

2.1 First special case: terminal case

When $n = 0$, the problem is of the form

$$\Gamma \vdash \begin{array}{cc} () & [\vdash ?P] \\ () & \begin{pmatrix} t_1 \\ \vdots \\ t_m \end{pmatrix} \end{array} : T$$

Since the first pattern always matches, this can be compiled into the single term t_1, so the only work to do is to check that t_1 has indeed the type T and to fix the value of $?P$ to T to ensure that the term is well typed. This is then compiled into the single term t_1.

At this point we may want to flag as used the return term from which t_1 derives, so that at the end of compilation we can check for unused pattern lines, that is pattern lines that are not flagged. These pattern lines are redundant, in the sense that all the term list that match with them also match with pattern lists higher in the pattern matrix. We may then raise an error for the user, as such a situation should not happen.

2.2 Second special case: only trivial patterns for x_1

When $n > 0$ and all patterns corresponding to the variable x_1 are variables themselves, i.e. we have

$$\begin{array}{cccc} & (x_1 \dots x_n) & [\overrightarrow{y_1}, x'_1, \dots, \overrightarrow{y_n}, x'_n \vdash ?P] & \\ \Gamma \vdash & \begin{pmatrix} v_1\ p_{1,2} \cdots p_{1,n} \\ \vdots \\ v_m\ p_{1,2} \cdots p_{m,n} \end{pmatrix} & \begin{pmatrix} t_1 \\ \vdots \\ t_m \end{pmatrix} & : T \end{array}$$

with all the v_i being variables.

Here as said before we can get rid of x_1 from the list of matching variables by transforming this matching into

$$\begin{array}{cccc} & (x_2 \dots x_n) & [\overrightarrow{y_2}, x'_2, \dots, \overrightarrow{y_n}, x'_n \vdash ?P] & \\ \Gamma \vdash & \begin{pmatrix} p_{1,2} \cdots p_{1,n} \\ \vdots \\ p_{1,2} \cdots p_{m,n} \end{pmatrix} & \begin{pmatrix} t_1\,[x_1/v_1] \\ \vdots \\ t_m\,[x_1/v_m] \end{pmatrix} & : T \end{array}$$

with $t[x/y]$ denoting the term t where all occurrences of y have been replaced by x.

2.3 General case

Here there is a little bit of work, the idea is to divide the main problem in different subproblems after the first constructor of the patterns $p_{i,1}$, and to then point to the good pattern by using a "master match" on x_1, which will be a simple match. The matching problem is of the form

$$\begin{array}{cccc} & (x_1 \dots x_n) & [\overrightarrow{y_1}, x'_1, \dots, \overrightarrow{y_n}, x'_n \vdash ?P] & \\ \Gamma'\Delta\Gamma'' \vdash & \begin{pmatrix} p_1\ \overrightarrow{p_1} \\ \vdots \\ p_m\ \overrightarrow{p_n} \end{pmatrix} & \begin{pmatrix} t_1 \\ \vdots \\ t_m \end{pmatrix} & : T \end{array}$$

where Δ is the first time x_1 or one of the $\overrightarrow{b_1}$ is declared. We demand that x_1 and be declared, because this ensures that the new context we will build for each branch (the ones we will call Γ_i later) are strictly more precise than Γ, so that no typing error occurs in Γ_i, as it may be the case if x_1 is defined.

More precisely, $I_1\overrightarrow{a_1}\overrightarrow{b_1}$ is the inductive type of x_1 so let

$$C_i : \forall\overrightarrow{\phi_i} : \overrightarrow{\Phi_i}, I_1\overrightarrow{a_1}, \overrightarrow{b_{1,i}}(\overrightarrow{\phi_i})$$

for $1 \leq i \leq k$ be the constructors of this type. Define also r_i the arity of the constructor C_i.

Define $p_{i,1} \dots p_{i,m_i}$ the patterns p_k with head constructor C_i, such that the order is preserved. Define also $\overrightarrow{p_{i,j}}$ and $t_{i,j}$ the pattern list (resp. output term) associated with $p_{i,j}$, and $\overrightarrow{q_{i,j}}$ the list of patterns such that $p_{i,j}$ is either $C_i(\overrightarrow{q_{i,j}})$ or $C_i(\overrightarrow{q_{i,j}})$ `as` $\theta_{i,j}$. Finally, the patterns may also be variables, in this case we do not have any head constructor, so we define $p_{\omega,1}$ to p_{ω,m_ω} the patterns reduced to a single variable, and again the associated $\overrightarrow{p_\omega, j}$ and $t_{\omega,j}$.

Having all this definition, we can for each constructor C_i build a new context Γ_i defined as

$$\Gamma', \Delta, \Gamma'', \overrightarrow{z} : \overrightarrow{\Phi_i}, \overrightarrow{b_1}' := \overrightarrow{b_{1,i}}(\overrightarrow{z}), x'_1 := C_i(z_1 \dots z_{r_i}) : I_1 \overrightarrow{a_1} \overrightarrow{b_1}', \Gamma''[\overrightarrow{b_1}'/\overrightarrow{b_1}][x'_1/x_1]$$

The first Γ'' is only kept for convenience with the manipulation of de Bruijn indexes but no term refers to it in the rest of the processing, and by substitution of the list $\overrightarrow{b_1}'$ to $\overrightarrow{b_1}$ we mean substitution of each element of the second list by the corresponding one of the first.

Using this context, define for each constructor C_i the subproblem

$$\Gamma_i \vdash \begin{array}{ccc} (z_1 \dots z_{r_i} \; x_2 \dots x_n) & [\overrightarrow{w_1}, z_1, \dots, \overrightarrow{w_{r_i}}, z_{r_i}, \overrightarrow{y_2}, x'_2, \dots, \overrightarrow{y_n}, x'_n \vdash ?Q] & \\ \begin{pmatrix} \overrightarrow{q_{i,1}} \; \overrightarrow{p_{i,1}} \\ \vdots \\ \overrightarrow{q_{i,m_i}} \; \overrightarrow{p_{i,m_i}} \\ \overrightarrow{q} \; \overrightarrow{p_{i,1}} \\ \vdots \\ \overrightarrow{q} \; \overrightarrow{p_{i,m_i}} \end{pmatrix} & \begin{pmatrix} t_{i,1}[x'_1/x_1][\overrightarrow{b_1}'/\overrightarrow{b_1}][x'_1/\theta_{i,1}] \\ \vdots \\ t_{i,m_i}[x'_1/x_1][\overrightarrow{b_1}'/\overrightarrow{b_1}][x'_1/\theta_{i,m_i}] \\ t_{\omega,1}[x'_1/x_1][\overrightarrow{b_1}'/\overrightarrow{b_1}] \\ \vdots \\ u_{\omega,m_\omega}[x'_1/x_1][\overrightarrow{b_1}'/\overrightarrow{b_1}] \end{pmatrix} & :?P \end{array}$$

With the variable list $\overrightarrow{q}$ being a list of fresh variables, and $\overrightarrow{w_i}$ have the type of the indexes of the (inductive) types $\overrightarrow{\Phi_i}$. The $\theta_{i,j}$ are the variables that may be bound to the pattern $p_{i,j}$ by an `as` clause.

Remark that the type of the whole term is now $?P$ (which has a dependence on the newly defined $\overrightarrow{y_1}'$ and x'_1), and that we introduced a new existential variable to be the return predicate of the subproblem. As before, the typing condition that determines $?Q$ is that it must be equivalent to $?P$.

If we name s_i the solution to the subproblem associated with the constructor C_i, and $\overrightarrow{\gamma}''$ the list of all variables defined in Γ'' with $\overrightarrow{\Gamma}''$ their types, then a solution to the whole problem is the term

$$\Gamma', \Delta, \Gamma'' \vdash \left(\begin{array}{ccc} (x_1) & [\overrightarrow{y_1}, x'_1 \vdash \forall \overrightarrow{\gamma}'' : \overrightarrow{\Gamma}'', ?P] & \\ \begin{pmatrix} C_1(z_1, \dots z_{r_1}) \; \texttt{as} \; x'_1 \\ \vdots \\ C_k(z_1 \dots z_{r_k}) \; \texttt{as} \; x'_1 \end{pmatrix} & \begin{pmatrix} \lambda \overrightarrow{\gamma}'' : \overrightarrow{\Gamma}'' \cdot s_1 \\ \vdots \\ \lambda \overrightarrow{\gamma}'' : \overrightarrow{\Gamma}'' \cdot s_k \end{pmatrix} & : \forall \overrightarrow{\gamma}'' : \overrightarrow{\Gamma}'', T \end{array} \right) \; \overrightarrow{\gamma}''$$

where the generalization over Γ'' is introduced in order to get rid of its duplication by forcing it to the value of the original one. It can seem very heavy to just replicate all of Γ'' without more selection, but the compilation of Coq already contains a phase to get rid of the useless generalization, so this phase will automatically simplify the expression.

3 Second Algorithm

In the second case we study, we have a return predicate P giving the awaited returned type, but it can now depend in a more complex way on the type of the x_i, namely instead of variables $\overrightarrow{y_i}$, we now use patterns.

The Gallina syntax is therefore:

$$\Gamma \vdash \begin{array}{l} \texttt{match } x_1 \texttt{ as } x'_1 \texttt{ in } I_1\_\overrightarrow{\pi_1}, \dots, x_n \texttt{ as } x'_n \texttt{ in } I_n\_\overrightarrow{\pi_n} \texttt{ return } P \texttt{ with} \\ \qquad\qquad\qquad\qquad \begin{array}{lccc} | & p_{1,1} \cdots p_{1,n} & \Rightarrow & t_1 \\ | & \vdots & \vdots & \vdots \\ | & p_{m,1} \cdots p_{m,n} & \Rightarrow & t_m \\ & \texttt{end.} & & \end{array} \end{array}$$

with the $\overrightarrow{\pi_i}$ being lists of patterns over the types of the indexes of x_i, and the predicate P valid in a context

$$\Gamma, \overrightarrow{w_1} : \overrightarrow{W_1}, x'_1 : I_1 \overrightarrow{a_1} \overrightarrow{\pi_1}(\overrightarrow{w_1}), \ldots, \overrightarrow{w_n} : \overrightarrow{W_n}, x'_n : I_n \overrightarrow{a_1} \overrightarrow{\pi_n}(\overrightarrow{w_n})$$

where $\overrightarrow{W_i}$ is a list of the types of the variables used in the patterns $\overrightarrow{\pi_i}$ and $\overrightarrow{\pi_i}(\overrightarrow{w_i})$ denotes the list of terms obtained when replacing the variables in the patterns $\overrightarrow{\pi_i}$ with the $\overrightarrow{w_i}$.

We will write the associated problem under the form:

$$\begin{array}{ccc} & (x_1 \ldots x_n) & [\overrightarrow{w_1}, x'_1(\overrightarrow{\pi_1}), \ldots, \overrightarrow{w_n}, x'_n(\overrightarrow{\pi_n}) \vdash P] \\ \Gamma \vdash & \begin{pmatrix} p_{1,1} \cdots p_{1,n} \\ \vdots \\ p_{m,1} \cdots p_{m,n} \end{pmatrix} & \begin{pmatrix} t_1 \\ \vdots \\ t_m \end{pmatrix} \end{array}$$

3.1 First special case

When $n = 0$ the problem is reduced to

$$\begin{array}{ccc} & () & [\vdash P] \\ \Gamma \vdash & () & \begin{pmatrix} t_1 \\ \vdots \\ t_m \end{pmatrix} \end{array}$$

The difference with section 2.1 is not very important, only that now P is given. So the only thing to do is checking that t_1 is of type P. If this is true, then a solution for the matching problem is the tern t_1.

Here flagging may be used the same way as in section 2.3 to spot the useless pattern lines and raise an error if needed.

3.2 Second special case

In this case all patterns corresponding to x_1 are reduced to variables, so the problem is of the form

$$\begin{array}{ccc} & (x_1 \ldots x_n) & [\overrightarrow{w_1}, x'_1(\overrightarrow{\pi_1}), \ldots, \overrightarrow{w_n}, x'_n(\overrightarrow{\pi_n}) \vdash P] \\ \Gamma \vdash & \begin{pmatrix} v_1 \; p_{1,2} \cdots p_{1,n} \\ \vdots \\ v_m \; p_{m,2} \cdots p_{m,n} \end{pmatrix} & \begin{pmatrix} t_1 \\ \vdots \\ t_m \end{pmatrix} \end{array}$$

where the v_i are all variables.

Here as the term P depends on the $\overrightarrow{w_1}$, the replacement is a bit more complex, namely we have to use a matching for the return predicate: call s_T a solution to the problem

$$\begin{array}{ccc} & (\overrightarrow{b_1}) & [\vdash S] \\ \Gamma, \overrightarrow{w_2}, x'_2(\overrightarrow{\pi_2}), \ldots, \overrightarrow{w_n}, x'_n(\overrightarrow{\pi_n}) \vdash & \begin{pmatrix} \overrightarrow{\pi_1} \\ v \end{pmatrix} & \begin{pmatrix} P[x_1/x'_1] \\ \text{True} \end{pmatrix} \end{array}$$

then a solution to the whole problem is

$$\begin{array}{ccc} & (x_2 \ldots x_n) & [\overrightarrow{w_2}, x'_2(\overrightarrow{\pi_2}), \ldots, \overrightarrow{w_n}, x'_n(\overrightarrow{\pi_n}) \vdash s_T] \\ \Gamma \vdash & \begin{pmatrix} p_{1,2} \cdots p_{1,n} \\ \vdots \\ p_{m,2} \cdots p_{m,n} \end{pmatrix} & \begin{pmatrix} t_1[x'_1/v_1] \\ \vdots \\ t_m[x'_1/v_m] \end{pmatrix} \end{array}$$

where S is the sort of the type P. The idea is that if $\overrightarrow{b_1}$ do indeed match with the pattern they are supposed to, then it gives the awaited P (that can now be defined since the $\overrightarrow{w_1}$ appear in the branch), and if not then it gives a filler (that we chose to be True). Remark that this matching is of the form studied in section 2, so it can be processed with the first algorithm, in such a way that we never create (at this point of the compilation) a new instance of the kind studied in section 3, which will be useful to prove termination.

3.3 General case

For the general case, we will introduce subproblems associated with constructors, as we did in section 2.3. This will however be a bit more complex, as we will use some more intermediate matching problems in order to filter the cases that are not allowed due to the patterns given for the types.

First, consider this matching problem, that we will use as a return predicate:

$$\Gamma, \overrightarrow{y_1} : \overrightarrow{B_1}, x_1' : I_1 \overrightarrow{a_1} \overrightarrow{y_1}, \overrightarrow{\gamma}'' : \overrightarrow{\Gamma}'' \vdash \begin{array}{c} (\overrightarrow{y_1}\ \overrightarrow{b_2}, \dots \overrightarrow{b_n}) \\ \begin{pmatrix} \overrightarrow{\pi_1}\ \overrightarrow{\pi_2} \dots\ \overrightarrow{\pi_n} \\ v \end{pmatrix} \end{array} \begin{array}{c} [\vdash S] \\ \begin{pmatrix} P \\ \text{True} \end{pmatrix} \end{array}$$

where S is the sort of the type P and Γ'' is the list of the types of Γ'', including the $\overrightarrow{b_i}$ that are used in the matching problem. We name s_T a solution to this matching problem.

As we did in section 2.3, classify the patterns with their head constructor, and define the associated $\overrightarrow{p_{i,j}}$, $\overrightarrow{q_{i,j}}$, $t_{i,j}$, $\overrightarrow{p_{\omega,j}}$, $t_{\omega,j}$ and $\theta_{i,j}$.

Now let us consider the constructor $C_i : \forall \overrightarrow{\phi_i} : \overrightarrow{\Phi_i}, I_1 \overrightarrow{a_1} \overrightarrow{b_{1,i}}(\overrightarrow{\phi_i})$ of the type I_1 of x_1. For this constructor define (again as in section 2.3) the context Γ_i to be:

$$\Gamma', \Delta, \Gamma'', \overrightarrow{z} : \overrightarrow{\Phi_i}, \overrightarrow{b_1}' := \overrightarrow{b_{1,i}}(\overrightarrow{z}), x_1' := C_i(z_1 \dots z_{r_i}) : I_1 \overrightarrow{a_1} \overrightarrow{b_1}', \Gamma''[\overrightarrow{b_1}'/\overrightarrow{b_1}][x_1'/x_1]$$

and if $\overrightarrow{W_1}$ is the type of the variables $\overrightarrow{w_1}$ appearing in the patterns $\overrightarrow{\pi_1}$, then define Γ_i' to be:

$$\Gamma_i, \overrightarrow{w_1} : \overrightarrow{W_1}$$

Here we can build the subproblem associated with the constructor C_i, that is:

$$\Gamma_i' \vdash \begin{array}{c} (z_1 \dots z_{r_i}\ x_2 \dots x_n) \\ \begin{pmatrix} \overrightarrow{q_{i,1}}\ \overrightarrow{p_{i,1}} \\ \vdots \\ \overrightarrow{q_{i,m_i}}\ \overrightarrow{p_{i,m_i}} \\ \overrightarrow{q}\ \overrightarrow{p_{i,1}} \\ \vdots \\ \overrightarrow{q}\ \overrightarrow{p_{i,m_i}} \end{pmatrix} \end{array} \begin{array}{c} [\overrightarrow{w_2}, x_2'(\overrightarrow{\pi_2}), \dots, \overrightarrow{w_n}, x_n'(\overrightarrow{\pi_n}) \vdash P] \\ \begin{pmatrix} t_{i,1}[x_1'/x_1][\overrightarrow{b_1}'/\overrightarrow{b_1}][x_1'/\theta_{i,1}] \\ \vdots \\ t_{i,m_i}[x_1'/x_1][\overrightarrow{b_1}'/\overrightarrow{b_1}][x_1'/\theta_{i,m_i}] \\ t_{\omega,1}[x_1'/x_1][\overrightarrow{b_1}'/\overrightarrow{b_1}] \\ \vdots \\ u_{\omega,m_\omega}[x_1'/x_1][\overrightarrow{b_1}'/\overrightarrow{b_1}] \end{pmatrix} \end{array}$$

with $\overrightarrow{q}$ having the same definition as in section 2.3, as well as x_1' and $\overrightarrow{b_1}'$. Remark that the variables x_1' and $\overrightarrow{w_1}$ now appear in the context Γ_i'. Define s_i a solution to this subproblem.

Here we introduce an intermediary matching problem to ensure that the $\overrightarrow{z}$ match with the pattern $\overrightarrow{\pi_1}$, this matching problem is

$$\Gamma_i \vdash \begin{array}{c} (\overrightarrow{b_1}') \\ \begin{pmatrix} \overrightarrow{\pi_1} \\ v \end{pmatrix} \end{array} \begin{array}{c} [\overrightarrow{b_1}' \vdash s_T] \\ \begin{pmatrix} s_i \\ \text{I} \end{pmatrix} \end{array}$$

with I the only constructor of the type True. Remark that in the branch where s_i appear the $\overrightarrow{w_1}$ are defined since they are bound by the pattern matching. Therefore the context in which we defined s_i is indeed the one it is used in in the branch.

If we have for each constructor a solution s'_i to the problem just above, then the master match is:

$$\Gamma', \Delta, \Gamma'' \vdash \left(\begin{array}{cc} (x_1) & [\overrightarrow{y_1}, x'_1 \vdash \forall \overrightarrow{\gamma}'' : \overrightarrow{\Gamma}'', s_T] \\ \left(\begin{array}{c} \mathcal{C}_1(z_1, \dots z_{r_1}) \\ \vdots \\ \mathcal{C}_{k'}(z_1 \dots z_{r_{k'}}) \end{array} \right) & \left(\begin{array}{c} \lambda \overrightarrow{\gamma}'' \cdot s'_1 \\ \vdots \\ \lambda \overrightarrow{\gamma}'' \cdot s'_k \end{array} \right) \end{array} \right) \overrightarrow{\gamma}''$$

Appendices

A Compilation example

To make things a bit clearer, here is an example of the compilation. The context is the following:

```
Inductive Ind : bool -> Set :=
  | C : forall b : bool, (if b then nat else bool) -> Ind b.

Variable v : Ind true.
```

and we want to compile the following definition

```
Definition w :=
match v in Ind true return nat with
  | C _ O => O
  | C _ (S n) => n
end.
```

The first round of compilation gives

```
Definition w :=
match v in Ind b return (if b then nat else True) with
  | C b t =>
  match b return (if b then nat else True) with
    | true =>
    match b, t return nat with
        | _, O => O
        | _, S n => n
    end
    | false => I
  end
end.
```

The second one processes the second level of matching, and gives, once we δ-reduce the `if-then-else` and simplify the abstraction, (as the Coq-compiler automatically does) for more readability:

```
Definition w :=
match v in Ind b return (if b then nat else True) with
  | C b t =>
  match b return ((if b then nat else bool) -> if b then nat else True) with
    | true => fun (t : nat) =>
    match return (forall t : nat, nat) with
      | =>
      match b, t return nat with
        | _, O => O
        | _, S n => n
      end
    end
    | false => fun (t : bool) =>
    match return (forall t : bool, True) with
      | => I
    end
  end t
end.
```

A third step is the application of the first special case for the matching with no variables, after applying it to the two places where it can be done, we get

```
Definition w :=
match v in Ind b return (if b then nat else True) with
  | C b t =>
  match b return ((if b then nat else bool) -> if b then nat else True) with
    | true => fun (t : nat) =>
    match b, t return nat with
      | _, O => O
      | _, S n => n
    end
    | false => fun (t : bool) => I
  end t
end.
```

Now we still have to compile the matching on `b` and `t`, it first gives (second special case)

```
Definition w :=
match v in Ind b return (if b then nat else True) with
  | C b t =>
  match b return ((if b then nat else bool) -> if b then nat else True) with
    | true => fun (t : nat) =>
    match t return nat with
      | O => O
      | S n => n
    end
    | false => fun (t : bool) => I
  end t
end.
```

and then, after two more steps of compilation, the first one expanding the matching on `t` and the second one making it more compact, the algorithm terminates, and gives

```
Definition w :=
match v in Ind b return (if b then nat else True) with
  | C b t =>
  match b return ((if b then nat else bool) -> if b then nat else True) with
    | true => fun (t : nat) =>
    match t return nat with
      | O => O
      | S n => n
    end
    | false => fun (t : bool) => I
  end t
end.
```

which is well-typed in Coq and composed only of simple matchings, as expected.