Certified functional programming

Program extraction within Coq proof assistant

Pierre Letouzey

July 2004

Summary

Acknowledgments

At the time of making the last adjustments to this manuscript, I'm tempted to just thank the whole world. Since that would risk to lengthen a document already extremely bulky, I will try to be a little more selective, with the risk to forget some people. May those forgotten ones forgive my ingratitude.

First, I would like to express my deep gratitude to Christine Paulin. During all these years, her supervision has really be exemplary. I was permanently impressed by her competence, her patience and her kindness. Lastly, she has always remained ready to listen to me in spite of her many obligations.

I then would like to thank the five other members cordially of the jury. I am very honored that they agreed to examine my work. Let's first thank Xavier Leroy: after being a teacher of mine during DEA, he is today the president of my jury. A big thank you also to the rapporteurs Stefano Berardi and Jean-François Monin for their attentive readings and their constructive comments. Thank you also to Véronique Benzaken, who accepted here to deviate from her predilection field. And a final thanks to Helmut Schwichtenberg, for having agreed to prolong our discussions on the extraction by this participation in my jury.

My gratitude also goes to all the members of the teams of research I belong or used to belong. Each one contributed to make my working environment quite exceptional, particularly pleasant and stimulating. That concerns of course the DÉMONS team of the LRI, with (in disorder) Jean-Christophe, Ralf, Claude, Evelyne, Judicaël, Xavier, Julien, Sylvain, Pierre, Nicolas, Benjamin, Jacek, Daria, Laurence, Jean-Pierre, Délia and Frédéric. Then comes the team LOGICAL from INRIA Rocquencourt, around people like Hugo Herbelin and Bruno Barras. In a broader way, I will like to greet the members of the LRI with who I have worked, with a special mention for Marwan Burelle. Lastly, I do not forget the team where I wrote my first Coq lines, namely the CROAP team (now LEMMA) from INRIA Sophia-Antipolis: a big thank you to Laurent, Yves, Laurence, Francis, Loïc and their colleagues.

In addition to the members of these teams, I would also like to thank the researchers having endure the initial youth problems of my new implementation of the Coq extraction. Some have contributed simple bugs reports, some others functionalities requests, some finally engaged more advanced discussions, anyway this new extraction owes them much. I think here for example the members of FT-R&D at Lannion, like Jean-François Monin, Laurent Gottely, and Cuihtlauac Alvarado. I am also deeply indebted with the group of Henk Barendregt in Nijmegen, in particular Milad Niqui, Bas Spitters and Luís Cruz-Filipe.

In the foreword of his thesis, Jean-Christophe Filliâtre thanked in particular some key proponents of free software: Linus Torvalds for Linux, Richard Stallman for Emacs, Donald Knuth for TEX, and Xavier Leroy for Objective Caml. The least which I can do is to thank them again. How indeed could I live without these tools? I add just here a huge thank you to Jean-Christophe for having familiarized me with quantity of such tools, and also provided a LATEX style for this document. By the way, thanks also to Vincent Zoonekynd, who is at the origin of the style for the boxes framing the examples of this manuscript.

If I look back in time, a certain number of people had a determining influence on my course until this thesis. I would like to greet here a certain number of my old professors of mathematics and computer science: Jean-Luc Ybert, André Lehault, Guy Méheut, Herve Gianella, Jacques Chevallet, Laurent Chéno and Roberto Di Cosmo. In particular, thank you Laurent Chéno for having taught at the same time the Caml language and the concept of proof of programs.

To finish, thanks to all my friends, Yann, Sandra, Gaël, Nathaëlle, Mylène, Nicolas, Eric, Mathilde, Louis, Lionel, Dimitri, Barbara and others, for their good mood, their touches of madness, and all their projects "foireux, but only halfway". Thanks infinitely to my family and especially to my parents. What would I be without them? My gratitude is quite difficult to translate into words. In addition, I wish the best to my sister Catherine, just accepted as "professeur agrégé". Lastly, thank you Antigone for all, and more than that.

Introduction

The need for certified programs

Today, it is obvious to say that software occupies a dominating place in our modern societies, including in critical roles. The list of these missions of confidence now filled by programs lengthens unceasingly. It can be a matter of controlling high-risk equipments like planes or nuclear thermal power stations, but it can also be more prosaic operations like the management of electronic payments. In any case, whether human lives or money are in questions, the stakes are huge.

Unfortunately, it is also a banality to note the perfectibility of these programs which surround us. Without yielding to catastrophism, we cannot but note that software failures regularly fill news headlines. The example the more frequently mentioned remains the explosion of the Ariane 5 rocket in 1996. More recently, one can quote the description by S. Humpich of a vulnerability in the credit cards system. More alarming, the software editor that currently dominates the PC market propose softwares that are repeatedly affected by safety faults, which open the door to all kinds of virus, worms or trojans. Finally, here is a testimony read in a discussion forum, answering an intervention opposing the low quality of PC software with the great reliability of those intended for aeronautics:

> *I will bring however one small nuance concerning your parallel with airliners. Their computer systems are not as solid as you say, I know something of it, I am pilot on A320! In fact we regularly have one or more calculators which fail. This is even normal, the software is regularly modified to answer the new legal or operational requirements, and so is also the hardware, without taking in account that the whole must undergo the attack of tens of electric transfers per day! Fortunately, all systems are doubled, tripled and even quintupled in some certain cases, with the result that while one reboots quietly and carries out its autotests, the other takes over, and safety is never compromised ...*

There either, the situation is thus not perfect[1], since the empirical solution of redundancy is not foolproof.

This unsatisfactory practical situation contrasts in a startling way with the point of view of scientists. “Computer science is an exact science”, wrote C. A. R. Hoare [45] in 1969. He and others like R. W. Floyd or E. W. Dijkstra have indeed build a precise mathematical framework for programming, specifying in the 1970s a concept of proof of program, as

[1] Even if in practical the great majority of air crashes are due to human errors...

rigorous as a proof of theorem. Alas, the programming is still too often approached like an experimental art: repetitions of tests, tests, errors and corrections. It is true that establishing formally a program's correctness asks currently very significant efforts, whereas it is often very easy to carry out tests. But these tests could finally prove to be quite expensive, and they can only rarely be exhaustive...

In any case, whether one wants to certify a program or simply to test it, the first step is to produce a specification, describing the awaited behavior of this program. The mathematical result to be proved in the case of a formal certification is that the program satisfies its specification. Obviously, this specification step is a key moment: if the specification is incomplete or incorrect, nothing will prevent a program that is certified to be correct with respect to this specification to behave badly at the time of its execution.

The previously mentioned works of C. A. R. Hoare, R. W. Floyd and E. W. Dijkstra are originally intended for imperative programs. For example, the Hoare logic is it made of assertions relating the values of the program variables. And a specification of a portion of code `C` has the form `{P}C{Q}`: if the pre-condition `P` is valid before the execution of the code `C`, then the post-condition `Q` will be valid after the action of `C` on the contents of the variables. In the same way, most tools for formal certification that are widespread in industry, like the B method [1], are focused on proof of imperative programs. This is undoubtedly explained simply by the omnipresence of imperative paradigm in industry, to the detriment of functional languages. The main exception to date is the creation and the use of Erlang at Ericsson. Nevertheless, we now will see that the functional languages are particularly well suited for the creation of certified programs.

The Curry-Howard isomorphism

All the functional languages have a common theoretical core, namely the λ-calculus. This is true of Lisp, Scheme, the members of the ML family like Ocaml, or Haskell. However in [26], H. Curry noticed in 1958 that in the simply typed λ-calculus, any well-typed term has necessarily a type that is a tautology in propositional intuitionistic logic[2], as soon as one assimilates functional type $A \rightarrow B$ and implication $A \Rightarrow B$.

In addition to this isomorphism between types and propositions, H. Curry also establishes a correspondence between terms and demonstrations. In particular, the constructive proof of an implication $A \Rightarrow B$ is a method which allows to transform any demonstration `a` of the property `A` into a proof `b` of the property `B`. Via the isomorphism, this proof of $A \Rightarrow B$ can thus be seen like a function which for any object `a` of type `A` associates an object `b` of type `B`.

In 1969, W A. Howard proposed a λ-calculus with dependent types that extends the isomorphism to first-order intuitionistic logic [46]. For example, a constructive proof $\exists$`x,P(x)` stipulating the existence of an object satisfying the property `P`, gives via this isomorphism a functional program building indeed this object. More precisely, this program will return a couple `(x,p)` in which `x` is the sought object, and `p` is a certificate showing that we indeed have `P(x)`.

[2] For more details on intuitionistic and constructive logics, one may consult [10] or [80].

Thereafter, this isomorphism was extended to all kinds of intuitionistic logical systems with increasing expressivity, like for example the types theory of Martin-Löf [58] or the Calculation of Inductive Constructions (CIC) which is used in the Coq proof assistant [78]. And extraction, that is the possibility of deriving a program from a proof, was used in practice in many systems, like PX [44], Nuprl [50], Coq, Minlog [30] or more recently Isabelle [13, 14].

One can make here a comparison with the Hoare method. If we wish to build a program certified taking as input an object `x` satisfying a pre-condition `P(x)` and then returning another object `y` satisfying a post-condition `Q(x,y)`, we only have to prove in a intuitionistic formalism the proposition $\forall$`x,P(x)`$\rightarrow\exists$`y,Q(x,y)`. The Curry-Howard isomorphism then allows us to automatically derive from our proof a functional program that is correct by construction.

In fact, the program obtained by a direct use of this isomorphism is not completely the one expected. Instead of a program taking a `x` and returning a `y`, one obtains rather a program with two arguments `x` and `p` and with two results `y` and `q`. And `p` and `q` are then two certificates corresponding respectively to the proofs of `P(x)` and `Q(x,y)`. The role of the extraction is then to generate the natural program, *i.e.* without logical certificates, and to justify this deletion of certificates. We first see how that is done in Coq, then we evoke the case of the other proof assistants.

The extraction in Coq

This thesis was thus devoted to the study of the extraction in Coq. In fact, it could have been entitled "Fifteen years of extraction in Coq", since it intervenes fifteen years after a first thesis on this same subject by C. Paulin [66, 67, 69]. We thus start by recalling what has been made at the time and then explaining why that has left the possibility of a new work in this field.

First of all, we should mentioned that Coq is built directly on top of Curry-Howard isomorphism: in particular a proof is directly represented internally by a λ-term. One is then tempted to say that a Coq proof *is* precisely a functional program, and that the extraction has only a recopy to make. But this approach, although correct, is too naive. Indeed, if one takes again the example of an existential proposition $\exists$`x,P(x)`, a intuitionistic proof of this proposition contains indeed the construction method for the witness `x`, but also logical justifications ensuring that `x` is appropriate, that is satisfies `P(x)`. From the programming point of view, the constructive skeleton will give us the wanted program, whereas the logical justifications are in general non-desirable in a program. The role of extraction is then to derive from one term `t` of type `T` a program `p` containing only the computational contents of `t`. And usually, correctness of this program `p` is guaranteed by a *realizability* relation `r` connecting `p` and the initial type `T`. This concept of realizability was initially introduced by S. C. Kleene in 1945 in [49]. The extraction function, which we will note $\mathcal{E}$, will be then correct if one can establish that a typing relation `t:T` implies the realizability relation $\mathcal{E}$`(t) r T` after extraction.

In her thesis, C. Paulin defines a realizability adapted to the Calculation of Constructions[3], and an associated extraction. A first version of this realizability is based on the semantic criterion of "pre-realized formulas". Then in a second step, C. Paulin proposes to replace this semantic criterion by one syntactic criterion, namely an object annotation by the user, to mark its computational nature or conversely its logical nature. For that purpose, instead of using only one `Prop` type of all propositions, one doubles this type with a `Set` type of informative propositions[4], whereas the propositions remaining in `Prop` are being considered as purely logical.

This theoretical extraction presented and proved correct by C. Paulin outputs extracted programs belonging to a restriction of the Calculation of Constructions, namely F_ω. This work were implemented in the assistant `Coq` thereafter, initially by C. Paulin for the extraction part towards F_ω, then by going B. Werner for the part going from F_ω to ML[5] (see [69]). Later on, J.-C. Filliâtre maintained and improved this implementation.

The contributions of this thesis

This initial work on the extraction `Coq` suffers from several limitations, some of them present at the creation of `Coq` extraction, others appeared during evolutions of the `Coq` system. Our work has primarily consisted in solving these limitations, which led us, as we will see, to an almost complete redesign of this extraction mechanism. At the same time, we made an effort to maintain as much as possible a compatibility with the previous extraction. And this new implementation of the extraction was gradually integrated in versions 7.0 and following of `Coq`.

Complete support of the universes

A first limitation relates to the question of `Coq`*universe* or *sorts*. We have already evoked the division of `Prop`, the universe of propositions, in two universes, one named `Set` for the computational propositions, the other, `Prop`, for the logical propositions. But these two universes `Prop` and `Set` are themselves parts of a higher level universe named `Type`. However `Coq` treats these universes like all other terms of the system. One can thus form a universal quantification on `Type`, which will concerns in particular `Set` and `Prop`. The extraction C. Paulin was not able to work with such terms that includes reasoning on universes. Any term where the higher universe `Type` was appearing was quite simply considered as not-extractable. And this was not an artificial restriction, but indeed an intrinsic limitation of this extraction method. A theoretical study as well as a practical study of the `Type` level extraction thus remained to be made, which appeared to be not-obvious. This limitation was becoming quite awkward as the use of the universe `Type` tends now to to spread in `Coq` developments. For example, this universe allows to write data types compatible at the same time with `Set` and with `Prop`. In addition, `Type` is also frequently used in association with strong elimination in developments based on reflexion (or two levels approaches, see

[3] These Constructions were not yet Inductive at this moment...

[4] In the thesis of C. Paulin, `Set` was named `Spec`

[5] The various languages "spoken" by the extraction were Caml (`Lourd` then `Light`), LazyML and now Ocaml, Haskell and Scheme.

for example [16]). These developments were thus at the origin out of field of application for Coq extraction.

Solving the typing problems

A second problem appears in the translation step from F_ω to one of the concrete ML-like functional languages. This step, present in the implementation, is not dealt with by C. Paulin's thesis. However the F_ω type system is much richer than that ML's one. The extraction can then produce a extracted term not typable in ML. In practice, such a conflict of typing seldom occurs, but occurs nonetheless. And the increasingly frequent use of the `Type` universe tends to multiply these situations. Moreover, this low frequency of the typing conflicts can be also explained by a form of user's self-censorship, not very inclined to use `Type` if he knows beforehand that this will leads to a not-extractable development. We have developed a method consisting in identifying the locations of these typing conflicts, then in solving them via the use of low-level functions influencing types. This way, the extracted code is always usable with standard compiler for the target language. For the moment this method was only implemented for the Ocaml language.

Correctness of strict evaluation

The principal problem of the old extraction is its lack of safety concerning the execution of extracted terms via a strict strategy, as in Ocaml. In fact, execution of some extracted terms in Coq version 5.x and 6.x can finish abnormally on several fatal errors, or quite to the contrary loop and never finish. For a good understanding of the problem, let's first come back to the original extraction towards F_ω. In her thesis, C. Paulin has proved that her extraction produces well-typed terms in F_ω. This system satisfying the strong normalization property, the reduction of these extracted terms is thus ensured to proceed correctly, whatever strategy is employed, either strict or lazy.

But let's now add an axiom `A` in Coq. When one extracts a proof that uses this axiom, the extracted program will be incomplete unless one provide manually a program p correspondent with this axiom `A`, that is realizing it: `p r A`. When this program `p` is typable in F_ω, the preceding result of correctness still holds, and any reduction will proceed without problem. Unfortunately, three axioms that are particularly natural and significant for the expressivity of the system do not have typable realizations in F_ω. Any extracted program using the realizations of these axioms can then theoretically see its execution fail, and that occurs indeed in some cases, at least when evaluation strategy is strict. Here are these three axioms:

$$\frac{\perp}{A} \qquad \frac{x = y \quad P(x)}{P(y)} \qquad \frac{\mathcal{WF}(R) \quad \forall X, (\forall y, R(y,x) \to P(y)) \to P(y)}{\forall X, P(x)}$$

- the first axiom is the contradiction elimination, which allows to treat the impossible cases of proofs and programs. This axiom corresponds naturally to a function raising an exception. But such a realization leads sometimes to evaluations finishing abnormally on an uncaught exception.

- the second axiom is the equality elimination. With the most general version of this axiom, an equality between types makes it possible to modify the apparent typing of a term. That does not pose any problems in Coq, but on the other hand after extraction one can have all kinds of execution failures related to typing errors, for example `(0 0)`.
- the third axiom allows to prove a property P by induction over a well-founded predicate R. This axiom is realized easily by an fixpoint operator like `Y`. But when this realization is used, the old extraction can then generate terms whose strict evaluation will loop.

In fact, finding F_ω-typable realizations for these three axioms would have implied to consider as computational the falsity, the equality, and the well-foundedness, ending finally with almost no logical parts left. Instead, these three categories of problems, including the two already evoked in [69], were ignored or minimized via empiric means[6] After all, if one wished to ensure the correctness of the Ocaml-extracted terms, one could always work in a stripped-down version of the system, without these three axioms – which however are part the library initially loaded by the system.

To solve these problems of execution, we had to alter the extraction function $\mathcal{E}$ significantly, and in particular the elimination of logical λ-abstractions, in order to guarantee a correct evaluation whatever strategy is employed.

Support of the system's evolutions

Since the first work on extraction, Coq evolution has accentuated the limitations of this old extraction. We have already mentioned for example the increasingly frequent use of the universe `Type`. In addition, the three axioms which put in danger the correctness of the execution of the extracted terms are nevertheless essential for the expressivity of the system. So, when it has been possible to modify Coq underlying logical system in order to reinforce it and being able *to prove* these axioms, that has been done. And the extraction was then facing potentially incorrect situations even without the least addition of axioms. It thus became crucial to correct these problems, which was made.

Among the other evolutions of Coq which had an impact on the extraction, one can quote of course the change from the Calculation of Constructions to Calculation of Inductive Constructions, that is the addition of the primitive inductive types, or much more recently the adoption of a system of modules and functors. Among these evolutions, some are benign for the extraction. For example, the addition of the record types basically has not change anything, since they are visible only by the user and are translated internally into inductive types. In the same way, extracting the co-inductive types does not ask any particular work if the target language is lazy like Haskell, and the old extraction was already supporting this case. On the other hand, some enrichments required more substantial modifications of the implementation, but without impact on the extraction theory. This is the case for example of the support by B. Werner of the inductive types extraction or even of our adaptation of the extraction for the new modules system.

[6] The function `False_rec`, raising an exception associated with the contradiction elimination, was in particular always unfolded, since its definition itself was raising an exception.

Modules and interfaces

Let's stop for a second at this last extension of extraction in order to support the new Coq modules, at least for Ocaml extraction, as well as the generation of an interface for any code extracted towards this language. These two new features may seem minors in comparison with the solving of flaws that were endangering the correctness of extracted code. But we consider as really essential that the extracted code can be integrated easily in a broader development. The generation of the interfaces is in fact only one positive side-effect of the evoked solving of typing problems, but it now allows to predict the type of a extracted function only by inspecting the Coq type of the initial object.

Our contribution in brief

Finally, our new extraction is thus characterized by the three following points:

1. It endeavors to manage any Coq term.
2. It ensures that the execution of the extracted terms will be correct, both with a lazy or strict strategy.
3. It guarantees the good typing of the extracted code and provides an interface with this code (in Ocaml only for the moment).

In fifteen years, the Coq extraction has been transformed from a still experimental tool to a mature framework for development of certified code. In particular this tool is now able to treat significant and realistic examples, like for example the library of finite sets used in practical by Ocaml developers, based on modules and functors (see chapter 7).

Comparison with other extraction systems

We now will compare the extraction Coq with the similar tools present in other systems. This comparison will be centered on the two essential points of Coq extraction, namely the deletion of the logical parts in proofs, and then the translation into a true functional programming language. For more details, one can consider the chapter 6 of [66], which remains largely up-to-date.

Deletion of logical parts in proofs

First of all let us reconsider the distinction between logical part and computational part in a Coq term. In the typical example of a term whose type is `∀x,P(x)→∃y,Q(x,y)`, we have seen that the Curry-Howard isomorphism gives us a function with two arguments `x` and `p` and with two results `y` and `q`. If one wants to obtain a final function producing only `y` from `x`, it is appropriate to make sure that the logical certificates `p` and `q` do not intervene during this construction of `y`. If it is indeed the case, then `p` and `q` are simple decorations or *dead code* from the point of view of the computation of `y`. And normally, the deletion of such a dead code brings very significant profits concerning the size of the final program and its speed of execution.

Much work has been accomplished concerning the topic of automatic control of dead code. In the beginning, S. Berardi studied this dead code elimination in context of simply typed λ-calculus [12], then L. Boerio has extended these techniques to the second order [15]. Then F. Prost generalized this work with Pure Type Systems [72].

We have however chosen to remain compatible with the old extraction of Coq, which always relies on the annotation of the logical parts by the user. In practice this is not so much a constraint, since it is enough to annotate only each new definition of object. One has only to determine in advance, and once for all, the role of each object. By the way, these annotations via the universes `Prop`, `Set` or `Type` also fulfill a another role apart from helping the extraction: see the question of the impredicativity in the following chapter.

Let us recall that `Prop` is mainly a copy of `Set`, but with one logical meaning, whereas any object placed in universe `Set` and `Type` will be considered informative. We will see later that Coq typing system ensures that computations on informative objects do not depend on the computation results in logical parts. This prevents for example an unwanted use of a auxiliary logical construction at a computational location, and thus justifies the deletion during extraction of the objects having `Prop` as universe.

In fact, this method and the automatic dead code analysis are relatively orthogonal: although the extraction cannot eliminate dead code placed in an informative universe, it can on the contrary simplifying subterms which do not satisfy the dead code criteria (see in particular the elimination of logical singleton inductive in section 2.3.2). Besides, it would not be absurd to combine these two techniques, by applying for example a dead code analysis to extracted code. The extraction, syntactic process, would then remove at low cost the broad logical parts while a more complex and expensive dead code analysis could then work better on the small extracted terms.

If we study now the PX system [44], we note that the elimination of the logical parts relies in this system on a syntactic concept of formulas without computational contents (known as of rank 0). One finds in particular among these formulas to eliminate the Harrop formulas[7], but also manual annotations `◇A`, which are the formulas `A` which computational contents is deliberately hidden.

In Nuprl [50], one first finds a certain number of formulas preset as being of null contents. This concerns in particular equality and inequality, falsity and the relation `a∈A` which express that `a` is a proof of `A`. Beside that, in a type subset `{x:A|B(x) }`, the role of `B` is rather similar to the one of a Coq object in `Prop` : one can make use of it only for establishing a formula of null contents or another property on the right-hand side of a subset type.

Concerning Minlog, one finds in [30] distinction between Harrop formulas and other formulas that are considered informative. Moreover, H. Schwichtenberg told us during a private communication its interest for the use of two kinds quantifiers $\forall$ and $\forall_{nc}$, the second one binding a variable without computational contents. By the way, let us mention here many works around Minlog that aim at extracting programs from classical proofs.

Lastly, in Isabelle, S. Berghofer also carries out an automatic analysis of computational contents for the terms being extracted, that amounts to eliminate the Harrop formulas. On

[7]This is a well-known class of formulas without computational contents, gathering the formulas without disjunction or existential quantification in positive location.

the other hand the status of predicate variables is not decided as long as these variables are not instantiated. For a initial theorem with n predicate variables, it may be necessary to generate at worst 2^n alternative extractions to manage all the situations during the later use of this theorem (see p.64 [14]). On another side, in the case of Isabelle/HOL, the extraction relies on a manual annotation concerning the computational contents of inductive predicates (see p.84 [14]). It should be noted that Isabelle is based on a classical logic. In fact, unlike Minlog, the extraction only identifies the constructive parts of the terms, without seeking to work on the classical parts.

Translation to a true functional language

Since the beginning, the successive versions of the extraction in Coq have all aimed at producing source code for widespread functional languages, and not just outputting the raw λ-terms. Currently, our implementation supports three target languages: Ocaml [53], Haskell [31] and Scheme [48]. Among these three languages, the extraction towards Ocaml is most complete and mature, whereas on the contrary the one towards Scheme is still experimental. There are three principal reasons for the use of such external target languages:

- First of all, the certified code produces can more easily be integrated in broader developments. This is made possible by the production of readable interfaces[8]. One can for example obtain one autonomous program by adding to extracted code a manually-written part for managing inputs and outputs (see examples of chapter 5). It is also possible to develop a certified extracted library, which can be re-used thereafter in multiple projects. For example, the chapter 7 presents the formalization of a library of finite sets. This way, a broad community of programmers can profit from the extraction, including Coq non-users.
- Secondly, the use of such external languages allows substantial speed gains. As it was previously said, one Coq proof can be directly seen as a program. And its direct execution in Coq is possible via $\beta\delta\iota\zeta$-reduction (cf. for example the command `Eval compute`). But during this execution, Coq behaves primarily like an interpreter. And it is well known that this is much less efficient than the use of a compiler. A natural idea is then to use already existing and widespread compilers, like those for Ocaml or Haskell. It should be noted however that the internal reduction in Coq is currently improving: B. Grégoire worked at the implementation a compiler for the Coq terms [39, 40]. But this compiler cannot ignore as much logical parts as the extraction, because it must be able to to work with unclosed terms and to reduce under lambdas. And this, mixed with elimination with certain logical parts, can lead to not-termination (see part 2.3.2).
- Lastly, a pragmatic reason for this choice in favor of external target languages is the impossibility of writing back some extracted terms in Coq. We saw that in the beginning, the work of C. Paulin on Coq extraction was using a internal intermediary step This idea is also found in internal extraction of P. Severi & N Szasz [76]. But the theoretical system currently used in Coq differs appreciably of those used in these

[8] In practice, we even try to also produce source code that is as readable as possible.

studies. And the same enrichments that we have seen calling into question the correctness of the old extraction, also prevent from realizing henceforth a complete internal extraction. In particular, Coq now accepts the definition of informative fixpoints based on a logical decreasing measure. So is defined for example the the accessibility relation `Acc` and of its associated fixpoints `Acc_rec` and `Acc_iter` that we will study later. An internal extraction of these terms would then lead to fixpoints without decreasing measure, and thus potentially to non-strongly normalizing terms (see an example in section 2.3.2). However such objects with infinite reduction are proscribed in Coq.

A contrario, it is obvious that the use of a target internal language simplifies greatly the justification of the extraction correctness. Indeed, starting from a (not necessarily formal) proof of correctness by realizability of an internal extraction, one can then easily implement a procedure which builds explicitly, in addition to the extracted term, the proof that this particular extracted term realize indeed its specification. Such tools of automatic generation for particular correctness proofs have been implemented in particular in Minlog and Isabelle. Of course, having a external extraction does not exclude the creation of a formal correctness proof. But a stage should then be added to the process, namely a formalization of our target language semantics, in order to be able to express the adequacy between the extracted term, this semantics and the initial specification. For lack of time, this correctness proof have only be made on paper during this work, and is the subject of chapter 2.

It should be noted that these two systems are not confronted with the problems coming from the richness of Coq logic. In particular, the extracted functions in Minlog are typable in Gödel's system T, and one cannot obtain by extraction any non-structural (but well-founded) inductions or any other object that prevent an Coq internal extraction. One must choose between a rich logic and a complex but expressive extraction on the one hand, and a more minimal logic and a simple extraction on the other hand.

Moreover, S. Berghofer also mentions in [14] the possibility of generating true ML code starting from its internally extracted terms. This step, even if it may seem obvious because of the proximity between his internal terms and the ML syntax, must nevertheless be made with the greatest caution. One can indeed make a parallel with the generation of F_ω internal terms in the old Coq extraction, and their later translation towards ML in an informal step. We have indeed just seen that this could end with extracted terms whose strict evaluation fails, in spite of one result of correctness on the F_ω level.

Finally, in Nuprl, the extraction also produced an internal λ-term, that can be reduced by using an internal reduction machine inside Nuprl. But it does not seem currently that the correctness proof of an extracted λ-term can be obtained automatically.

Summary

This work begins with a progressive introduction to the Coq proof assistant. The first part of the chapter 1 presents via examples the principal features of this system, and in particular those which influence the extraction. The second part of this chapter exposes more formally the CIC, which is Coq's underlying theoretical system.

After this brief review of Coq and CIC, we start a first half of this manuscript dedicated to the presentation of our new extraction. The chapter 2 first of all present our redesign of the extraction function $\mathcal{E}$ for terms, and complementary two proofs of its correctness. The first proof is a syntactic proof stating that reduction of extracted terms cannot not fail, the second proof is a semantic proof ensuring the correctness of extracted terms with respect to theirs specifications.

The chapter 3 is then dedicated to the typing of the extracted terms. We start by studying in detail which kind of non-typability the extracted terms may presents in a ML-like typing system, then we propose a solution for skirting these difficulties.

Lastly, the chapter 4 supplements the description of our new extraction by presenting the last aspects of our work: extraction of modules, of co-inductive types, of record types, and lastly optimization of the extracted code. Finally we briefly present the implementation carried out, from both developer's and user's point of view.

The chapter 5 starts the second half of the manuscript, devoted to case studies. In this chapter, we first review the users contributions from the point of view of extraction, these users contributions forming an already consequent library of examples of Coq developments. Then we detail the case of two of these contributions, which put at fault the old extraction: the first on exceptions by continuations, due to J.-F. Monin, and the second on Higman's lemma, by H. Herbelin.

The chapter 6 gives a progress report on an ambitious project consisting in developing a library of arithmetic real exact by extraction of a development of constructive real analysis. The development in question is the C-CoRN project at the university of Nijmegen. We will see that one is still far from the goal, but that at the same time enormous progress were already accomplished. In addition, we present a small alternative experimentation around constructive reals made in collaboration with H. Schwichtenberg.

Lastly, the chapter 7 presents the certification of the Ocaml library of finite sets which we carried out in collaboration with J.-C. Filliâtre. This development is one of first to combine modules, functors and extraction.

Syntactic conventions

Throughout this document, fragments of source code given in example will be highlighted as follows:

```
Definition zero := 0.
```

The contents of these fragments could be in Coq, Ocaml or Haskell according to the example. The short examples will be mentioned with only font change, such as `(plus 0 0)`.

Some examples will also show user's interactions with Coq :

```
Coq < Eval compute in 1+1.
    = 2
    : nat
```

We reuse here the syntax of the old `coqtop` text interface: the lines preceded by the prompt "`Coq <`" correspond to user's inputs, whereas the other lines are Coq outputs. In the same way, the examples of use of Ocaml interactive loop will be presented as follows, with the character "`#`" as prompt:

```
# 1+1;;
- : int = 2
```

The Coq examples are intended to be used with Coq version 8.0 or later [78]. It should be noted that this version inaugurates a new syntax, much more pleasant. Beyond the use of this new syntax, we also have improved the readability of examples using some recent advanced features of the system:

- Some original ASCII keywords were replaced by theirs Unicode equivalents[9]:

`forall`	`exists`	`->`	`<->`	`<>`	`~`	`/\`	`\/`	`<=`
$\forall$	$\exists$	$\rightarrow$	$\leftrightarrow$	$\neq$	$\neg$	$\wedge$	$\vee$	$\leq$

- We used the most readable syntax for arithmetic expressions, for example `0+1` instead of `(plus O (S O))`, as allowed by the `Scope` mechanism.
- When the type of a quantification can be deduced from the context, Coq now authorizes its omission. We sometimes use this possibility.

[9] This can indeed be done in Coq, via the `Notation` command and the use of an Unicode-compatible interface, such as CoqIDE [78] for example.

Chapitre 1

A presentation of Coq

This chapter has a double goal:

- First of all, the reader discovering the `Coq` proof assistant will find here a quick description of this system, and more particularly its underlying logical system, namely the Calculus of Inductive Constructions (CIC in summary). Ideally, the only requirement for reading this part is a basic knowledge in type theory and in λ-calculus. Of course, this chapter does not intend to replace the documentation coming with `Coq`. The reader may thus consult as well:
  - the Tutorial [47] for a more gradual introduction,
  - the Reference Manual [78], and in particular its chapter 4 on the CIC, for an exhaustive formal description of `Coq`.
- At the same time, this presentation of `Coq` is of course centered on the extraction. Even if this extraction is introduced only in the following chapter, we will detail here all the features the CIC logical system that will influence the extraction.

1.1 An introduction via examples

1.1.1 Coq as a logical system

The objective goal of `Coq` is to be an proof assistant, and thus to allow the formalization of mathematical reasoning. Let us take an obvious statement: $A \to A$, where A is one unspecified proposition. Here is a proof in natural deduction (cf [71]):

$$\dfrac{\dfrac{}{A \vdash A}(Ax)}{\vdash A \to A}(\to I)$$

This proof can be directly transcribed in the `Coq` system. First, one is given a propositional variable.

```
Parameter A : Prop.
```

We will detail later this `Prop`, but here one can consider it simply as the set of all logical propositions.

```
Lemma easy : A → A.
Proof.
 intro.
 assumption.
Qed.
```

This portion of Coq script starts with the name and the statement of our lemma. Then between the keywords `Proof` and `Qed` is the proof of this statement, made of directives or *tactics*. These tactics reflect here the structure of the natural deduction proof. The `intro` tactic corresponds to the use of introduction rule for the arrow ($\rightarrow I$). And the `assumption` tactic corresponds to the axiom rule (Ax).

The script presented here is complete, but the system Coq, by its *interactive* nature, allows to build this proof step by step. Thus, if one stops after the order `intro`, the system prints the current state of the goal(s) to prove:

```
1 subgoal

  H : A
  ============================
   A
```

Here we have to find a proof of `A` under the assumption `A` (this assumption being named `H`). Many interfaces exist to facilitate this interaction with the Coq system, such as Proof-general [4], Pcoq [3], or more recently CoqIDE [78]. Since the extraction works on finished proof, we will not detail more this concept of interaction.

It should also be noted that Coq provides to the user a large variety of tactics. Here for example, for a statement that simple, the automatic research tactic `auto` would have been able to build the proof directly. There again, we will not detail these tactics. Indeed, the extraction does not depend on the way followed during the proof building, but only on the final form of the proof. In fact, `auto` builds here the same internal proof that `intro` followed by `assumption`.

Proofs as λ-terms

What is this internal representation of the proof? The system uses in fact λ-terms, following the Curry-Howard isomorphism. The underlying logical system of Coq, the CIC, is indeed a λ-calculus with a powerful type system. And always in application of the Curry-Howard isomorphism [46, 7, 38], the statements that can be expressed in Coq are the CIC types. Lastly, to check if `t` is indeed a valid proof of the statement `T`, we only have to checking if the type `T` is a legal type of the λ-term `t`.

Let us ask Coq which is the internal representation of the lemma `easy`:

```
Coq < Print easy.
easy = fun H : A ⇒ H
```

.../...

.../...

```
      : A → A
```

The syntax `fun x:X ⇒ T` is the `Coq` notation for the typed λ-abstraction λ`x:X.t`. This abstraction is here the effect of the `intro` tactic. As for `assumption`, its effect is the use of a variable in the context, here `H`. In a more general way, any tactic contributes to build piece by piece the complete λ-term of the proof. And finally, it is clear that `fun H:A ⇒ H` is indeed of type `A→A`.

These proofs in the form of λ-terms are usually handled only by the system. But nothing prevents the user to provide whole or part of its proof in this form, for example via the `exact` tactic.

```
Lemma easy : A → A.
Proof.
 exact (fun a ⇒ a).
Qed.
```

One can notice here that `Coq` is able to infer the type of `a`, which is necessarily A. It is hence optional to write this type. We will frequently use this possibility thereafter.

Finally, if the complete λ-term is known in advance, we can write it directly in the form of a `Definition`:

```
Definition easy : A → A := fun a ⇒ a.
```

Alternatively, one can also use a definition style where arguments are named during the declaration of the type, making useless the writing of the head λ-abstractions :

```
Definition easy (a:A) : A := a.
```

This style is certainly more concise, but not necessarily more readable. We will thus avoid it, except in the case of the recursive functions where it is essential (cf. lower).

The higher order

Up to now, the only basic type `A` we met had been fixed as a parameter, and the statement of the lemma `easy` related to this particular `A`. But one can also consider in `Coq` statements (that is types) speaking of sets of types. For example ∀`A:Prop, A→A` is a statement (*i.e.* a `Coq` type) that reads: "for any proposition `A` (that is object `A` belonging to the type `Prop`), `A` implies `A`". Such a universal typed quantification ∀`x:X, T` is also named *product* in `Coq`. This quantification not being restricted, the CIC is thus a higher order logic. One also speaks of *dependent types*, since generally the body `T` of the product depends on the variable `x` in the head of the product.

What can be the proof of the statement ∀`A:Prop, A→A`? The introduction of the "forall" consists in picking an unspecified object in the domain, and then carrying out the rest of the proof with this object. On the λ-calculus level, this results in one λ-abstraction: in response to a statement ∀`A:Prop, ...`, the proof will thus begin with `fun A:Prop ⇒...`. The rest of the proof is then the same one as the one of our `easy` lemma, which finally gives us the complete λ-term `fun A:Prop ⇒ fun a:A ⇒ a`, or simply `fun (A:Prop)(a:A) ⇒ a`.

It is noticeable that the λ-abstractions are used at the same time for building the proof of products and arrow types. This is no coincidence, one can indeed see a arrow type `A→B` as a non-dependent product `∀x:A,B` for which `x` does not appear in `B`. In fact, in Coq, syntax `A→B` is directly syntactic sugar for `∀_:A,B`.

1.1.2 Coq as a programming language

One can also approach Coq via the other side of Curry-Howard isomorphism, and look at the CIC not as a logical system, but more as a λ-calculus, that is a programming language. For example, with this vision, our `easy` lemma is the identity function on the type `A`.

Some standard inductive datatypes

For considering Coq as a programming language, it is necessary for us to be able to define the datatypes that one usually meet in other programming languages. The definition of these datatypes can be made easily in Coq via the use of inductive types. All inductive types that we present in this chapter belong to the standard library of Coq.

The boolean type is obtained via the following declaration:

```
Inductive bool : Set := true : bool | false : bool.
```

This declaration creates a new type, named `bool`, with two constructors `true` and `false` of type `bool`. The `Set` annotation indicates what should be the type of the `bool` type. We will reconsider this point thereafter; meanwhile `Set` can be seen as the set of all datatypes. In the same way, one defines the Peano integers this way:

```
Inductive nat : Set := O : nat | S : nat → nat.
```

The constructor `O` codes for zero, and `S` codes the successor function. A last usual example is the one of parametric lists:

```
Inductive list (A:Set) : Set :=
 | nil : list A
 | cons : A → list A → list A.
```

Here, the syntax `(A:Set)` expresses the fact that `list` depends on one parameter `A`. And for each use of this data structure, the parameter will be provided as an argument, a list of integers being for example `list nat`.

It should be noted that the system refuses certain inductive types whose definition is syntactically valid. These restrictions, which ensure the coherence of the system from the logical point of view, consist mainly in a positivity condition. The interested reader may consult the chapter 4 of [78] or even [68].

Pattern matching over inductive terms

To take advantage of these inductive types, the CIC is equipped with a primitive operator allowing pattern matching, whose syntax is `match... with...`. This allows for example to define the predecessor of an integer.

```
Definition pred : nat → nat :=
 fun n ⇒
  match n with
   | O ⇒ O
   | S m ⇒ m
  end.
```

The primitive matching of Coq is a simple, first level only matching:

- Each branch corresponds to a constructor and only one.
- In the branch corresponding to the constructor `C`, the pattern matched is inevitably the application of `C` to variables.

In addition to this primitive matching, we can also in Coq define more complex matchings, as in this double predecessor:

```
Definition predpred : nat → nat :=
 fun n ⇒
  match n with
   | S (S m) ⇒ m
   | _ ⇒ O
  end.
```

We will not detail these complex matchings, which are in fact translated at the internal Coq level into a succession of primitive matchings. Our double predecessor is thus the combination of two simple predecessors.

Lastly, let us mention to be complete that the syntax `match... with` accepts additional annotations via the keywords `as`, `in` and `return`. These annotations are almost always optional. But in certain situations Coq does not know how to infer automatically a type for the whole matching, or does it badly. These additional annotations allow to provide explicitly this type. We will encounter this situation later on, for example in the term `Acc_inv` (see section 1.1.4).

The structural induction

The last primitive construction of the CIC is the possibility of to define a term by structural induction on an inductive object. In its simplest version, this structural induction corresponds to the `Fixpoint` syntax. Let us define for example the addition of two unary integers:

```
Fixpoint plus (n m:nat) {struct n} : nat :=
 match n with
  | O ⇒ m
  | S n' ⇒ S (plus n' m)
```

.../...

.../...

```
 end.
```

The complete type nat→nat→nat of plus is here divided between two named arguments n and m on a side of the " : " and one result nat of the other. This makes it possible to specify via struct on which inductive argument will be made the induction. One speaks then of "guard" inductive argument, or "decreasing" inductive argument. Another possible choice of presentation would have been to draw aside only the first argument, which is then implicitly the "guard" argument, and to leave as result type nat→nat:

```
Fixpoint plus (n:nat) : nat → nat := fun m ⇒
 match n with
  | 0 ⇒ m
  | S n' ⇒ S (plus n' m)
 end.
```

A recursive definition is accepted only if any internal recursive call is done on a recursive argument which is structurally smaller that the initial recursive argument. Here for example n' is indeed a subterm of n. One will find a more precise definition of this "structurally smaller" in [78]. The reason for this constraint is, there again, the logical coherence. Without this condition on recursive calls, it would be very easy to build terms of any type A:

```
Fixpoint loop (n:nat) : A := loop n.

Definition impossible : A := loop 0.
```

The logical system would be then incoherent.

In fact, the syntax Fixpoint is not the most general one, because it declares the name of the function immediately, and does not allow recursive anonymous function. In particular one cannot imbricate one Fixpoint in another. Such a anonymous fixpoint can be defined in Coq via the syntax fix. Here an alternative definition of plus using this syntax:

```
Definition plus :=
 fix plusrec (n m:nat) {struct n} : nat :=
  match n with
   | 0 ⇒ m
   | S n' ⇒ S (plusrec n' m)
  end.
```

On Coq internal level, these two definitions of plus are identical. In fact, Fixpoint is only syntactic sugar for a Definition followed by a fix, as a Print plus can show. A typical example of use for an anonymous fix is the merge of two sorted lists, that can be found for example p.193 in one of the implementations of finite sets.

The term reduction

The principal interest of a programming language is to be able to execute the programs. This is possible with Coq. Its logical system CIC is indeed equipped with reductions rules:

(*beta*) CIC being basically a λ-calculus, one obviously finds the β-reduction

(*delta*) As this system authorizes the addition of new constants (via `Definition`, `Lemma`, etc), a rule of reduction named δ allows to replace a constant name by the body of this constant.

(*zeta*) `Coq` now includes primitive "let-in", that is local abbreviations `let x:=t in U`. A system rule named ζ allows to unfold these abbreviations, by replacing `x` by `t` everywhere in `u`.

(*iota*) The system also includes two rules dedicated to inductive types, both named ι. The first rule allows to reduce a matching, if the matched term starts with a constructor. By example, `match S N with O ⇒ O|S m ⇒ m end` can be reduced to `n`. The other rule related to inductive is the one can reduce a recursive function, as soon as its guard recursive argument is present and that it starts with a constructor.

These reductions can be used in `Coq` at any time, for example via the command `Eval compute`[1]:

```
Coq < Eval compute in pred (plus 2 2).
    = 3
     : nat
```

These computations in `Coq` have very strong properties in comparison with executions in more usual programming languages:

(i) First of all, the `Coq` reduction is a *strong* reduction, that is allowed at any place, even under a lambda or in the body of a recursive function. For example:

```
Coq < Eval compute in fun n ⇒ plus 1 n.
     = fun n : nat ⇒ S n
     : nat→nat
```

In comparison, the majority of usual languages freeze the body of functions, to reduce them only when all waited arguments are present.

(ii) Then, any reduction `Coq` is finite. This property is named strong normalization. Starting with a given term, any chain of reductions thus leads in a finite number of steps to a *normal form*, that is a non-reducible term. This results from the multiple constraints required before a term is accepted as valid in the CIC, as well as the former constraints on ι-reductions. On the contrary, in the immense majority of languages, it is very simple to write a not-terminating program.

(iii) In addition, `Coq` also satisfies the confluence property: if one considers two reduced of the same term, there exist two derivations of these two terms towards a common reduced term. This, associated with (ii), allows to show that for each initial term there exists in fact one and only one normal form. One is thus sure to obtain it in finite time, in whatever order the computations are made. This property is for example not

[1] The syntax `2` is by default translated into `S (S O)`, and so on for the other numbers. The new `Scope` mechanism [78] allows to change this translation according to needs ($\mathbb{Z}$, $\mathbb{R}$...).

satisfied by languages with side effects, in which evaluation order can influence the result.

The counterpart of these strong meta-theoretical properties is a certain lack of expressivity of CIC considered as a programming language. It is indeed impossible to speak there directly of partial functions, that is non everywhere defined. It is also impossible to define directly a general, non-structural, recursive function. In fact we will see later how to circumvent these two limitations, respectively via the use of pre-conditions and well-founded induction operators.

1.1.3 The universes of CIC

We now approach the mysteries of `Coq` that form the core of the extraction mechanism, namely the *univers* or *sorts* of `Coq`. It should be known that there is no `Coq` syntactic distinction between the basic terms (like `0`) and the types (like `nat`). In both cases, they are terms of the CIC. However any term of the CIC has a type, which is again a term of the CIC. One can thus wonder how looks a type of a type, or a type of a type of a type. Let us ask the system:

```
Coq < Check 0.
0
     : nat

Coq < Check nat.
nat
     : Set

Coq < Check Set.
Set
     : Type

Coq < Check Type.
Type
     : Type
```

Here appear `Set` and `Type`, which with `Prop` are three special objects of `Coq` named *sorts*. And these sorts will have a close relationship with types of types in the system:

- First of all, it is clear that `Set` and `Type` are types of types (of `0` and `nat` respectively). As for `Prop`, it is enough to define an inductive type in `Prop`, which can be done similarly to definition (already seen) of inductive types in `Set` :

```
Inductive True : Prop := I : True.
```

  `Prop` is then the type of the type of the constructor `I`.

- In addition, a property of the CIC states that all type of type can be reduced towards `Set`, `Prop` or `Type`.

Lastly, to be complete, let us note that the type of `Prop` is `Type`.

The `Type` universe

Let us first examine the `Type` sort. It is a strange object who seems to be his own type. If a set-theoretic vision is taken, that corresponds to a set of sets containing itself, which leads to Russel paradox. Even if this set-theoretic vision is here imperfect, it is nevertheless true that `Type:Type` allows to prove the inconsistency of the system [20]. A remedy is then to take an infinite hierarchy of sorts. In Coq, the `Type` sorts are implicitly subscripted by an integer, and for any index i one has: $\texttt{Type}_i\texttt{:Type}_{i+1}$. From the point of view the extraction, we will not need to distinguish among this infinity of sorts, and we will thus remain on the level of the approximation `Type:Type`.

Classification of universes

One can then ask for the reason of the presence of three sorts when only one, `Type`, could have been enough. Two independent criteria allow in fact to distinguish these three sorts:

- the predicative or impredicative nature
- the logical or computational nature (we will also say informative).

We now detail these two criteria successively. But before that, here how the Coq sorts are located with respect to these two criteria. Until version 7.4 of Coq, the situation could be summarized by the following diagram:

	imprédicatif	predicative
calculative	`Set`	`Type`
logic	`Prop`	

As from version 8.0, the `Set` sort is now by default predicative, which gives us the new following diagram:

	imprédicatif	predicative
calculative		`Set`, `Type`
logic	`Prop`	

Impredicativity

Even if the (im)predicative nature does not influence directly the extraction, we nevertheless will try to illustrate this concept. For an impredicative sort like `Prop`, one authorizes the creation of an element of `Prop` via an universal quantification on all elements of `Prop` (and thus in particular on this new element currently being defined). For example, we have already met the identity type on `Prop` :

```
Definition typeId : Prop := ∀A:Prop, A → A.
```

On the other hand, it is not possible to define the same term with predicative sort `Type` instead of `Prop`. That apparently works, but this is only an appearance, since the implicit indices of the two `Type` occurrences are in fact different:

```
Definition typeId' : Type_{i+1} := ∀A:Type_i, A → A.
```

The problem is even more obvious with `Set` and `Coq` 8.0. The following definition:

```
Definition typeId'' : Set := ∀A:Set, A → A.
```

generates an error explaining that `typeId''` is of type `Type` and not `Set`.

As mentioned previously, version 8.0 of `Coq` now has by default a predicative `Set` sort. In fact, a command line option for `Coq` 8.0 allows to return to the previous situation. In all the following, we use the default situation for `Coq` 8.0, namely with predicative `Set`, except otherwise mentioned (see by example the study of Higman's lemma in section 5.4). Anyway, the theoretical study of extraction that follows is independent of this question of impredicativity. We will thus keep on speaking of CIC for one or the other logical system obtained with or without the impredicativity of `Set`, even if the official name for the system without impredicativity is now pCIC (for "Predicative Calculus of Inductive Constructions").

Logical and informative universes

The other distinction between sorts relates to their logical or computational nature. First, we have initially a simple usage convention concerning `Prop` and `Set` :

- `Set` is intended to contain all the objects having computational contents, in particular the previously considered examples of usual datatypes for the programmer.
- The role of `Prop`, on the contrary, is to contain everything related with pure logic, such as for example the various justifications, pre- and post-conditions, in other words everything that can be ignored during computations.

The user thus chooses at the time of the definition of an object whether to place it in `Prop` or another sort. And this choice will be used by the extraction, which will ignore the logical parts placed in `Prop`.

With respect to the logic/computational duality, `Type` plays an ambiguous role. In particular, CIC contains a *cumulativity* principle, which allows to state that if `t:Set`, then one also has `t:Type`, and similarly if `t:Prop`, then one also has `t:Type`. Let us consider the identity on `Type` :

```
Definition id : ∀X:Type, X → X := fun (X:Type) (x:X) ⇒ x.
```

The cumulativity ensures that the following terms are well-typed: `(id Set nat)` or `(id nat O)` or `(id Prop True)` or finally `(id True I)`. By the cumulativity, certain terms of sort `Type` are thus in fact logical because originally belonging to `Prop`, and others are computational. In doubt, we must then consider that the objects in `Type` can have computational contents. This explains that on the extraction level, `Set` and `Type` will not be distinguished.

The logical inductive types

Concerning the logical operations, we have only presented yet the implication and the universal quantification. Let us now see how to define the other usual logical operators via inductive types in `Prop`. We have already met `True`, with its single constructor `I`. At the logical level the statement `True` admits thus an immediate proof, which is simply `I`. On the contrary, `False` is an inductive type without constructor, and is thus not provable in a empty context (the opposite would have as a consequence the inconsistency of the system).

```
Inductive False : Prop := .
```

The negation `¬A` is then defined as `A→False`.

The logical connectors `or` and `and` are defined as follows:

```
Inductive or (A B:Prop) : Prop :=
 | or_introl : A → or A B
 | or_intror : B → or A B.

Inductive and (A B:Prop) : Prop :=  conj : A → B → and A B.
```

One will meet these two inductive with Coq syntaxes ∧ and ∨.

Here is now the existential quantification:

```
Inductive ex (A:Type)(P:A→Prop) : Prop :=
   ex_intro : ∀x:A, P x → ex A P.
```

The Coq syntax for this existential is `∃x:A, P` (or `∃x, P` if `A` can be infered).

Finally the equality corresponds to inductive having only one constructor that simulates the reflexivity:

```
Inductive eq (A:Type)(x:A) : A → Prop :=  refl_equal : eq A x x.
```

The equality will be noted `=`, and the difference (that is the negation of the equality) will be noted $\neq$.

Thanks to pattern matching, one can prove the usual (intuitionistic) properties of all these logical operators. Here is one example:

```
Definition proj1 : ∀A B:Prop, A∧B → A :=
 fun (A B:Prop)(ab:A∧B) ⇒ match ab with (conj a b) ⇒ a end.
```

The elimination rules for inductive types

In fact, the dichotomy between `Prop` used as logical sort and `Set` used as computational sort is more than just a matter of usage convention. Rules authorizing or not the elimination of an inductive term (that is a matching on this term) differ indeed according to the sort of this inductive term. If it is `Prop`, then this elimination can only be used to build a term in `Prop`. On the other hand, one authorizes eliminations of inductive on `Set` and `Type` to build terms of all sortes[2]. the idea is that a logical inductive, therefore without computational contents, cannot influence a computation in `Set` or `Type`. And the CIC typing system guarantees this property. Let us examine the following example:

[2] When `Set` is impredicative, there exists however a restriction on the elimination of certain inductive in `Set` : they must be *small* before being authorized to build a term of sort `Type` (cf section 4.7 of [78])

```
Definition or_carac : ∀A B:Prop, A∨B → nat :=
 fun A B ab ⇒
  match ab with
   | or_introl _ ⇒ 0
   | or_intror _ ⇒ 1
  end.
```

If this example were legal, to know if an application of `or_carac` is reduced into 0 or 1, the contents of the logical proof `A∨B` would have to be calculated, which one wants precisely to avoid. The response of the system to this definition attempt is:

```
Error: Incorrect elimination of ab in the inductive type or.
The elimination predicate fun _:A∨B ⇒ nat has type A∨B → Set.
It should be one of : Prop.

Elimination of an inductive object of sort : Prop
is not allowed on a predicate in sort : Set
because non-informative objects may not construct informative ones.
```

There are nevertheless two exceptions allowing the elimination of one inductive on `Prop` in order to build an informative term. In these two cases, the form of inductive ensure that its matching brings no computational information.

- The first case is the one of an empty logical inductive, that is without constructor, like `False`. This elimination of `False` corresponds to the "ex-falso quodlibet" principle: if we have one term of type `False`, this implies that we are in a contradictory context, and we are then authorized to construct a term of any type.

```
Definition False_rect: ∀A:Type, False → A :=
 fun (A:Type) (f:False) ⇒ match f with end.
```

- The second exception deals with the logical singleton inductive. They are logical inductive with only one constructor, and this only constructor does not have informative arguments. One can give as example `and`, and especially `eq`. Having a term in such inductive types does not bring any computational content, since one knows inevitably which is the constructor of this term, and the arguments of this constructor are again without computational contents. Here follows for example the informative induction principle associated with `eq`, which in fact describes how to get from `(P y)` to `(P x)` when it is known that `x=y`:

```
Definition eq_rect: ∀A:Type, ∀x:A, ∀P:A→Type, P x → ∀y:A, x=y → P y
 := fun (A:Type)(x:A)(P:A→Type)(f:P x)(y:A)(e:x=y) ⇒
     match e with refl_equal ⇒ f end.
```

This principle and its logical sibling `eq_ind` are the basis of the `rewrite` tactic of replacement of terms by equal terms. By the way, it should be noted that these induction principles ... _rect and ... _ind are generated automatically by Coq when the corresponding inductive type is defined.

1.1.4 Prop/Set mixed terms

Until now, we have seen how to use the CIC as a programming language, and how to calculate with terms written with this CIC. One may wonder then why we need an automatic mechanism for extraction of programs starting from CIC terms, since we now that such a term *is* already a program? It is indeed true that the extraction of purely calculative terms is only one problem of translation[3] of a source language (CIC) towards target languages (Ocaml or Haskell). But the things get more complex (and more intersting) when we consider mixed terms, with logical and informative parts interlaced. The use of this style of terms, allowed by the CIC, is very convenient in many situations. One can in particular enrich a informative term with pre- and post-conditions, or use a well-founded induction for justifying the decrease of a measure at each recursive call of a fixpoint.

Pre-conditions

In addition to the natural need to express a specification of functions in the pre- and post-conditions form, the logical pre-conditions also bring a solution to the problem of partial functions definition. Let us consider, for example, an integer division function div of type nat→nat→nat, which is not defined when its second argument is zero. There are then (at least) three ways of proceeding:

(i) One can nonetheless define (div N O), using one arbitrary value, for example O. The problem is that a integer division can return O either as a legitimate result, or as mark of an abnormal situation. One cannot thus prove the following lemma any more:

```
Lemma div_gives_zero : ∀n m:nat, (div n m)=0 → n<m.
```

(ii) the second solution is to simulate a exceptions mechanism. That can be done by extending the output domain nat into a $\mathtt{nat}_\perp$, as is domain theory. Rather than modifying all the types used, there is one generic method, namely the use of an inductive option:

```
Inductive option (A:Set) : Set :=
 | Some : A → option A
 | None : option A.
```

Thus, div will turn over None if its second argument is O, and (Some r) if not, r being then the true result of the computation. The disadvantage of this method is the heaviness of this encoding: for each use of a division, it will be necessary to carry out one matching for either reaching the true result, or again to turn over None.

[3] We will see in fact that even this translation is not so simple, since the type system of CIC is much more powerful than the ones of the functional target languages.

(iii) the last possibility is to express the fact that the second argument must be non-null via a logical pre-condition. The type of `div` then becomes `nat→∀n:nat, n≠0→nat`. The type of the second argument is not given any more via an arrow type, that is an anonymous product, but by a named product (by `n`) in order to be able to refer to it in the logical assertion. It is the method which approaches the most the concept of partial definition. Indeed the function `div` is not defined outside the validity domain of the logical assertion. The counterpart is that one must now provide a logical proof of non-nullity as third argument during each call to `div`.

```
Coq < Lemma two_non_zero : 2≠0. auto. Qed.
two_non_zero is defined

Coq < Eval Compute in (div 3 2 two_non_zero).
     = 1
     : nat
```

Post-conditions

In addition to pre-conditions, it is also possible to express logical post-conditions, and hence combining both pre- and post-conditions allows to give the specification of a function in a Hoare-like style [45]. Thus a function of type $A \to B$, pre-condition P and of post-condition Q corresponds to a constructive proof of the formula:

$$\forall \mathtt{x:A},\ \mathtt{(P\ x)} \to \exists \mathtt{y:B},\ \mathtt{(Q\ x\ y)}$$

We will not transcribe this formula in `Coq` by using its already encountered existential quantifier `ex` on `Prop`. Indeed, that would amount to regard the result of the function as being not-informative. We rather use a informative existential quantification, named `sig`.

```
Inductive sig (A:Set)(P:A→Prop) : Set :=
 exist : ∀x:A,P x → sig A P.
```

It should be noted that `sig A (fun x ⇒ P x)`, which expresses the existence of one informative object `x` checking the logical property `(P x)`, is also written `{x:A|(P x)}`. This thus gives us the following general form for a function with pre- and post-conditions:

$$\forall \mathtt{x:A},\mathtt{(P\ x)} \to \{\ \mathtt{y:B} \mid \mathtt{Q\ x\ y}\ \}$$

For example a version with post-condition of our integer division function can be specified as follows:

```
Definition div : ∀a b:nat,b≠0 → { q:nat | q*b ≤ a ∧ a < (S q)*b }.
```

One can then find the calculative result of `div` via one matching[4]

[4] The syntax `let...:=... in...` of this example is a shortcut for a matching on an inductive type with only one constructor. In the same way a syntax `if... then... else...` exists as a shortcut for any matching on inductive types with two constructors.

```
Coq < Eval compute in let (q,_) := div 3 2 two_non_zero in q.
     = 1
     : nat
```

This inductive `sig` can be also used to solve the partial function problem in a fourth way, via a restriction of the starting domain. One can indeed express the type of the non-null integers by `{ n:nat | n≠0 }`. This approach is equivalent to the one via by pre-conditions, and as it is slightly less natural in Coq, we will not use it.

Informative disjunction

Beside `sig`, another type is frequently used that combines informative and logical parts. This is `sumbool`, which is a counterpart of the logical disjunction `or`, except that it is placed in the `Set` universe:

```
Inductive sumbool (A B: Prop) : Set :=
  | left : A → sumbool A B
  | right : B → sumbool A B.
```

The type `(sumbool A B)` is also noted `{A}+{B}`. The consequence of the use of `Set` in the definition of `sumbool` is that one can test if an object of type `sumbool` starts with `left` or `right`, even if one is in an informative part. In comparison, that would be illegal with an object of type `or`. On the contrary, the arguments of `left` and `right` remain logical, which prevent to analyze it for constructive ends. From a computational point of view, `sumbool` is thus a type that is really similar to `bool`, it simply contains in addition some logical annotations. This type is primarily used to express results of decidability, and can be read "there exists an algorithm to determine if `A` or `B`". For example:

```
Theorem eq_nat_dec : ∀n m, {n = m} + {n ≠ m}.
```

Or:

```
Lemma le_lt_dec : ∀n m, {n ≤ m} + {m < n}.
```

Well-founded induction

The definitions by well-founded induction constitute a last example of use of logical parts in a computational function. First, we need to formalize the fact for a relation of being well-founded.

```
Section Well_Founded.
 Variable A : Set.
 Variable R : A→A→Prop.
 Inductive Acc : A→Prop :=
```

.../...

.../...

```
  Acc_intro : ∀x:A,(∀y:A,R y x → Acc y) → Acc x.

  Definition Well_founded := ∀a:A,Acc a.
```

The use of a `Section` and `Variable` allows to factorize the common dependencies, here over a type `A` and over a logical relation `R` on this type. An element is known as accessible `(Acc x)` with respect to `A` and `R` iff all predecessors of `x` by `R` are themselves accessible. This inductive definition may seem strange, because of the lack of basic case. In fact, the universal quantification ensures that an initial element (without predecessor by `R`) is directly accessible. The finiteness of the inductive objects implies that one can interpret `Acc` as follows: we have `(Acc x)` if all the sequences of successive predecessors by `R` starting from `x` are finite. Finally the relation `R` is said to be well-founded if all the points in `A` are accessible by `R`.

We then need an inversion of the `Acc` inductive: if one point `x` is accessible, then all its predecessors are accessible. This is obtained by pattern matching on `(Acc x)`:

```
Definition Acc_inv : ∀x:A,Acc x → ∀y:A,R y x → Acc y :=
 fun x a ⇒
  match a in Acc x return ∀y,R y x → Acc y with
   | Acc_intro x' f ⇒ f end.
```

The two annotations "`in Acc x`" and "`return ∀y,R y x → Acc y`" can be ignored during first reading. They are needed to specify which must be the type of the `match ... with...` subterm. These annotations are optional in the many simple situations where the system can infer a suitable type. It is thus the case of all matchings encountered until now, since they produce objects of obvious types like `nat`. But here, the type of the second argument `f` of `Acc_intro` depends on the first argument `x'`, which is not visible outside of the term. In fact, `x'` is necessarily `x`, but the system currently does not know how to discover that, hence the need for a manual annotation. For a more general discussion on the need for annotations, see page 84 of [68].

We now show that if an informative predicate `P` is propagated by `R`, then `P` is valid in any accessible point.

```
 Section Acc_iter.
  Variable P : A→Type.
  Variable F : ∀x,(∀y,R y x → P y) → P x.
  Fixpoint Acc_iter (x:A)(a:Acc x) {struct a} : P x :=
   F x (fun y h ⇒ Acc_iter y (Acc_inv x a y h)).
 End Acc_iter.
End Well_founded.
```

There are two astonishing things in this `Acc_iter` definition:

- First of all we build an informative term (since its sort is `Type`) by induction over `a`, which is a logical inductive object. This is legal in `Coq`, but can seem to be a violation of the principle of the `Prop`/`Set` duality, according to which a logical part should not influence a informative computation. In fact this influence is limited to just provide the insurance that this recursion will terminate. True computation is in fact carried out in the function `F`, which cannot use the contents of `a` because of the restrictions on sorts in pattern matchings.

- the second shocking point is that the recursive call in `(Acc_iter X a)` is done on the recursive argument `(Acc_inv x a y H)`, which does not seem structurally smaller than the initial recursive argument `a`. But if `a` is of form `(Acc_intro _ F)`, then `(Acc_inv x a y H)` can be reduced in `(f y H)`. As `f` is the function contained in `a`, this definition satisfies indeed the structurally decreasing criterion of Coq.

Let us mention a last function related to well-foundedness:

```
Coq < Check well_founded_induction.
well_founded_induction
     : ∀(A:Set) (R:A→A→Prop),
       well_founded R →
       ∀P:A→Set,
       (∀x:A, (∀y:A, R y x → P y) → P x) →
       ∀a:A, P a
```

This is an alternative to `Acc_iter`, in with the relation `R` is supposed well-founded, and that allows us to obtain `(P x)` for all `x` without any restriction.

As an application, this well-founded induction allow us to define functions. Let us consider for example `A = nat` and `R = lt`, that is the strict order on `nat`. The standard library of Coq contains a proof named `lt_wf` stating that this order is well-founded. We can then define our integer division `div` by successive subtractions rather than by structural induction:

```
Definition div : ∀a b:nat,b≠0 → { q:nat | q*b ≤ a ∧ a < (S q)*b }.
Proof.
 intro a; pattern a; apply (well_founded_induction lt_wf); clear a.
 intros a Hrec b Hb.
 elim (le_lt_dec b a); intros Hab.

  assert (H : a-b < a). omega.
  elim (Hrec (a-b) H b Hb); simpl; intros q (Hq,Hq').
  exists (S q); simpl; omega.

  exists 0; omega.
Qed.
```

1.1.5 Coq extensions

A certain number of Coq extensions were voluntarily not presented here. They will be the subject of specific studies in chapter 4. This concerns in particular:

- the new system of modules of Coq (section 4.1)
- the co-inductive types (section 4.2)

1.2 A formal presentation of the CIC

We now will give a more formal presentation of the CIC. This formalism will be used in the theoretical chapter which comes next. The notations used here correspond as much as possible to those of chapter 4 of the Reference Manual [78]. The principal exception concerns the environments and the contexts. For simplicity reasons, we will merge these two concepts. First, let us specify the terms syntax of the CIC.

1.2.1 Syntax

Definition 1 (terms) *Terms of the* CIC *are given by the following grammar:*

$$\begin{array}{rcl} T & ::= & s \\ & | & x \;\;|\;\; c \;\;|\;\; C \;\;|\;\; I \\ & | & \forall x : t,\, t \;\;|\;\; \lambda x : t,\, t \;\;|\;\; \texttt{let}\ x := t\ \texttt{in}\ t \;\;|\;\; (t\ t) \\ & | & \texttt{case}(t,\, t,\, t\ \ldots\ t) \\ & | & \texttt{fix}\ x_i\ \{x_1/k_1 \!:\! t \!:=\! t\ \ldots\ x_n/k_n \!:\! t \!:=\! t\} \end{array}$$

where:

- *s indicates a sort, among* `Set`*,* `Prop` *or* $\texttt{Type}_i$*. We will omit the index of* $\texttt{Type}_i$ *as long as it does not intervene explicitly.*
- *x, c, C, I are identifiers, that respectively refer to variables, constants, inductive constructors and inductive types.*
- *for the fixpoint, the k_i are integers that correspond to the number of arguments awaited by the x_i components.*

To lighten syntactic expressions, we sometimes use vectorial notations:

- $(f\ \overrightarrow{x})$ for $(f\ x_1\ \ldots\ x_n)$
- $\forall\overrightarrow{x : X},\, T$ for $\forall x_1 : X_1,\, \ldots \forall x_n : X_n,\, T$
- $\lambda\overrightarrow{x : X},\, t$ for $\lambda x_1 : X_n,\, \ldots \lambda x_n : X_n,\, t$

And we use the notation $|\overrightarrow{x}|$ to specify the size of vector $\overrightarrow{x}$, when this size is significant and non-obvious.

It should be noticed that the syntax chosen here differs somewhat from the concrete syntax `Coq` :

- the λ is more concise than the keyword `fun`.
- the `match... with` syntax, even if it is a improvement with respect to usage comfort, is hardly adapted to theoretical reasoning, especially in the presence of additional annotations `as`, `in` and `return`. We use instead a $\texttt{case}(P,\, e,\, \overrightarrow{f})$ syntax in which e is the matched object, P is the elimination predicate and the $\overrightarrow{f}$ functions correspond to the branches put in fonctional form[5]: the equation $(C\ x\ y\ z) \Rightarrow t$ becomes the function $\lambda x,\, \lambda y,\, \lambda z,\, t$. By the way, let us mention that a pattern matching can perfectly have

[5] In fact, this syntax is similar to the one used in versions 5.x of `Coq`. It was still usable in versions 7.x via the keyword `Case... of`.

zero branch, in which case we will note it $\texttt{case}(e,\, P,\, \varnothing)$. Concerning the predicate P, if e is an inductive object of type $(I\ \overrightarrow{q}\ \overrightarrow{u})$, then the typing rules that follow force P to be of the form $\lambda\overrightarrow{u},\, \lambda x : (I\ \overrightarrow{q}\ \overrightarrow{u}),\, P_0$. The equivalent in Coq concrete syntax is then:

$\texttt{match E have X in } (I\ \overrightarrow{\_}\ \overrightarrow{u}) \texttt{ return } P_0 \texttt{ with ... end}$

- The Coq concrete syntax allows the definition of a block of mutually recursive anonymous functions via:

  $\texttt{fix } f_1 \texttt{:} T_i \texttt{:=} t_i \texttt{ with ... with } f_n \texttt{:} T_n \texttt{:=} t_n \texttt{ in } f_i$

  We will use following shortened syntax here:

  $\texttt{fix } f_i\ \{f_1/k_1 : T_1 := t_1\ \ldots\ f_n/k_n : T_n := t_n\}$

  where for each f_i component, the integer k_i indicates the rank of the inductive argument on which induction is performed. This integer corresponds to the `struct` annotation of Coq concrete syntax. We will name this inductive argument the "guard", because it will be used to control the fixpoint reduction (see reductions below). This is then called guarded induction.

Definition 2 (contexts) *A context Γ is a list that can contain the following elements:*

- *assumptions* $(x : t)$
- *definitions* $(c := t : t')$
- *inductive declarations* $\texttt{Ind}_n(\Gamma_I := \Gamma_C)$, *where n is the number of parameters, and Γ_I and Γ_C are two contexts respectively containing some inductive types and theirs constructors.*

Compared to the notations of the Reference Manual, we have chosen not to place the parameters in a specialized context, but to let them appear at the same time in Γ_I and Γ_C. the n annotation allows to emphasize that the n first products in all the elements of Γ_I and Γ_C correspond to parameters. Besides, this is closer to the actual Coq implementation. For example, let us write the declarations of unary integers and of polymorphic lists in this syntax:

$$\texttt{Ind}_0(\texttt{nat} : \texttt{Set} := \texttt{O} : \texttt{nat};\ \texttt{S} : \texttt{nat} \rightarrow \texttt{nat})$$
$$\texttt{Ind}_1(\texttt{list} : \texttt{Set} \rightarrow \texttt{Set} := \\ \quad \texttt{nil} : \forall A{:}\texttt{Set},\, \texttt{list}\ A;\ \texttt{cons} : \forall A{:}\texttt{Set},\, A \rightarrow \texttt{list}\ A \rightarrow \texttt{list}\ A)$$

1.2.2 Reductions

Definition 3 (reductions) *The reductions of the* CIC *are as follows:*

(*beta*) $((\lambda x : X,\, t)\ u) \rightarrow_\beta t\{x \leftarrow u\}$

(*delta*) $c \rightarrow_\delta t$ *if current context* Γ *contains* $(c := t : T)$.

(*zeta*) $\texttt{let } x := t \texttt{ in } u \rightarrow_\zeta u\{x \leftarrow t\}$

(*iota*) $\texttt{case}(C_i\ \overrightarrow{p}\ \overrightarrow{u},\, P,\, f_1\ \ldots\ f_n) \rightarrow_\iota f_i\ \overrightarrow{u}$
when C_i is the i-th constructor of an inductive type having $|\overrightarrow{p}|$ parameters.

(*iota*) *if* F *is the recursive block* $f_1/k_1 : A_1 := t_1 \ \ldots\ f_n/k_n : A_n := t_n$*, then:*
$(\texttt{fix}\ f_i\ \{F\}\ u_1\ \ldots\ u_{k_i}) \rightarrow_\iota (t_i\{f_j \leftarrow \texttt{fix}\ f_j\ \{F\}\}_{\forall J}\ u_1\ \ldots\ u_{k_i})$
if the "guard" argument u_{k_i} *starts with a constructor.*

These reductions are *strong*: they are authorized at any position inside a term, via the usual compatibility rules. On the contrary, we will also have to consider later *weak* reductions, that is reductions that can only happen at the top of the term, or either on the left or on the right of an application or at the head of a `Case`. Said otherwise, reductions that can only happen outside any binder. We will use $\rightarrow_r$ to indicate a step of any of the strong reductions $\rightarrow_\beta$, $\rightarrow_\delta$, $\rightarrow_\iota$ or $\rightarrow_\zeta$.

Starting from these reductions, one then defines the convertibility relation $=_{\beta\delta\iota\zeta}$ and the cumulativity order $\leq_{\beta\delta\iota\zeta}$ which are used below in the typing rule (Conv).

Definition 4 (convertibility) *Two terms* u *and* v *are convertible (noted* $u =_{\beta\delta\iota\zeta} v$*) if they have a common reduced form* w*, that is such as* $u \rightarrow_r^* w$ *and* $v \rightarrow_r^* w$*.*

Definition 5 (cumulativity) *The cumulativity order* $\leq_{\beta\delta\iota\zeta}$ *is recursively defined by:*

- *If* $u =_{\beta\delta\iota\zeta} v$ *then* $u \leq_{\beta\delta\iota\zeta} v$
- $\texttt{Type}_i \leq_{\beta\delta\iota\zeta} \texttt{Type}_j$ *as soon as* $i \leq j$
- $\texttt{Set} \leq_{\beta\delta\iota\zeta} \texttt{Type}$
- $\texttt{Prop} \leq_{\beta\delta\iota\zeta} \texttt{Type}$
- *If* $T =_{\beta\delta\iota\zeta} T'$ *and* $U \leq_{\beta\delta\iota\zeta} U'$ *then* $\forall x : T, U \leq_{\beta\delta\iota\zeta} \forall x : T', U'$

It is noticeable that due to the δ-reduction, these two concepts depend implicitly on a context.

Definition 6 (`arity`) *An arity is a term convertible to a sort or a product* $\forall x : T, U$ *with* U *being again a arity. After reduction an arity can thus be written as* $\forall x_1 : X_1, \ldots \forall x_n : X_n, s$*. One then speaks of an arity of sort* s*.*

1.2.3 Typing

We now give a condensed definition of the typing rules of the Cic. Once again, we refer to the Reference Manual [78] for a detailed version of these rules and their explanations.

Definition 7 (typing rules) *The typing judgement* $\Gamma \vdash t : T$*, which means that* T *is one valid type for* t *in the context* Γ*, is defined simultaneously with the property* $\mathcal{WF}(\Gamma)$ *of context good formation, via the rules of figure 1.1.*

Let us now give the side conditions of the typing rules for inductive types:

1. In rule (Prod), the condition $\mathcal{P}(s_1, s_2, s_3)$ on sorts is:
   $(s_2 = s_3 = \texttt{Prop}) \vee$
   $(s_2 = s_3 = \texttt{Set} \wedge s_1 \neq \texttt{Type}) \vee$
   $(s_1 = \texttt{Type}_i \wedge s_2 = \texttt{Type}_j \wedge s_3 = \texttt{Type}_k \wedge i \leq K \wedge j \leq k)$
   By the way, to let `Set` be impredicative (again), it is enough to remove the condition $s_1 \neq \texttt{Type}$ on the second line.

$$\mathcal{WF}(\varnothing) \qquad \frac{\Gamma \vdash T : s \quad x \notin \Gamma}{\mathcal{WF}(\Gamma; (x : T))} \qquad \frac{\Gamma \vdash t : T \quad c \notin \Gamma}{\mathcal{WF}(\Gamma; (c := t : T))} \qquad \text{(WF)}$$

$$\frac{\mathcal{WF}(\Gamma)}{\Gamma \vdash \texttt{Set} : \texttt{Type}_i} \qquad \frac{\mathcal{WF}(\Gamma)}{\Gamma \vdash \texttt{Prop} : \texttt{Type}_i} \qquad \frac{\mathcal{WF}(\Gamma) \quad i < j}{\Gamma \vdash \texttt{Type}_i : \texttt{Type}_j} \qquad \text{(Ax)}$$

$$\frac{\mathcal{WF}(\Gamma) \quad (x : T) \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{\mathcal{WF}(\Gamma) \quad (c := t : T) \in \Gamma}{\Gamma \vdash c : T} \qquad \text{(Var)(Cst)}$$

$$\frac{\Gamma \vdash T : s_1 \quad \Gamma; (x : T) \vdash U : s_2 \quad \mathcal{P}(s_1, s_2, s_3)}{\Gamma \vdash \forall x : T,\, U : s_3} \qquad \text{(Prod)}$$

$$\frac{\Gamma \vdash \forall x : U, T : s \quad \Gamma; (x : U) \vdash t : T}{\Gamma \vdash \lambda x : U,\, t : \forall x : U,\, T} \qquad \frac{\Gamma \vdash t : \forall x : U, T \quad \Gamma \vdash u : U}{\Gamma \vdash (t\ u) : T\{x \leftarrow u\}} \qquad \text{(Lam)(App)}$$

$$\frac{\Gamma \vdash t : T \quad \Gamma; (x := t : T) \vdash u : U}{\Gamma \vdash \texttt{let}\ x\ \texttt{:=}\ t\ \texttt{in}\ u : U\{x \leftarrow u\}} \qquad \text{(Let)}$$

$$\frac{\Gamma \vdash U : s \quad \Gamma \vdash t : T \quad T \leq_{\beta\delta\iota\zeta} U}{\Gamma \vdash t : U} \qquad \text{(Conv)}$$

$$\frac{\mathcal{WF}(\Gamma) \quad \texttt{Ind}_n(\Gamma_I := \Gamma_C) \in \Gamma \quad (I : A) \in \Gamma_I}{\Gamma \vdash I : A} \qquad \text{(I-Type)}$$

$$\frac{\mathcal{WF}(\Gamma) \quad \texttt{Ind}_n(\Gamma_I := \Gamma_C) \in \Gamma \quad (C : T) \in \Gamma_C}{\Gamma \vdash C : T} \qquad \text{(I-Cons)}$$

$$\frac{\begin{array}{c}\text{for all } (I : A) \in \Gamma_I, \quad \Gamma \vdash A : s \\ \text{for all } (C : T) \in \Gamma_C, \quad \Gamma; \Gamma_I \vdash T : s_C \\ \mathcal{I}_n(\Gamma_I, \Gamma_C)\end{array}}{\mathcal{WF}(\Gamma; \texttt{Ind}_n(\Gamma_I := \Gamma_C))} \qquad \text{(I-WF)}$$

$$\frac{\begin{array}{c}\texttt{Ind}_n(\Gamma_I = \Gamma_C) \in \Gamma \quad (I : \forall \overrightarrow{p : T},\, A) \in \Gamma_I \quad \sigma = \{\overrightarrow{p} \leftarrow \overrightarrow{q}\} \\ |\overrightarrow{p}| = n \quad \Gamma \vdash e : I\ \overrightarrow{q}\ \overrightarrow{u} \quad \Gamma \vdash P : B \quad \mathcal{C}(I\ \overrightarrow{q} : A_\sigma; B) \\ \text{for all constructor } C_i : \forall \overrightarrow{p : T},\, \forall \overrightarrow{x : X},\, I\ \overrightarrow{p}\ \overrightarrow{y}, \\ \Gamma \vdash f_i : \forall \overrightarrow{x : X_\sigma},\, P\ \overrightarrow{y_\sigma}\ (C_i\ \overrightarrow{q}\ \overrightarrow{x})\end{array}}{\Gamma \vdash \texttt{case}(e,\, P,\, f_1 \ldots f_m) : P\ \overrightarrow{u}\ e} \qquad \text{(Case)}$$

$$\frac{\forall i,\ \Gamma \vdash A_i : s_i \quad \forall i,\ \Gamma; \overline{(f : A)} \vdash t_i : A_i \quad \mathcal{F}(\overrightarrow{f}, \overrightarrow{A}, \overrightarrow{k}, \overrightarrow{t})}{\Gamma \vdash \texttt{fix}\ f_j\ \{f_1/k_1 : A_1 := t_1\ \ldots\ f_n/k_n : A_n := t_n\} : A_j} \qquad \text{(Fix)}$$

FIG. 1.1: Typing rules for CIC

2. In rule (I-WF) of good formation for an inductive definition, $\mathcal{I}_n(\Gamma_I, \Gamma_C)$ gathers all the side conditions that must be fulfilled for this definition to be valid:
   - All the names contained in Γ_I and Γ_C must be new and distinct.
   - As we have not made explicit the parameters, but only their number n, it should be checked that all the declarations of Γ_I and Γ_C start with the same n products $\forall\overrightarrow{p : P}$, and that all occurrences of the inductive type I in Γ_C is applied to at least $\overrightarrow{p}$.
   - For all $(I : A) \in \Gamma_I$, A must be an arity of a sort s_I.
   - For all $(C : T) \in \Gamma_C$, T must be a type of constructor for one of the inductive types I defined in Γ_I, *i.e.* T must be of the form $\forall\overrightarrow{p : P}, \forall\overrightarrow{x : X}, I\ \overrightarrow{p}\ \overrightarrow{y}$. Moreover the s_C sort in the typing premise of T must be s_I.
   - T must also check the positivity condition with respect to all the types of Γ_I. This condition is essential to guarantee strong normalization, but does not intervene in the extraction. We will thus not detail it.
3. In rule (Case), the σ substitution replaces the n formal parameters $\overrightarrow{p}$ by n concrete parameters $\overrightarrow{q}$. And the condition $\mathcal{C}(I\ \overrightarrow{q} : A_\sigma; B)$ expresses the fact that the arrival type of the `case` must be compatible with the inductive I which is matched:
   - One has $\mathcal{C}(I : (\forall x : X,\ A); (\forall x : X,\ B))$ iff for all x, one has $\mathcal{C}(I\ x : A; B)$.
   - One has $\mathcal{C}(I : \texttt{Prop}; I \rightarrow \texttt{Prop})$
   - One has $\mathcal{C}(I : \texttt{Prop}; I \rightarrow s)$ for a $s \neq \texttt{Prop}$ sort iff I is an empty or singleton logical inductive type.
   - One has $\mathcal{C}(I : \texttt{Set}; I \rightarrow s)$ for any s sort.
   - One has $\mathcal{C}(I : \texttt{Type}; I \rightarrow s)$ for any s sort.

   In the previous definition:
   - An empty logical inductive type is an inductive of sort `Prop` with zero constructor.
   - An singleton logical inductive type is an inductive of sort `Prop` with only one constructor whose non-parametric arguments are all of sort `Prop`.

   Let us note that if `Set` is taken impredicative, one should accept $\mathcal{C}(I : \texttt{Set}; I \rightarrow \texttt{Type})$ only when I is a small inductive, that is one whose constructors cannot have not-parametric argument of sort `Type`.
4. In rule (Fix), the condition $\mathcal{F}(\overrightarrow{f}, \overrightarrow{A}, \overrightarrow{k}, \overrightarrow{t})$ requires:
   - for all i, A_i must be of the form $\forall\overrightarrow{x : X}, A'_i$, with at least k_i products, and the X_{k_i} type of the k_i-th product must be inductive.
   - moreover t_i can only contain decreasing recursive calls: if f_j appears in t_i, then it must have at least k_j arguments, and its k_j-th argument must be structurally smaller than the initial inductive argument x_{k_i}. The exact definition of this "structurally smaller" can be found in the Reference Manual. Informally, it is equivalent to say that any subterm of an inductive term obtained by going through at least one constructor is structurally smaller than the starting term.

1.2.4 Properties

A first property of CIC states that if there is $\Gamma \vdash t : T$, then there is a sort $s \in \{\texttt{Prop}, \texttt{Set}, \texttt{Type}\}$ such as $\Gamma \vdash T : s$. Any t well-typed term thus admits at least a sort s.

In fact, one cannot speak rigorously of "the" type and of "the" sort of a t term, because there is no unicity of types in the CIC. For example an object of type `Prop` also admits the type `Type` via the rule (Conv) and cumulativity.

However, a term cannot admit at the same time `Prop` and `Set` as a sort. One can thus speak about the smallest sort of a term, with respect to the cumulativity order ($\texttt{Prop} \leq \texttt{Type}$ and $\texttt{Set} \leq \texttt{Type}$). This smallest sort will also be named *principal sort*.

In the same way, all types of a term are comparable via the cumulativity order. One can even show that there exists a type, unique modulo conversion, which is smaller than all the other types with respect to the conversion order. It will be named *principal type*. In fact, this concept has only interest for types which are arities. In the other cases, all the possible types are equal modulo conversion. And we will then speak sometimes, by abuse, of "the" type.

With respect to a context Γ, a term T is a type if it admits for type a sort s in this context. In fact, rather than these `Coq` types, it is a superset of those which will play a crucial role for the extraction:

Definition 8 (type scheme) *A type scheme is a well-typed term which admits at least one arity as type, that is has the form $\forall x_1 : X_1, \ldots \forall x_n : X_n,\ s$ with s being a sort.*

In other words, a type scheme is a term which will become a type as soon as it is applied to sufficiently many arguments. For example $\lambda X : \texttt{Type},\ X \to X$ is a type scheme: once applied to a type, one obtains the corresponding arrow type. On the contrary, $\lambda X : \texttt{Type},\ \lambda x : X,\ x$ is not a type scheme: it may happen that a type is obtained by applying it (for example with `Set` and `nat`), but one can also not obtain a type (for example by the application to `nat` and `0`).

Lemma 1 (stability) *the* CIC *admits the following results:*

1. (*Subject Reduction*) *Let t be a term that reduces to u. Then any type T of t is also a type of u. And any s sort of t is also a sort of u.*
2. *During the substitution of a variable in a term, the type of this term can obviously change. More precisely, if T is a type of t, then $T\{x \leftarrow u\}$ is a type of $t\{x \leftarrow u\}$. But the following properties are nonetheless preserved:*
   - *If t has as a* `Prop` *sort, it is the same for $t\{x \leftarrow u\}$*
   - *If t is a type scheme, it is the same for $t\{x \leftarrow u\}$*
   - *If t has an inductive type, it is the same for $t\{x \leftarrow u\}$*
3. *Lastly, concerning the applications, if t has* `Prop` *as sort, it is also true for $(t\ u)$, and if t is a type scheme, so is $(t\ u)$.*

PROOF. We will admit these results here. Please refer to theoretical studies of the CIC, such as for example [83]. □

It should be noted that for each stability property above, the reciprocal is false:

1. Let us take an unspecified type in `Prop`, say `True`. Then $((\lambda X : \texttt{Type},\, X)\ \texttt{True})$ can be reduced to `True`. However this last term accepts `Prop` as type, and not the first one. And in this example, $\texttt{Type}_0$ is a sort of `True`, but not of the initial term, which has only $\texttt{Type}_1$ as principal sort.
2.
   - Let us use the context $\Gamma = (X : \texttt{Type})$. Then $(\lambda x : X,\, x)\{X \leftarrow \texttt{True}\}$ admits `Prop` as sort whereas $\lambda x : X,\, x$ has `Type` for principal sort.
   - In this same context, $(\lambda x : X,\, x)\{X \leftarrow \texttt{Set}\}$ is one type scheme whereas $\lambda x : X,\, x$ was not one originally.
   - Let us take now $\Gamma = (b : \texttt{bool})$. If $T = \texttt{case}(b,\, \texttt{Type},\, \texttt{nat},\, \texttt{bool})$, then the term $t = \texttt{case}(b,\, T,\, \texttt{0},\, \texttt{true})$ does not admit an inductive type, whereas $t\{b \leftarrow \texttt{true}\}$ admits as a type $T\{b \leftarrow \texttt{true}\}$ which is convertible to `nat`.
3. Let us take $\texttt{Id} = \lambda X : \texttt{Type},\, \lambda x : X,\, x$. It is not a term of sort `Prop`, nor a type scheme. Yet (`Id True`) has sort `Prop`, and (`Id Set`) is a type scheme.

In all the cases, the counterexamples make use of the sort `Type`, which thus illustrates the precautions to be taken for in order to conceive an extraction able to manage `Type`. This being said, it is nevertheless necessary to relativize the importance of the problems implied by the possible change of the principal type during reduction:

Lemma 2 *Let t be a well-typed term of* CIC *that reduces to u, and let T and U be respectively principal types of t and u.*

(i) *One has $U \leq_{\beta\delta\iota\zeta} T$.*

(ii) *If t is not a type scheme, $T =_{\beta\delta\iota\zeta} U$.*

(iii) *t is a type scheme iff u is a type scheme.*

(iv) *t and u have the same principal sort (if we consider the various $\texttt{Type}_i$ sorts as only one sort `Type`).*

PROOF.

(i) It is enough to point out that by "subject reduction", T is always a type of u. Consequently, the property of principality of U gives us the desired result.

(ii) If t is not a type scheme, that means that T is not an arity. However, if we have $U \leq_{\beta\delta\iota\zeta} T$ without $U =_{\beta\delta\iota\zeta} T$, it means that these two types are arities with final sorts $s_U < s_T$.

(iii) $U \leq_{\beta\delta\iota\zeta} T$ implies that:

- eiher U and T are equal modulos $\beta\delta\iota\zeta$, and in particular are jointly arities or not.
- either U and T are distinct modulos $\beta\delta\iota\zeta$, both of them being arities, with final sorts $s_U < s_T$.

(iv) There are two cases: if t and u are type schemes, then they have `Type` as principal sort. If they are not, then they share exactly the same types (modulo $\beta\delta\iota\zeta$), and thus the same sorts.

□

The conservation of the principal sorts during reduction (iv) seem to be invalidated by the first of the preceding counterexamples. Is this conservation indeed true? With Normandy as native country, it is normal to answer yes and no. Yes, if one is interested just in the distinction `Set`/`Prop`/`Type`, and forgets the indices of $\texttt{Type}_i$. And no, in the opposite case, as shown by the counterexample. In fact the point of view of the extraction will be it first.

One thus finds again a border between the logical terms and the informative ones: a logical term (*i.e.* of principal sort `Prop`) cannot become informative (*i.e.* of principal sort `Set` or `Type`) during a reduction, and reciprocally.

We will also have to study the possible forms of the CIC terms that are closed and in normal form.

Lemma 3 *Let t be a well-typed term of* CIC*, closed and in normal form modulo $\beta\delta\iota\zeta$.*

1. *If t has an inductive type $(I\ \overrightarrow{v})$, it starts with a constructor of this I type.*
2. *If t has a product type, it is either:*
   a. *a partially applied inductive constructor*
   b. *an inductive type which can be alone or applied to arguments, including partially.*
   c. *an λ-abstraction*
   d. *a fixpoint $(\boldsymbol{fix}\ f_i\ \{\ldots\}\ \overrightarrow{u})$. And in this case, if the "guard" argument is the k_i-th, then $|\overrightarrow{u}| < k_i$.*
3. *If t has a sort s as type (*i.e. *t is a type), it is either:*
   a. *a sort*
   b. *a totally applied inductive type*
   c. *a product*

PROOF. We proceed by induction on the typing of t, and by case on last rule:

- First of all, the rule (VAr) cannot produce closed term, and the rules (Cst) and (Let) build not-normal terms.
- The rule (Ax) corresponds to the case (3a).
- The rule (Prod) corresponds to the case (3b).
- (Lam): One cannot be in the parts (1) or (3) of the statement, and the part (2c) is clearly checked.
- (App): Let us take $t = (t'\ u)$. The induction hypothesis on $\Gamma \vdash t' : \forall x : U, T$ leave us four cases:
  a. $t' = (C\ \overrightarrow{v})$. Then $t = (C\ \overrightarrow{v}\ u)$. If C is still partially applied in t, then t have a product type, and we are in part (2a) of the statement. And when C is completely applied, t has an inductive type: the case (1) of the statement is fulfilled.
  b. $t' = (I\ \overrightarrow{v})$. Then t is only an inductive type with more arguments. If this inductive type is now completely applied, it has a sort as type, and one is in case (3b). If not, its type is still a product, and one is in case (2b).

c. t' cannot be an λ-abstraction, otherwise $(t'\ u)$ would be reducible.

d. t' is then a fixpoint with missing arguments. Let us suppose that u is the "guard" argument awaited by this fixpoint. This u is then an inductive term, and the induction hypothesis corresponding to $\Gamma \vdash u : U$ shows that it starts with one constructor. The fixpoint is thus reducible, which is contradictory with the initial assumptions. One is thus still in the situation of a fixpoint missing some arguments. Lastly, this lack of arguments necessarily implies that this fixed point has a product type. The parts (1) and (3) of the statement are thus excluded, and the part (2) is correct via the case (2d).

- (Conv): We directly use the induction hypothesis.
- (I-Type): If $\Gamma \vdash I : A$ and arity A has at least one product, then the situation (2b) is checked. And if A is directly a sort, we are in case (3b).
- (I-Cons): We are in the situation (1) if the constructor awaits no argument, and otherwise in the situation (2a) .
- (Box): The induction hypothesis concerning the head of the `case`, which is an inductive term, shows that this term starts by a constructor. The `case` is thus reducible, and that contradicts the original assumption.
- (Fix): A fixpoint without argument always expects at least one (the guard argument). One cannot thus be in parts (1) and (3) of the statement. On the other hand, the part (2d) of the statement is clearly valid.

□

Curiously, the statement itself of this lemma shows that there are not other cases to study for the type T of t. Indeed, T can also be picked as closed and in normal form. The part (3) of the statement then affirms that it is either a sort, a product or an inductive type.

Let us also notice that the proof of this result only requires the absence of redex at precise places, namely at the top, on the left and right handside of an application and at the head of a `case`. This result will then remain perfectly valid when we will study the weak reduction of the CIC.

1.2.5 CIC$_m$: an CIC variant adapted to the semantic study

The presentation of the CIC given above is appropriate for the more syntactic part of the theoretical study we will make concerning extraction. On the other hand, in the second time, we will make a more semantic study of the extraction correctness in the section 2.4. And for that, the CIC as formulated here will be slightly unsuited. Let us present here the alternative CIC$_m$ that we will then use.

The principal concern in this part 2.4 will be that CIC authorize the silent promotion of a proposition to the rank of informative type. For example `True`, originally of `Prop` type, can be also seen as having the type `Type`, via the typing rule (Conv). This will be awkward later one, since one will wish to give a different semantics to a proposition and an informative type, and this without having to look at principal type derivation to know in which case we are.

A solution is then to use a mark on the syntax level to announce the use of a cumulativity $\texttt{Prop} \leq \texttt{Type}$. Thus $\texttt{True}^\dagger$ will indicate the `True` object promoted to the `Type` level. This marking technique dates back to our DEA work [54], itself reusing the idea of [28]. For similar reasons, though less fundamental, it will be also interesting to mark the promotion of a `Set` object to the `Type` level. Thus $\texttt{nat}^\ddagger$ will indicate the `nat` type seen at `Type` level. On the syntax level of `Coq`, these marks would translate into explicit "casts"; $\texttt{True}^\dagger$ and $\texttt{nat}^\ddagger$ would be written `(Prop_Type True)` and `(Set_Type nat)`, with:

```
Definition Prop_Type (t:Prop) : Type := t.
Definition Set_Type (t:Set) : Type := t.
```

Before continuing the presentation of this system CIC_m with marks, let us announce first that we will somewhat restrict the power of cumulativity in this system, for simplifying the presentation. For example, the initial (Conv) rule authorizes to write:

$$\frac{\Gamma \vdash t : \texttt{nat} \to \texttt{Set} \quad (\texttt{nat} \to \texttt{Set}) \leq_{\beta\iota} (\texttt{nat} \to \texttt{Type})}{\Gamma \vdash t : \texttt{nat} \to \texttt{Type}}\text{(Conv)}$$

From now on, we will authorize the use of the cumulativity only on types and not on type schemes: $\texttt{Set} \leq \texttt{Type}$ is accepted but not $\texttt{nat} \to \texttt{Set} \leq \texttt{nat} \to \texttt{Type}$ any more. This prohibits in particular the preceding example, but one can approximate it via one η-expansion: η-expansion:

$$\cfrac{\cfrac{\cfrac{\Gamma \vdash t : \texttt{nat} \to \texttt{Set}}{\Gamma; (x : \texttt{nat}) \vdash (t\ x) : \texttt{Set}}\text{(App)}}{\Gamma; (x : \texttt{nat}) \vdash (t\ x) : \texttt{Type}}\text{(Conv)}}{\Gamma \vdash \lambda x : \texttt{nat},\ (t\ x) : \texttt{nat} \to \texttt{Type}}\text{(Lam)}$$

For a more detailed study on the cumulativity in higher order type theories, one can look at [57].

Let us now return to our marks † and ‡. They are annotations that can be applied to any term of CIC. And we now replace the rule (Conv) by four new rules (Conv) (CumT) (CumS) (CumP), in order to make compulsory the presence of marks after the use of the cumulativity $\texttt{Prop} \leq \texttt{Type}$ or $\texttt{Set} \leq \texttt{Type}$:

$$\frac{\Gamma \vdash U : s \quad \Gamma \vdash t : T \quad T =_{\beta\iota\delta\zeta} U}{\Gamma \vdash t : U}\text{(Conv)} \qquad \frac{\Gamma \vdash t : \texttt{Type}_i \quad i < j}{\Gamma \vdash t : \texttt{Type}_j}\text{(CumT)}$$

$$\frac{\Gamma \vdash t : \texttt{Set}}{\Gamma \vdash t^\ddagger : \texttt{Type}}\text{(CumS)} \qquad \frac{\Gamma \vdash t : \texttt{Prop}}{\Gamma \vdash t^\dagger : \texttt{Type}}\text{(CumP)}$$

The cumulativity of $\texttt{Type}_i$ towards $\texttt{Type}_j$, with no consequence for the extraction, remains implicit. It should also be noticed that a well-typed marked term cannot have two successive marks: $t^\dagger$ being of `Type` type, it is impossible to form $t^{\dagger\dagger}$.

Let us stress the fact that are ignored by typing rules when they appear on the right of a judgement. We only care about the influence of marks inside the term being typed. For example the following application is quite legal:

$$\frac{\Gamma \vdash t : \forall x : U^\dagger, T \quad \Gamma \vdash u : U}{\Gamma \vdash t\ u : T\{x \leftarrow u\}}\text{(App)}$$

It would be possible to make formal this aspect of typing, by adding the following four rules:

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash t : T^\dagger} \qquad \frac{\Gamma \vdash t : T^\dagger}{\Gamma \vdash t : T} \qquad \frac{\Gamma \vdash t : T}{\Gamma \vdash t : T^\ddagger} \qquad \frac{\Gamma \vdash t : T^\ddagger}{\Gamma \vdash t : T}$$

In practice, these rules will remain implicit throughout our study.

We now specify the status of these marks (for example †) with respect to substitution:

- A substitution without effect does not change anything concerning the marks, for example $y\{x \leftarrow t^\dagger\} = y$
- Of course, a substitution without mark does not create any: $x\{x \leftarrow t\} = t$
- A variable replaced by a marked term gives a marked term: $x\{x \leftarrow t^\dagger\} = t^\dagger$
- A marked variable gives a marked term after substitution: $x^\dagger\{x \leftarrow t\} = t^\dagger$
- Lastly, the case $x^\dagger\{x \leftarrow t^\dagger\}$ is prohibited by typing. Indeed, $x^\dagger$ is necessarily of type `Type`, and x has the type `Prop`. However the $t^\dagger$ that one wants to substitute for it is also in `Type`, so the substitution is impossible.

Concerning the reductions, they will influence the marks only via substitutions. A priori, one could need to specify what becomes a mark partially covering a redex, as in $(\lambda x : U,\ t)^\ddagger\ u$. But in fact, this case is excluded because a marked term is inevitably a type, which cannot begin with one λ-abstraction. In the same way, the head of a ι-redex cannot be marked because it has an inductive type. It is thus not a type.

We must now justify again a certain number of properties that our marking could have modified. Let us start with the property of "subject reduction", and before that by the conservation of typing by substitution.

Lemma 4 *Let t and u be two terms of* CIC_m *admitting for respective types T and U. If x is a free variable of t of type U, then $t\{x \leftarrow u\}$ admits $T\{x \leftarrow u\}$ for type in the* CIC_m.

PROOF. By induction on the typing derivation of t. □

Lemma 5 *Let t be a term of* CIC_m *admitting T as type, and u a reduced form of t. Then u also admits T as type.*

PROOF. It is the same proof as for CIC, but with the previous lemma instead of the initial subtitution lemma. □

Now, the principal effect of the marks presence is the unicity of types:

Lemma 6 *Let t be a term of* CIC_m *admitting T and U as types. Then $T =_{\beta\delta\iota\zeta} U$, as soon as you aggregate all the the different sorts* $\texttt{Type}_i$ *into only one, and you ignore the marks inside types T and U.*

PROOF. In CIC, the non-unicity of the types comes from the possibility to use the cumulativity part of rule (Conv) at any time. But with the presence of the marks, the rules (CumS) and (CumP) are now rules whose use is controlled by the term syntax, like the majority of other rules. And the rule (CumT) does not really modify the types, at least according to the point of view that we choose here.

Once this is said, one proceeds by induction on typing derivation $t : T$, and then compares this derivation with the one of $t : U$. Except for $\beta\delta\iota\zeta$-conversions (Conv) that we can ignore here, these two derivations have necessarily the same form. $\square$

In particular the concept of principal type becomes now a commonplace. Let us look again at the counterexample which shows that a principal type in CIC is not necessarily preserved during reduction: $((\lambda X : \texttt{Type}, X)\ \texttt{True})$. It is now badly typed in CIC_m. To make a valid term from it, a mark would have to be added: $((\lambda X : \texttt{Type}, X)\ \texttt{True}^\dagger)$. But then its reduced form $\texttt{True}^\dagger$ admits only type `Type`, and not `Prop` any longer.

To finish this presentation of CIC_m, we will clarify the possible translations between CIC and CIC_m. In the more simple direction, it is immediate to take a t term of type T in CIC_m and to obtain a corresponding well-typed term in CIC : it is enough to remove the marks. And T (without its marks) then remains a valid type for it in CIC. In the opposite direction, starting form a term t in CIC having for principal type T, one can obtain a corresponding well-typed term in CIC_m that differs from t only by the presence of marks (and possibly of η-expansion, cf. restriction on the cumulativity). Introducing these marks is easy: we just look at a typing derivation of $t : T$, and adapt all the uses of the rule (Conv) from top to bottom. This way, the type of the CIC_m obtained term is still T (or rather the version of T with marks).

Chapitre 2

The extraction of Coq terms

This chapter is devoted to the presentation of our extraction function $\mathcal{E}$ over CIC terms. For the moment, this function will return terms in an intermediate theoretical language named $\text{CIC}_\Box$, untyped, which will play here a role similar to the one of F_ω in the old extraction. The next chapter then deals with the translation into the final language, and in particular with the problem of typing the code extracted in this language.

In addition to the presentation of this function $\mathcal{E}$, this chapter also contains the correctness proof of $\mathcal{E}$, or rather the correctness *p*roofs, since this theoretical study is divided into two parts:

- a first rather syntactic study enables us to guarantee that any reduction of extracted terms will succeed, regardless of the strict or lazy evaluation strategy used. This part is a revised version of the results presented in [55], themselves inspired by [54].
- In a second time, we will establish in section 2.4 the correctness of these extracted terms with respect to the original `Coq` specifications, by using a semantic approach inspired by realizability.

Let us start first by detailing the limitations of the old function $\mathcal{E}$ of C. Paulin, which led us to build the current version of $\mathcal{E}$.

2.1 The difficulties in the removal of logical parts

A potential danger induced by removal of logical parts is that this removal can modify the evaluation order. Let us take for example a function of type $\forall$`x:A,(P x)`$\rightarrow$ `B`, with `A` and `B` being informative types, and `P` a logical property. Our function `f` thus awaits an informative argument `x` and a proof that this argument satisfies the pre-condition `P x`. If one also has a term `t` of type `A` and a proof `p` of type `(P t)`, one can then form the two well-typed `Coq` terms `(f t)` and `(f t p)`. On the `Coq` level, these two terms have appreciably different nature. For example, the evaluation of `(f t)` will probably be quickly blocked by the lack of the second argument `p`, whereas `(f t p)` is a total application which can normally be reduced towards a value of type `B` (when for example these terms `f`, `t` and `p` are closed).

Now let us examine the action of a extraction function $\mathcal{E}$ that completely removes the logical parts, like the old extraction does. The two preceding terms `(f t)` and `(f t p)` are then extracted into the same term `(`$\mathcal{E}$`(f)` $\mathcal{E}$`(t))`, because `p`, logical, disappears. Since the type of $\mathcal{E}$`(f)` is then of form $\mathcal{E}$`(A)`$\rightarrow$$\mathcal{E}$`(B)`, the extracted term `(`$\mathcal{E}$`(f)` $\mathcal{E}$`(t))`, being a total application, behaves rather like `(f t p)`, and thus can be reduced completely. Of course, this behavior is quite different from the one of `(f t)`.

This modification of the order of evaluation has, of course, an influence on the efficiency when executing extracted terms. But it can even be fatal to the good progress of this execution. The old extraction could in particular generate extracted terms whose evaluation stopped prematurely on an uncaught exception, or on similar other execution error. And conversely one could also meet terms whose evaluation did not finish.

The typical example of code that can raise an exception in case of naive extraction involves the constant `False_rec`. This `Coq` term of type `∀P:Set, False→P` is used to treat the contradictory sub-cases of a proof. For example, the `Coq` tactic `contradiction` generates proof terms that use this constant. When one defines a function `f` of the type `∀x:nat, (x≠0)→nat`, it can thus be useful to use this constant when `x=0`. During the extraction, this `False_rec` is translated into an exception, which means that the execution must never reach this absurd sub-case. Now, let us consider again a partial application, `(f 0)`, legal in `Coq`, that will normally never receive its second argument of type `0≠0`. The extraction `(`$\mathcal{E}$`(f) 0)` can then be executed without waiting for the removed logical argument, and thus raise the exception associated with `False_rec`. Our new extraction solves this problem by giving back to the extraction of `(`$\mathcal{E}$`(f) 0)` its status of closure. For that, we leave artificial abstractions `fun _ → ...` each time that it is necessary.

We will see thereafter in part 2.3.2 that it is also possible to generate other types of errors with the old execution, by combining extraction, partial application and some `Coq` constants like `eq_rec`. There also exists examples of extracted code that were not terminating with the old extraction. These examples are based on the constant `Acc_rec`, which offers the possibility in `Coq` of defining an informative fixpoint justified by a decreasing logical measure. It is then possible, under an contradictory context, to provide one false logical justification. The extracted fixpoint, without its logical justification, may then loop forever.

A family of limitations of the old extraction relates to the `Coq` universes. Indeed, distinction in `Coq` between the informative and logical parts is made fuzzy by the presence of the `Type` universe. One can, indeed, form hybrid terms like `if B then nat else True`, where `b` is a boolean. This term will be either informative or logical according to value of this boolean `b`. This construction is allowed by the existence of rules known as cumulativity rules, which express that `Type` contains at least `Set` and `Prop`. In our example, `nat:Set` thus imply also `nat:Type` and similarly `True:Prop` imply `True:Type`. And finally, our hybrid term is well-typed with the `Type` type. The preceding extraction simply refused to extract such a type, and more generally any term using directly or indirectly the `Type` sort. This drastic restriction allowed a complete elimination of logical parts (at least in a system prohibiting `False_rec`, `eq_rec` and `Acc_rec`). On the contrary, the goal of our work is to be able to treat any `Coq` term. We then have to use one ad-hoc constant (denoted $\square$) to mark the sites previously occupied by logical parts, like the `True` above. Our approach is thus similar to the pruning methods [12, 15].

2.2 The new extraction function $\mathcal{E}$

This function $\mathcal{E}$ of extraction will eliminate any subterm of sort `Prop`, because these subterms correspond to logical parts, as explained in the previous chapter. But in addition to that, we also eliminate the subterms corresponding to types, and more generally to type schemes. Why remove these type schemes? This choice is less natural than the one leading to the elimination of the `Prop` parts. In particular, there exists at least one Coq development whose principal result is the construction of a type representing a particular lattice [62]. The extraction of such a development will then produce only one arbitrary constant replacing this type. But this situation is exceptional. In usual developments, the results concern only datatypes, such as for example inductive types like `bool`, `nat` or `Z`. And in these cases, we will show that the type schemes correspond to dead code from the point of view of computation. Another justification of this choice of elimination is that unlike Coq, our target languages (Ocaml and Haskell) make a clear distinction between the level of types and the level of terms, and in particular do not allow the use of types like ordinary terms.

We define $\mathrm{CIC}_\square$ starting from the same language as CIC, with in addition one special constant $\square$. On the other hand the terms of $\mathrm{CIC}_\square$ will not be typed. Indeed, we will not adapt the typing relation of CIC to $\mathrm{CIC}_\square$. Lastly, the reductions in $\mathrm{CIC}_\square$ are defined as being exactly those of CIC, with $\square$ seen like a non-reducible constant.

Let us now define our extraction function of CIC towards $\mathrm{CIC}_\square$.

Definition 9 (function $\mathcal{E}$) *The extraction function $\mathcal{E}$ is defined by structural induction on any term t typable in a context Γ:*

($\square$) *If t is a type scheme or admits* `Prop` *as sort in the context Γ, then $\mathcal{E}_\Gamma(t) = \square$*

If not, one proceeds according to the structure of t:

(*id*) $\mathcal{E}_\Gamma(a) = a$ *if a is a variable x, a constant c or a constructor C.*

(*lam*) $\mathcal{E}_\Gamma(\lambda x : T,\, t) = \lambda x : \square,\, \mathcal{E}_{\Gamma'}(t)$ *where* $\Gamma' = \Gamma; (x : T)$

(*let*) $\mathcal{E}_\Gamma(\texttt{let}\ x := t\ \texttt{in}\ u) = \texttt{let}\ x := \mathcal{E}_\Gamma(t)\ \texttt{in}\ \mathcal{E}_{\Gamma'}(u)$ *where* $\Gamma' = \Gamma; (x := t : T)$ *and T is a type of t*

(*app*) $\mathcal{E}_\Gamma(u\ v) = (\mathcal{E}_\Gamma(u)\ \mathcal{E}_\Gamma(v))$

(*cases*) $\mathcal{E}_\Gamma(\texttt{case}(e,\, P,\, f_1\ \ldots\ f_n)) = \texttt{case}(\mathcal{E}_\Gamma(e),\, \square,\, \mathcal{E}_\Gamma(f_1)\ \ldots\ \mathcal{E}_\Gamma(f_n))$

(*fix*) $\mathcal{E}_\Gamma(\texttt{fix}\ f_i\ \{f_1/k_1 : A_1 := t_1\ \ldots\ f_n/k_n : A_n := t_n\}) =$
$\texttt{fix}\ f_i\ \{f_1/k_1 : \square := \mathcal{E}_{\Gamma'}(t_1)\ \ldots\ f_n/k_n : \square := \mathcal{E}_{\Gamma'}(t_n)\}$
where $\Gamma' = \Gamma; (f_1 : A_1); \ldots; (f_n : A_n)$

And the extraction of a context is defined by:

(*nil*) $\mathcal{E}(\varnothing) = \varnothing$

(*def*) $\mathcal{E}(\Gamma; (c := t : T)) = \mathcal{E}(\Gamma); (c := \mathcal{E}_\Gamma(t) : \square)$

(*ax*) $\mathcal{E}(\Gamma; (x : T)) = \mathcal{E}(\Gamma); (x : \square)$

(*ind*) $\mathcal{E}(\Gamma; \texttt{Ind}_n(\Gamma_I := \Gamma_C)) = \mathcal{E}(\Gamma); \texttt{Ind}_n(\mathcal{E}(\Gamma_I) := \mathcal{E}(\Gamma_C))$

Clearly, $\mathcal{E}$ is a "pruning" function: its only task is to replace certain subterms by $\Box$. In particular there is no modification of the structure. In the currently implemented Coq extraction, a second phase is dedicated to some structural modifications. This phase will be described in the chapter 4. This "pruning" is different from what the previous extractions were doing, since they were in particular not withdrawing the types, and were removing completely the logical λ-abstractions, via a rule like:

(lam') $\mathcal{E}(\lambda x : P, t) = \mathcal{E}(t)$ *if* P *admits* `Prop` *for type.*

In term of realizability, this last rule (lam') corresponds to modified realizability, whereas our new rule (lam) corresponds more to recursive realizability. But as we have explained in the previous section, the rule (lam') is not correct if one combines it with strict evaluation à la Ocaml.

The reader may also have noticed the lack of an explicit rule dedicated to products. But a product is always a type, and thus a fortiori a type scheme. The rule $(\Box)$ thus applies.

2.3 Syntactic study of the reduction of extracted terms

Initially, we study the reduction of extracted terms, and we prove in particular that this reduction necessarily terminates in a finite amount of time. This will be done by means of a syntactic method: we will simulate the derivations of the extracted terms by those of the Coq initial terms.

But this first approach, relatively simple, is not very suitable to establish more semantic properties of correctness, in particular when functional values and/or unclosed terms are concerned. The section 2.4 is then devoted to a complementary study, based on an extension of the concept of realizability.

The syntactic study which follows now is split in two parts. The section 2.3.1 leads to theorem 1 which establishes the strong normalization of the extracted terms, but only in a slightly weaker version of Cic. Then the section 2.3.2 treats the complete Cic, which on the other hand obliges to restricted oneself to weak reductions for the extracted terms. In this situation, the main result of correctness is the theorem 5.

2.3.1 Strong reduction in a restriction of $\text{Cic}_\Box$

We wish here to establish that the evaluation of an extracted term finishes, and that the result of this evaluation has a correct meaning, for example `true` or `false` for an original term in Cic of the `bool` type. And of course, we also wish this result to be coherent with answer what would give the evaluation of the original term in Cic.

We will thus proceed by simulation in Cic of derivations possible in $\text{Cic}_\Box$, and vice versa. The problem is that this simulation can lead to terms that still comprise Cic redex, whereas their analogues in $\text{Cic}_\Box$ are not reducible any more. In fact, there are three potential categories of Cic redex corresponding to non-redex zones of $\text{Cic}_\Box$:

1. a β-redex $(\lambda x : X, t)\ u$ corresponding to a non-redex $\Box\ u'$
2. a ι-redex $\texttt{case}(e, \ldots, \ldots)$ corresponding to a non-redex $\texttt{case}(\Box, \ldots, \ldots)$

3. a ι-redex ($\texttt{fix}\ f_i\ \{\ldots\}\ u_1 \ldots\ u_n$) corresponding to a non-redex, which can be:
   a. either ($\Box\ u'_1\ \ldots\ u'_n$)
   b. either ($\texttt{fix}\ f_i\ \{\ldots\}\ u_1\ \ldots\ \Box$) (the "guard" is now a $\Box$ blocking the reduction)

In case 1, we would like to have $(\lambda x : X,\ t)\ u$ directly corresponding to $\Box$ instead of $\Box\ u$. Indeed the stability lemma claims that the application preserves the fact of being of Prop sort or of being a type scheme. If $\lambda x : X,\ t$ could become $\Box$, then it should thus be "morally" the same with $(\lambda x : X,\ t)\ u$. This "lack of precision" of an extracted term can in fact appears after some steps of reduction, as shown by the following example:

Example 1

$$t = (\lambda X : \texttt{Type},\ \lambda f : \texttt{nat}{\to}X,\ \lambda g : X \to \texttt{nat},\ (g\ (f\ \texttt{0})))\ \ \texttt{Prop}\ \ (\lambda\_\,,\ \texttt{True})$$
$$\mathcal{E}(t) = (\lambda X : \Box,\ \lambda f : \Box,\ \lambda g : \Box,\ (g\ (f\ \texttt{0})))\ \ \Box\ \ \Box$$
$$\to^*_\beta\ \ \lambda g : \Box,\ (g\ (\Box\ \texttt{0}))$$

We will solve this problem of case 1 (and at the same time the case 3a) thanks to an ad-hoc reduction:

Definition 10 ($\Box$-reduction) *The $\Box$-reduction is defined by the rule* $(\Box\ u) \to_\Box \Box$

The situation of case 2 is rather different. Unlike the the previous example where a lambda became logical after a reduction, a pattern matching cannot change the inductive type on which it is performed. And $\mathcal{E}$ eliminates all pattern matchings that produce objects of sort Prop. Since a matching on an inductive type in Prop can normally only build an object in Prop, case 2 should normally not occur. But this restriction on logical matchings has two exceptions, concerning logical empty inductive types and logical singleton inductive types (see the previous chapter). For example Cic authorizes the following derivations:

$$\frac{p : \texttt{False} : \texttt{Prop} \quad T : \texttt{Set}}{\texttt{case}(p,\ T,\ \varnothing)\ : T} \qquad \frac{p : x = y : \texttt{Prop} \quad q : P\ x : \texttt{Set}}{\texttt{case}(p,\ P,\ q)\ : P\ y : \texttt{Set}}$$

The first derivation corresponds to the Coq constant named False_rec, while the second corresponds to eq_rec.

More generally, a logical elimination case can produce something informative if the elimination is carried out on a term whose inductive type which:

1. has zero constructor (empty inductive, like False in Coq)
2. has only one constructor whose arguments are all logical, put aside possible parameters (inductive singleton logical, like eq)

This is in fact a first exception to the slogan: "logical objects are never taken into consideration during computations of informative objects". The second distortion to this principle is the case 3b: the "guard" of a fixpoint can be a logical inductive term whereas the complete fixpoint is informative.

Let us suppose one moment that these characteristics of typing Coq are deactivated. Until the end of this section 2.3.1, we will consider two systems Cic$^-$ and Cic$^-_\Box$ which are respectively Cic and Cic$_\Box$ with the following restrictions:

(i) The elimination of logical empty inductive terms cannot produce informative terms.

(ii) It is the same for elimination the logical singleton inductive terms.

(iii) For any component f_i of a fixpoint, its "guard" cannot be logical unless the type of f_i is also logical.

We now compare the respective results of reductions in $\text{CIC}_\square^-$ and CIC^-. To simplify this comparison, we will speak here only about types without logical contents:

Definition 11 *A T type of* CIC *is said to have no logical content if for any closed normal form t of type T we have $\mathcal{E}(t) = t$.*

The usual datatypes, like `bool` or `nat` fulfill this condition. Of course, a object in a non-reduced form in such a type can contain logical parts, but they will disappear during reduction to finish on `true` or `(S (S O))` for example. We then have the following result concerning the strong reduction of terms extracted in these types without logical contents:

Theorem 1 *Let t be a closed term, well-typed in CIC^-, whose T type has no logical contents. Then any reduction of $\mathcal{E}(t)$ terminates on the normal form (in CIC^-) of t.*

We will not prove this result, because it has less importance that the theorem 5 of the following section, while the proofs of the two theorems are similar. Similarly, it is possible to establish a more general version of this result, dealing also of types which are not without logical contents. But as the normal form of $\mathcal{E}(t)$ can then contain $\square$, we need new tools to compare it with the normal form of t. There again, this is not done here, but rather in the following section.

Lastly, let us note that the use of $\square$-reductions is not necessary in this particular result. This is explained by the conjunction of the restrictions (i), (ii) and (iii) and the assumption that T is not-logic. On the other hand the results of the following section will have to use this $\square$-reduction.

2.3.2 Weak reduction in the complete $\text{CIC}_\square$

As our objective is a extraction mechanism accepting all the `Coq` terms, we must from now on remove these restrictions (i), (ii) and (iii). The restriction (i) concerning empty inductives is in fact easy to remove, since a ι-reduction on one empty inductive in fact cannot occur, for lack of constructor to start the reduction. We can thus just ignore these `case`, and translate them later into exceptions (see the study of `False_rec` in section 2.1). On the other hand deletion of the restrictions (ii) and (iii) will oblige us to adapt authorized reductions on the extracted terms: we must give up strong reduction (*i.e.* reduction under the lambdas) and restrain to weak reduction. In any event, our functional languages target do not authorize strong reduction. To center our study on weak reduction is thus perfectly legitimate.

Singleton elimination

If H is an equality (hence logical), `case`$(H,$ `nat`$,$ `O`$)$ can be reduced and give `O` even without knowing the exact value of H, hidden behind a $\square$. In a similar way, we can reduce systematically all elimination of logical singleton inductive. But that is dangerous when combined with strong reduction, and can lead to execution errors. Let us consider for example the following function `cast` that transforms an integer into a boolean on the condition of being able to prove that boolean and integers are identical[1]:

```
Definition cast : (nat=bool) → nat → bool :=
 fun (H:nat=bool)(n:nat) ⇒
  match H in (_=bool) return bool with
   | refl_equal ⇒ n
  end.
```

Let us take then the following example:

```
Definition example :=
 fun (H:nat=bool) ⇒
  let b : bool := cast H 0 in
  match b with
   | true ⇒ 0
   | false ⇒ 1
  end.
```

If now one carries out the a priori reduction of the `case` in this example, the subterm `(cast H 0)` would be reduced to integer 0, whereas the `match` which follows awaits a boolean, and that will result in an execution error.

A similar example can also lead the integer `0` to be considered as being a function if one has in assumption the equality `nat = (nat→nat)`. And if one applies this "function" `0`, we can end with an execution error `(0 0)`, in the event of strong reduction of the extracted terms

Clearly, if one prohibits the reduction under lambdas, these problems disappear. Indeed, an singleton inductive term out of the lambdas is inevitably closed, and can thus be always reduced to a constructor, which legitimates our reduction of logical singleton eliminations out of any lambda.

Fixpoint with logical "guards"

The problem is now to reduce informative fixpoint whose argument being used as "guard" is logical. Of course, the immediate temptation is to remove this "guard" condition, at least for this category of fixpoint. But this, combined with strong reduction, can lead to an evaluation that does not terminate. The following function `loop` is built on the model of `Acc_iter` (see the previous chapter). It expects an hypothetical proof of the false statement claiming the accessibility of 0 by the relation `gt` (that is $>$ on the natural numbers of `Coq`). If this proof were provided, `loop` would then go into an infinity of recursive calls: `F N` would call `F (S N)` and so on.

[1] We will use here `Coq` syntax, more readable. The equivalent in our theoretical syntax of this `match` with annotations is `case`$(H,\ (\lambda t : \texttt{Set},\ \lambda H : (\texttt{eq Set nat}\ t),\ t),\ n)$

```
Definition loop :=
 fun (Ax:Acc gt 0) ⇒
  (fix F (n:nat)(a:Acc gt n) {struct a} : nat :=
    F (S n) (Acc_inv a (S n) (gt_Sn_n n)))
  0 Ax.
```

The `(Acc_inv a...)` subterm is a proof of accessibility for `(S N)`, using the constants `Acc_inv` and `gt_Sn_n` provided by the standard library of Coq. The extraction $\mathcal{E}$ gives then:

$$\begin{aligned}\mathcal{E}(\texttt{loop}) = \ & \lambda Ax : \Box, \\ & \quad \texttt{fix}\ F\ \{F/2 : \Box := \\ & \qquad \lambda n : \texttt{nat},\ \lambda a : \Box,\ (F\ (\texttt{S}\ n)\ \Box)\} \\ & \quad \texttt{0}\ \Box\end{aligned}$$

And if one withdraws here the "guard" condition, then this term can be strongly reduced even without being applied, and then give $\lambda Ax : \Box$, $\texttt{fix}\ F\ \{\ldots\}\ (\texttt{S}\ \texttt{0})\ \Box$, and so on...

Modification of reduction

To manage these exceptional cases of ι-reduction on logical terms, we must first of all add an additional annotation on the CIC terms. The `match` bearing on inductive types with only one constructor are normally written this way in Coq :

`match` e `with C` $\overrightarrow{x}$ `⇒ t end`

With the "functional syntax" used in this theoretical study, this becomes:

$\texttt{case}(e, \ldots, (\lambda \overrightarrow{x}, t))$

The latter form presents the disadvantage of losing track of the number of the arguments $\overrightarrow{x}$ for the single constructor `C` of our inductive type. In particular, with this syntax, it is not correct to count the number of lambdas, because `t` can contain additional ones. We will cure this problem by marking this number of arguments in index for these `case` with single branch:

$\texttt{case}_n(e, \ldots, (\lambda \overrightarrow{x}, t))$

And of course, the extraction function $\mathcal{E}$ will keep these annotations. To avoid obscuring (more) the notations, we will omit sometimes these annotations in situations where they are not used.

Here now come the modifications to be made to the reductions $\text{CIC}_\Box$ in order to be able to manage these $\Box$ that may block ι-reductions.

Definition 12 (new ι-reduction) *The ι-reduction on $\text{CIC}_\Box$ terms is from now on:*

($iota$) $\texttt{case}(C_i\ \overrightarrow{p}\ \overrightarrow{u}, P, f_1\ \ldots\ f_m) \rightarrow_\iota f_i\ \overrightarrow{u}$

($iota$) $\texttt{case}_n(\Box, P, f) \rightarrow_\iota f\ \underbrace{\Box\ \ldots\ \Box}_{n}$

($iota$) *Let F be the recursive block $f_1/k_1 : A_1 := t_1\ \ldots\ f_n/k_n : A_n := t_n$. Then:*
$(\texttt{fix}\ f_i\ \{F\}\ u_1\ \ldots\ u_{k_i}) \rightarrow_\iota (t_i\{f_j/\texttt{fix}\ f_j\ \{F\}\}_{\forall j}\ u_1\ \ldots\ u_{k_i})$

when u_{k_i} is $\Box$ or starts with a constructor.

We also restrict the reductions and prohibit strong reduction: for each possible reduction, one associates to it a weak reduction.

Definition 13 (weak reductions) *The reductions $\to_{\beta_w}$, $\to_{\iota_w}$, $\to_{\delta_w}$, $\to_{\zeta_w}$ and $\to_{\Box_w}$ are defined from the same basic rules as respectively $\to_\beta$, $\to_\iota$, $\to_\delta$, $\to_\zeta$, $\to_\Box$, but with restraining the compatibility rules to only the following ones:*

$$\frac{u \to_? v}{(u\ t) \to_? (v\ t)} \qquad \frac{u \to_? v}{(t\ u) \to_? (t\ v)}$$

$$\frac{u \to_? v}{\mathtt{case}(u,\ P,\ \ldots) \to_? \mathtt{case}(v,\ P,\ \ldots)}$$

Lastly, as for $\to_r$, the complete weak reduction $\to_{r_w}$ is $\to_{\beta_w} \cup \to_{\iota_w} \cup \to_{\delta_w} \cup \to_{\zeta_w}$.

In fact, this reduction $\to_{r_w}$ can be seen as a common generalization of the strategies of calls per value and call by name of Ocaml and Haskell. The last step towards the reductions actually implemented in these languages is to fix an evaluation order. To reduce the arguments initially will give us strict strategy of Ocaml. And on the contrary reducing first of all the head of the term corresponds to the lazy strategy of Haskell.

A significant point to mention is that all this theoretical study is to be done in axiom-free contexts. Indeed, to study reduction in the presence of axioms is equivalent to study the strong reduction under the lambdas corresponding to these axioms, which we wish to precisely avoid. In particular, we will use on several occasions the fundamental property according to which an inductive term closed in a axiom-free context reduces inevitably towards a term starting with a constructor. Clearly, the presence of an axiom can invalidate this property. Of course, all axioms do not have this effect, but for reasons of simplicity, we prohibit them all.

To study the evaluation of the extracted terms, we will compare it with the evaluation of the initial terms. We hence need an invariant relating initial terms and extracted terms that will be stable by reduction. Of course, one can try to directly use the function $\mathcal{E}$ for this invariant. Unfortunately, that is not a good choice, because $\mathcal{E}$ behaves badly with respect to reduction: if $t \to_r u$, one may not have $\mathcal{E}(t) \to_r \mathcal{E}(u)$ in some cases[2]. Instead of $\mathcal{E}$, we define and use a non-deterministic relation $\to_\mathcal{E}$ which will have good invariance properties.

Definition 14 (relation $\to_\mathcal{E}$) *The non-deterministic relation $\to_\mathcal{E}$ relating a* Cic *term and a* Cic$_\Box$ *term, and depending implicitly on one context Γ, is defined by the following rules:*

[2]The term t in example 1 is a counterexample to that. Indeed, $\mathcal{E}(t)$ can be reduced into one term containing $(\Box\ \mathtt{0})$, whereas $\mathcal{E}$ never produces such subterm, but directly $\Box$.

$$\frac{\Gamma \vdash t : \forall \overrightarrow{x : X},\, s}{t \to_{\mathcal{E}} \Box}\ (\mathcal{E}\text{–}\Box_1) \qquad \frac{\Gamma \vdash t : T : \mathtt{Prop}}{t \to_{\mathcal{E}} \Box}\ (\mathcal{E}\text{–}\Box_2)$$

$$\frac{a = x \text{ or } a = c \text{ or } a = C}{a \to_{\mathcal{E}} a}\ (\mathcal{E}\text{–}id) \qquad \frac{t \to_{\mathcal{E}} t'}{\lambda x : T,\, t \to_{\mathcal{E}} \lambda x : \Box,\, t'}\ (\mathcal{E}\text{–}lam)$$

$$\frac{t \to_{\mathcal{E}} t' \quad u \to_{\mathcal{E}} u'}{\mathtt{let}\ x := t\ \mathtt{in}\ u \to_{\mathcal{E}} \mathtt{let}\ x := t'\ \mathtt{in}\ u'}\ (\mathcal{E}\text{–}let) \qquad \frac{t \to_{\mathcal{E}} t' \quad u \to_{\mathcal{E}} u'}{(t\ u) \to_{\mathcal{E}} (t'\ u')}\ (\mathcal{E}\text{–}app)$$

$$\frac{e \to_{\mathcal{E}} e' \quad \forall i,\ f_i \to_{\mathcal{E}} f_i' \quad \mathit{Info}(e)}{\mathtt{case}(e,\, P,\, f_1\ \ldots\ f_n) \to_{\mathcal{E}} \mathtt{case}(e',\, \Box,\, f_1'\ \ldots\ f_n')}\ (\mathcal{E}\text{–}case)$$

$$\frac{\forall i,\ t_i \to_{\mathcal{E}} t_i'}{\mathtt{fix}\ f_i\, \{f_1/k_1 : A_1 := t_1 \ldots f_n/k_n : A_n := t_n\} \to_{\mathcal{E}} \mathtt{fix}\ f_i\, \{f_1/k_1 : \Box := t_1' \ldots f_n/k_n : \Box := t_n'\}}\ (\mathcal{E}\text{–}fix)$$

The condition $\mathit{Info}(e)$ requires e to be of an informative inductive type (or empty logical or singleton logical). One naturally extends $\to_{\mathcal{E}}$ to extract the contexts.

The non-determinism comes from the two rules $(\mathcal{E}\text{–}\Box_1)$ and $(\mathcal{E}\text{–}\Box_2)$. When these rules can be applied, one can indeed use them, or instead use one of the other structural rules. For example, if a variable x has sort `Prop`, one have both $x \to_{\mathcal{E}} \Box$ and $x \to_{\mathcal{E}} x$. Obviously, the function $\mathcal{E}$ is just a way to make this relation deterministic, by always choosing to prune as soon as possible:

Lemma 7 *If t is a* CIC *term typable in a context Γ, then $T \to_{\mathcal{E}} \mathcal{E}(t)$.*

PROOF. One only have to choose the rules extracting towards $\Box$ as soon as a type scheme or a logical term is reached. One must only check the auxiliary condition *Info(e)* of the rule $(\mathcal{E}\text{–}case)$: If a complete term `case` is not of sort `Prop`, then the matched e term is necessarily of an informative inductive type (or logical empty or logical singleton). The condition *Info(e)* is thus fulfilled. $\Box$

Lemma 8 *Let t be a* CIC *term typable in a context Γ, and t' a* CIC$_\Box$ *term such as $t \to_{\mathcal{E}} t'$. We have the following immediate properties:*

1. *t and t' differ only at positions where t' contains $\Box$.*
2. *any subterm of 't corresponding to a $\Box$ in t is of sort* `Prop` *or is a type scheme.*
3. *all the* `case` *remaining in t' relate to inductive types that are informative or logical singleton or logical empty types.*

This relation $\to_{\mathcal{E}}$ has the advantage of being stable by substitution, contrary to the function $\mathcal{E}$:

Lemma 9 *Let t, u, T, U be four terms of* CIC *and Γ a context such that:*

$$\begin{cases} \Gamma; (x : U) \vdash t : T \\ \Gamma \vdash u : U \end{cases}$$

Let also t', u' be CIC$_\Box$ *terms such that $t \to_{\mathcal{E}} t'$ and $u \to_{\mathcal{E}} u'$. One then has:*

$$t\{x \leftarrow u\} \to_{\mathcal{E}} t'\{x \leftarrow u'\}$$

PROOF. By induction on the derivation of $t \rightarrow_{\mathcal{E}} t'$, and by case analysis according to the last rule used in this derivation:

- ($\mathcal{E}$–$\Box_1$) or ($\mathcal{E}$–$\Box_2$): if t is a type scheme or has sort `Prop`, then it is the same for $t\{x \leftarrow u\}$ according to the stability lemma 1. We then have indeed $t\{x \leftarrow u\} \rightarrow_{\mathcal{E}} \Box$ by the same rule.
- ($\mathcal{E}$–id): the case where t is not the variable x is obvious, since there is then nothing to substitute. On the opposite, if $t = x = t'$, then $x\{x \leftarrow u\} = u \rightarrow_{\mathcal{E}} u' = x\{x \leftarrow u'\}$.
- ($\mathcal{E}$–lam): we have $t = \lambda y : Y,\, t_0$ and $t' = \lambda y : \Box,\, t'_0$ with $t_0 \rightarrow_{\mathcal{E}} t'_0$. The induction hypothesis gives us $t_0\{x \leftarrow u\} \rightarrow_{\mathcal{E}} t'_0\{x \leftarrow u'\}$. However $(\lambda y : Y,\, t_0)\{x \leftarrow u\} = \lambda y : Y\{x \leftarrow u\},\, t_0\{x \leftarrow u\}$ and $(\lambda y : \Box,\, t'_0)\{x \leftarrow u'\} = \lambda y : \Box,\, t'_0\{x \leftarrow u'\}$. These two last terms are indeed related by $\rightarrow_{\mathcal{E}}$, thanks to the rule ($\mathcal{E}$–$lam$).
- ($\mathcal{E}$–$case$): as for the rule ($\mathcal{E}$–lam), one obtains as induction hypothesis the good behavior of each sub-expression of the `case` term with respect to substitution. Before using rule ($\mathcal{E}$–$case$) on the whole substituted `case`, it is just necessary to make sure that the condition ***`Info`*** (e) remains true after substitution of the term e matched in the `case`. However this is obvious, because an inductive term does not change its inductive type by substitution.
- the remaining structural rules are treated like ($\mathcal{E}$–lam).

$\Box$

The following theorem expresses that one can simulate on the `Coq` level all weak reduction of an extracted term.

Theorem 2 *Let t be a* CIC *closed well-typed term and t', u' two* CIC$_\Box$ *terms such that $t \rightarrow_{\mathcal{E}} t'$ and $t' \rightarrow_{r_w} u'$. There exists then a* CIC *term u such as $u \rightarrow_{\mathcal{E}} u'$ and $t \rightarrow_{r_w+} u$.*

$$\begin{tikzcd} t \arrow[r, dashed, "r_w+"] \arrow[d, "\rightarrow_{\mathcal{E}}"'] & u \arrow[d, dashed, "\rightarrow_{\mathcal{E}}"] \\ t' \arrow[r, "r_w"'] & u' \end{tikzcd}$$

PROOF. One proceeds by case analysis according to the reduction employed between t' and u'. We will start with the two difficult cases:

- the reduction carried out is a ι_w singleton reduction like

  $$\texttt{case}_n(\Box,\, \ldots,\, f') \rightarrow_\iota (f'\ \Box\ \ldots\ \Box).$$

  The compatibility rules for the ι_w-reduction implies that this reduction occurs out of any binder. Moreover axioms have been prohibited, so the a subterm of t which corresponds to the eliminated $\Box$ is thus typable in a context without assumption. As a is an inductive term, it can then be reduced to a term having a constructor at the head: $(C\ \overrightarrow{p}\ \overrightarrow{v})$.

It is even possible to carry out this reduction to a constructor in a weak way. In fact, C is the single constructor of this singleton logical inductive type, and C have exactly n arguments apart from the parameters: $|\overrightarrow{v}| = n$. Thus in t the subterm $\texttt{case}_n(a, \ldots, f)$ can be reduced to $(f\ v_1\ \ldots\ v_n)$ via at least one step of r_w-reduction. So we can take for u the term resulting from t by these reductions. To check that $u \to_{\mathcal{E}} u'$, we need only check that $(f\ v_1\ \ldots\ v_n) \to_{\mathcal{E}} (f'\ \Box\ \ldots\ \Box)$. And that is obvious, because all the v_i have sort `Prop` since they are arguments of the constructor of a logical singleton inductive type.

- the reduction carried out is a ι_w-reduction of a fixpoint whose "guard" argument in t' is $\Box$. Then the "guard" argument g corresponding to this $\Box$ in t has an inductive type. Moreover, as in the previous case, g is typable in one context without assumption, and can thus be reduced to a term h beginning with a constructor. One can then reduce the fixpoint in t. And finally the terms obtained this way in CIC and in $\text{CIC}_\Box$ are still related by $\to_{\mathcal{E}}$.

The other cases are much easier. Let us consider for example the case where the reduction carried out is a β_w-reduction. According to the definition of the extraction relation, the β-redex in t' necessarily have a counterpart β-redex in t. We then just have to reduce this redex t in order to obtain a suitable u. And one has indeed $u \to_{\mathcal{E}} u'$, using the previous substitution lemma for $\to_{\mathcal{E}}$. Finally, all remaining cases ($\to_{\delta_w}$, $\to_{\zeta_w}$ and the end of $\to_{\iota_w}$) are similar to this case $\to_{\beta_w}$. $\Box$

Corollaire 1 *Let t be a closed well-typed* CIC *term and t' a* $\text{CIC}_\Box$ *term such that $t \to_{\mathcal{E}} t'$. Then any sequence of derivations $\to_{r_w}$ starting from t' is finite.*

PROOF. Thanks to repeated applications of the previous theorem, one can in fact build a corresponding succession of derivations in CIC starting from t, with at least as many steps. However the strong normalization of CIC implies that this last sequence is finite. $\Box$

This termination is obviously a good property, but is not enough to ensure that the weak reduction of an extracted term proceeds without problems. Indeed a reduction with a with a premature, abnormal end is no more desirable that a reduction with no end. Could we finish on a normal term which is not a value, such as `(O true)` or `match (fun x ⇒ x) with ...`? For these two extreme examples, it is clear that the answer is no: otherwise the previous theorem would show that these two terms could be related by $\to_{\mathcal{E}}$ with well-typed terms of CIC, which is here impossible.

On the other hand a perfectly possible normal form is the application $(\Box\ \texttt{0})$. For example, we may start with a predicate $P : \forall n : \texttt{nat}, \texttt{True}$. We then have $(P\ \texttt{0}) \to_{\mathcal{E}} (\Box\ \texttt{0})$, the former being normal with respect to r_w. Of course, we also have $(P\ \texttt{0}) \to_{\mathcal{E}} \Box$, and this is the choice made by the function $\mathcal{E}$. But we have already seen with the example 1 that this $(\Box\ \texttt{0})$ can appear as subterm during a reduction. And with a strategy à la Ocaml which requires to first evaluate arguments, one then finds oneself to seek a value for $(\Box\ \texttt{0})$. At the theoretical level, the answer is the ad-hoc reduction $\to_\Box$ already evoked. We will see in sections 2.6.3 and 3.3.2 how to deal with this ad-hoc reduction in practice.

Lemma 10 *In* $\text{CIC}_\Box$*, any sequence of reductions $\to_{\Box_w}$ is finite.*

PROOF. A reduction $\rightarrow_{\Box_w}$ strictly decreases the size of the term. $\Box$

Lemma 11 *Let t be well-typed* CIC *term and t', t'' be* CIC$_\Box$ *terms such that $t \rightarrow_{\mathcal{E}} t'$ and $t' \rightarrow_{\Box_w} t''$. Then $t \rightarrow_{\mathcal{E}} t''$.*

PROOF. It is enough to consider the redex $(\Box\ v')$ of t' that one reduces to get t''. This redex corresponds to a $(u\ v)$ subterm of t. One knows that u is either a type scheme or has sort `Prop`. (cf lemma 8). However this fact being stable by application, it is thus the same for the subterm $(v\ u)$. We can then conclude by applying $(\mathcal{E}\text{–}\Box_1)$ or $(\mathcal{E}\text{–}\Box_2)$ a level earlier in t. $\Box$

Theorem 3 *Let t be a closed well-typed* CIC *term and t' such that $t \rightarrow_{\mathcal{E}} t'$. then any sequence of derivations $\rightarrow_{(r_w|\Box_w)}$ starting from t' is finite.*

PROOF. Let us call $t'_0 \dots t'_n \dots$ this sequence in CIC$_\Box$. One can then build a sequence t_n in CIC verifying at each c that $t_n \rightarrow_{\mathcal{E}} t'_n$:

- if $t'_n \rightarrow_{\Box_w} t'_{n+1}$, we take $t_{n+1} = t_n$ and the invariant corresponds to the previous lemma.
- if $t'_n \rightarrow_{r_w} t'_{n+1}$, we use the theorem 2, and obtain t_{n+1} such that $t_n \rightarrow_{r_w} t_{n+1}$.

First of all, in this CIC sequence, there can be only be a finite number of successive steps equality, because they correspond to successive reductions $\rightarrow_{\Box_w}$ on the CIC$_\Box$ level. This CIC sequence thus consists of reduction steps possibly mixed with a finite number of step of equality at each time. However, because of the strong normalization, there must be only a finite number of such reduction step. The sequence in CIC is thus finite, just as the initial sequence. $\Box$

The integration of this reduction $\rightarrow_{\Box_w}$ does not affect termination during the evaluation of an extracted term. One can now tackle the question of the shape of normal forms with respect to the reduction $\rightarrow_{(r_w|\Box_w)}$. To answer this question, we will need a result dual to the theorem 2:

Theorem 4 *Let t, u be* CIC *well-typed terms (not necessarily closed) and t' a* CIC$_\Box$ *term such that $t \rightarrow_{\mathcal{E}} t'$ and $t \rightarrow_{r_w} u$. Then it exists a* CIC$_\Box$ *term u' such that $u \rightarrow_{\mathcal{E}} u'$ and verifying $t' \rightarrow_{r_w} u'$ or $t' \rightarrow^*_{\Box_w} u'$.*

$$\begin{tikzcd}
t \arrow[r, "r_w"] \arrow[d, "\rightarrow_{\mathcal{E}}"'] & u \arrow[d, dashed, "\rightarrow_{\mathcal{E}}"] \\
t' \arrow[r, dashed, "r_w|\Box_w*"'] & u'
\end{tikzcd}$$

PROOF.

- If the redex r reduced in t corresponds to a similar redex in t' which is complete, then we just have to reduce this redex of t' to obtain a u' which is appropriate.

- If r is completely contained in a subterm of t corresponding to a $\Box$ of t', it is enough to take $u' = t'$.
- We now will consider the intermediate cases where r corresponds to a redex of t' that is incomplete because partly hidden by a $\Box$. These situations correspond to cases 1, 2, 3a and 3b of section 2.3.1.
  - If r is a β-redex, the only situation to be considered is $r = (\lambda x : X,\, a)\ b$ in t corresponding to $(\Box\ b')$ in t'. One can then simulate the β-reduction of t by one $\Box$-reduction in t'.
  - If r is a δ- or ζ-redex, there is no problematic situation to consider.
  - If r is a ι-redex for a `case`, the only remaining case is

    $\texttt{case}(e,\ P,\ \ldots)$

    in t corresponding in t' to

    $\texttt{case}(\Box,\ P',\ \ldots).$

    The properties of $\rightarrow_{\mathcal{E}}$ (lemma 8) ensure that e is either of sort `Prop` or is a type scheme. As e is an inductive term, which thus cannot be a type scheme, e is necessarily of sort `Prop`. In addition, the condition ***Info*** states that this `case` in t concerns one inductive type that is either informative, logical empty, or logical singleton. Taking into account the sort of e, the informative case is impossible. The empty inductive case is also impossible, since the lack of constructor prevents any reduction to occur. So we are in presence of a logical singleton elimination, which we can now to reduce at the $\text{C}\textsc{ic}_\Box$ level thanks to the new ι-reduction.
  - If r is a ι-redex for a `fix`, there are two sub-cases. If the `fix` disappears in t' but not all the arguments composing the initial redex (case 3a), then one can simulate the ι-reduction of t by some $\Box$-reductions in t'. And if the `fix` is present in t', it means that the "guard" argument has become $\Box$ (case 3b). We can then reduce it thanks to the news ι-reduction for `fix`.

$\Box$

Theorem 5 *Let t be a closed well-typed* Cic *term and t' a* $\text{C}\textsc{ic}_\Box$ *term such that $t \rightarrow_{\mathcal{E}} t'$. Then any normal form t'_0 of t' modulo $\rightarrow_{(r_w|\Box_w)}$ corresponds via $\rightarrow_{\mathcal{E}}$ to a weak normal form t_0 of t. More precisely, we are in one of the four following cases:*

(i) $t'_0 = \Box$

(ii) $t'_0 = C\ \overrightarrow{v}$ *where the arguments $\overrightarrow{v}$ are also in normal form modulo $\rightarrow_{(r_w|\Box_w)}$.*

(iii) t'_0 *starts with an λ-abstraction.*

(iv) $t'_0 = \texttt{fix}\ f_i\ \{\ldots\}\ \overrightarrow{v}$. *If the guard argument is k_i-th, then $|\overrightarrow{v}| < k_i$ and $\overrightarrow{v}$ is also in normal form modulo $\rightarrow_{(r_w|\Box_w)}$.*

Proof. As in the proof of theorem 3, we build a sequence of derivations starting from t and corresponding on the Cic level to the derivation sequence leading from t' to t'_0. This gives us a Cic term t_1 such that $t \rightarrow^*_{r_w} t_1$. If we now continue to apply weak reductions to

t_1, we end up with a Cic term t_0 in weak normal form. One can then reflect this derivation $t_1 \rightarrow^*_{r_w} t_0$ on the Cic$_\Box$ level via the theorem 4, what gives us a sequence of derivations $\rightarrow_{(r_w\Box_w)*}$ starting from t'_0. But this t'_0 is in normal form with respect to these reductions, therefore does not change. Finally, we have the following diagram:

$$\begin{tikzcd}
t \arrow[r, "r_w*"] \arrow[d, "\rightarrow_\mathcal{E}"'] & t_1 \arrow[r, "r_w*"] \arrow[d, "\rightarrow_\mathcal{E}"] & t_0 \arrow[d, "\rightarrow_\mathcal{E}"] \\
t' \arrow[r, "(r_w\Box_w)*"'] & t'_0 \arrow[r, "="'] & t'_0
\end{tikzcd}$$

The remainder of the statement comes directly from the study of the possible shape of a closed weak normal form in Cic (cf lemma 3). Indeed t_0 can be:

- a sort, an applied inductive type or a product, which can only give $\Box$ according to the rules of $\rightarrow_\mathcal{E}$.
- a applied inductive constructor, that gives by $\rightarrow_\mathcal{E}$ either $\Box$ or the same constructor applied to extracted arguments.
- a λ-abstraction, which gives by $\rightarrow_\mathcal{E}$ either $\Box$ or another λ-abstraction.
- a fixpoint lacking some arguments, which gives by $\rightarrow_\mathcal{E}$ either $\Box$ or a corresponding fixpoint.

$\Box$

Even if it mentions only the general reduction $\rightarrow_{r_w}$, this theorem is also interesting from the point of view of the particular evaluation strategies, strict (à la Ocaml) or lazy (à la Haskell). Indeed, it was already mentioned previously that these two strategies can be seen like restrictions of $\rightarrow_{r_w}$, with the additional condition that the right part (resp. left) of an application should be in normal form before being able to reduce the other side. In all the cases, the returned term at the end of a strict evaluation is clearly normal with respect to $\rightarrow_{r_w}$, and in the same way for a lazy evaluation. To be able to use the previous theorem, it is just needed that these strict or lazy evaluations also integrate the reduction $\rightarrow_{\Box_w}$. In section 2.6, we will see how to reconcile our requirements in term of reduction rules with the reduction mechanism implemented in Ocaml and Haskell.

It is also possible to give some more precise details concerning the four case of the previous theorem. If $t'_0 = \Box$ then t_0 is a type scheme or has a `Prop` sort. The lemma 2 shows that it in is then also the same for t. Said differently (by contraposition): if t is informative and is not a type scheme, then the reduction of its extraction cannot finish on $\Box$.

Reciprocally, we would like to have that $t'_0 = \Box$ as soon as t is logical or is a type scheme. That is inaccurate, because of the non-determinism of $\rightarrow_\mathcal{E}$, as shown by the example $\lambda x : \texttt{True},\, x \rightarrow_\mathcal{E} \lambda x : \Box, \Box$. But of course, we only have to specialize the previous theorem with $t' = \mathcal{E}(t)$ instead of $T \rightarrow_\mathcal{E} t'$ to ensure this property. Anyway, $\rightarrow_\mathcal{E}$ has no interest except as intermediate invariant.

Concerning the other cases of the result:

- For (ii): if t'_0 starts with a constructor, then t has an inductive type if the constructor is completely applied, and otherwise a product type if not.
- Reciprocally: if $t' = \mathcal{E}(t)$, and if t has an informative inductive type, then the reduction will necessarily ends in in case (ii).
- If we are in case (iii) or (iv), this implies that t had initially a product type.
- On the other hand if we just know that t has a product type, we can end in any case (i) (ii) (iii) or (iv).

The main result of this study is that the reduction of extracted terms can be done without problems. If the reduction mechanism must carry out an application, it well will find at the head a value accepting one argument: closure, fixpoint or $\square$ (or an inductive constructor as long as one keeps their curryfied notation). And if the reduction mechanism must carry out a pattern matching, the matched object will be indeed reducible towards one constructor totally applied to arguments, except in the special case of logical singleton matchings.

The question is now to know if the final result of the reduction of an extracted term is indeed correct with respect to the initial term. Obviously, the work makes up to now will allow to give a first answer, at least in simple cases, such as for example the particular case of terms belonging to a datatype:

Definition 15 *a datatype is an inductive type D whose constructors have as arguments only objects of type D or of type another datatype.*

For example, an inductive type I with a constructor of type $(\texttt{nat} \to I) \to I$ (*i.e.* encapsulating a function) is not a datatype. On the other hand usual types like `bool`, `nat` or `Z` are datatypes in this sense. One can then specialize our previous result for this particular case:

Theorem 6 *Let t be a closed well-typed* Cic *term whose type T is a datatype without logical contents. Then all derivations of $\mathcal{E}(t)$ via $\to^*_{r_w \square_w}$ ends on the normal form* Cic *of t.*

Proof. We just have to consider again the proof of the previous theorem, except that now t_0 is in normal form and not only in weak normal form. This comes from the fact that T is a datatype: a closed weak normal form in this type, not being able to contain closures, is thus also in normal form. And consequently, the definition of being without logical contents shows that $\mathcal{E}(t_0) = t_0$. There is thus nothing to extract in t_0, and we have $t'_0 = t_0$. $\square$

For example, if we build in Coq an arbitrarily complex term answering `true` or `false` to a particular question, one is sure that its extraction will reduce to the same answer as in Coq. The same applies if our term returns the hundredth Fibonacci number or the list of the first thousand π digits.

This result, although interesting, is in fact strongly limited. In particular, it says nothing about the correctness of non-closed terms or functions (which is in fact the same, modulo λ-abstractions). To treat these cases, it is first of all necessary to specify what one means by correct functions, and then establish this correctness. The simple idea is to show that if all the arguments are correct in a certain way, then so is output computed by the function. The formalization of this idea and its proof turned out to be much more difficult than initially planned. This is the object of all the following section.

2.4 Semantic study of extraction correctness

In the following study, we focus on clarifying what it mean for an extracted term to be correct, and we do it by means of semantic considerations. Up to now we indeed used a purely syntactic correctness property, namely the comparison between the structures of an extracted term and of the initial term. But such an approach does not allow to treat functions in a satisfactory way.

This study is intended in the long term to allow the generation in Coq of the correctness proof of extracted objects. This aim influences largely this study, first in the choice of the logical framework, then in the definitions and the proofs that follow, made in the most detailed and mechanical possible manner.

We first of all will define a transformation $[\![.]\!]$, that will give us in particular, once applied to a type T, the correctness predicate that must verified by any extracted term $\mathcal{E}(t)$ from an object t of type T. Then we will establish successively:

- the preservation of this transformation $[\![.]\!]$ by substitution;
- the preservation of $[\![.]\!]$ via reduction;
- the fact that the objects built by $[\![.]\!]$ are correctly typed;
- the fact that the extracted terms verify indeed the correctness predicates given by $[\![.]\!]$.

Warning: to try to simplify this study, we do not treat the cases of "let-in" or constants or fixpoint with more than one component. Anyway, these cases are not the critical ones, and taking them into account would be a tedious but a priori straightforward job. On the other hand the principal change compared with the previous section is the use of the modified system $\mathrm{C{\scriptsize IC}}_m$ with explicit marking of cumulativities, introduced in section 1.2.5.

2.4.1 The logical framework

We use here a point of view as close as possible to a real formalization in Coq of the extraction correctness. Even if this formalization has remained a paper one during this thesis, it is possible that we try in the future to make a true development Coq of it. The logical system in which we will express our correctness properties is here the $\mathrm{C{\scriptsize IC}}_m$.

We already evoked in page 9 of introduction the fact that it is not not possible in general to be able to type-check the extracted terms in $\mathrm{C{\scriptsize IC}}_m$. Amongst other reasons, we can mention here the possible disappearance of logical decreasing certificates in fixpoints. It will thus be necessary to embed these extracted terms into one datatype, like:

```
Inductive expr : Set :=
 | Var : identifier → expr
 | Lam : identifier → expr → expr
 | App : expr → expr → expr
 | ...
```

We suppose from now on that we have in $\mathrm{C{\scriptsize IC}}_m$ such a concrete type, that we will name Λ thereafter, internalizing the syntax of untyped $\mathrm{C{\scriptsize IC}}_m$ terms (or pre-terms), plus one constant

$\Box$. Generally, we will note t_Λ the object t internalized in Λ. With a slight abuse of notation, we will continue to use the CIC_m syntax for these internalized objects. Thus we will write $(\mathtt{S}_\Lambda\ \mathtt{O}_\Lambda)$ rather than $(\mathtt{App}\ \mathtt{S}_\Lambda\ \mathtt{O}_\Lambda)$.

Just as two convertible objects of CIC_m have the same properties, and in particular are equal via the standard equality `eq`, we require two convertible Λ objects to be equal. In particular, we will use the fact that `((λx:X, t) x)` and `t` are the same object. We do not specify more precisely, for the moment, which notion of convertibility is necessary on the Λ level, leaving that to be clarified later on.

This logical framework being fixed, the function $\mathcal{E}$ extraction seen at the beginning of this chapter is now a meta-level function, transforming any CIC_m term into an object in the concrete type Λ.

2.4.2 The simulation predicates

In similar works on extraction correctness, the usual method is to define the correctness of the extracted terms with respect to the initial type of extracted object. Theses works exhibit a realizability predicate $p\ \mathtt{r}\ T$, which is read as follows: "the program p realize the type T". And the goal is then to show that $\mathcal{E}(t)\ \mathtt{r}\ T$ when $t : T$. The critical point is the realization of functions. The natural rule to realize a functional type is as follows:

$$p\ \mathtt{r}\ A \to B \ \text{ iff }\ \forall a,\ a\ \mathtt{r}\ A \Rightarrow (p\ a)\ \mathtt{r}\ B$$

If one wants to generalize this to the dependent product type $\forall x : A, B$, it is necessary to take into account the possible appearance of x in B, that we underline by the notation $B(x)$. Let us try:

$$p\ \mathtt{r}\ \forall \mathrm{x{:}A, B(x)} \ \text{ iff }\ \forall a,\ a\ \mathtt{r}\ A \Rightarrow (p\ a)\ \mathtt{r}\ B(a)$$

In a system allowing an internal extraction, this may be appropriate, although $a\ \mathtt{r}\ A$ does not necessarily imply that $a : A$, and hence $B(a)$ may be badly typed. In any case, like the extracted parts have here the type Λ, there is no chance that this rule is well typed. One can then try to be rely on an element x of type A:

$$p\ \mathtt{r}\ \forall \mathrm{x{:}A, B(x)} \ \text{ iff }\ \forall x{:}A,\ \forall a,\ a\ \mathtt{r}\ A \Rightarrow (p\ a)\ \mathtt{r}\ B(x)$$

In this formulation, the annoying point is now that x and a are not correlated, although they must morally correspond respectively to a CIC_m term and to a possible extraction of this term. To express this correlation, we have chosen to introduce a simulation predicate $T \sim p$ relating a CIC_m term t and an extracted term p. More precisely, we define a family of predicates $\sim_T$ indexed by CIC_m types T:

$$\sim_T\ :\ T \to \Lambda \to \mathtt{Prop}$$

The realization rule for a product now becomes:

$$f \sim_{\forall x:A,\, B(x)} p \quad \text{iff} \quad \forall x{:}A,\ \forall a,\ x \sim_A a\ \Rightarrow\ (f\ x) \sim_{B(x)} (p\ a)$$

In fact, we will center everything on these predicates $\sim_T$, and relegate to the background the realizability predicate $\mathtt{r}$, which will be in fact defined via the simulation predicates:

$$p \;\mathtt{r}\; T \;\text{ iff }\; \exists t{:}T,\; t \sim_T p$$

And to achieve the initial goal, which was to establish $\mathcal{E}(t)\;\mathtt{r}\;T$, one now need to prove that $t \sim_T \mathcal{E}(t)$, which is more precise.

At the technical level, these predicates $\sim_T$ for $T : s$ are not yet general enough to be defined and handled directly. We indeed need to extend them for type schemes $T : K$. As in general, these predicates will not be binary relations anymore, we will not keep the infix notation $\sim$, and will speak instead of predicates $\widehat{T}$. When T is a type, $t \sim_T p$ will then be just an abbreviation for $(\widehat{T}\;t\;p)$.

In practice, we will not be able to define directly these predicates $\widehat{T}$, but rather some dependent pairs $[\![T]\!]$ which will have these predicates $\widehat{T}$ as second components, and an enriched alternative $\overline{T}$ for T as first components. We will thus use three types of dependent pairs adapted to our needs:

```
Record Type⁺ : Type := mk_Type
 { type_Type :> Type;
   pred_Type : type_Type → Λ → Prop }.
Record Set⁺ : Type := mk_Set
 { type_Set :> Set;
   pred_Set : type_Set → Λ → Prop }.
Record Prop⁺ : Type := mk_Prop
 { type_Prop :> Prop;
   pred_Prop : type_Prop → Λ → Prop := fun _ _ ⇒ True }.
```

If s is one of the three sort `Set`, `Prop` or `Type`, the constructor of a dependent pair of type s^+ is `mk_`s, and the two projections are `type_`s and `pred_`s. These two projections respectively give again the type contained in s^+ and the predicate of simulation associated with this type. Let us look at the types of theses projections now. The type of the first is very simple:

```
type_s : s⁺ → s.
```

The one of the second presents a dependency:

```
pred_s : ∀T:s⁺, (type_s T) → Λ → Prop.
```

In addition, the first projection can be seen as a coercion of s^+ into s, which is announced by the Coq syntax `:>` instead of the usual `:` syntax. In Coq, this coercion allow to avoid writing the first projection explicitly. In the theoretical study that follows, we will continue to clarify these projections. On the other hand, we will do it using a lighter syntax:

- $T.\mathbb{1}$ for (`type_`s T)
- $T.\mathbb{2}$ for (`pred_`s T)

These abbreviations are voluntarily ambiguous, because they do not specify the initial type $\texttt{Type}^+$, $\texttt{Set}^+$ or $\texttt{Prop}^+$. When needed, typing the expression T allows to solve this ambiguity.

It should be noted that the case of $\texttt{Prop}^+$ is a little particular. We have indeed definitively fixed the contents of the field `pred_Prop` during the definition of the $\texttt{Prop}^+$ type. Thus the predicate associated with an object in $\texttt{Prop}^+$ is necessarily the trivial predicate, always equivalent to `True`. The idea is that the extraction of a logical part can be chosen arbitrarily, without that having repercussion over the correctness of the extracted term. At the practical level, the only difference between the $\texttt{Prop}^+$type and the $\texttt{Type}^+$types and $\texttt{Set}^+$is that it constructor `mk_Prop` awaits one argument instead of two. On the other hand two projections `type_Prop` and `pred_Prop` exist and operate as described previously.

The homogeneity between the s^+ will allow us to plunge $\texttt{Set}^+$ and $\texttt{Prop}^+$ in $\texttt{Type}^+$ in order to mimic cumulativities $\texttt{Set} < \texttt{Type}$ and $\texttt{Prop} < \texttt{Type}$. In $\textsc{Cic}_m$, these cumulativities are announced by the marks ‡ and †. These marks will here be mirrored by the two following functions:

```
Definition Set⁺_Type⁺ := fun T:Set⁺ ⇒ let (t,p):=T in mk_Type t‡ p.
Definition Prop⁺_Type⁺ := fun T:Prop⁺ ⇒ let (t,p):=T in mk_Type t† p.
```

2.4.3 The transformation $[\![.]\!]$

We now define a transformation $[\![.]\!]$ for all object of $\textsc{Cic}_m$. In fact, only the transformation of types really imports. But these types can be found on any positions, for example in the arguments of an inductive constructor, and then re-appear during a pattern matching. And a type can also be in fact the application of a type scheme to arguments. In short, $[\![.]\!]$ needs to deal with any $\textsc{Cic}_m$ term.

In the particular case of a type $T : s$, $[\![T]\!]$ will be a dependent pair whose second component $[\![T]\!].2$ is the expected simulation predicate. In this precise case, we will shorten $[\![T]\!].1$ in $[\![T]\!]_1$ and $[\![T]\!].2$ in $[\![T]\!]_2$. To help the reading, it can be interesting to note that this meta-theoretical transformation $[\![.]\!]$ preserve typing judgments: if $t : T$, then $[\![t]\!] : [\![T]\!]_1$. Here comes now the definition of $[\![.]\!]$ by structural induction:

- $[\![x]\!] = x$
- $[\![t\ t']\!] = [\![t]\!]\ [\![t']\!]$
- $[\![\lambda x : T,\ t]\!] = \lambda x : [\![T]\!]_1,\ [\![t]\!]$
- $[\![\forall x : T,\ T']\!] =$

  $$\texttt{mk\_}s\ \ \forall x : [\![T]\!]_1,\ [\![T']\!]_1\quad \lambda t, \lambda p, \forall x : [\![T]\!]_1, \forall x' : \Lambda,\ [\![T]\!]_2\ x\ x' \to [\![T']\!]_2\ (t\ x)\ (p\ x')$$

  if the type[3] of $\forall x : T, T'$ is $s \neq \texttt{Prop}$. The type of the two shortened abstractions are respectively $\forall x : [\![T]\!]_1,\ [\![T']\!]_1$ and Λ. And if $s = \texttt{Prop}$, one removes the second argument of $\texttt{mk\_}s$.
- $[\![s]\!] = \texttt{mk\_Type}\ \ s^+\ \ \lambda\_, \lambda\_, \texttt{True}$
- $[\![t^\ddagger]\!] = \texttt{Set}^+\texttt{\_Type}^+\ [\![t]\!]$

[3]Notice: in $\textsc{Cic}_m$, we have uniqueness of types modulo conversion.

– $[\![t^\dagger]\!] = \texttt{Prop}^+\texttt{\_Type}^+\ [\![t]\!]$

– $[\![I]\!] = \lambda\overrightarrow{u : [\![U]\!]_1},\ (\texttt{mk\_}s\ (\overline{I}\ \overrightarrow{u})\ (\widehat{I}\ \overrightarrow{u}))$

when the inductive type I admits $\forall\overrightarrow{u : U},\ s$ as arity, with $s \neq \texttt{Prop}$. See the context transformation for the definition of $\overline{I}$ and $\widehat{I}$. And if s is Prop, we just remove the second argument of $\texttt{mk\_}s$.

– $[\![C]\!] = \overline{C}$ for a constructor C of the inductive type I. And $\overline{C}$ is then a constructor of the inductive $\overline{I}$. See the context transformation for the definition of $\overline{I}$.

– $[\![\texttt{case}(e,\ P,\ \overrightarrow{f_i})]\!] = \texttt{case}([\![e]\!],\ \overline{P},\ \overrightarrow{[\![f_i]\!]})$

Here, for P of form $\lambda\overrightarrow{u},\ \lambda x : (I\ \overrightarrow{q}\ \overrightarrow{u}),\ T$, one notes $\overline{P} = \lambda\overrightarrow{u},\ \lambda x : (\overline{I}\ \overrightarrow{[\![q]\!]}\ \overrightarrow{u}),\ [\![T]\!]_1$.

– $[\![\texttt{fix}\ x : T := t]\!] = \texttt{fix}\ x : [\![T]\!]_1 := [\![t]\!]$

Finally this transformation $[\![.]\!]$ extends to contexts as follows:

– $[\![\Gamma; (x : T)]\!] = [\![\Gamma]\!]; (x : [\![T]\!]_1)$

– Each declaration of inductive a I of arity $K = \forall\overrightarrow{u : U},\ s$ and of constructors $C_i : T_i$ is replaced by two inductive $\overline{I}$ and $\widehat{I}$, the second being useful only if $s \neq \texttt{Prop}$.

1. $\overline{I}$ is simply the propagation of $[\![.]\!]_1$ to I. Its arity is $\overline{K} = \forall\overrightarrow{u : [\![U]\!]_1},\ s$ and its constructors $\overline{C}_i$ has type $[\![T_i]\!]_1$.
2. $\widehat{I}$ is used as simulation predicate for $\overline{I}$. Its arity is $\widehat{K} = \forall\overrightarrow{u : [\![U]\!]_1},\ (\overline{I}\ \overrightarrow{u}) \to \Lambda \to \texttt{Prop}$. And its constructors $\widehat{C}_i$ have type $([\![T_i]\!]_2\ \overline{C}_i\ C^i_\Lambda)$, with C^i_Λ being the i-th constructor of the inductive $\mathcal{E}(I)$ extracted from I and internalized in Λ.

Let us reconsider one moment the definitions of $\overline{I}$ and $\widehat{I}$. It can to seem indeed surprising that $[\![T_i]\!]_1$ on the one hand and $([\![T_i]\!]_2\ \overline{C}_i\ C^i_\Lambda)$ on the other hand are valid types of constructors. In fact if T_i is form $\forall\overrightarrow{v : V},\ (I\ \overrightarrow{w})$, one then has:

$[\![T_i]\!]_1 = \forall\overrightarrow{v : [\![V]\!]_1},\ (\overline{I}\ \overrightarrow{[\![w]\!]})$

and also:

$([\![T_i]\!]_2\ \overline{C}_i\ C^i_\Lambda) = \forall[\![\overrightarrow{v : V}]\!],\ (\widehat{I}\ \overrightarrow{S(w)}\ (\overline{C}_i\ \overrightarrow{v})\ (C^i_\Lambda\ \overrightarrow{v}^\Lambda))$

We used here two new notations:

- $\forall[\![\overrightarrow{x : T}]\!],\ \ldots$ which denotes $\forall\overrightarrow{x : [\![T]\!]_1},\ \ldots$ plus one additional program variable x' associated with each variable initial x, and the proof H_x of correctness connecting x and x'. More precisely:

  $\forall[\![(x : T)\overrightarrow{(v_i : V_i)}]\!],\ \ldots = \forall x : [\![T]\!]_1,\ \forall x' : \Lambda,\ \forall H_x : ([\![T]\!]_2\ x\ x'),\ \forall[\![\overrightarrow{v_i : V_i}]\!],\ \ldots$
- $\overrightarrow{u}^\Lambda$ is the extracted version of the variables: each variable x is replaced by its associated extracted variable x'.

For a context Γ, one will need later to handle an even richer alternative for $[\![\Gamma]\!]$, in which a declaration $(x : T)$ generates, in addition to $(x : [\![T]\!]_1)$, the declaration of a associated program variable $(x' : \Lambda)$ and a proof H_x connecting x and x', of type $([\![T]\!]_2\ x\ x')$. We will note this extended context $[\![\Gamma]\!]_+$.

2.4.4 One example

It is now time to test all this pretty formalism on one example. We will seek to know which correctness condition must satisfy a program extracted $\mathcal{E}(\texttt{div})$ when the original term `div` is an integer division:

```
div : ∀a b:nat,b≠0 → { q:nat | q*b ≤ a ∧ a < (S q)*b }
```

First of all, as the arity of `nat` is directly `Set`, one has $[\![\texttt{nat}]\!] = \texttt{mk\_Set}\ \ \overline{\texttt{nat}}\ \ \widehat{\texttt{nat}}$, and thus $[\![\texttt{nat}]\!]_{\mathbb{1}} = \overline{\texttt{nat}}$. Types of new constructors $\overline{\texttt{O}}$ and $\overline{\texttt{S}}$ of $\overline{\texttt{nat}}$ are then respectively $\overline{\texttt{nat}}$ and $\overline{\texttt{nat}} \to \overline{\texttt{nat}}$, with the result that $\overline{\texttt{nat}}$ is exactly isomorphic with `nat`. We will thus identify them. And concerning $\widehat{\texttt{nat}}$, one has:

$$\begin{array}{l}
\texttt{Inductive }\widehat{\texttt{nat}}\texttt{ : nat} \to \Lambda \to \texttt{Prop :=}\\
\ \texttt{| }\widehat{\texttt{O}}\texttt{ : }\widehat{\texttt{nat}}\texttt{ O O}_\Lambda\\
\ \texttt{| }\widehat{\texttt{S}}\texttt{ : }\forall\texttt{n:nat,}\forall\texttt{n':}\Lambda\texttt{,}\widehat{\texttt{nat}}\texttt{ n n'} \to \widehat{\texttt{nat}}\texttt{ (S n) (S}_\Lambda\texttt{ n').}
\end{array}$$

This inductive predicate expresses simply the fact that $\texttt{n':}\Lambda$ is a correct extraction of `(S (S ... (S O)...))` iff $\texttt{n' = (S}_\Lambda\texttt{ (S}_\Lambda\texttt{ ... (S}_\Lambda\texttt{ O}_\Lambda\texttt{)...))}$. Another inductive type used in the type of `div` is `sig` :

$$\begin{array}{l}
\texttt{Inductive }\overline{\texttt{sig}}\texttt{ (A:Set}^+\texttt{)(P:A.}\mathbb{1} \to \texttt{Prop}^+\texttt{) : Set :=}\\
\ \texttt{| }\overline{\texttt{exist}}\texttt{ : }\forall\texttt{x:A.}\mathbb{1}\texttt{, (P x).}\mathbb{1} \to \overline{\texttt{sig}}\texttt{ A P.}
\end{array}$$

This definition is not, in fact, so different from that of `sig`, in particular if one omits the coercions $.\mathbb{1}$ inferable by `Coq`. There is even the following relation:

$\overline{\texttt{sig}}\ A\ P \leftrightarrow \texttt{sig}\ \ A.\mathbb{1}\ \ \lambda x, ((P\ x).\mathbb{1})$

Here is now the definition of the inductive $\widehat{\texttt{sig}}$:

$$\begin{array}{l}
\texttt{Inductive }\widehat{\texttt{sig}}\texttt{ (A:Set}^+\texttt{)(P:A.}\mathbb{1} \to \texttt{Prop}^+\texttt{) : }\overline{\texttt{sig}}\texttt{ A P} \to \Lambda \to \texttt{Prop :=}\\
\ \texttt{| }\widehat{\texttt{exist}}\texttt{ : }\forall\texttt{x:A.}\mathbb{1}\texttt{,}\forall\texttt{x':}\Lambda\texttt{, A.}\mathbb{2}\texttt{ x x'} \to\\
\quad\quad \forall\texttt{h:(P x).}\mathbb{1}\texttt{,}\forall\texttt{h':}\Lambda\texttt{, (P x).}\mathbb{2}\texttt{ h h'} \to\\
\quad\quad \widehat{\texttt{sig}}\texttt{ A P (}\overline{\texttt{exist}}\texttt{ x h) (exist}_\Lambda\texttt{ x' h')).}
\end{array}$$

Let us name `Div` the type of `div` and let `P` be `(fun a b q ⇒ q*b ≤ a ∧ a < (S q)*b)`. We obtain then that $[\![\texttt{Div}]\!]_{\mathbb{1}}$ is isomorphic with `Div`, and that $[\![\texttt{Div}]\!]_{\mathbb{2}}$ has for type $\texttt{Div} \to \Lambda \to \texttt{Prop}$ and is:

$$\begin{array}{l}
[\![\texttt{Div}]\!]_{\mathbb{2}}\texttt{ = fun (f:Div)(p:}\Lambda\texttt{)} \Rightarrow\\
\ \forall\texttt{a:nat,}\forall\texttt{a':}\Lambda\texttt{,}\widehat{\texttt{nat}}\texttt{ a a'} \to\\
\ \forall\texttt{b:nat,}\forall\texttt{b':}\Lambda\texttt{,}\widehat{\texttt{nat}}\texttt{ b b'} \to\\
\ \forall\texttt{h:b}\neq\texttt{O,}\forall\texttt{h':}\Lambda\texttt{, }[\![\texttt{b}\neq\texttt{O}]\!]_{\mathbb{2}}\texttt{ h h'} \to\\
\ \texttt{(}\widehat{\texttt{sig}}\texttt{ }[\![\texttt{nat}]\!]\texttt{ (}[\![\texttt{P}]\!]\texttt{ a b) (f a b h) (p a' b' h')).}
\end{array}$$

However, since $\texttt{b}\neq\texttt{O}$ is a logical proposition, the predicate $[\![\texttt{b}\neq\texttt{O}]\!]_{\mathbb{2}}$ which is associated for him is the trivial predicate. And similarly, for any triplet of integer `a`, `b` and `q`, the predicate $([\![\texttt{P}]\!]\ a\ b\ q).\mathbb{2}$ is also $\lambda\_,\ \lambda\_,$ `True`. Finally, $[\![\texttt{Div}]\!]_{\mathbb{2}}$ is equivalent to:

```
fun (f:Div)(p:Λ) ⇒
 ∀a:nat,∀a':Λ,nat^ a a' →
 ∀b:nat,∀b':Λ,nat^ b b' →
 ∀h:b≠0,∀h':Λ,
 ∃q,∃q',(f a b h) = (exist q _) ∧ (p a' b' h') = (exist_Λ q' _) ∧
         nat^ q q' ∧ P a b q.
```

Our extracted function `div` will thus be correct iff for two arguments corresponding to `Coq` integers (the second being non-null) and a third unspecified argument, it returns a constructor $\texttt{exist}_\Lambda$ whose first argument has a `Coq` counterpart that check the post-condition `P`. Despite all the delays resulting of the formalism complexity, the informal meaning of pre- and post-conditions is indeed there.

2.4.5 Substitution properties of the transformation $[\![.]\!]$

The first step on the way to the correctness proof for the extraction is to establish the substitution properties what verifies the transformation $[\![.]\!]$. This will concern here only the substitution of the last variable of a context, which in particular does not influence inductive types defined previously.

Lemma 12 *The transformation $[\![.]\!]$ preserves substitutions: $[\![t\{x\leftarrow r\}]\!] = [\![t]\!]\{x\leftarrow[\![r]\!]\}$*

Proof. This proof is done by structural induction, and is purely syntactic. Because of the important number of cases, we will not treat them all. Here a typical case:

$$\begin{array}{rcl}
[\![(t\ t')\{x\leftarrow r\}]\!] & \overset{subst.}{=} & [\![t\{x\leftarrow r\}\ t'\{x\leftarrow r\}]\!] \\
& \overset{def.\ [\![.]\!]}{=} & [\![t\{x\leftarrow r\}]\!]\ [\![t'\{x\leftarrow r\}]\!] \\
& \overset{hyp.\ rec.}{=} & [\![t]\!]\{x\leftarrow[\![r]\!]\}\ [\![t']\!]\{x\leftarrow[\![r]\!]\} \\
& \overset{subst.}{=} & ([\![t]\!]\ [\![t']\!])\{x\leftarrow[\![r]\!]\} \\
& \overset{def.\ [\![.]\!]}{=} & [\![t\ t']\!]\{x\leftarrow[\![r]\!]\}
\end{array}$$

All the cases other than the basic cases follow this scheme:

1. propagation of substitution
2. use of $[\![.]\!]$ definition
3. repeated uses of the induction hypothesis
4. factorization of substitution
5. reversed use of $[\![.]\!]$ definition

There is nevertheless a delicate point: nothing guarantees a priori that the same definition rule for $[\![.]\!]$ will apply before (point 5) and after (point 2) the propagation of substitution. Fortunately a substitution does not upset the structure: a substituted application remains an application, a substituted matching remains a matching, etc. Of course, a substituted variable can give anything else, but this case is correctly managed:

$$\begin{aligned}[\![x\{x\leftarrow r\}]\!] &\overset{subst.}{=} [\![r]\!] \\ &\overset{subst.}{=} [\![x]\!]\{x\leftarrow [\![r]\!]\}\end{aligned}$$

Lastly, even if the same definition rule for $[\![.]\!]$ is used before and after substitution, it is still necessary that this rule produces similar objects. In particular, the transformation $[\![I]\!]$ of one inductive type and the transformation $[\![\forall x : T,\, T']\!]$ of a product type use both a constructor $\texttt{mk\_}s$ that depend on a sort s. This sort is respectively the sort at the end of the arity of I and the type of the product. It is thus essential for the validity of the present lemma that a substitution cannot modify this sort s. This is a commonplace in the inductive case: the sort at the end of the arity of I cannot change via substitution. On the other hand this is much less obvious in the product case. In the initial CIC system, they is even false: if $x : \texttt{Type}$, then $\forall y : Y,\, x$ has type $\texttt{Type}$, whereas $(\forall y : Y,\, x)\{x\leftarrow \texttt{True}\}$ has type $\texttt{Prop}$. What about CIC$_m$ now? If one consider again this example, one obtain:

$[\![\forall y : Y,\, x]\!] = \texttt{mk\_Type}\ \forall y : [\![Y]\!]_{\mathbb{1}},\, [\![x]\!]_{\mathbb{1}}\ \ldots$

To be legal, the substitution must now be $\{x\leftarrow \texttt{True}^\dagger\}$. One then has:

$$\begin{aligned}&[\![\forall y : Y,\, x\{x\leftarrow \texttt{True}^\dagger\}]\!] \\ &\quad = [\![\forall y : Y,\, \texttt{True}^\dagger]\!] \\ &\quad = \texttt{mk\_Type}\ \forall y : [\![Y]\!]_{\mathbb{1}},\, (\texttt{Prop}^+\texttt{\_Type}^+\ [\![\texttt{True}]\!]).\mathbb{1}\ \ldots\end{aligned}$$

On the other side, the substitution to be applied after transformation is $\{x\leftarrow [\![\texttt{True}^\dagger]\!]\} = \{x\leftarrow \texttt{Prop}^+\texttt{\_Type}^+\ [\![\texttt{True}]\!]\}$. And we finally have the same final term.

In a general way, our modified system CIC$_m$ always verifies the conservation of the type of product by substitution. Indeed, if s is the initial type, then $s\{x\leftarrow r\} = s$ is clearly a type of the substituted product according to the lemma 4. We can then conclude using the uniqueness of types in CIC$_m$. □

2.4.6 Reduction properties of the transformation $[\![.]\!]$

As we excluded from our study the constants and the “let-in”, we consider here only the reductions β and ι.

Theorem 7 *The transformation $[\![.]\!]$ preserves the reduction: $t \to_{\beta\iota} t' \Rightarrow [\![t]\!] \to_{\beta\iota} [\![t']\!]$*

PROOF. Let us start first with the cases where the reduction is performed at the top:

- $t = (\lambda u : U,\, a)\ b$ being reduced by β in $t' = a\{u\leftarrow b\}$. Then $[\![t]\!]$ is $(\lambda u : [\![U]\!]_{\mathbb{1}},\, [\![a]\!])\ [\![b]\!]$ and can be indeed reduced to $[\![a]\!]\{u\leftarrow [\![b]\!]\}$. And the former is equal via the previous lemma of substitution to $[\![a\{u\leftarrow b\}]\!] = [\![t']\!]$.
- If a β-reduction occurs at the head of a type scheme, we proceed in the same way.
- $t = \texttt{case}(C_i\ \overrightarrow{p}\ \overrightarrow{u},\, P,\, f_j)$ being reduced by ι in $t' = (f_i\ \overrightarrow{u})$. Then $[\![t]\!]$ is in fact $\texttt{case}(\overline{C}_i\ \overrightarrow{[\![p]\!]}\ \overrightarrow{[\![u]\!]},\, \overline{P},\, \overrightarrow{[\![f_j]\!]})$ which can be indeed reduced to $([\![f_i]\!]\ \overrightarrow{[\![u]\!]}) = [\![t']\!]$.

- $t = (\texttt{fix}\ x : T := t_0)\ \overrightarrow{u}$ being reduced by ι in $t' = t_0\{x \leftarrow (\texttt{fix}\ x : T := t_0)\}\ \overrightarrow{u}$. Then $[\![t]\!] = (\texttt{fix}\ x : [\![T]\!]_{\mathbb{1}} := [\![t_0]\!])\ \overrightarrow{[\![u]\!]}$. In particular the inductive guard argument, which started with one inductive constructor, preserves his head structure. One can thus reduce $[\![t]\!]$ in $[\![t_0]\!]\{x \leftarrow (\texttt{fix}\ x : [\![T]\!]_{\mathbb{1}} := [\![t_0]\!])\}\ \overrightarrow{[\![u]\!]}$. The previous substitution lemma shows that this term is indeed equal to $[\![t']\!]$.
- If a ι-reduction occurs at the head of a type scheme, we proceeds in the same way.

When the reduction takes place deep inside a term, one proceeds by induction over the structure of the initial object t. As these cases bring no surprises, we will not detail them. □

2.4.7 Validity of terms produced by the transformation $[\![.]\!]$

Theorem 8 *The transformation $[\![.]\!]$ preserves typing judgment:* $\Gamma \vdash t : T \ \Rightarrow\ [\![\Gamma]\!] \vdash [\![t]\!] : [\![T]\!]_{\mathbb{1}}$

PROOF. By induction over the initial typing derivation, and at the same time over the good formation of transformed contexts $[\![\Gamma]\!]$.

(WF) $\mathcal{WF}(\varnothing) \Rightarrow \mathcal{WF}([\![\varnothing]\!])$

$$\text{(WF)}\quad \frac{\Gamma \vdash U : s \quad u \notin \Gamma}{\mathcal{WF}(\Gamma; (u : U))} \Rightarrow \frac{\dfrac{[\![\Gamma]\!] \vdash [\![U]\!] : [\![s]\!]_{\mathbb{1}}}{[\![\Gamma]\!] \vdash [\![U]\!]_{\mathbb{1}} : s} \quad u \notin [\![\Gamma]\!]}{\mathcal{WF}([\![\Gamma]\!]; (u : [\![U]\!]_{\mathbb{1}}))}$$

Indeed $[\![s]\!]_{\mathbb{1}} = s^+$ and thus the first projection of $[\![U]\!]$ has type s.

$$\text{(Ax)}\quad \frac{\mathcal{WF}(\Gamma) \quad s \in \{\texttt{Set}, \texttt{Prop}, \texttt{Type}_i\} \quad i < j}{\Gamma \vdash s : \texttt{Type}_j} \Rightarrow \frac{\mathcal{WF}([\![\Gamma]\!]) \quad s \in \{\texttt{Set}, \texttt{Prop}, \texttt{Type}_i\} \quad i < j}{[\![\Gamma]\!] \vdash [\![s]\!] : [\![\texttt{Type}_j]\!]_{\mathbb{1}}}$$

Indeed $[\![s]\!] = \texttt{mk\_Type}\ s^+\ \lambda\_,\ \lambda\_,\ \texttt{True}$ and $[\![\texttt{Type}_j]\!]_{\mathbb{1}} = \texttt{Type}^+$.

$$\text{(Var)}\quad \frac{\mathcal{WF}(\Gamma) \quad (u : U) \in \Gamma}{\Gamma \vdash U : U} \Rightarrow \frac{\mathcal{WF}([\![\Gamma]\!]) \quad (u : [\![U]\!]_{\mathbb{1}}) \in [\![\Gamma]\!]}{[\![\Gamma]\!] \vdash [\![u]\!] : [\![U]\!]_{\mathbb{1}}}$$

First of all, the variables are invariant by $[\![.]\!]$, therefore $[\![u]\!] = u$. And in addition, types present in environment $[\![\Gamma]\!]$ have indeed the form $[\![U]\!]_{\mathbb{1}}$ (cf. rule (WF)).

$$\text{(Prod)}\quad \frac{\Gamma \vdash T : s_1 \quad \Gamma; (x : T) \vdash T' : s_2 \quad \mathcal{P}(s_1, s_2, s_3)}{\Gamma \vdash \forall x : T, T' : s_3}$$
$$\Rightarrow$$
$$\frac{\dfrac{[\![\Gamma]\!] \vdash [\![T]\!] : [\![s_1]\!]_{\mathbb{1}} \qquad [\![\Gamma]\!]; (x : [\![T]\!]_{\mathbb{1}}) \vdash [\![T']\!] : [\![s_2]\!]_{\mathbb{1}} \qquad \mathcal{PROD}(s_1, s_2, s_3)}{\dots}}{[\![\Gamma]\!] \vdash \texttt{mk\_}s_3\ \ \forall x : \overline{T}, \overline{T'}\ \ \lambda t, \lambda p, \forall x : \overline{T}, \forall x' : \Lambda, (\widehat{T}\ x\ x') \rightarrow (\widehat{T'}\ (t\ x)\ (p\ x'))\ :\ [\![s_3]\!]_{\mathbb{1}}}$$

With $\overline{T} = [\![T]\!]_{\mathbb{1}}$ and $\widehat{T} = [\![T]\!]_{2}$ and idem for T'.

The derivation outlined here is the one for $s_3 \neq \texttt{Prop}$. Instead of writing all the details, we will just describe the general scheme. $[\![s_1]\!]_{\mathbb{1}} = s_1^+$ and $[\![s_2]\!]_{\mathbb{1}} = s_2^+$. Thus $\overline{T}$ and $\overline{T'}$ have as respective types s_1 and s_2, and $\widehat{T}$ and $\widehat{T'}$ have as respective types $\overline{T} \to \Lambda \to \texttt{Prop}$ and $\overline{T'} \to \Lambda \to \texttt{Prop}$. A use of the typing rule for the product allows us then to affirm that $\forall x : \overline{T}, \overline{T'}$ admits s_3 for type. It is then easy to see that the predicate part of the dependent pair is indeed of type $\forall x : \overline{T}, \overline{T'} \to \Lambda \to \texttt{Prop}$. It is then quite legal to form this dependent pair, which has thus type $[\![s_3]\!]_{\mathbb{1}} = s_3^+$.
The case $s_3 = \texttt{Prop}$ is a simplified version of what precedes, since $\texttt{mk\_}s_3$ only has one argument.

$$\text{(Lam)}\ \frac{\Gamma \vdash \forall u : U,\, V : s \quad \Gamma; (u : U) \vdash v : V}{\Gamma \vdash \lambda u : U,\, v : \forall u : U,\, V}$$
$\Rightarrow$
$$\frac{\dfrac{[\![\Gamma]\!] \vdash [\![\forall u : U,\, V]\!] : [\![s]\!]_{\mathbb{1}}}{[\![\Gamma]\!] \vdash \forall u : [\![U]\!]_{\mathbb{1}},\, [\![V]\!]_{\mathbb{1}} : s} \quad [\![\Gamma]\!]; (u : [\![U]\!]_{\mathbb{1}}) \vdash [\![v]\!] : [\![V]\!]_{\mathbb{1}}}{[\![\Gamma]\!] \vdash \lambda u : [\![U]\!]_{\mathbb{1}},\, [\![v]\!] : [\![\forall u : U,\, V]\!]_{\mathbb{1}}}$$

Using the induction hypothesis for $[\![\forall u : U,\, V]\!]$, one deduces via the first projection that $[\![\forall u : U,\, V]\!]_{\mathbb{1}} = \forall u : [\![U]\!]_{\mathbb{1}},\, [\![V]\!]_{\mathbb{1}}$ has type s. This plus the other induction hypothesis for $[\![v]\!]$ allows us to apply the rule (Lam) and to conclude.

$$\text{(App)}\ \frac{\Gamma \vdash v : \forall u : U,\, V \quad \Gamma \vdash w : U}{\Gamma \vdash (v\ w) : V\{u \leftarrow w\}} \Rightarrow \frac{[\![\Gamma]\!] \vdash [\![v]\!] : [\![\forall u : U,\, V]\!]_{\mathbb{1}} \quad [\![\Gamma]\!] \vdash [\![w]\!] : [\![U]\!]_{\mathbb{1}}}{[\![\Gamma]\!] \vdash ([\![v]\!]\ [\![w]\!]) : [\![V]\!]_{\mathbb{1}}\{u \leftarrow [\![w]\!]\}}$$

As previously, one use the equality $[\![\forall u : U,\, V]\!]_{\mathbb{1}} = \forall u : [\![U]\!]_{\mathbb{1}},\, [\![V]\!]_{\mathbb{1}}$. Lastly, one has indeed $[\![V]\!]_{\mathbb{1}}\{u \leftarrow [\![w]\!]\} = [\![V\{u \leftarrow w\}]\!]_{\mathbb{1}}$ via the previous substitution lemma.

$$\text{(Conv)}\ \frac{\Gamma \vdash U : s \quad \Gamma \vdash t : T \quad T =_{\beta\iota} U}{\Gamma \vdash t : U}$$
$\Rightarrow$
$$\frac{\dfrac{[\![\Gamma]\!] \vdash [\![U]\!] : [\![s]\!]_{\mathbb{1}}}{[\![\Gamma]\!] \vdash [\![U]\!]_{\mathbb{1}} : s} \quad [\![\Gamma]\!] \vdash [\![t]\!] : [\![T]\!]_{\mathbb{1}} \quad [\![T]\!]_{\mathbb{1}} =_{\beta\iota} [\![U]\!]_{\mathbb{1}}}{[\![\Gamma]\!] \vdash [\![t]\!] : [\![U]\!]_{\mathbb{1}}}$$

Note that $T =_{\beta\iota} U$ implies indeed $[\![T]\!]_{\mathbb{1}} =_{\beta\iota} [\![U]\!]_{\mathbb{1}}$, because of the previous theorem of reduction preservation for $[\![.]\!]$.

$$\text{(CumT)}\ \frac{\Gamma \vdash t : \texttt{Type}_i \quad i < j}{\Gamma \vdash t : \texttt{Type}_j}$$
$\Rightarrow$
$$\frac{[\![\Gamma]\!] \vdash [\![t]\!] : [\![\texttt{Type}_i]\!]_{\mathbb{1}} \quad i < j}{[\![\Gamma]\!] \vdash [\![t]\!] : [\![\texttt{Type}_j]\!]_{\mathbb{1}}}$$

This is immediate, since $[\![\texttt{Type}_i]\!]_{\mathbb{1}} = [\![\texttt{Type}_j]\!]_{\mathbb{1}} = \texttt{Type}^+$.

(CumP) $\dfrac{\Gamma \vdash t : \texttt{Prop}}{\Gamma \vdash t^{\dagger} : \texttt{Type}}$
$\Rightarrow$

$$\frac{[\![\Gamma]\!] \vdash [\![t]\!] : [\![\texttt{Prop}]\!]_{\mathbb{1}}}{[\![\Gamma]\!] \vdash [\![t^{\dagger}]\!] : [\![\texttt{Type}]\!]_{\mathbb{1}}}$$

Here $[\![t^{\dagger}]\!] = \texttt{Prop}^{+}\texttt{\_Type}^{+}\ [\![t]\!]$, which is then indeed of type $[\![\texttt{Type}]\!]_{\mathbb{1}} = \texttt{Type}^{+}$.

(CumS) $\dfrac{\Gamma \vdash t : \texttt{Set}}{\Gamma \vdash t^{\ddagger} : \texttt{Type}}$
$\Rightarrow$

$$\frac{[\![\Gamma]\!] \vdash [\![t]\!] : [\![\texttt{Set}]\!]_{\mathbb{1}}}{[\![\Gamma]\!] \vdash [\![t^{\ddagger}]\!] : [\![\texttt{Type}]\!]_{\mathbb{1}}}$$

Here $[\![t^{\ddagger}]\!] = \texttt{Set}^{+}\texttt{\_Type}^{+}\ [\![t]\!]$, which is then indeed of type $[\![\texttt{Type}]\!]_{\mathbb{1}} = \texttt{Type}^{+}$.

(I-Type) For an arity $K = \forall \overrightarrow{u : U} s$, with $s \neq \texttt{Prop}$, one has:

$$\frac{\mathcal{WF}(\Gamma) \quad \texttt{Ind}_n(\Gamma_I := \Gamma_C) \in \Gamma \quad (I : K) \in \Gamma_I}{\Gamma \vdash I : K}$$
$\Rightarrow$

$$\cfrac{\cfrac{\mathcal{WF}([\![\Gamma]\!]) \quad \texttt{Ind}_n(\Gamma_{\overline{I},\widehat{I}} := \Gamma_{\overline{C},\widehat{C}}) \in [\![\Gamma]\!] \quad (\overline{I} : \overline{K}) \in \Gamma_I \quad (\widehat{I} : \widehat{K}) \in \Gamma_I}{\dots}}{[\![\Gamma]\!] \vdash \lambda \overrightarrow{u : [\![U]\!]_{\mathbb{1}}},\ (\texttt{mk\_}s\ (\overline{I}\ \overrightarrow{u})\ (\widehat{I}\ \overrightarrow{u})) : [\![K]\!]_{\mathbb{1}}}$$

with the abbreviations $\overline{K} = \forall \overrightarrow{u : [\![U]\!]_{\mathbb{1}}},\ s$ and $\widehat{K} = \forall \overrightarrow{u : [\![U]\!]_{\mathbb{1}}},\ (\overline{I}\ \overrightarrow{u}) \to \Lambda \to \texttt{Prop}$. And one has $[\![K]\!]_{\mathbb{1}} = \forall \overrightarrow{u : [\![U]\!]_{\mathbb{1}}},\ s^{+}$. The dotted lines in the derivation tree correspond to a double uses of the rule (I-Type), to deduce that $\overline{I}$ and $\widehat{I}$ have respective type $\overline{K}$ and $\widehat{K}$, and then of the typing of the lambdas and of the dependent pair.

Lastly, the case $s = \texttt{Prop}$ is only one simplification of what precede, because $\texttt{mk\_}s$ has then only one argument.

(I-Cons) $\dfrac{\mathcal{WF}(\Gamma) \quad \texttt{Ind}_n(\Gamma_I := \Gamma_C) \in \Gamma \quad (C : T) \in \Gamma_C}{\Gamma \vdash C : T}$
$\Rightarrow$

$$\frac{\mathcal{WF}([\![\Gamma]\!]) \quad \texttt{Ind}_n(\Gamma_{\overline{I},\widehat{I}} := \Gamma_{\overline{C},\widehat{C}}) \in [\![\Gamma]\!] \quad (\overline{C} : [\![T]\!]_{\mathbb{1}}) \in \Gamma_C}{[\![\Gamma]\!] \vdash \overline{C} : [\![T]\!]_{\mathbb{1}}}$$

(I-WF) Concerning the declaration of an inductive I, of arity $K = \forall \overrightarrow{u : U},\ s_I$ and constructors $C_i : T_i$, if we pose $\Gamma_I = (I : K)$ and $\Gamma_C = (C_1 : T_1); \dots; (C_k : T_k)$, we then originally have :

$$\frac{\Gamma \vdash K : s \quad \Gamma; \Gamma_I \vdash T_i : s_I \quad \mathcal{I}_n(\Gamma_I, \Gamma_C)}{\mathcal{WF}(\Gamma; \texttt{Ind}_n(\Gamma_I := \Gamma_C))}$$

Let us first examine the definition of $\overline{I}$. Its arity is $\forall \overrightarrow{u : [\![U]\!]_{\mathbb{1}}},\ s_I$. However one has :

$[\![\Gamma]\!] \vdash [\![\forall \overrightarrow{u : U},\ s_I]\!] : [\![s]\!]_{\mathbb{1}}$	induction hypothesis for K
$[\![\Gamma]\!] \vdash \forall \overrightarrow{u : [\![U]\!]_{\mathbb{1}}},\ s_I^+ : s$	via the first projection, since $[\![s]\!]_{\mathbb{1}} = s^+$
$[\![\Gamma]\!] \vdash \forall \overrightarrow{u : [\![U]\!]_{\mathbb{1}}},\ s_I : s$	since s_I^+ has a larger type than s_I.

And concerning the new types $[\![T_i]\!]_{\mathbb{1}}$ of the constructors $\overline{C}_i$, we have:

$[\![\Gamma]\!]; [\![\Gamma_I]\!] \vdash [\![T_i]\!] : [\![s_I]\!]_{\mathbb{1}}$	induction hypothesis for T_i
$[\![\Gamma]\!]; [\![\Gamma_I]\!] \vdash [\![T_i]\!]_{\mathbb{1}} : s_I$	via the first projection

In fact, $[\![\Gamma_I]\!] = (I : \forall \overrightarrow{u : [\![U]\!]_{\mathbb{1}}},\ s_I^+)$ and not $(\overline{I} : \forall \overrightarrow{u : [\![U]\!]_{\mathbb{1}}},\ s_I)$. But I will appear in $[\![T_i]\!]_{\mathbb{1}}$ only behind the first projection, in a form like $(I\ \overrightarrow{w}).\mathbb{1}$. One can then change these occurrences into $(\overline{I}\ \overrightarrow{w})$ and replace $[\![\Gamma_I]\!]$ by $(\overline{I} : \forall \overrightarrow{u : [\![U]\!]_{\mathbb{1}}},\ s_I)$.
Finally the side conditions $\mathcal{I}_n$ are indeed satisfied for $\overline{I}$. In particular the $[\![T_i]\!]_{\mathbb{1}}$ are always types of constructors for sorts s_I. As for the condition of positivity, without going into to much details, the intuition is that $[\![.]\!]_{\mathbb{1}}$ preserve the structure of the T_i, and in particular positivity.
Let us pass now to the definition of $\widehat{Q}$. Its arity is $\forall \overrightarrow{u : [\![U]\!]_{\mathbb{1}}},\ (\overline{I}\ \overrightarrow{u}) \to \Lambda \to \texttt{Prop}$, which is indeed typable of type `Type`. As for constructors $\widehat{C}$, the chosen definition ensures that they are types of constructors for $\widehat{I}$ of sort `Prop`. And one more time, we will not detail the positivity verification, but here again it does not seem to pose any problem.

(Case) Let us now look at the case of a matching on an object of inductive type I, whose arity is $K = \forall \overrightarrow{p : P},\ K'$ with $K' = \forall \overrightarrow{u : U},\ s$ and whose constructors are $C_i : T_i$. We note $T_i = \forall \overrightarrow{p : P},\ \forall \overrightarrow{v : V},\ (I\ \overrightarrow{p}\ \overrightarrow{w})$ the types of constructors C_i. Let finally σ be the substitution of formal parameters $\overrightarrow{p}$ by concrete parameters q.

$$\frac{\Gamma \vdash e : I\ \overrightarrow{q}\ \overrightarrow{u} \quad \Gamma \vdash P : B \quad \mathcal{C}(I\ \overrightarrow{q} : K'_\sigma; B) \quad \forall i, \Gamma \vdash f_i : \forall \overrightarrow{v : V_\sigma},\ P\ \overrightarrow{w_\sigma}\ (C_i\ \overrightarrow{q}\ \overrightarrow{v})}{\Gamma \vdash \texttt{case}(e,\ P,\ f_1 \ldots f_k) : P\ \overrightarrow{u}\ e}$$

$\Rightarrow$

$$\frac{\begin{array}{c}[\![\Gamma]\!] \vdash [\![e]\!] : \overline{I}\ \overrightarrow{[\![q]\!]}\ \overrightarrow{[\![u]\!]} \qquad [\![\Gamma]\!] \vdash [\![P]\!] : [\![B]\!]_{\mathbb{1}} \qquad \mathcal{C}(\overline{I}\ \overrightarrow{[\![q]\!]} : \forall \overrightarrow{u : [\![U_\sigma]\!]_{\mathbb{1}}},\ s;\ \overline{B}) \\ \forall i, [\![\Gamma]\!] \vdash [\![f_i]\!] : \forall \overrightarrow{v : [\![V_\sigma]\!]_{\mathbb{1}}},\ [\![P\ \overrightarrow{w_\sigma}\ (C_i\ \overrightarrow{q}\ \overrightarrow{v})]\!]_{\mathbb{1}}\end{array}}{[\![\Gamma]\!] \vdash \texttt{case}([\![e]\!],\ \overline{P},\ [\![f_1]\!] \ldots [\![f_k]\!]) : [\![P\ \overrightarrow{u}\ e]\!]_{\mathbb{1}}}$$

The transformed version above is not directly a legal application of the rule (Case): there are some adjustments to be made. First of all let us study the predicate P, which form is $\lambda \overrightarrow{u : U},\ \lambda x : (I\ \overrightarrow{q}\ \overrightarrow{u}),\ T$. Its type B is of the form $\forall \overrightarrow{u : U},\ (I\ \overrightarrow{q}\ \overrightarrow{u}) \to s_P$. Thus $[\![B]\!]_{\mathbb{1}} = \forall \overrightarrow{u : [\![U]\!]_{\mathbb{1}}},\ (\overline{I}\ \overrightarrow{[\![q]\!]}\ \overrightarrow{[\![u]\!]}) \to s_P^+$. And $[\![P]\!] = \lambda \overrightarrow{u : [\![U]\!]_{\mathbb{1}}},\ \lambda x : (\overline{I}\ \overrightarrow{[\![q]\!]}\ \overrightarrow{[\![u]\!]}),\ [\![T]\!]$. This implies that our new predicate $\overline{P} = \lambda \overrightarrow{u : [\![U]\!]_{\mathbb{1}}},\ \lambda x : (\overline{I}\ \overrightarrow{[\![q]\!]}\ \overrightarrow{u}),\ [\![T]\!]_{\mathbb{1}}$ is well typed, of type $\forall \overrightarrow{u : [\![U]\!]_{\mathbb{1}}},\ (\overline{I}\ \overrightarrow{[\![q]\!]}\ \overrightarrow{[\![u]\!]}) \to s_P$. Let us note $\overline{B}$ this last type. Since $\forall \overrightarrow{u : [\![U_\sigma]\!]_{\mathbb{1}}},\ s$ and $\overline{B}$ are arities on the same sort than their original versions, the condition $\mathcal{C}$ is always checked afterward transformation. Finally the following equalities hold:

$$[\![P\ \overrightarrow{u}\ e]\!]_{\mathbb{1}} = ([\![P]\!]\ \overrightarrow{[\![u]\!]}\ [\![e]\!]).\mathbb{1} = (\overline{P}\ \overrightarrow{[\![u]\!]}\ [\![e]\!])$$

and similarly:

$$[\![P\ \overrightarrow{w_\sigma}\ (C_i\ \overrightarrow{q}\ \overrightarrow{v})]\!]_{\mathbb{1}} = (\overline{P}\ \overrightarrow{[\![w_\sigma]\!]}\ (\overline{C}_i\ \overrightarrow{[\![q]\!]}\ \overrightarrow{[\![v]\!]}))$$

This now allows a legal application of the rule (Case), modulo some permutations between substitution $[\![.]\!]$ and σ.

$$\text{(Fix)}\ \frac{\Gamma \vdash T : s \quad \Gamma;(x:T) \vdash t : T \quad \mathcal{F}(x,T,k,t)}{\Gamma \vdash (\texttt{fix}\ x/k : T := t)\ :\ T}$$

$\Rightarrow$

$$\frac{\dfrac{[\![\Gamma]\!] \vdash [\![T]\!] : [\![s]\!]_{\mathbb{1}}}{[\![\Gamma]\!] \vdash [\![T]\!]_{\mathbb{1}} : s} \quad [\![\Gamma]\!];(x:[\![T]\!]_{\mathbb{1}}) \vdash [\![t]\!] : [\![T]\!]_{\mathbb{1}} \quad \mathcal{F}(x,[\![T]\!]_{\mathbb{1}},k,[\![t]\!])}{[\![\Gamma]\!] \vdash (\texttt{fix}\ x/k : [\![T]\!]_{\mathbb{1}} := [\![t]\!])\ :\ [\![T]\!]_{\mathbb{1}}}$$

Concerning side conditions $\mathcal{F}(x,[\![T]\!]_{\mathbb{1}},K,[\![t]\!])$:

- the argument number awaited by $[\![t]\!]$ is the same one as the one of t, and the k-th argument of $[\![t]\!]$ still has an inductive type, which is now $\overline{I}$ instead of I.
- For each recursive call $(x\ \overrightarrow{u})$ in t there exists a correspond one (or several) recursive call(s) $(x\ \overrightarrow{[\![u]\!]})$ in $[\![t]\!]$. The important point is that the "guard" argument u_k undergoes only the transformation $[\![.]\!]$ to become $[\![u_k]\!]$, which does not change the fact that it is structurally smaller than the initial inductive argument, since $[\![.]\!]$ preserve the structure of t.

□

2.4.8 Correctness of $\mathcal{E}$ with respect to the transformation $[\![.]\!]$

Let us start with two auxiliary results, which confirm that the elimination cases of the extraction are quite valid with respect to $[\![.]\!]$.

Lemma 13 *For any* Cɪᴄ$_m$ *arity* K*, well-typed in a context* Γ*, one can prove* $\forall x : [\![K]\!]_{\mathbb{1}},\ \forall x' : \Lambda,\ [\![K]\!]_{2}\ x\ x'$ *in the context* $[\![\Gamma]\!]$*.*

Pʀᴏᴏꜰ. This is shown by induction on the structure of K:

- If $K = s$, then $[\![K]\!]_2$ is directly $\lambda x : [\![K]\!]_{\mathbb{1}},\ \lambda x' : \Lambda,\ \texttt{True}$. The term $\lambda\_,\ \lambda\_,\ I$ is then appropriate as proof of the required property.
- If now K is a product $\forall x : T,\ K'$, we have:

  $$[\![K]\!]_2 = \lambda t : [\![K]\!]_{\mathbb{1}},\ \lambda p : \Lambda,\ \forall x : [\![T]\!]_{\mathbb{1}},\ \forall x' : \Lambda,\ [\![T]\!]_2\ x\ x' \to [\![K']\!]_2\ (t\ x)\ (p\ x')$$

  However the induction hypothesis for K' gives us a term H_r whose type is $\forall y : [\![K']\!]_{\mathbb{1}},\ \forall y' : \Lambda,\ [\![K']\!]_2\ y\ y'$ in the context $[\![\Gamma]\!];(x : [\![T]\!]_{\mathbb{1}})$. The following term is then appropriate:

  $$\lambda z : [\![K]\!]_{\mathbb{1}},\ \lambda z' : \Lambda,\ \lambda x : [\![T]\!]_{\mathbb{1}},\ \lambda x' : \Lambda,\ \lambda\_,\ H_r\{x \leftarrow x\}\ (z\ x)\ (z'\ x')$$

$\square$

Lemma 14 *For any* $\textsc{Cic}_m$ *type* T *admitting* `Prop` *like sort in a context* Γ*, one can prove* $\forall x : [\![T]\!]_{\mathbb{1}},\ \forall x' : \Lambda,\ [\![T]\!]_{\mathbb{2}}\ x\ x'$ *in the context* $[\![\Gamma]\!]$.

PROOF. According to theorem 8, $[\![T]\!]$ has type $\texttt{Prop}^+$. Considering the definition of $\texttt{Prop}^+$, the predicate $[\![T]\!]_{\mathbb{2}}$ is thus $\lambda\_,\ \lambda\_,\ \texttt{True}$. $\square$

We now will state and prove the principal result of this semantic study: the extraction $\mathcal{E}$ is correct with respect to the simulation predicates obtained via the transformation $[\![.]\!]$:

Theorem 9 *For any well-typed* $\textsc{Cic}_m$ *object* t*, if* $\Gamma \vdash t : T$*, then* $([\![T]\!]_{\mathbb{2}}\ [\![t]\!]\ \mathcal{E}(t))$ *is provable in the context* $[\![\Gamma]\!]_+$.

PROOF. First of all, the basic cases, which are arities and logical parts, are treated using the two previous lemmas. The remainder of the proof is done by induction over the typing derivation $\Gamma \vdash t : T$.

(Ax) In this rule that allow the typing of sorts, T is a sort, and thus a fortiori an arity. We thus use lemma 13.

(Prod) Here T is again a sort, hence the use of lemma 13.

(Var) $$\frac{\mathcal{WF}(\Gamma) \quad (x : T) \in \Gamma}{\Gamma \vdash x : T}$$

If T admits `Prop` as type or is an arity, then $\mathcal{E}(x) = \square$ and we use lemma 14 or lemma 13. If not, $\mathcal{E}(x)$ is the program variable associated with x in the context $[\![\Gamma]\!]_+$, namely x'. And this context also contains a proof H_x of type $([\![T]\!]_{\mathbb{2}}\ x\ x')$, which gives us the claimed $([\![T]\!]_{\mathbb{1}}\ [\![x]\!]\ \mathcal{E}(x))$.

(Lam) $$\frac{\Gamma \vdash \forall x : T_0,\ T' : S \quad \Gamma; (x : T_0) \vdash t' : T'}{\Gamma \vdash \lambda x : T_0,\ t' : \forall x : T_0,\ T'}$$

If $T = \forall x : T_0,\ T'$ is an arity or has type `Prop`, we apply one of the previous lemmas. If not, the induction hypothesis for t' states that:

$$[\![\Gamma]\!]_+; (x : [\![T_0]\!]_{\mathbb{1}}); (x' : \Lambda); (H_x : [\![T_0]\!]_{\mathbb{2}}\ x\ x') \vdash H_R : [\![T']\!]_{\mathbb{2}}\ [\![t']\!]\ \mathcal{E}(t')$$

However we have:

$$[\![T]\!]_{\mathbb{2}} = \lambda z : [\![T]\!]_{\mathbb{1}},\ \lambda z' : \Lambda,\ \forall x : [\![T_0]\!]_{\mathbb{1}},\ \forall x' : \Lambda,\ \forall H_x : [\![T_0]\!]_{\mathbb{2}}\ x\ x',\ [\![T']\!]_{\mathbb{2}}\ (z\ x)\ (z'\ x')$$

The desired property is then:

$$\forall x : [\![T_0]\!]_{\mathbb{1}},\ \forall x' : \Lambda,\ \forall H_x : [\![T_0]\!]_{\mathbb{2}}\ x\ x',\ [\![T']\!]_{\mathbb{2}}\ ((\lambda x : [\![T_0]\!]_{\mathbb{1}},\ [\![t']\!])\ x)\ ((\lambda x' : \square,\ \mathcal{E}(t'))\ x')$$

And this can be simplified using $(\lambda x : [\![T_0]\!]_{\mathbb{1}},\ [\![t']\!])\ x = [\![t']\!]$ and $(\lambda x' : \square,\ \mathcal{E}(t'))\ x' = \mathcal{E}(t')$. Finally, the following term is a correct proof of the wanted property:

$$\lambda x : [\![T_0]\!]_{\mathbb{1}},\ \lambda x' : \Lambda,\ \lambda H_x : [\![T_0]\!]_{\mathbb{2}}\ x\ x',\ H_r\{x, x', H_x \leftarrow x, x', H_x\}$$

(App) $$\frac{\Gamma \vdash t' : \forall x : T_0,\ T' \quad \Gamma \vdash t_0 : T_0}{\Gamma \vdash (t'\ t_0) : T'\{x \leftarrow t_0\}}$$

If $T'\{x \leftarrow t_0\}$ is an arity or has type `Prop`, one applies one of the previous lemmas. If not, one will use the two induction hypothesis:

$$[\![\Gamma]\!]_+ \vdash H^1_r : \forall x : [\![T_0]\!]_1, \forall x' : \Lambda, [\![T_0]\!]_2\ x\ x' \to [\![T']\!]_2\ ([\![t']\!]\ x)\ (\mathcal{E}(t')\ x')$$

$$[\![\Gamma]\!]_+ \vdash H^2_r : [\![T_0]\!]_2\ [\![t_0]\!]\ \mathcal{E}(t_0)$$

However the desired property is:

$$[\![T'\{x \leftarrow t_0\}]\!]_2\ [\![t'\ t_0]\!]\ \mathcal{E}(t'\ t_0) = [\![T']\!]_2\{x \leftarrow [\![t_0]\!]_1\}\ ([\![t']\!]\ [\![t_0]\!])\ (\mathcal{E}(t')\ \mathcal{E}(t_0))$$

It is then enough to take as proof:

$$H^1_r\ [\![t_0]\!]\ \mathcal{E}(t_0)\ H^2_r$$

(Conv) $$\frac{\Gamma \vdash T : S \quad \Gamma \vdash T : T' \quad T' =_{\beta\iota} T}{\Gamma \vdash T : T}$$

We have $[\![T']\!] =_{\beta\iota} [\![T]\!]$, and in particular the second projections are convertible. It is thus enough to use exactly the proof term coming from the induction hypothesis for $t : T'$.

(CumT) $$\frac{\Gamma \vdash t : \mathtt{Type}_i \quad i < j}{\Gamma \vdash t : \mathtt{Type}_j}$$

$\mathtt{Type}_j$ is a sort thus a fortiori an arity. We use lemma 13.

(CumP) $$\frac{\Gamma \vdash t : \mathtt{Prop}}{\Gamma \vdash t^\dagger : \mathtt{Type}}$$

`Prop` is a sort thus a fortiori an arity. We use lemma 13.

(CumS) $$\frac{\Gamma \vdash t : \mathtt{Set}}{\Gamma \vdash t^\ddagger : \mathtt{Type}}$$

`Set` is a sort thus a fortiori an arity. We use lemma 13.

(I-Type) An inductive type necessarily has an arity as type. We use lemma 13.

(I-Cons) $$\frac{\mathcal{WF}(\Gamma) \quad \mathtt{Ind}_n(\Gamma_I := \Gamma_C) \in \Gamma \quad (C : T) \in \Gamma_C}{\Gamma \vdash C : T}$$

If the constructor belongs to a logical inductive type, the lemma 14 is used. If not, an unfolding of $([\![T]\!]_2\ [\![C]\!]\ \mathcal{E}(C))$ show that it is exactly the type of the constructor $\widehat{C}$ of the inductive $\widehat{I}$. It is then enough to take this $\widehat{C}$ as proof term.

(Case) Let us now look at the case of a pattern matching on an object of inductive type I whose arity is $K = \forall \overrightarrow{p : P}, K'$ with $K' = \forall \overrightarrow{u : U}$, s and whose constructors are $C_i : T_i$. We note $T_i = \forall \overrightarrow{p : P}, \forall \overrightarrow{v_i : V_i}, (I\ \overrightarrow{p}\ \overrightarrow{w_i})$ the types of the constructors C_i. Let finally σ be the substitution of formal parameters $\overrightarrow{p}$ by concrete parameters q.

$$\frac{\Gamma \vdash e : I\ \overrightarrow{q}\ \overrightarrow{u} \quad \Gamma \vdash P : B \quad \mathcal{C}(I\ \overrightarrow{q} : K'_\sigma; B) \quad \forall i, \Gamma \vdash f_i : \forall \overrightarrow{v_i : V_{i\sigma}},\ P\ \overrightarrow{w_{i\sigma}}\ (C_i\ \overrightarrow{q}\ \overrightarrow{v_i})}{\Gamma \vdash \mathtt{case}(e,\ P,\ f_1 \ldots f_k) : P\ \overrightarrow{u}\ e}$$

First of all, if $T = (P\ \overrightarrow{u}\ e)$ admits `Prop` as type, or is an arity, we use one of the previous lemmas. Let us suppose now that this matching is informative. The induction hypothesis give us:

$[\![\Gamma]\!]_+ \vdash H_r : [\![I]\!]_2\ \overrightarrow{[\![q]\!]}\ \overrightarrow{[\![u]\!]}\ [\![e]\!]\ \mathcal{E}(e)$ with $[\![I]\!]_2 = \overline{I}$ if $s \neq$ `Prop` or $\lambda\overrightarrow{\_}$, `True` otherwise

$[\![\Gamma]\!]_+ \vdash H_r^i : [\![\forall \overrightarrow{v_i : V_{i\sigma}},\ P\ \overrightarrow{w_{i\sigma}}\ (C_i\ \overrightarrow{q}\ \overrightarrow{v_i})]\!]_2\ \ [\![f_i]\!]\ \ \mathcal{E}(f_i)$ for all i

Said otherwise, with the notations of page 65:

$[\![\Gamma]\!]_+ \vdash H_r^i : \forall \overrightarrow{[\![v_i : V_{i\sigma}]\!]},\ ([\![P]\!]\ \overrightarrow{[\![w_{i\sigma}]\!]}\ (\overline{C}_i\ \overrightarrow{[\![q]\!]}\ \overrightarrow{v_i})).2\ \ ([\![f_i]\!]\ \overrightarrow{v})\ \ (\mathcal{E}(f_i)\ \overrightarrow{v_i}^\Lambda)$

On the other hand, one wishes to prove:

$([\![P]\!]\ \overrightarrow{[\![u]\!]}\ [\![e]\!]).2\ \ \mathtt{case}([\![e]\!],\ \overline{P},\ \overrightarrow{[\![f_i]\!]})\ \ \mathtt{case}(\mathcal{E}(e),\ \square,\ \overrightarrow{\mathcal{E}(f_i)})$

The head e of this matching is then in one of the three following cases:

- Its inductive type I is an logical empty inductive. We are then in a situation similar to a proof of `False` : we can just eliminate this proof e via a pattern matching to be able to prove anything. And the post-transformation version $[\![e]\!]$ has still an empty inductive type, which is now $\overline{I}$. The proof will be thus of the form $\mathtt{case}([\![e]\!],\ \ldots,\ )$ with the good head predicate.

- Its type inductive I is a logical singleton inductive. There is thus only one constructor C_1 for I, and all the non-parametric arguments of C_1 are logical. Let n be the number of such arguments. The expression $\mathtt{case}(\mathcal{E}(e),\ \square,\ \overrightarrow{\mathcal{E}(f_1)})$ is in fact $\mathtt{case}_n(\square,\ \square,\ \mathcal{E}(f_1))$, which can be reduced to $(\mathcal{E}(f_1)\ \overrightarrow{\square})$.

  We then define a new predicate $\widehat{P}$ whose body is:

  $\lambda \overrightarrow{u : [\![U]\!]},\ \lambda x : \overline{I}\ \overrightarrow{[\![q]\!]}\ \overrightarrow{u},\ ([\![P]\!]\ \overrightarrow{u}\ x).2\ \ \mathtt{case}(x, \overline{P},\ \overrightarrow{[\![f_i]\!]})\ \ (\mathcal{E}(f_i)\ \square\ \ldots\ \square)$

  The proof to be built then starts with a matching on $[\![e]\!]$ according to this predicate $\widehat{P}$. This matching have then only one branch, whose type must be:

  $\forall \overrightarrow{v_1 : [\![V_1]\!]_{[\![\sigma]\!]}},\ \widehat{P}\ \overrightarrow{[\![w_1]\!]_{[\![\sigma]\!]}}\ (\overline{C}_1\ \overrightarrow{[\![q]\!]}\ \overrightarrow{v_1}) =$

  $\forall \overrightarrow{v_1 : [\![V_{1\sigma}]\!]},\ ([\![P]\!]\ \overrightarrow{[\![w_{1\sigma}]\!]}\ (\overline{C}_1\ \overrightarrow{[\![q]\!]}\ \overrightarrow{v_1})).2\ \ ([\![f_1]\!]\ \overrightarrow{v_1})\ \ (\mathcal{E}(f_1)\ \overrightarrow{\square})$

  modulo a ι-reduction and some permutations between $[\![.]\!]$ and σ. Now, for each variable v_{1j} of the sequence $\overrightarrow{v_1}$, one knows that it is a logical variable. By using repeatedly the lemma 14, one builds for each j a term H_j of type $([\![V_{1j\sigma}]\!]_2\ v_{1j}\ \square)$. The required branch of the matching is then:

  $\lambda \overrightarrow{v_1 : [\![V_{1\sigma}]\!]},\ (H_r^1\ v_{11}\ \square\ H_1\ \ldots\ v_{1n}\ \square\ H_n)$

- Its inductive type I is informative. There again, we proceed by pattern matching on $[\![e]\!]$, but we need to destruct H_r during matching. One thus starts with a generalization (`Generalize` in `Coq`) with respect to the type of H_r. On the level of the final proof term, this corresponds to an application to H_r. And the head of this application now will be a pattern matching on $[\![e]\!]$ according to following predicate $\widehat{P}$:

$\lambda \overrightarrow{u : [\![U]\!]}, \lambda x : \overline{I}\ \overrightarrow{[\![q]\!]}\ \overrightarrow{u},\ \widehat{I}\ \overrightarrow{[\![q]\!]}\ \overrightarrow{u}\ x\ \mathcal{E}(e) \rightarrow$

$([\![P]\!]\ \overrightarrow{u}\ x).2\quad \texttt{case}(x, \overline{P}, \overrightarrow{[\![f_i]\!]})\quad \texttt{case}(\mathcal{E}(e), \Box, \overrightarrow{\mathcal{E}(f_i)})$

The j-th branch of this matching must then have for type:

$\forall \overrightarrow{v_j : [\![V_j]\!]_{[\![\sigma]\!]}}, \widehat{P}\ \overrightarrow{[\![w_j]\!]_{[\![\sigma]\!]}}\ (\overline{C}_j\ \overrightarrow{[\![q]\!]}\ \overrightarrow{v_j}) =$

$\forall \overrightarrow{v_j : [\![V_{j\sigma}]\!]}, \widehat{I}\ \overrightarrow{[\![q]\!]}\ \overrightarrow{[\![w_{j\sigma}]\!]}\ (\overline{C}_j\ \overrightarrow{[\![q]\!]}\ \overrightarrow{v_j})\ \mathcal{E}(e) \rightarrow$

$([\![P]\!]\ \overrightarrow{[\![w_{j\sigma}]\!]}\ (\overline{C}_j\ \overrightarrow{[\![q]\!]}\ \overrightarrow{v_j})).2\quad ([\![f_i]\!]\ \overrightarrow{v_i})\quad (\texttt{case}(\mathcal{E}(e), \Box, \overrightarrow{\mathcal{E}(f_j)}))$

This matching branch is then built as follows:

1. We starts by introducing the variables $\overrightarrow{v_j}$.
2. We introduce the specialized version of H_r, where $[\![e]\!]$ has become $(\overline{C}_j\ \overrightarrow{[\![q]\!]}\ \overrightarrow{v_j})$.
3. It is then possible to inverse this new H_r (`Inversion` in `Coq`). And according to the form of its type, we then obtain the existence of variables v'_{jk} associated to each variable v_{jk} of $\overrightarrow{v_j}$, as well as proof H_k of type $([\![V_{jk\sigma}]\!]_2\ v_{jk}\ v'_{jk})$, and finally we also obtain the fact that $\mathcal{E}(e)$ is equal to $(C^j_\Lambda\ \overrightarrow{v_j}^\Lambda)$.
4. We can then rewrite $\mathcal{E}(e)$ in the current goal, in which the part concerning $\mathcal{E}$ becomes convertible then with $(\mathcal{E}(f_j)\ \overrightarrow{v_j}^\Lambda)$.
5. The only remaining step is to apply H^j_r to the good arguments v_{jk}, v'_{jk} and H_k.

$$\text{(Fix)}\quad \frac{\Gamma \vdash T : s \quad \Gamma; (f : T) \vdash t : T \quad \mathcal{F}(f, T, k, t)}{\Gamma \vdash (\texttt{fix}\ f/k : T := t)\ :\ T}$$

If T is of type `Prop` or is an arity, we use one of the previous lemmas. If not, we use the induction hypothesis:

$[\![\Gamma]\!]_+; (f : [\![T]\!]_1; (f' : \Lambda); (H_f : [\![T]\!]_2\ f\ f') \vdash H_r : [\![T]\!]_2\ [\![t]\!]\ \mathcal{E}(t)$

However one wishes to establish the following statement:

$[\![T]\!]_2\ \ (\texttt{fix}\ f : [\![T]\!]_1 := [\![t]\!])\ \ (\texttt{fix}\ f' : \Box := \mathcal{E}(t))$

Thereafter, we note $\overline{\texttt{fix}}$ and $\texttt{fix}_\Lambda$ the respective fixpoints $(\texttt{fix}\ f : [\![T]\!]_1 := [\![t]\!])$ and $(\texttt{fix}\ f' : \Box := \mathcal{E}(t))$. The idea is then to make a proof by fixpoint, something like:

$\texttt{fix}\ F : [\![T]\!]_2\ \overline{\texttt{fix}}\ \texttt{fix}_\Lambda := H_r\{f \leftarrow \overline{\texttt{fix}}\}\{f' \leftarrow \texttt{fix}_\Lambda\}\{H_f \leftarrow F\}$

Unfortunately, this term is slightly inaccurate. The type of H_r after substitution is indeed $([\![T]\!]_2\ [\![t]\!]\{f \leftarrow \overline{\texttt{fix}}\}\ \mathcal{E}(t)\{f' \leftarrow \texttt{fix}_\Lambda\})$ instead of $([\![T]\!]_2\ \overline{\texttt{fix}}\ \texttt{fix}_\Lambda)$. However unfolded versions of fixpoint $\overline{\texttt{fix}}$ and $\texttt{fix}_\Lambda$ are convertible with their initial versions only when a guard argument is present and starts with a constructor. It is in fact possible to correct these inaccuracies. To simplify the writing, we will proceed only in the situation where the guard argument is in the first position. But of course, this method extends to the general case. We thus suppose that T can be written $\forall x : I\ \overrightarrow{u}, T'$. The type ($[\![T]\!]_2\ [\![t]\!]\ \mathcal{E}(t)$) of H_r rewrites to:

$$\forall x : \overline{I}\ \overrightarrow{[\![u]\!]}, \forall x' : \Lambda, \forall H_x : \widehat{I}\ \overrightarrow{[\![u]\!]}\ x\ x',\ \ [\![T']\!]_2\ ([\![t]\!]\ x)\ (\mathcal{E}(t)\ x')$$

However one can prove that:

$$\forall x : \overline{I}\ \overrightarrow{[\![u]\!]},\ \ [\![t]\!]\{f \leftarrow \overline{\texttt{fix}}\}\ x\ \ =\ \ \overline{\texttt{fix}}\ x$$

To establish that, we can just reason by case over x, and in each case x is then replaced by a constructor, which implies that the equality becomes trivial thanks to a ι-conversion. And we can also prove similarly that:

$$\forall x : \overline{I}\ \overrightarrow{[\![u]\!]}, \forall x' : \Lambda, \forall H_x : \widehat{I}\ \overrightarrow{[\![u]\!]}\ x\ x',\ \ \mathcal{E}(t)\{f' \leftarrow \texttt{fix}_\Lambda\}\ x'\ \ =\ \ \texttt{fix}_\Lambda\ x'$$

There again one proceeds by matching on x. Then, in order to deduce the form of x', we "inverse" H_x as in the previous sub-case for (case). This shows us that x' starts with one constructor, and there again the required equality becomes trivial after a ι-conversion. The next step is to use these two equalities to rewrite the type of the substituted version of H_r, so that its type becomes indeed ($[\![T]\!]_2\ \overline{\texttt{fix}}\ \texttt{fix}_\Lambda$).

Finally, it should be checked that this fixpoint proof that we have just built is indeed legal: are its recursive calls structurally decreasing? It is a matter of locating the uses of H_f in H_r. However the occurrences of f in the t term initial are applied to decreasing arguments. And the construction of H_r respects the structure of t: a `case` produces a `case`, one application generates an application, etc. To each application ($f\ y$) in t will correspond an application ($H_f\ [\![y]\!]\ \mathcal{E}(y)\ \ldots$) in H_r, which relies ($f\ [\![y]\!]$) and ($f'\ \mathcal{E}(y)$). And if y was structurally smaller than the initial inductive argument, it is indeed the same for $[\![y]\!]$.

□

2.5 Summary of the correctness results

At the end of this double study of correctness, it is time to make an assessment of the properties of correctness which we have proved. In fact, the syntactic part (section 2.3) and the semantic part (section 2.4) are quite complementary.

- the syntactic study allows to establish the termination without anomaly of any weak reduction of a closed extracted term. On the other hand if the extracted term is a function, this is not very instructing. This part is a relatively simple methodology, obtained early during this thesis works.
- Such as we formulated it, our semantic analysis does not allow to prove by itself the termination of our extracted programs. Perhaps that would be it possible by adding termination conditions among the definitions of $[\![]\!]$, but we do not have pushed further in this direction for lack of time. Indeed, this semantic analysis has required an important quantity of efforts to be achieved. In particular, issues like cumulativity of Cic have been quite hard to deal with, and that has only be done in the last months of this thesis. In any case, this analysis allows to give a meaning to extracted functions, and

affirms that they preserve the properties of the initial Coq functions. This analysis also opens the door to one study of the extraction in the presence of axioms, something excluded by our first study.

2.6 Toward a more realistic extraction

We have seen before that the reduction r_w studied in section 2.3.2 corresponded to the respective reduction strategies of Haskell and Ocaml, depending on whether one reduced initially the head or the argument of the applications.

There remain nevertheless four substantial differences between our theoretical model and the reductions actually used by Haskell and Ocaml :

1. empty inductive types and their eliminations
2. the ad-hoc rule $\rightarrow_i$ of elimination of logical singleton inductive types
3. the ad-hoc rule $\rightarrow_\Box$ eliminating the arguments of constants $\Box$
4. the rule $\rightarrow_i$ concerning the fixpoints and their concept of "guard"

2.6.1 The empty inductive types

Our target languages do not authorize the definition of empty inductive types, and a fortiori the matching over such an inductive. But in fact, if one handles an object having an empty inductive type, that means that we are in some impossible situation. A matching on such an object is in particular a portion of code that will never be executed.

More precisely, let us suppose that the evaluation of an extracted object meet a term of the form $\texttt{case}(e,\, P,\, \varnothing)$. The theorem 2 enables us to obtain a corresponding well-typed CIC term $\texttt{case}(e_0,\, P_0,\, \varnothing)$. In particular e_0 is an object having an empty inductive has type. We can then continue with a reasoning similar to the one used during the proof of theorem 2: since these reductions are done in a weak manner, *i.e.* in an empty context and outside all lambdas, the inductive object e_0 is thus closed. It can then to be reduced to a term having a constructor as head. This is obviously impossible, because an empty inductive has by definition no constructor.

One can thus freely replace any pattern matching on an empty inductive object by arbitrary code. In practice, a natural choice is to replace this matching by the raising of an exception. This allows to underline the inaccessible character of this position of the code, while giving the most general type to this portion of code. We thus use in Ocaml the construction `assert false`, and `error` in Haskell.

One may note that this treatment of empty inductives is completely independent of the logical or informative character of this inductive type. Whether we deal with the logical inductive `False` or with its dual in `Set` named `empty`, their two eliminations `False_rec` and `empty_rec` result in the raising of an exception.

2.6.2 The elimination of logical singleton inductive types

In the study of section 2.3.2, we used one ι-reduction rule adapted to treat the elimination of logical singleton inductive types:

$$\texttt{case}_n(\Box,\ P,\ f) \rightarrow_\iota f\ \underbrace{\Box\ \ldots\ \Box}_{n}$$

The reason for such an ad-hoc rule is to allow the function $\mathcal{E}$ to disturb as less as possible the structure of the terms, which simplify its study. This reduction thus does not have anything fundamental, so much so that it is perfectly possible to integrate it into the extraction: rather than to encode this special rule in our languages targets, we can "pre-compile" during the extraction all these eliminations of logical singleton inductive, without losing correctness of extraction.

Justifying such an optimization of the extracted terms is not completely immediate. Indeed this trick implies to reduce all the eliminations of logical singleton, including those located under lambdas, whereas our correctness results of the section 2.3.2 are valid only with the use of the *weak* version of this ι-reduction.

It can seem paradoxical besides to want to carry out strong reductions after having warned against the dangers of such reductions in the examples of the beginning of section 2.3.2. But let us consider again the function `cast` of page 51, which translated into a matching on a logical singleton inductive. It is not the ι-reduction itself of the body of `(cast H 0)` to `0` that generate an error in the execution of the function `example`, but rather the following matching where `0` is seen as a boolean. And this second non-singleton matching will remain quite prohibited because it is done under a lambda.

We will note $\rightarrow_{\iota_s}$ the *strong* version of this ad-hoc ι-reduction, usable even under lambdas, contrary to the weak version $\rightarrow_{\iota_w}$. From one extracted term t, we now justify the correctness of one term t' obtained via $t \rightarrow^*_{\iota_s} t'$, using a simulation of t' by t.

Theorem 10 *Let t, t' and u' be three terms of* $\text{CIC}_\Box$ *such that $t \rightarrow^*_{\iota_s} t'$ and $t' \rightarrow_{r_w} u'$. There exists a* $\text{CIC}_\Box$ *term u such that $u \rightarrow^*_{\iota_s} u'$ and $t \rightarrow_{r_w+} u$.*

$$\begin{tikzcd}
t \arrow[r, dashed, "r_w+"] \arrow[d, "\iota_s*"'] & u \arrow[d, dashed, "i_s*"] \\
t' \arrow[r, "r_w"'] & u'
\end{tikzcd}$$

Proof. Taking into account the rule $\rightarrow_\iota$ for logical singleton inductive, the terms t and t' have extremely close structures. It is enough then to compare the position of the redex r which is reduced in t' with the corresponding position in t.

- If this position is apart from any branch of pattern matching on a logical singleton inductive, then the same reduction can be carried out in t, which gives us a suitable term u.

- Suppose now that this position in t is under one or more pattern matching(s) over logical singleton inductives. First of all let us consider the case of a single matching on the way to the redex corresponding to r in t. The reduction which one carries out in t' is weak. In t, that mean that our logical matching singleton is a fortiori located outside any binder, these binder being lambdas or matchings. One can then start by using $\to_{\iota_w}$ to reduce this matching over a logical singleton inductive, before reducing the redex corresponding to r. And in the case of multiple matchings, the argument is the same: the most external matching can be reduced in a weak way, then the second, and so on, before finishing by the redex corresponding to r.

$\square$

Theorem 11 *Let t, t' and u be three terms of* $\text{Cic}_\square$ *such that $t \to^*_{\iota_s} t'$ and $t \to_{r_w} u$. There exists a* $\text{Cic}_\square$ *term u' such that $u \to^*_{\iota_s} u'$ and verifying $t' \to_{r_w} u'$ or $t' = u'$.*

$$\begin{tikzcd}
t \arrow[r, "r_w"] \arrow[d, "\iota_s *"'] & u \arrow[d, dashed, "\iota_s *"] \\
t' \arrow[r, dashed, "r_w | \epsilon"'] & u'
\end{tikzcd}$$

Proof. First of all, if the reduction of t to u is a ι-reduction of a logical singleton term already carried out between t and t', then taking $u' = t'$ is appropriate. If not, the weak reduction between t and u is done outside of any matching, including matching on logical singleton. There is thus one exactly identical redex in t', that one can reduce for getting a suitable u'. $\square$

In addition to the reductions $\to_{r_w}$, the reductions $\to_\square$ can also be simulated between the initial extracted term t and its optimized version t'. Finally, any sequence of reductions starting from t' is finite, otherwise the theorem 10 would allow us to build an infinite sequence reductions for the extracted term t. And by combining the two previous theorems it can be shown that the weak normal forms of t and t' are related by $\to_{\iota_s}$, and in particular equal if they contain no more λ-abstractions.

During the extraction, we are indeed in right to carry out all the ι-reductions of logical singleton inductive, even strongly, on the raw extracted term generated by $\mathcal{E}$. Moreover the head of a matching over a logical singleton is inevitably `Prop` as sort, and thus is extracted by $\mathcal{E}$ into $\square$. This thus implies that all matchings over logical singletons $\texttt{case}_n$ are indeed reduced by $\to_{\iota_s}$ and disappear completely.

2.6.3 The elimination of the possible arguments of $\square$

Let us now consider the ad-hoc reduction $(\square\ x) \to_\square$. Unlike the previous ad-hoc reduction on logical singleton inductive types, it will not be possible to avoid it completely after extraction. Of course, the extraction $\mathcal{E}$ never generates itself such $(\square\ x)$ terms, since the whole initial term $(f\ x)$ is then identified as worth eliminating, and gives directly $\Box$. On the other hand, the example 1 of page 49 shows that such a subterms can appear after some reductions inside an extracted term.

Ocaml

If one evaluates à la Ocaml the extracted terms, *i.e.* with a strict strategy, it is then necessary to be sure that this $(\Box\ x)$ application will not fail. In practice that can be done by choosing carefully the concrete term which will implement $\Box$. Usually, one chooses the term `()` of type `unit` to replace an arbitrary constant like our $\Box$, but that is thus not appropriate here. One can then try try to rather use `fun _ → ()`, or `fun _ _ → ()` when it is known that $\Box$ can receive up to two arguments, and so on. Unfortunately, we have been unable to simply determine the maximum number of arguments that can receive a particular $\Box$ term. We thus have chosen a more general though less elegant solution, namely one fixpoint that absorbs its arguments. In Ocaml, that can be written as follows:

```
let rec f x = f
```

Of course, this definition is badly typed, and we will see in the next chapter how to circumvent this problem. In any case, from the point of view of the execution, a constant $\Box$ implemented this way can indeed receive an arbitrary number of arguments without failing.

Haskell

In the case of a lazy evaluation to the Haskell, one could perfectly reuse this idea of a fixpoint absorbing its arguments. But that is in fact not necessary. If one studies again the example 1, the extracted object is after reduction $\lambda g : \Box, (g\ (\Box\ \texttt{0}))$. The $(\Box\ \texttt{0})$ subterm do not appear at the head, and the function g may completely not use its argument, in which case the $(\Box\ \texttt{0})$ application will never be computed. This situation is in fact quite common, and one will in fact never need to use the rule $\to_\Box$ with lazy strategy. This lazy strategy implies that the head of terms will always be reduced first, according to the two following compatibility rules (see definition 13):

$$\frac{u \to_? v}{(u\ t) \to_? (v\ t)} \qquad \frac{u \to_? v}{\texttt{case}(u,\ P,\ \ldots) \to_? \texttt{case}(v,\ P,\ \ldots)}$$

Let us take a initial closed term t_0, well-typed in the CIC, and $t = \mathcal{E}(t_0)$. Let also u be one of the successive reduced forms of t by the lazy strategy. If a reduction $(\Box\ x) \to_\Box \Box$ can intervene in u at a position permitted by these two compatibility rules, there are two possible situations:

- the $\Box$ can be at the top of u, which is then of the form $(\Box\ \overrightarrow{a})$. According to the theorem 2, this term corresponds to a term $u_0 = (f_0\ \overrightarrow{a_0})$ of CIC, which is a reduced form of the initial term t_0. And f_0 is then either of sort `Prop` sort or is a type scheme. According to the stability lemma 1, it is the same for the whole term $u_0 = (f_0\ \overrightarrow{a_0})$. Then the lemma 2 shows that the initial term t_0 is also of sort `Prop` or is a type scheme. And thus $\mathcal{E}(t_0) = \Box$, which show that the case considered here is impossible.

- the $\square$ can be at the beginning of the head of a matching, that is u contains a subterm of the form `case`$(\square\ \overrightarrow{a}, \ldots, \ldots)$. A similar reasoning to the one of the previous case shows that the inductive term corresponding in CIC to $(\square\ \overrightarrow{a})$ is of sort `Prop`. But in the extracted term $t = \mathcal{E}(t_0)$, all `case` on logical inductive inevitably has $\square$ as head, whether this logical inductive is empty or singleton. And that cannot change under reduction, which contradicts the presence in u of the subterm of the form `case`$(\square\ \overrightarrow{a}, \ldots, \ldots)$.

The rule $\rightarrow_\square$ is thus never used in a reduction with Haskell. $\square$ can thus always be implemented by an arbitrary term.

In fact one can even go further and consider that $\square$ is an abnormal final result during the evaluation of an extracted term. Indeed, the computation of an extracted term only have interest when its result is ... informative, such as for example `true`, `3`, or `fun x⇒x`. This result can of course contain residues $\square$ of logical parts or type schemes, for example inside a function. But the whole result being $\square$ is abnormal. We thus made the choice to implement $\square$ by the raising of an exception. And that does not disturb the computations on informative parts of terms:

- We have just seen that $\square$ cannot be evaluated as the head of an application.
- $\square$ will never be evaluated either like head of a matching on a logical inductive term, since we saw in the two previous sections how to make disappear these matchings over the logical empty of singleton inductive.

2.6.4 Toward a usual reduction of fixpoints

The reduction of fixpoint is the last major difference between the reductions in our system $\text{CIC}_\square$ and those in the target languages. Indeed in these target languages there is obviously no concept of "guard" argument, that should start with a constructor for the reduction to be possible.

Ocaml

In the case of Ocaml, the difference is not so important. Indeed one fixpoint with two arguments `let rec f x y = t` will only be reduced when it receives its two arguments. And in the case of an inductive argument, Ocaml will evaluate completely this argument, whose value will thus start indeed with one constructor, before continuing the evaluation of the fixpoint. We can then just simply translate the `fix` $f\ \{f/2 : \square := \lambda x,\ \lambda y,\ t\}$ example into `let rec f x y = t in f`. And it is not abusive to suppose that the body of a fixpoint starts with enough λ-abstractions, here at least two. In fact, the concrete Coq syntax for the `Fixpoint` and the `fix` obliges us to declare the arguments at least up to the guard argument. Lastly, dealing with mutual fixpoints brings no more difficulties. There is no additional concerns about logical guard argument, since the condition "being $\square$" has been substituted to the original condition "starting by one constructor" in our new ι-reduction. And the complete evaluation of a logical argument cannot indeed finish on anything else than $\square$.

Haskell

On the other hand, for Haskell, the situation is more delicate. Unfolding of a fixpoint `f` is done when `f` comes to the head of the term to be evaluated, and this is done with no initial reduction on the arguments. The constraint on the (at least partial) evaluation of the guard argument, prior to any unfolding, has no particular reason to be fulfilled. It is thus not sure that the order of the reductions in Haskell can be reflected on the CIC level in the spirit of the theorem 2.

Let us try to informally show that everything work correctly nonetheless. The decreasing constraints that the CIC imposes imply that the following recursive call is done with a "smaller" guard argument. And to obtain such a "smaller" argument, it is necessary to destruct at least one level of the initial argument. The evaluation of the guard argument is thus simply pushed back from one recursive call to the next one.

The previous justification is unsatisfactory. First, such a modification of the evaluation order is not necessarily safe. Moreover our reasoning assumes that the analysis of the guard argument, that produces the new recursive argument, is made at the head of term, and hence that Haskell will carry out it without additional delay. Usually, it is true that the body of one fixpoint starts immediately with a `match`. But that is not a general rule, as the example of `Acc_iter` shows (see page 29). In this case, the analysis of the argument of guard is pushed back *in* the term being used as following argument. `Acc_iter` is in fact a bad example, because its guard argument is logical, and thus disappears with the extraction. But we can imagine a similar example with an informative guard argument, and this argument could indeed never be evaluated by Haskell.

We now will outline a rigorous justification of the simple translation of $\text{CIC}_\square$ fixpoints into Haskell fixpoints. For that, we show that the computation order during a reduction of a fixpoint in Haskell can be simulated in CIC using a modified version of the initial fixpoint. For that we will use a remark that C. Paulin made p. 103 of [68]: every structural fixpoint on an inductive I can be transformed into a definition by well-founded induction over an order $<_I$, that is in a structural fixpoint over the inductive `Acc` having for parameter $<_I$. The interest of such a translation is to add an argument of logical accessibility which will serve as new guard argument. This way, there is not need anymore in CIC to evaluate the old guard argument even partially before an unfolding, since the constraints are now being concentrated on the new logical argument.

We will not propose here a proof of C. Paulin's remark, already justified in [68]. Instead, we try to illustrate this mechanism on a simple example, that is the addition of Peano integers. For $I =$ `nat`, an appropriate order $<_I$ is the standard order `lt` (or `<`) over integers.

We also use the large version `le` (or $\leq$) of this order. Here come a version of `plus` in which the guard argument is not one of the two main arguments, but a third argument of logical accessibility:

```
Definition plus (n m:nat) : nat :=
  (fix plusrec (n m:nat) (a:Acc lt n) {struct a} : nat :=
     match n as n0 return n0≤n → nat with
      | 0 ⇒ fun h ⇒ m
      | (S n') ⇒ fun h ⇒ S (plusrec n' m (Acc_inv a n' h))
     end (le_refl n))
  n m (lt_wf n).
```

Let us compare with the last version of `plus` presented page 20:

- One added an additional accessibility argument `a`, that is used as decreasing argument. And for each recursive call, its new accessibility argument is obtained via the function `Acc_inv` already met page 29.
- the true fixpoint, `plusrec`, awaits now three arguments. The function `plus` is thus a encapsulation of this `plusrec` in which one provides the proof `(lt_wf n)` stating that `n` is indeed accessible as third argument.
- Lastly, the typing of the `match` is much more complex than in the initial `plusrec`, and requires special annotations, as well as an artificial abstraction on a variable `h` of type `n≤n`. And a proof of this `n≤n` is provided immediately after the `match` via the term `(le_refl N)`.

To be convinced that this new `plus` allows to simulate on the Cic level any reduction made at the Haskell level with the extraction of the initial `plus`, it is enough to compare the extractions of both `plus`. Here in particular is what $\mathcal{E}$ gives for our new `plus`[4]:

```
Definition plus (n m:□) : □ :=
  (fix plusrec (n m:□) (_:□) : □ :=
     match n with
      | 0 ⇒ fun _ ⇒ m
      | (S n') ⇒ fun _ ⇒ S (plusrec n' m □)
     end □)
  n m □.
```

One notes the presence of residues of logical parts, which differentiate this extraction from that of the `plus` initial. But these residues do not have an influence on the order of computations. For example, anonymous abstractions present in the branches of `match` receives immediately their argument □ located after the `match`. Any computation in Haskell with the initial extraction thus corresponds to a computation with this news extraction, computation that finally be simulated on the Cic level, which guarantees the correctness of the initial extraction.

Finally let us note that in the extraction of new the `plus`, all these logical, constant residues □ and anonymous abstractions, are in practice detected and removed by extraction optimizations that we describe in section 4.3. And finally the extracted term from this `plus` is *exactly* the same as that extracted from the old `plus`.

[4] We use here a syntax à la `Coq`, more readable, instead of our $\mathrm{Cic}_\square$ syntax.

Chapitre 3

Typing the extracted terms

We have built in the previous chapter an extraction from Coq terms to raw, untyped, λ-terms. We have in particular shown that the execution of these extracted terms was necessarily finite and without error. It now remains to study the final translation towards a true functional language, and that is the subject of this chapter.

As we mentioned in introduction, we wish that extracted code can be integrated into a broader development. It is thus necessary to be able at least to include/understand the signature of the extracted objects. Two choices arise then: one can generate source code for a particular language, or directly some byte-code or even assembly code associated with a readable interface. But in addition to the difficulty of generating binary code, this would lead to a " black box" solution, without possibility of a posteriori control. We preferred to produce source code, which leaves the user the possibility of reading this code, and which also allows to profit from the optimizing compilers already existing. The "open source" movement showed in particular that accessibility and readability of sources largely increase the confidence in a program.

This first choice rises a new question: which language should we use as target of the extraction? All we need is a λ-calculus with inductive types. This explains the choice of languages derived from ML, namely Ocaml and Haskell. But these languages are typed, and their typing systems a la Hindley-Milner [59, 27] are appreciably different from that of Coq. In particular one cannot express in these languages any dependent types nor universes. As already mentioned previously, the old extraction made the pragmatic choice of refusing any Coq term using the Type sort. But even this restriction was not sufficient to be certain to obtain well-typed ML terms. For example, the concept of polymorphism differs between ML and Coq. It is thus clear that a simplistic translation of λ-terms Coq into ML λ-terms can lead to non-typable terms.

We should thus adapt our terms if we want to use the standard compilers like the ones of Ocaml or Haskell, not modified for the extraction. At the same time, we would wish to remain as close as possible of a direct translation, and this for several reasons. First, within the ordinary examples of Coq terms, a large majority have indeed a ML counterpart that is correctly typed. Secondly, the need for an interface to the extracted code also militates in favor of a simple and natural translation. Lastly, any effort aiming at circumventing these typing problems by an ad-hoc encoding seems to lead to a stage of precompilation which we precisely wish to avoid. An particular encoding was tested by L. Pottier [70], but this coding may still produce ML non-typable terms.

How then can we use compilers for typed languages when extracted terms are potentially non-typable? Concerning Ocaml, we use now, just like L. Pottier, an undocumented features of this language, called `Obj.magic`. This function allows to give a generic type `'a` to any term. By using this function, one can thus locally circumvent the type checking done by Ocaml compiler. We will see in a second time how the current extraction allows to automatically generate `Obj.magic` in the extracted code. And concerning Haskell, some of the implementations of this language propose a undocumented function `unsafeCoerce` which seems equivalent to `Obj.magic`. It should then be possible in the future to extend to Haskell this automatic generation of artificially well-typed terms.

Let us mention by the way the realization of an experimental extraction towards the language Scheme. Like this functional language derived from Lisp is untyped, it seemed to us one for a moment that it could be an promising target language for the extraction, since it allow to avoid the problem of non-typable terms. Unfortunately, Scheme does not have natively any inductive types nor pattern matching over such types. It is thus necessary to encode them using macros, except in the particular case of the Bigloo implementation [75]. In addition to that, preliminary tests showed a significant difference in efficiency of extracted code, clearly in favor of Haskell/Ocaml to the detriment of Scheme. The effort in favor of this Scheme extraction thus have not been continued.

In addition, we should emphasize that it is not even sufficient to simply ensure the existence of a correct type for each extracted term if we cannot determine this type in advance. Indeed, as we want to allow an easy integration of the extracted code in broader developments, we must be able to envisage which will be the types of the extracted terms, and to produce files interfaces. Moreover, if one wants to be able to extract modules and functors of Coq (cf section 4.1), it will be necessary to be able to do such type prediction.

This study of the typing of extracted terms is organized as follows. First of all, we describe the typing errors that our extraction on the terms can produce. Then we present an automatic method for bypassing these errors, via insertion in the code of type-changing primitives, for making this code artificially typable. To guide this insertion, we first compute a "reasonable" extracted type starting from the Coq initial type of the considered object. We describe this extraction $\widehat{\mathcal{E}}$ of types. Then, we force this desired extracted type to really become the type of the extracted raw term, initially untyped :

$$t : T \Rightarrow \mathcal{E}(t) : \widehat{\mathcal{E}}(T)$$

This is done via an alternative of the algorithm $\mathcal{M}$ of type checking/inferring, modified so that each detected typing error produces an insertion of a type-changing primitive.

3.1 Analysis of the typing problems

The problems start with types declarations, that is declarations of inductive types or of type abbreviations. Coq includes indeed typing features without counterparts in Ocaml or Haskell, and in particular the following ones:

- matchings on the type level (`match`)

- fixpoint on the type level (`fix`)
- polymorphism by universal quantifications at non-prenexe positions or in the types of inductive constructors.

The rest of this section details each of these situations. In versions 6.x and earlier of Coq, these declarations of untranslatable types were refused by the extraction, with message of the kind: `Error: .... is not an ML type`.

The situation of terms is different. The use of an untranslatable type in a Coq term does not prevent, a priori, its typability in Haskell or Ocaml, since these languages do not include type annotations on the lambdas and matchings: there is so no explicit references to the untranslatable types. The old extraction then generated code without being concerned with its typability. The type inference on this extracted term could then either find a simpler type, acceptable in Haskell or Ocaml, or fail.

In fact, it should be said that the frequency of the typing errors in practical in the extracted code is low. For example, there is not a single one in the extraction of Coq standard library. And in all the contributions Coq studied from the extraction point of view (see chapter 5), only 4 contributions present some errors: `Lyon/Circuits`, `Lannion`, `Rocq/Higman`, and `Nijmegen/C-CoRN`. We will examine the three last in more details in chapters 5 and 6. This scarcity can be explained by the fact that, most of the time, the users write theirs informative functions as they would have written them in Ocaml or Haskell. Thus the dependent types are generally used only as a form of polymorphism, or to express logical properties (pre/post-conditions for example). One can also see a form of self-censorship there: the majority of these examples date back to before this renewal of the extraction, hence the use of too advanced features gave at that time the insurance of not being able to extract the development.

Let us see now more in detail several typical situations that generate typing errors during extraction.

3.1.1 The type "integer or boolean"

Let us take for example a predicate `P` that depends on a boolean, whose value is either `nat` or `bool`:

```
Definition P (b:bool) : Set := if b then nat else bool.
```

This predicate now allows to aggregate two values of different types:

```
Definition p (b:bool) : P b :=
 match b return P b with
  | true ⇒ O
  | false ⇒ true
 end.
```

The type of `p` hence depends on `b`, because `(p true)` has for type `(P true) = nat`, while `(p false)` has for type `(P false) = bool`. The extraction of the previous chapter then proposes for `p` an non-typable extracted term:

```
let p = function
  | True → O
  | False → True
```

There is obviously no equivalent neither in Ocaml nor in Haskell of one such dependent type over a boolean. A possibility is then to forget this dependence by using an approximation of `P` as being a disjoint union of `nat` and `bool`:

```
type approx_P = Nat of nat | Bool of bool
let p = function
  | True → Nat O
  | False → Bool True
```

It would then be necessary to also allow the return to `nat` or `bool` as soon as the first argument is known. That implies to locate each place in Coq where the conversion typing rule `(P true) → nat` is used. One would insert there the following unwrapping during extraction:

```
let nat_of_P = function (Nat n) → n | _ → assert false
```

And idem with `false` and `bool`. Such a transformation presents several disadvantages. First, it denatures the initial program by introducing wrappings/unwrappings which involve computations without equivalents in the original term. And secondly addition, the automatic generation of these unwrappings is far from being obvious. Indeed, the use of conversion `(P true) → nat` is transparent for Coq, and is recorded nowhere. As it thereafter will be seen, it is thus not this method which was selected to treat this example.

3.1.2 A more realistic version

It should be noted than even if the previous example is so simplified that it may look useless and unrealistic, it is nonetheless inspired by real developments. For example, let us try to formalize the semantics of an imperative mini-language à la Pascal, using a memory model. If our language has only integers as basic type, the access function to a storage cell will have a very simple type:

```
Parameter get_value : memory → address → nat.
```

with `memory` representing a state of the memory and `address` the address of a memory location. Now let us add other basic types, and references. One can write for example:

```
Inductive types : Set :=
 | Nat : types
 | Bool : types
 | Address : types → types. (* type of a reference to another type *)
```

Which type could now have our `get_value` function ? Of course, we can represent the values with a sum type:

```
Inductive values : Set :=
 | Val_nat : nat → values
 | Val_bool : bool → values
 | Val_ref : address → values.

Parameter get_value : memory → address → values.
```

The problem of such a representation is that it implies to continuously reason by case on this type `values`, even if it is already known that we are in one of the situations.

If we now use dependent types, much of these case reasoning will become simple computation. For that, let us associate to a memory location the type of its content:

```
Parameter get_type : memory → address → types.
```

And let us use a function returning the domain associated with each type name:

```
Definition domain (t:types) : Set :=
 match t with Nat ⇒ nat | Bool ⇒ bool | Address _ ⇒ address end.
```

One can then give the following signature to `get_value`:

```
Parameter get_value : ∀m:memory,∀a:address,domain (get_type m a).
```

This way, if one really implements the function `get_type` in a computable way, then the type `(domain (get_type m a))` can be reduced to `nat`, `bool` or `address` according to the case via a simple conversion.

During its extraction, an implementation of this `get_value` will then present exactly the same kind of typing difficulty that our simplistic example `p`.

3.1.3 The type of integer functions with arity n

The situation is even worse when the dependent type may be reduced to an infinity of different types, depending of the input value. For example the following predicate `F` associates to the integer `n` the type of the integer functions with `n` arguments:

```
Fixpoint F (n:nat) : Set :=
 match n with
  | O ⇒ nat
  | S n ⇒ nat → F n
 end.
```

One can then build a function whose arity depends on its first argument:

```
Fixpoint f (n:nat) : F n :=
 match n with return F n with
  | O ⇒ O
```

.../...

…/…

```
  | S n ⇒ fun _ ⇒ f n
 end.
```

There again, the raw extraction of `f` is untypable:

```
let rec f = function
  | O → O
  | S n → (fun x → f n)
```

One can still propose a typable version via the use of an union type designed to simulate the structure of `F`, but that becomes really difficult to describe this workaround in the most generic case of a `Fixpoint` on types. Moreover, it is possible to hide this fixpoint behind a constant. Indeed, `F` can be also written as:

```
Definition F := nat_rect (fun _ ⇒ Set) nat (fun _ t ⇒ nat → t).
```

3.1.4 Untranslatable inductive types

The inductive types also raise problems, because Coq allows to write types without equivalent in Ocaml or Haskell. For example, we can write in Coq an inductive type being able to contain objects of any type:

```
Inductive any : Type := Any : ∀A:Type, A → any.
```

A naive typed extraction `type 'a any = Any of 'a` would be unsatisfactory. One would indeed propagate a type variable `'a` with no counterpart in Coq. An initial version of our extraction implementation was proceeding of the kind, and was sometimes producing inductive types with hundredths of type variables, because of this propagation of variables.

In the same spirit, one can obtain a non-homogeneous list:

```
Inductive anyList : Type :=
 | AnyNil : anyList
 | AnyCons : ∀A:Type, A → anyList → anyList.

Definition my_anyList := AnyCons bool true (AnyCons nat O AnyNil).
```

Here, a naive translation (which would here give in fact the usual polymorphic lists) would be not only unsatisfactory, but even worse, completely incorrect from the point of view of types. This example does not seem to be adaptable into Ocaml code that would be typable and equivalent. The only solution here is then the one presented in the following section.

To show that these lists are perfectly usable in Coq, here comes for example the function returning the head of a non-empty list, or an object of type `unit` in the case of an empty list. The type of this function uses a predicate that depends on the list:

```
Definition getHeadType (l:anyList) : Type :=
 match l with
```

…/…

.../...

```
  | AnyNil ⇒ unit
  | AnyCons A _ _ ⇒ A
 end.

Definition getHead (l:anyList) : getHeadType l :=
 match l return getHeadType l with
  | AnyNil ⇒ tt
  | AnyCons _ a _ ⇒ a
 end.
```

The extraction of `getHead` presents typing problems close to those of the initial example `p`.

3.1.5 Dependent types and polymorphism

Even when dependent types are used to express simple polymorphism, one can have nasty surprises. The classical example is a derived form of the `distr_pair` property:

```
Definition distr_pair : (∀X:Set, X → X) → nat*bool :=
 fun f ⇒ (f nat 0, f bool true).
```

It is known that `fun f → (f 0, f true)` is untypable in ML. And there again, no simple adaptation into an equivalent typable code.

3.1.6 Contradictory case and typing

A last example, particularly awkward, is the use of an contradictory assumption for changing types. Let us suppose for example that an axiom makes the assumption that `nat = bool`

```
Section Strange.
 Variable absurd : nat = bool.
```

We can then use this false equality to show that `0` is a boolean.

```
 Definition 0_as_bool : bool.
  rewrite <- absurd; exact 0.
 Defined.
```

And consequently, nothing prohibits to define a term by case on `0` being either `true` or `false`!

```
 Definition strange := if 0_as_bool then 0 else (S 0).
End Strange.
```

It should be noted that internally, `0_as_bool` is a simple call to `eq_rec`, the induction principle for the equality.

```
Coq < Print O_as_bool.
O_as_bool =
fun absurd : nat = bool ⇒ eq_rec nat (fun P : Set ⇒ P) O bool absurd
    : nat = bool → bool
```

As explained in the previous chapter, `eq` is the emblematic example of logical singleton inductive types, it thus disappears during extraction, leaving for `eq_rec` only the identity. Moreover, the logical argument `nat=bool` is ignored and becomes a `_`. the extraction of `O_as_bool` is then:

```
let O_as_bool _ = O
```

Concerning `strange`, it gives (as soon as we replace `O_as_bool` by its definition[1]):

```
let strange _ =
 match O with
   | True → O
   | False → S O
```

It should be noted that even if a term extracted from a Coq proof can contain a call to `strange`, this call will never be fully evaluated, because of the lack of the argument that would start the evaluation of the matching. The body of `strange` is here in fact dead code. But even if the extraction of `strange` is safe from the point of view of execution, it is undeniable that this extracted term leads to typing problems.

3.2 An artificial correction of typing errors

3.2.1 Obj.magic and unsafeCoerce

We have seen that some of these typing errors could be solve if required by rewriting the produced code. But some other situations cannot be treated this way, which is anyway quite difficult to automatize and moreover undesirable. This is why we followed a uniform approach consisting in the use of low-level and not-documented primitives which allow to artificially change the type of an expression. For example, in Ocaml, we use `Obj.magic : 'a → 'b`. Such a primitive, implemented internally as the identity, is obviously not normally definable in Ocaml, and its use voids the vast majority of theoretical results concerning Ocaml : in particular, an Ocaml well-typed term with `Obj.magic` is not anymore guaranteed to be executable without error.

Some implementations of Haskell contain a primitive named `unsafeCoerce`, which should be usable exactly in the same way that `Obj.magic`. Because of lack of time, we did not implemented the automatic generation of `unsafeCoerce` in the Haskell extracted code. But this generation is a priori no more difficult than the generation of `Obj.magic`. The rest of this chapter is thus devoted to Ocaml.

[1] See later in section 4.3.3 the study of such replacements.

Let us make for example transform a boolean into an integer via `Obj.magic`.

```
# (Obj.magic true) + 1;;
- : int = 2
```

The fact that this computation succeeds is here a consequence of the internal representation of objects in Ocaml : `true` being coded by 1, one obtains 2 as final result. On the other hand, if the chimera created by the use of `Obj.magic` is not compatible with the internal representation of objects, the sanction is immediate:

```
# (Obj.magic 1) 2;;
Segmentation fault
```

Indeed, one tries here to use 1 as pointer to the code of a function, which generates a memory error.

The use of `Obj.magic` by the programmer, without being prohibited, is thus in any case strongly not recommended, and must be done with great caution. But one should not either disregard these `Obj.magic`, that are quite practical in some situations. For example there are some in the current implementation of the Queue module of the Ocaml standard library, or in the Dyn module of dynamic typing in the sources of Coq.

Within the extraction framework, fortunately, we already know that an execution error is not possible. We then have the freedom to place as many place `Obj.magic` as necessary in order to ensure typing. A corrected extraction of the example `p` of section 3.1.1 can for example be:

```
let p = function
  | True → Obj.magic O
  | False → Obj.magic True
```

How to place these `Obj.magic` ? A possibility is obviously to put some at each node of the program. But that presents two disadvantages:

- the code becomes completely unreadable, which contradicts one of our objectives.
- more serious, the performances are decreased, as showed by L. Pottier [70]. This is explained (at least partly) by the fact that the Ocaml compiler skips a certain number of optimizations around the `Obj.magic`.

Since we have seen that the typing errors generally remain very marginal in extracted code, it is thus really interesting to finely locate these error locations in order to insert as less `Obj.magic` as possible.

3.2.2 A first attempt at correcting the types

We thus have embedded in our extraction an type-checker that also behaves as a type-corrector: for each detected error, it inserts a `Obj.magic` that solves it. First, we present a relatively simple way of doing that. Since this first manner is also little satisfactory, we then describe another method, more complex, used in practice in the extraction implementation.

The algorithms $\mathcal{W}$ and $\mathcal{W}$'

The simplest way to detect and correct the typing errors is to proceed in a *lazy* way. For that we use one algorithm for inference/verification of types, such as the $\mathcal{W}$ algorithm of Damas-Milner [27]. The type of this algorithm is the following:

```
W : env * expr → type * subst
```

And its correctness is expressed by $\mathcal{W}(\Gamma, t) = (T, \sigma) \Rightarrow \sigma(\Gamma) \vdash T : \sigma(T)$.

This algorithm can fail to type a term only through a failure of the type unification subprocess `mgu`. All that remains to be done is to catch these unification errors and transform them into success thanks to new `Obj.magic`. For that we adapt $\mathcal{W}$ so that it also returns modifications made under the terms. Its type is now:

```
W' : env * expr → type * subst * expr
```

And the desired property is now $\mathcal{W}'(\Gamma, t) = (T, \sigma, T') \Rightarrow \sigma(\Gamma) \vdash t' : \sigma(T)$ and $t' \sim t$.

The $\mathcal{W}$ algorithm being now more than famous, we just give here a presentation of $\mathcal{W}$' centered on our modifications. A more formal presentation of $\mathcal{W}$ can be found in [51]. The presentation given here has been inspired by a course of X. Leroy.

First of all, let us point out the mechanisms of instantiation and of generalization, that allow to go from a type scheme to an Ocaml type and reciprocally:

$\mathtt{Inst}(\forall \overrightarrow{\alpha_i}.\tau) = \tau[\overrightarrow{\alpha_i}/\overrightarrow{\beta_i}]$, with $\overrightarrow{\beta_i}$ fresh variable.

$\mathtt{Gen}(\tau, \Gamma) = \forall \overrightarrow{\alpha_i}.\tau$, with $\overrightarrow{\alpha_i}$ being the variables of τ free in Γ

The figure 3.1 contains the definition of a minimalist version of $\mathcal{W}$', parameterized by an environment E of declarations of constants provided with their types. This presentation does voluntarily include no recursivity: on the typing level, we can present recursivity via an fixpoint operator `fix` having for signature $\forall \alpha,\ (\alpha \to \alpha) \to \alpha$.

The encoding of inductive types

The inductive types will also be represented by encoding via constants, in order to leave the central algorithm as simple as possible. Consider an inductive type Ocaml t defined by:

$\mathtt{type}\ (\alpha_1, \ldots, \alpha_2)\ t = C_1\ \mathtt{of}\ \tau_{1,1} * \ldots * \tau_{1,n_1} | \ldots |\ C_p\ \mathtt{of}\ \tau_{p,1} * \ldots * \tau_{p,n_p}$

We represent the constructors of t by constants $C_1 \ldots C_p$ having for types:

$C_i\ :\ \forall \alpha_1, \ldots, \alpha_n.\tau_{i,1} \to \ldots \to \tau_{i,n_i} \to (\alpha_1, \ldots, \alpha_n)\ t$

And we represent pattern matching on t by an operator F_t with the following type:

$$F_t\ :\ \forall \alpha_1, \ldots, \alpha_n, \beta.(\alpha_1, \ldots, \alpha_n)\ t \to \\ (\tau_{1,1} \to \ldots \to \tau_{1,n_1} \to \beta) \to \ldots \to (\tau_{p,1} \to \ldots \to \tau_{p,n_p} \to \beta) \to \beta$$

A simplification to be avoided

At first sight, it may seem that our algorithm $\mathcal{W}$' includes an overly complex generation mechanism for `Obj.magic`. Indeed, for an application $(a_1\ a_2)$, we distinguish two cases:

$$
\begin{array}{ll}
\mathcal{W}'(\Gamma,\ x) = (\mathtt{Inst}(\Gamma(x)),\ \mathtt{id},\ x) & \text{for a variable } x \\[1ex]
\mathcal{W}'(\Gamma,\ c) = (\mathtt{Inst}(E(x)),\ \mathtt{id},\ c) & \text{for a constant } c \\[1ex]
\mathcal{W}'(\Gamma,\ \mathtt{fun}\ x \to a) = & \\
\quad \mathtt{let}\ (\tau_1,\ \phi_1,\ \tilde{a}) = \mathcal{W}'(\Gamma + \{x : \beta\},\ a)\ \mathtt{in} & \text{with } \beta \text{ fresh variable} \\
\quad (\phi_1(\beta) \to \tau_1,\ \phi_1,\ \mathtt{fun}\ x\ \to \tilde{a}) & \\[1ex]
\mathcal{W}'(\Gamma,\ a_1\ a_2) = & \\
\quad \mathtt{let}\ (\tau_1,\ \phi_1,\ \tilde{a_1}) = \mathcal{W}'(\Gamma,\ a_1)\ \mathtt{in} & \\
\quad \mathtt{let}\ (\tau_2,\ \phi_2,\ \tilde{a_2}) = \mathcal{W}'(\phi_1(\Gamma),\ a_2)\ \mathtt{in} & \\
\quad \mathtt{let}\ \sigma = \mathtt{mgu}(\phi_2(\tau_1),\ \alpha \to \beta)\ \mathtt{in} & \text{with } \alpha \text{ and } \beta \text{ fresh} \\
\quad \mathtt{if}\ \sigma = \mathtt{error}\ \mathtt{then} & \\
\quad\quad (\gamma,\ \phi_2 \cdot \phi_1,\ (\mathtt{Obj.magic}\ \tilde{a_1})\ \tilde{a_2}) & \text{with } \gamma \text{ fresh} \\
\quad \mathtt{else} & \\
\quad\quad \mathtt{let}\ \mu = \mathtt{mgu}(\tau_2,\ \sigma(\alpha))\ \mathtt{in} & \\
\quad\quad \mathtt{if}\ \mu = \mathtt{error}\ \mathtt{then} & \\
\quad\quad\quad (\sigma(\beta),\ \ \phi \cdot \phi_2 \cdot \phi_1,\ \ \tilde{a_1}\ (\mathtt{Obj.magic}\ \tilde{a_2})) & \\
\quad\quad \mathtt{else} & \\
\quad\quad\quad (\mu(\beta),\ \ \mu \cdot \sigma \cdot \phi_2 \cdot \phi_1,\ \ \tilde{a_1}\ \tilde{a_2}) & \\[1ex]
\mathcal{W}'(\Gamma,\ \mathtt{let}\ x = a_1\ \mathtt{in}\ a_2) = & \\
\quad \mathtt{let}\ (\tau_1,\ \phi_1,\ \tilde{a_1}) = \mathcal{W}'(\Gamma,\ a_1)\ \mathtt{in} & \\
\quad \mathtt{let}\ (\tau_2,\ \phi_2,\ \tilde{a_2}) = \mathcal{W}'(\phi_1(\Gamma) + \{x : \mathtt{Gen}(\tau_1,\ \phi_1(\Gamma))\},\ a_2)\ \mathtt{in} & \\
\quad (\tau_2,\ \phi_2 \cdot \phi_1,\ \mathtt{let}\ x = \tilde{a_1}\ \mathtt{in}\ \tilde{a_2}) &
\end{array}
$$

FIG. 3.1: Definition of $\mathcal{W}$'

- if the head a_1 does not have an arrow type, we surround it by an `Obj.magic`.
- if the argument a_2 does not have a type compatible with the beginning of the arrow type, then we place an `Obj.magic` around this argument.

If an `Obj.magic` is necessary, an simpler solution, also correct, is to always place the `Obj.magic` around the head of the application. The problem is that the constants C_i and F_t that encode inductive terms can then be surrounded by `Obj.magic`, and this forbids their transformation back to Ocaml syntax. Our solution with two unifications avoids this problem.

Correctness of $\mathcal{W}$'

This $\mathcal{W}$' algorithm satisfies the property $\mathcal{W}'(\Gamma, t) = (T, \sigma, t') \;\Rightarrow\; \sigma(\Gamma) \vdash t' : \sigma(T)$. The proof is similar to the one of the unmodified $\mathcal{W}$ algorithm. It is just necessary to check in addition the two cases that generate `Obj.magic`. However we have by induction hypothesis $\phi_1(\Gamma) \vdash \tilde{a_1} : \tau_1$ and $\phi_2(\phi_1(\Gamma)) \vdash \tilde{a_2} : \tau_2$. From the first relation we obtain $\phi_2(\phi_1(\Gamma)) \vdash \tilde{a_1} : \phi_2(\tau_1)$.

- The first case generating `Obj.magic` is the simplest: the type scheme $\forall\alpha\beta.\alpha \rightarrow \beta$ allows to give any type to (`Obj.magic` $\tilde{a_1}$), and in particular $\tau_2 \rightarrow \gamma$. The application ((`Obj.magic` $\tilde{a_1}$) $\tilde{a_2}$) thus accepts γ as type in the environment $\phi_2(\phi_1(\Gamma))$
- In the second case, we have $\sigma(\phi_2(\phi_1(\Gamma))) \vdash \tilde{a_1} : \sigma(\phi_2(\tau_1))$. Taking into account the definition of σ, this rewrites into $\sigma(\phi_2(phi_1(\Gamma))) \vdash \tilde{a_1} : \sigma(\alpha) \rightarrow \sigma(\beta)$. It is now (`Obj.magic` $\tilde{a_2}$) that can accept any type and in particular $\sigma(\alpha)$, and finally the term ($\tilde{a_1}$ (`Obj.magic` $\tilde{a_2}$) thus accepts $\sigma(\beta)$ as type in the environment $\sigma(\phi_2(\phi_1(\Gamma)))$.

Behavior of $\mathcal{W}$' on an example

Let us now test our $\mathcal{W}$' algorithm on the extraction of the example `p`. In our encoding, the body of `p` is written: `fun` $b \rightarrow F_{bool}\ b$ `0 True`. The successive calls to $\mathcal{W}$' are then, from the more internal to the most external:

$\mathcal{W}'(\{b : \alpha\}, F_{bool}) = (\texttt{bool} \rightarrow \beta \rightarrow \beta \rightarrow \beta,\ \texttt{id},\ F_{bool})$

$\mathcal{W}'(\{b : \alpha\}, F_{bool}\ b) = (\beta \rightarrow \beta \rightarrow \beta,\ \{\alpha \leftarrow \texttt{bool}\},\ F_{bool}\ b)$

$\mathcal{W}'(\{b : \alpha\}, F_{bool}\ b\ \texttt{0}) = (\texttt{nat} \rightarrow \texttt{nat},\ \{\alpha \leftarrow \texttt{bool}; \beta \leftarrow \texttt{nat}\},\ F_{bool}\ b\ \texttt{0})$

$\mathcal{W}'(\{b : \alpha\}, F_{bool}\ b\ \texttt{0 True}) = (\texttt{nat},\ \{\alpha \leftarrow \texttt{bool}; \beta \leftarrow \texttt{nat}\},\ F_{bool}\ b\ \texttt{0 (Obj.magic True)})$

$\mathcal{W}'(\varnothing, \texttt{fun}\ b \rightarrow F_{bool}\ b\ \texttt{0 True}) = (\texttt{bool} \rightarrow \texttt{nat}, \ldots, \texttt{fun}\ b \rightarrow F_{bool}\ b\ \texttt{0 (Obj.magic True)})$

On the fourth line, `True` cannot have the type `nat` that would be necessary for a correct application. It is then surrounded by an `Obj.magic`. Finally, the corrected term is thus (in Ocaml syntax):

```
let p = function
  | True → 0
  | False → Obj.magic True
```

The limitations of $\mathcal{W}$'

If this method works indeed and produce typable terms, it is nevertheless not very satisfactory. In fact, the inferred Ocaml types are sometimes strange and asymmetrical, the order of unification playing a significant role in the final result. In our example `p` the typing problem is in particular discovered only on the second branch of `match`, and it is thus only this second branch that carries an `Obj.magic`. This implies that the type of `p` is `bool`→ `nat`, and that the occurrences of `(p false)` will be surrounded by `Obj.magic`, whereas the occurrences of `(p true)` will not.

This asymmetry forbids to guess a priori what type will be inferred for a given extracted term. The only way of calculating this type is to apply the $\mathcal{W}$' algorithm. In particular two Coq terms with the same type will not necessarily have the same Ocaml type once extracted. To illustrate that point, it is enough to consider `p` and its following alternative, where one filters in the other direction by using the opposite boolean:

```
Definition p' :=
 fun b: bool ⇒
 match negb b as b' return P (negb b') with
  | true ⇒ true
  | false ⇒ O
 end.
```

The extraction of `p'` via $\mathcal{W}$' gives then:

```
let p' b =
  match negb b with
    | True → Obj.magic True
    | False → O
```

And this term extracts has as a type `bool→bool` instead of `bool→nat` of `p`. A slight abuse should be noted here: the Coq type of `p'` is `∀b:bool,P (negb (negb b))`, and that is not exactly the type of `p`, which is `∀b:bool,P b`. To correct this abuse, we can for example consider `(p true)` and `(p' true)` instead of `p` and `p'`. In this case the two terms have both the type `nat` since `(P true)` = `(P (negb (negb true)))` = `nat`. And after extraction `(p true)` and `(p' true)` will have for respective Ocaml types `nat` and `bool`[2].

This impossibility of knowing in advance the types of the extracted terms has forced us to stop using this method. Indeed, this lack of determinism in types is really annoying when interfacing with extracted code in broader developments. And even without speaking of integration with external code, the extraction of Coq modules to Ocaml modules (cf. section 4.1) is strongly complicated by such a property.

3.2.3 The algorithm $\mathcal{M}$

To solve these difficulties, we proceed in two stages.

- the first stage is to compute a type Ocaml $\widehat{\mathcal{E}}(T)$ awaited for the extracted term starting from $t : T$. This computation is done independently of the body of the Coq term t, and uses only its type T, in order to remain modular. This extraction stage $\widehat{\mathcal{E}}$ will be detailed later on.
- once we known this awaited type, we use it as goal, and force the extraction of t to accept indeed $\widehat{\mathcal{E}}(T)$ as Ocaml type, always thanks to `Obj.magic`.

We start now by detailing this second stage, which is done once again by adapting an typing algorithm. But now the stress is more laid on the checking of types that on the inference. And to this change of approach corresponds a change of algorithm: we replace

[2] Another way of obtaining two adequate terms is to wrap `p'` into a term `p''` having the same type as `p`. Indeed, since the types of `p` and `p'` are provably equal instead of being convertible, one can pass from the one to the other via the lemma `negb_elim: ∀b:bool,negb (negb b)=b`. This gives us:

```
Definition p'' : ∀b:bool,P b. intro b; rewrite <- negb_elim; exact (p' b). Defined.
```

$\mathcal{W}$ by the $\mathcal{M}$ algorithm. This algorithm, described in 1998 by O. Lee and K. Yi in [51], is itself an alternative of $\mathcal{W}$. But unlike the latter, $\mathcal{M}$ proceeds via top-down analysis instead of bottom-up, hence its reversed name compared with $\mathcal{W}$. For the record, this $\mathcal{M}$ algorithm has been used until 1993 in Caml Light prior to version 0.7. And even if we will take advantage of this property, we nevertheless note that $\mathcal{M}$ detects the typing errors in a finer way than $\mathcal{W}$. The interested reader can find more details on this subject in [51].

The difference between $\mathcal{M}$ and $\mathcal{W}$ is already visible on the level of their types:

```
W : env * expr → type * subst
M : env * expr * type → subst
```

Instead of infer a result type, $\mathcal{M}$ asks rather for a type as input and checks that this type is be appropriate, modulo possible substitution. Its correctness is expressed by:

$\mathcal{M}(\Gamma, t, T) = \sigma \;\Rightarrow\; \sigma(\Gamma) \vdash t : \sigma(T)$.

In practice, the difference between the two algorithms is not not so large. We can always, indeed, use $\mathcal{M}$ with an initial type `'x` in order to retrieve an inference algorithm, and the inferred type obtained at the end is $\sigma($`'x`$)$. And anyway, even an algorithm centered on type-checking like $\mathcal{M}$ must make steps of inference, for example when it meets a `let-in`.

To use $\mathcal{M}$ within the framework of the extraction, we adapt it the same way we adapted $\mathcal{W}$ in $\mathcal{W}$':

```
M' : env * expr * type → subst * expr
```

And one wishes now that $\mathcal{M}'(\Gamma, t, T) = (\sigma, t') \;\Rightarrow\; \sigma(\Gamma) \vdash t' : \sigma(T)$. The figure 3.2 gives the definition of $\mathcal{M}$', which uses the same notations as the definition of $\mathcal{W}$'. We also reuse the same even encoding of recursivity and inductive types.

If we compare $\mathcal{M}$' and $\mathcal{W}$', we notices that the `let-in` case is similar in the two algorithms. This is explained by the need for inferring the type of the local definition. For the other cases, the roles are reversed concerning the use of the `mgu` unifier. The application does not require any unification, whereas on the opposite the last cases (functions, constants and variables) now contain some.

The correctness of $\mathcal{M}$' will not be developed. Just as for $\mathcal{W}$ and $\mathcal{W}$', this correctness relies on the correctness of $\mathcal{M}$, which is proved in [51], and on the study of the cases generating `Obj.magic`.

But we still cannot use $\mathcal{M}$' directly for extraction. Indeed we wish, for a given type τ, to force a term a to accept *exactly* τ as type. However $\mathcal{M}'$ returns a substitution σ to be applied on the variables of τ before getting the final appropriate type of a. Hence the final type $\sigma(\tau)$ of a can be strictly less general than τ. To solve this problem, we have used two types of variables:

- the substitutable variables, which are those used until now, but which will not be used any more except internally inside $\mathcal{M}$', during creations of fresh variables.
- non-substitutable variables, only authorized in extracted types that are used as initial arguments from $\mathcal{M}$'. If the unifier `mgu` must solve an equation of the form $\alpha =? \tau$ with α non-substitutable variable and τ a type different from α, a unification error is raised

$$
\begin{array}{ll}
\mathcal{M}'(\Gamma,\ x,\ \tau) = & \\
\quad \texttt{let}\ \sigma = \texttt{mgu}(\tau,\ \texttt{Inst}(\Gamma(x)))\ \texttt{in} & \\
\quad \texttt{if}\ \sigma = \texttt{error then}\ (\texttt{id},\ \texttt{Obj.magic}\ x)\ \texttt{else}\ (\sigma,\ x) & \\
& \\
\mathcal{M}'(\Gamma,\ c,\ \tau) = & \\
\quad \texttt{let}\ \sigma = \texttt{mgu}(\tau,\ \texttt{Inst}(E(c)))\ \texttt{in} & \\
\quad \texttt{if}\ \sigma = \texttt{error then}\ (\texttt{id},\ \texttt{Obj.magic}\ c)\ \texttt{else}\ (\sigma,\ c) & \\
& \\
\mathcal{M}'(\Gamma,\ \texttt{fun}\ x \to a,\ \tau) = & \\
\quad \texttt{let}\ \sigma = \texttt{mgu}(\tau,\ \alpha \to \beta)\ \texttt{in} & \text{with } \alpha \text{ and } \beta \text{ fresh} \\
\quad \texttt{if}\ \sigma = \texttt{error then} & \\
\qquad \texttt{let}\ (\phi,\ \tilde{a}) = \mathcal{M}'(\Gamma + \{x : \alpha\},\ a,\ \beta)\ \texttt{in} & \\
\qquad (\phi,\ \texttt{Obj.magic}\ (\texttt{fun}\ x \to \tilde{a})) & \\
\quad \texttt{else} & \\
\qquad \texttt{let}\ (\phi,\ \tilde{a}) = \mathcal{M}'(\Gamma + \{x : \sigma(\alpha)\},\ a,\ \sigma(\beta))\ \texttt{in} & \\
\qquad (\phi \cdot \sigma,\ \texttt{fun}\ x \to \tilde{a}) & \\
& \\
\mathcal{M}'(\Gamma,\ a_1\ a_2) = & \\
\quad \texttt{let}\ (\phi_1,\ \tilde{a_1}) = \mathcal{M}'(\Gamma,\ a_1,\ \alpha \to \tau)\ \texttt{in} & \text{with } \alpha \text{ fresh} \\
\quad \texttt{let}\ (\phi_2,\ \tilde{a_2}) = \mathcal{M}'(\phi_1(\Gamma),\ a_2,\ \phi_1(\alpha))\ \texttt{in} & \\
\quad (\phi_2 \cdot \phi_1,\ \tilde{a_1}\ \tilde{a_2}) & \\
& \\
\mathcal{M}'(\Gamma,\ \texttt{let}\ x = a_1\ \texttt{in}\ a_2) = & \\
\quad \texttt{let}\ (\phi_1,\ \tilde{a_1}) = \mathcal{M}'(\Gamma,\ a_1,\ \alpha)\ \texttt{in} & \text{with } \alpha \text{ fresh} \\
\quad \texttt{let}\ (\phi_2,\ \tilde{a_2}) = \mathcal{M}'(\phi_1(\Gamma) + \{x : \texttt{Gen}(\phi_1(\alpha),\ \phi_1(\Gamma))\},\ a_2,\ \phi_1(\tau))\ \texttt{in} & \\
\quad (\phi_2 \cdot \phi_1,\ \texttt{let}\ x = \tilde{a_1}\ \texttt{in}\ \tilde{a_2}) &
\end{array}
$$

FIG. 3.2: Definition of $\mathcal{M}$'

instead of returning the substitution $\{\alpha \leftarrow \tau\}$. And this unification error will make $\mathcal{M}$' generate an `Obj.magic` which will allow to keep the most general type, here α.

From now on, by ensuring that the τ type provided to $\mathcal{M}$' only contains non-substitutable variables, we thus guarantee that the substitution result σ leaves τ invariant. We could then ignore this substitution. Finally, starting from a Coq term $t : T$, the extracted typable term t_{ml} and its type T_{ml} are obtained by: $T_{ml} = \widehat{\mathcal{E}}(T)$ and $t_{ml} = \texttt{snd}(\mathcal{M}'(\varnothing, \mathcal{E}(t), \widehat{\mathcal{E}}(T)))$. And whatever is the choice of $\widehat{\mathcal{E}}(T)$, we are sure that t_{ml} has indeed T_{ml} as type and differs from $\mathcal{E}(t)$ only by possible insertion of `Obj.magic`.

3.3 The extraction of Coq types

We now present the extraction $\widehat{\mathcal{E}}$ of Coq types into Ocaml types. This extraction of types, as we have seen, is used by our $\mathcal{M}$' algorithm. But it is anyway made necessary by the presence of inductive types in Coq : starting from the Coq types of inductive constructors, we must deduce the Ocaml types that are the most faithful for the extracted constructors. As long as execution is concerned, the only constraint concerning the inductive types is to preserve the number of their constructors and the argument numbers of these constructors. But ensuring only this constraint would oblige to put far too many `Obj.magic`, and would produce unreadable extracted terms and types. Moreover, this extraction of types will enable us to produce a interface file `.mli` for each produced file `.ml`, and that in a foreseeable way: only the Coq types will influence this interface, and not the contents of the terms to be extracted.

3.3.1 A approximation of Coq types

The previous examples showed that the richness of the Coq types could not always be translated in Ocaml in an faithful way. We then proceed by approximation. For that, we use a most general Ocaml type (or most unknown), that we will note $\mathbb{T}$. We have implemented[3] this $\mathbb{T}$ type thanks to the internal Ocaml type of all the objects, `Obj.t`. And conversions between `Obj.t` and the other types are carried out via `Obj.magic`.

This $\mathbb{T}$ type allow to give a satisfactory response to the previous examples. For example, the function `p` which returns an integer or a boolean will have the type `bool` $\rightarrow \mathbb{T}$. And concerning the inductive type `any` containing any object, it becomes:

```
type any = Any of 𝕋
```

It must then be clear that the extraction of types which we propose is necessarily arbitrary at certain places, even if it gives good results in practice. This "efficiency" of the extraction of types is measured:

- by the precision of the approximation carried out. One extraction of types that would always answer $\mathbb{T}$, although possible, is of course not interesting.
- by the low number of `Obj.magic` necessary to force the correspondence between extracted terms and extracted types.

In fact, the following definition of $\widehat{\mathcal{E}}$ proceeds only by replacement of subterms by $\mathbb{T}$, if one puts aside the syntax questions. It would be then possible to give to the set of all types with $\mathbb{T}$ a structure of semi-lattice upward, whose maximal element would be $\mathbb{T}$. And we would then have for any type Coq U that $U \leq \widehat{\mathcal{E}}(U) \leq \mathbb{T}$. This order would then constitute a measure of the degree of approximation made during the extraction of types.

[3]In the extracted code, the ASCII notation for this $\mathbb{T}$ type is `__`. Note the absence of conflict with the notation of the term $\square$ which is also `__`: in Ocaml names of types and terms do not interfere

3.3.2 The type of logical residues

We saw in the previous chapter that the extraction of the terms was not able to completely remove logical parts from terms, and that it could remain some the residual constants □. Which type could we then give to these residues □? Ideally any constant type with at least a value could be appropriate, such as for example `unit`. Unfortunately, it can happen during a reduction in Ocaml that a residue is found applied (see example 1 in the previous chapter). The rule of reduction to be used is then:

$$(\square\ X) \to \square$$

Implementing □ by `()` is thus incorrect from the point of view of the execution. We can nevertheless get the good behavior in Ocaml, provided we cheat with the typing[4]:

```
# let rec __ x = Obj.magic __;;
val __ : 'a → 'b = <fun>
# __ 1 true [];;
- : '_a = <poly>
```

The only difficulty is that the type `'a→'b` given for □ is harmful for the readability of signatures, because it multiply the useless type variables. We then decided to "cast" this type into `Obj.t`. This can be done by using instead of `Obj.magic` its alternative `Obj.repr: 'a → Obj.t`. We have finally chosen the following implementation of □:

```
# let __ = let rec f _ = Obj.repr f in Obj.repr f
val __ : Obj.t = <fun>
```

In the following definition of the extraction $\widehat{\mathcal{E}}$ of types, we will use the symbol □ to indicate the type of the constant □. This symbol of the type is defined as an alias to $\mathbb{T}$. In fact, one syntactically separates these two symbols only during the extraction, during which they will play distinct roles:

- If the extraction of a type gives `nat`→□, one knows that all the type is logical, and it is desirable to produce the extraction □ for the whole.
- Conversely if a type is extracted in `nat`→$\mathbb{T}$, one knows that it acts as the type of a function with an integer argument and unknown result. And there, an approximation of the whole type with $\mathbb{T}$ is not desirable because any information would then been lost.

3.3.3 The border between types and terms

Let us recall that Coq does not make syntactic distinction between terms and types, unlike Ocaml. The extraction need then to decide to place an Coq object in the world of extracted terms or in the world of extracted types. In the previous chapter concerning the

[4]The ASCII notation of □ is `__`.

extraction of terms, we pruned all type schemes[5] into the constant $\square$. The reason of this pruning was that a type scheme, which will become a type once applied to sufficiently many arguments, does not have real calculative content.

For extraction of types, in a coherent way with the extraction of terms, type schemes are once again used as border between types and terms. In fact, the extraction of types accept any Coq term, but immediately returns the unknown type $\mathbb{T}$ if the input term is not a type scheme, in the same way that the extraction of terms returns $\square$ in the degenerated cases.

3.3.4 The Coq types, from simple to complex

Before entering the definition itself for extraction of types, we try here to give an intuition of its behavior. For that, we first present a certain number of examples of Coq types sufficiently simple to have a faithful equivalent in Ocaml. Then we continue with increasingly complex Coq types.

The inductive types

First of all, One can obviously mention the inductive types whose constructors are constant, such as `bool`.

```
Inductive bool : Set := true : bool | false : bool.
```

$\Downarrow$

```
type bool = True | False.
```

More generally inductive types without parameter and whose constructors have simply translatable types are themselves simply translatable. For example:

```
Inductive nat : Set := O : nat | S : nat → nat.
```

$\Downarrow$

```
type nat = O | S of nat.
```

When we consider parameters of type `Set` (or `Type`), the situation remains natural. One can indeed see `Set` as the set of all types. The Coq parameter then becomes an Ocaml type variable. For example:

```
Inductive list (A:Set) : Set :=
 | nil : list A
 | cons : A → list A → list A.
```

$\Downarrow$

```
type 'a list = Nil | Cons of 'a * 'a list
```

[5]As a reminder, one type scheme is a term admitting a type of the form: $\forall x_1 : X_1, ... \forall x_n : X_n, s$

Note the transformation of the constructor type, from a curryfied functional form in Coq to a sequence of products in Ocaml. Of course, the application of such an inductive type is obviously translated into an Ocaml application of types: a `(list nat)` of Coq becomes a `(nat list)` in Ocaml.

There is no reason to limit ourself to parameters, the products present in the signature of inductive types playing a similar role:

```
Inductive list2 : ∀A:Set,Type :=
 | nil2 : ∀A:Set,list2 A
 | cons2 : ∀A:Set,A → list2 (A*A) → list2 A.
```

$\Downarrow$

```
type 'a list2 = Nil2 | Cons2 of 'a * ('a * 'a) list2
```

But we enter here a dangerous zone. Changing slightly the previous definition leads to a problem.

```
Inductive list3 : ∀A:Set,Type :=
 | nil3 : ∀A:Set,list3 A
 | cons3 : ∀A:Set,A → list3 A → list3 (A*A).
```

This definition is the dual of the previous one, in which the pairs will accumulate on the right of lists. But in Ocaml, for an inductive `list3` with one type variable `'a`, the constructors are used to build a `'a list3` and no other type. There is thus no way of translating accurately this inductive, and we need to choose an approximation, for example

```
type 'a list3 = Nil3 | Cons3 of 𝕋 * 𝕋 list3
```

With these parameters or variables in the type `Set`, we have already reached the whole of definable inductive types in Ocaml. Any inductive type Coq more complex will thus bring issues. For example, if a parameter is a type scheme, to represent it by a type variable is imperfect: this variable, instead of representing a whole potential family of types, represents only one of them. We take nevertheless this representation, because one cannot do better in Ocaml. For example:

```
Inductive poly (X:nat→Set) : Set := Poly : ∀n:nat,X n → poly X.
```

$\Downarrow$

```
type 'x poly = Poly of nat * 'x
```

Finally, last case, if a parameter or a variable is on the terms level, it will not be translatable in Ocaml. The classical example is the lists of size n.

```
Inductive listn (A:Set) : nat → Set :=
 | niln : listn A O
```

.../...

.../...

```
 | consn : ∀n:nat, A → listn A n → listn A (S n).
```

A solution is then to replace each untranslatable argument by $\mathbb{T}$.

```
type ('a,'n) listn = Niln | Consn of nat * 'a * ('a,𝕋) listn
```

But we will see later than one can always locate such dependencies with respect to terms and completely remove them.

The type schemes

The type schemes correspond rather naturally to Ocaml types with type variables. Let us take for example the case of a type scheme having a dependency with respect to a type:

```
Definition Sch1 : Set → Set := fun X:Set ⇒ X → X.
```

$\Downarrow$

```
type 'x sch1 = 'x → 'x
```

And if one applies this type scheme in Coq, one also obtains an application in Ocaml : `(Sch1 nat)` gives `(nat  sch1)` in Ocaml syntax.

As for the inductive types, as long as the dependency is with respect to a type, everything remains simple. Let us consider now the case of a dependency with respect to a term. It is enough to take again the constant `P` of section 3.1:

```
Definition P (b:bool) : Set := if b then nat else bool.
Definition Sch2 (b:bool) : Set := P b.
```

There is obviously no such possible dependency in Ocaml. To remain uniform, a solution is to produce nonetheless a type variable which is in fact never used:

```
type 'b p = 𝕋
type 'b sch2 = 𝕋 p
```

We will see later that it is in fact possible to locate such dependencies with respect to terms and remove them completely, in order to obtain:

```
type p = 𝕋
type sch2 = p
```

Lastly, let us now consider the last case, the one of a dependency with respect to a new type scheme:

```
Definition Sch3 : (bool → Set) → Set :=
 fun (X:bool→Set) ⇒ X true → X false.
```

A priori, an Ocaml type variable is insufficient to represent the type scheme `X`. We must thus approximate, which can be made in several manners.

- The most cautious answer is to consider `(X true)` and `(X false)` as unknown. Indeed, if one instantiate `X` with the constant `P` as defined above, one obtains for example:

```
Sch3 P = nat → bool
```

This leads us to a first possibility for extraction: all variable which is not a type is seen as unknown, with its arguments. Here:

```
type 'x sch3 = 𝕋 → 𝕋
```

- But this answer is in practice too fuzzy to be interesting. Indeed, in a situation not really dependent like `(Sch3 (fun _ ⇒ nat))`, one obtains as final type $\mathbb{T} \to \mathbb{T}$ instead of `nat → nat`. And this occurs frequently in practice. On the other hand, one obtains `nat → nat` if one takes for extraction:

```
type 'x sch3 = 'x → 'x
```

And concerning the dependent situations like `(Sch3 P)`, this new extraction behaves correctly. Here for example, `P` will be regarded as an unknown type $\mathbb{T}$. And one thus still obtains $\mathbb{T} \to \mathbb{T}$.

Between the first version, more systematic, and the second, sharper, we have chosen the second.

The types themselves

The type construction which is immediately translatable in Ocaml is the functional arrow, i.e. the non-dependent product. Clearly, `nat → nat` is directly expressible both in Coq and in Ocaml.

On the opposite, the dependent product is in general not expressible in Ocaml. In particular, it induces a local binding of a variable inside a type. Let us look at the type of our `distr_pair` example of the section 3.1:

```
((X:Set) X → X) → nat*bool
```

If one tries at all costs to generate a type variable for `X`, then this variable will be visible in the whole type. In addition to distorting the semantics of the Coq type, that has harmful consequences: each declaration containing products will see its parameter number explode. The only reasonable translation is in fact to be unaware of the dependency induced by the internal product, and to replace each occurrence of the variable by $\mathbb{T}$. One obtains here:

```
(𝕋 → 𝕋 → 𝕋) → nat*bool
```

Only the products at the head of the type can be translated slightly more faithfully (cf. section 3.5).

To study the other cases, it should be noted that we are allowed to reduce in the types during their extraction, in order to limit the number of situations that are unsupported by Ocaml types. This is new compared to the extraction of terms, during which any reduction is obviously out of question. In the rest of the chapter, all is thus done modulo $\beta\iota\zeta$, in order to be as precise as possible. The situation of the δ-reduction, more complex, will be evoked together with the extraction of type constants. All these reductions are obviously dangerous for the efficiency of the extraction function, but fortunately, in practical, extraction times remain reasonable, even on consequent examples.

One will thus consider any type in its head normal form, which has the following shape: $\forall \overrightarrow{x_i : X_i}, (t\ \overrightarrow{a_j})$. It was already seen how to treat the head products. The extraction of what remains is done according to the structure of the head t, which can be:

- a s sort (which is then without argument)
- a constant c
- an inductive type I
- a variable X
- a non-reducible `match`
- a non-reducible `fix`

If this head t is a sort, any term of type t is a type scheme, and its extraction gives $\Box$. In order to remain coherent with this extraction of terms, the t type must be extracted to the type $\Box$.

Let now us consider the case of an applied constant c. One easy solution is then to δ-reduce this constant. But this is not satisfactory because that leads to larger and less readable types. And in addition, this is not always feasible, since all the constants do not necessarily have a usable body. Consider for example the abstract constants in a module signature, or type axioms[6] We then distinguish three situations for constants:

1. The most favorable case is when this constant is a type scheme. We then want, as for `(Sch1 nat)`, to translate the arguments and to return the application of the Ocaml types. But nothing forces the arguments to be types:

    - If an argument is a term, the approximation by $\mathbb{T}$ is mandatory.
    - Si now an argument `a` is a type scheme waiting n arguments, one can try to see it as a true type gathering all the possible evolutions of `a`. For that we apply n times $\mathbb{T}$ to this type scheme, except that in practice, instead of leave these $\mathbb{T}$ arguments, we reduce: this is equivalent to removing the n head lambdas[7] of `a`, and replace with $\mathbb{T}$ the variables created by these lambdas.

2. If now we are in the unusual situation where a type includes at the head a constant which is not a type scheme, there is little chance to translate this constant into a type constant. The best approach is then to reduce the constant, if possible, and then to extract the reduced version. The usual example (though not very realistic) illustrating this situation is the identity.

[6] In practice the current extraction allows realization of informative axioms by manually provided code. But this provided code is not analyzed by the extraction (cf section 4.4.2).

[7] If there are less than n lambdas at the head of `a`, we adds some via η-expansion

```
Definition id := fun (X:Type)(x:X) ⇒ x.
```

It is clear that to translate `(id Set nat)` into $\mathbb{T}$ only because the head constant is not a type scheme is too coarse, since a step of δ-reduction leads to `nat`.

3. Finally for the constants not type schemes and non-reducible, the last solution is the approximation by $\mathbb{T}$.

The case of an applied inductive type is similar to the one of a applied constant, just simpler since an inductive I has by construction a type of the form $\forall a : A, \ldots \forall z : Z,\ s$. We thus leave always the inductive type at the head, and we just need to extract the arguments as previously.

Concerning variables, everything has already been said, depending of the origin of this variable. If this variable comes from a dependent product, the variable and its arguments become $\mathbb{T}$ since the dependency disappears in Ocaml. If this variable comes from the parameters of a type scheme or an inductive type, it then gives an Ocaml type variable, its arguments being ignored.

Concerning the last cases not yet evoked for a type, namely the non-reducible types built with `match` or `fix`, they are too much complex for Ocaml, and are thus translated into $\mathbb{T}$.

3.3.5 The function $\widehat{\mathcal{E}}$ of extraction of types

We now describe again, in a formal way, the different situations mentioned up to now.

Definition 16 *The function $\widehat{\mathcal{E}}$ of extraction of types, from* CIC *to Ocaml types, is defined in a mutually recursive way. It uses a set v of type variables to be translated, noted in index and initially empty.*

Let us start with the types themselves, i.e. the Coq terms accepting a sort as type. The first case relates to the logical parts.

(prop) if U is of type `Prop`, then $\widehat{\mathcal{E}}_v(U) = \Box$.

The other cases are done by case on the head of the type after $\beta\iota\zeta$-reduction:

(sort) $\widehat{\mathcal{E}}_v(s) = \Box$

(prod1) if $\widehat{\mathcal{E}}_v(B) = \Box$, then $\widehat{\mathcal{E}}_v(\forall x : A,\ B) = \Box$

(prod2) if $\widehat{\mathcal{E}}_v(B) \neq \Box$, then $\widehat{\mathcal{E}}_v(\forall x : A,\ B) = \widehat{\mathcal{E}}_v(A) \to \widehat{\mathcal{E}}_v(B)$

(case) $\widehat{\mathcal{E}}_v(\texttt{case}(\ldots,\ \ldots,\ \ldots)\ \overrightarrow{a_i}) = \mathbb{T}$

(fix) $\widehat{\mathcal{E}}_v((\texttt{fix}\ \ldots)\ \overrightarrow{a_i}) = \mathbb{T}$

(var1) if $X \in v$, then $\widehat{\mathcal{E}}_v(X\ \overrightarrow{a_i}) = {}'X$

(var2) if $X \notin v$, then $\widehat{\mathcal{E}}_v(X\ \overrightarrow{a_i}) = \mathbb{T}$

(ind1) if I is an inductive type, we pose[8]: $\widehat{\mathcal{E}}_v(I\ a_1\ \ldots\ a_n) = (\widehat{\mathcal{E}}_v(a_1), \ldots, \widehat{\mathcal{E}}_v(a_n))\ I$

[8] we use as output the postfix syntax of Ocaml for the type applications

(cst1) *if the constant c is a type scheme, then:*
$\widehat{\mathcal{E}}_v(c\ a_1\ \ldots\ a_n) = (\widehat{\mathcal{E}}_v(a_1), \ldots, \widehat{\mathcal{E}}_v(a_n))\ c$

(cst2) *Otherwise, and if the constant can be reduced, we do it:*
if $(c\ \overrightarrow{a_i}) \rightarrow_\delta U$ *then* $\widehat{\mathcal{E}}_v(c\ \overrightarrow{a_i}) = \widehat{\mathcal{E}}_v(U)$

(cst3) *Finally in the case of a non-reducible constant that is not a type scheme, one uses an ultimate approximation:* $\widehat{\mathcal{E}}_v(c\ \overrightarrow{a_i}) = \mathbb{T}$

The recursive calls of $\widehat{\mathcal{E}}$ *on the arguments of inductive or constant type do not necessarily apply only to* Coq *types. We then extend* $\widehat{\mathcal{E}}$ *in the following way:*

(sch) *If* U *is a type scheme, one can thus write it modulo* η*-expansion in the form:* $\lambda a : A, \ldots\lambda z : Z, V$*, with* V *being a type. We then pose:*
$\widehat{\mathcal{E}}_v(U) = \widehat{\mathcal{E}}_v(V)$*, by thus ignoring the variables.*

(term) *If* U *is not a type scheme, then* $\widehat{\mathcal{E}}_v(U) = \mathbb{T}$.

Finally, the extraction of an environment of Coq *declarations is done as follows:*

(nil) $\widehat{\mathcal{E}}([]) = []$

(def1) *For the declaration of a type scheme c whose body can be written* $t =_\eta \lambda a : A, \ldots\lambda z : Z, U$*, we pose* $v = \{a, \ldots, z\}$ *and*
$\widehat{\mathcal{E}}(\Gamma; (c := t : T)) = \widehat{\mathcal{E}}(\Gamma); (\texttt{type}\ ('a, \ldots, 'z)\ c = \widehat{\mathcal{E}}_v(U))$

(def2) *For the declaration of a constant c that is not a type scheme, we do not produce anything:* $\widehat{\mathcal{E}}(\Gamma; (c := t : T)) = \widehat{\mathcal{E}}(\Gamma)$

(ax1) *For an axiom whose type* T *is of the form* $\forall a : A, \ldots\forall z : Z, s$ *we produce a declaration of abstract type:*
$\widehat{\mathcal{E}}(\Gamma; (ax : T)) = \widehat{\mathcal{E}}(\Gamma); (\texttt{type}\ ('a, \ldots, 'z)\ ax)$

(ax2) *For the other axioms we do not produce anything:* $\widehat{\mathcal{E}}(\Gamma; (ax : T)) = \widehat{\mathcal{E}}(\Gamma)$

(ind2) *For an inductive type* I *having* n *parameters* $\overrightarrow{p_i}$ *and whose constructors are* $\overrightarrow{C_i}$*, we pose:*

$$\widehat{\mathcal{E}}(\Gamma; \texttt{Ind}_n((I : \forall\overrightarrow{p_i : P_i}, \forall\overrightarrow{x_i : X_i}, s)\ :=\ (\overrightarrow{C_i : T_i})))$$
$$= \widehat{\mathcal{E}}(\Gamma); (\texttt{type}\ (\overrightarrow{'p_i}, \overrightarrow{'x_i})\ I\ \texttt{=}\ C_1\ \texttt{of}\ \pi(\widehat{\mathcal{E}}_v(T_1))\ |\ \ldots\ |\ C_n\ \texttt{of}\ \pi(\widehat{\mathcal{E}}_v(T_n)))$$

with $v = \{p_1, \ldots, p_n\}$
$\pi(\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau) = \tau_1 * \ldots * \tau_n$

This algorithm terminates even if it uses the Coq reductions, thanks to the strong normalization of CIC. It should also be noted that it produces well-formed Ocaml types. In particular each application of inductive or constant type is done with the good number of arguments. Let us take for example the case of a constant type $c : \forall\overrightarrow{x : X}, s$. When we extract a Coq type having c at its head, we knows that c has exactly n arguments, since it is the only way to satisfy the condition that $(c\ \overrightarrow{a_i})$ should have a sort as type. And each argument a_i will give an extracted argument $\widehat{\mathcal{E}}(a_i)$ in Ocaml. In addition, the number of type variables in the declaration of c in Ocaml is indeed n. The case of inductive type is similar.

3.4 Extraction, typing and correctness

As explained previously, the typing of extracted terms we ensure by the action of our algorithm $\mathcal{M}$' does not brings by itself any correctness property for the execution of these terms, because of the presence of the `Obj.magic`. On the other hand a "raw" extracted term $\mathcal{E}(t)$ is modified during the typing stage only via this insertion of `Obj.magic`. However these `Obj.magic` do not have any influence on the execution of a term: from the point of view of untyped λ-calculus, they are only identity functions. The results of correctness of the previous chapter thus still apply to the final well-typed extracted term.

3.5 Differences with the implemented extraction of types

For reasons of simplicity and concision, the $\widehat{\mathcal{E}}$ definition is incomplete. The mutual inductive types, for example, although not evoked here, are treated without more problems. And concerning co-inductive type, the section 4.2 is dedicated to their study. A certain number of particular cases are also distinguished in the implementation, like the empty inductive types, the singleton inductive types and the records. These cases will be evoked in section 4.3 about optimizations.

3.5.1 Dealing with type parameters

One difference between the actual implementation and the previous description of $\widehat{\mathcal{E}}$ concerns the type parameters and type arguments We have seen that the arguments which are not type schemes are translated in $\mathbb{T}$. One can in fact remove completely those arguments, because they can always be identified, and only thanks to their typing. That corresponds to the rules (ind1) and (cst1). Obviously, in order to remain coherent, one must then also filter the variables from the types we generate, according to whether they correspond or not to type schemes. The rules (def1) (ax1) and (ind2) should then be re-examined. It is this optimized version which has been implemented. The previous example of lists of size `n` is thus extracted to:

```
type 'a listn = Niln | Consn of nat * 'a * 'a listn
```

This corresponds to the usual lists, put besides this `nat` argument witness of the size of the list, which must be preserved.

3.5.2 Dealing with the nonparametric arguments of inductive types

Then, it should be noted that for simplicity reasons, the function $\widehat{\mathcal{E}}$ manages correctly only the parameters of the inductive types. Let us take again our example, the inductive type `list2` previously defined:

```
Inductive list2 : ∀A:Set,Type :=
```

.../...

.../...

```
  | nil2 : ∀A:Set, list2 A
  | cons2 : ∀A:Set, A → list2 (A*A) → list2 A.
```

If $\widehat{\mathcal{E}}$ is followed scrupulously, the extraction of `list2` is the one desired, but rather:

```
type 'a list2 = Nil2 of 𝕋 | Cons2 of 𝕋 * 𝕋 * (𝕋 * 𝕋) list2
```

In fact, $\widehat{\mathcal{E}}$ does not establish a link between the product $\forall A$: `Set`, ... in the type of the inductive and the products $\forall A$: `Set`, ... in the constructors types. The first gives a `'a` whereas the others give $\mathbb{T}$. In practice, the extraction of types currently implemented tries as much as possible to locate these "pseudo-parameters", and on this example `list2` give:

```
type 'a list2 = Nil2 | Cons2 of 'a * ('a * 'a) list2
```

3.5.3 Reduction of some types constants

Another difference between $\widehat{\mathcal{E}}$ described above and the implementation relates to the type constants. The rule (cst1) above always translates an applied type scheme constant into the same applied constant. We have seen in the example `Sch3` that `(Sch3 P)` gave in Ocaml the type `p sch3`, which is in fact $\mathbb{T} \to \mathbb{T}$ when reduced. But if we δ-reduces `(Sch3 P)` at the Coq level, we obtains `nat → bool`, which is extracted to itself, and a more precise extraction than before is thus obtained The currently implemented strategy is to test the two. Let us consider a Coq type t which δ-reduces to u:

- if the δ-reduced form of $\widehat{\mathcal{E}}(t)$ in Ocaml is equal to $\widehat{\mathcal{E}}(u)$, owe keep the most compact version, namely $\widehat{\mathcal{E}}(t)$.
- if not, the most precise version is kept, even if it is not the most compact, namely $\widehat{\mathcal{E}}(u)$.

3.5.4 Special treatment of head products

Finally, a last optimization of implementation deals with the head products. Indeed, the approximation of all variables of products by $\mathbb{T}$ is sometimes too extreme. For example, the type of the polymorphic identity `id` is not the expected one:

```
Ê(∀X:Set, X → X) = 𝕋 → 𝕋 → 𝕋
```

instead of $\mathbb{T}\to$ `'x → 'x`. This is partly corrected in the implementation:

- In the case of an extraction of type $\widehat{\mathcal{E}}(T)$, in order to determine the type of a extracted term $t : T$, we authorizes the head products of T to generate type variables. This allows to obtain finer types for the extracted terms In particular, the type of `id` is indeed as expected.
- The other situation is that of an extraction of type $\widehat{\mathcal{E}}(T)$ performed in order to transform a Coq type declaration into a Ocaml type declaration. In this case, we generates no variable, to respect the following principle of the type schemes: as many type variables as head lambdas.

Chapitre 4

Extraction in practice: refinements and implementation

In this chapter, we describe the current state of the extraction such as it is implemented in version 8.0 of Coq. This description consists of two parts.

First of all, we present a certain number of features implemented in the extraction tool, but not yet evoked up to now. These features are presented only in an informal way, because their theoretical study is for the moment not also complete that the one of the $\mathcal{E}$ function of chapter 2. These refinements are:

- the support of the new modules system of Coq,
- the support of co-inductive types,
- the addition of several optimizations intended to improve the efficiency of the extracted code.

In a second time, we present the implementation which was done during this thesis and describe briefly its internal behavior and then detail its usage from the point of view of the user.

4.1 Extraction of the new Coq modules

During this thesis, a significant innovation appeared in Coq, and we adapted the extraction to it. This new Coq feature is the new module system, that allows to structure the developments in a dramatically more flexible way, as well as do some abstract reasoning and re-use codes and/or proofs. From the point of view of extraction, this module system opens new prospects in term of proof of programs, by facilitating the design of autonomous certified elements. Whether they are modules or functors, these certified parts can then be easily assembled via theirs interfaces with other parts, certified or not, in order to constitute one application of greater scale. The extraction of modules extends in fact the effort described in the previous chapter aiming at obtaining a interface `.mli` for any extracted file `.ml`. From now on, all structures extracted, modules and functors, will have their interfaces.

The development that we present later in chapter 7 constitutes indeed a first stone towards the construction of a library of certified modules, usable by any programmer, whether he wishes to build an completely certified application, or quite simply to re-use building blocks of good quality. In this development about the structures of finite sets, we certify several implementations of functors having the same interface, which allows any application built on this interface to make its choice among various implementations, several being certified. In addition, we implemented many derived properties starting from the Coq interface of finite sets. These properties are thus automatically shared by all the implementations of this interface, allowing, let us hope, to easily certify programs using this library of finite sets.

Before presenting these new Coq modules and their extraction, we start first with a rapid overview of the module system of Ocaml, which inspired deeply the one of Coq.

4.1.1 The Ocaml modules

The module system of Ocaml [52] is itself derived from the original system of SML [42]. The latter has also evolved/changed in return, and the two systems are now very close and known under the name of Harper-Lillibridge-Leroy module system. One can find a complete presentation of this system in the Ocaml reference manual [53]. To be short, one can say that this module system adds three kinds of structures above the basic declarations of the language for types and constants:

- A first additional structure is named *module*. It allows to group several declarations which can correspond to values, types or possibly new modular substructures.

```
module OneModule = struct
 type 'a oneType = OneConstructor of 'a
 let oneFunction = function OneConstructor x → x
end
```

  The significant point is that the internal declarations inside the module share the same name scope, distinct from the external name scope. We then access the internal objects via the qualified notation, for example here `OneModule.oneType`.

- The *signatures* or interfaces form a second kind of structures. They also consist of a group of declarations, but these declarations relate only to types. These signatures allow to type the preceding modules. For example, if we enter in Ocaml the declaration of the module `OneModule`, the system returns us a corresponding signature:

```
# module OneModule = struct
   type 'a oneType = OneConstructor of 'a
   let oneFunction = function OneConstructor x → x
  end;;
module OneModule :
 sig
   type 'a oneType = OneConstructor of 'a
```
.../...

.../...

```
  val oneFunction : 'a oneType → 'a
 end
```

As previously the module bodies were delimited by the keywords `struct...end`, here a signature is introduced by `sig...end`. In the particular case of the preceding signature, inferred by Ocaml, we see that each type declaration has been left unchanged, whereas any value declaration is translated into a `val` construct specifying its type. This signature is thus the most precise specification possible. But other signatures are also acceptable for typing our module:

- Compared to the most precise signature, one can remove certain declarations. This way, one obtains local objects inside the module, inaccessible from outside.
- One can also hide the contents of certain types, in order to get *abstract* types, whose objects can only be handled via the primitives provided in the module. Here:

```
module type OneRestrictedSignature = sig
 type 'a oneType
 val oneFunction : 'a oneType → 'a
end
```

- Lastly, the *functors* are generalizations of modules. Roughly speaking, a functor is a function taking module(s) as argument(s) and manufacturing a new module:

```
module OneFunctor = functor (M:OneRestrictedSignature) → struct
 type 'a oneOtherType = ('a M.oneType) list
 let oneOtherFunction l = List.map M.oneFunction l
end
```

The body of this functor is thus parameterized with respect to a module variable `M`. And any module admitting the signature `OneRestrictedSignature` as type can then be applied to `OneFunctor`, be substituted for this module variable `M` and then give a new module:

```
# module OneNewModule = OneFunctor(OneModule);;
module OneNewModule :
 sig
  type 'a oneOtherType = 'a OneModule.oneType list
  val oneOtherFunction : 'a OneModule.oneType list → 'a list
 end
```

To conclude this short presentation of Ocaml modules, we obviously need to mention that these Ocaml modules present several other subtleties that we do not describe here, such as for example the `with` construct in the types of modules. Lastly, for more advanced examples of module usage, one can consult the chapter 7 about a formalization of finite sets by the means of modules.

4.1.2 The Coq modules

A proper module system for Coq has long been a desired but missing feature. Previously, some existing tools were intended to organize a development in a modular way, such as for example the splitting into several files, the mechanism of `Section` within a file or the use of axioms. But these methods were quickly reaching theirs limits. For example, starting from a development parameterized by an axiom, the only way of re-using this development in a concrete case was to duplicate its code and to replace the axiom by the concrete definition. One finds this process for example in the C-CoRN project (see chapter 6) for the real number structure.

The first study of a module system for Coq was made by J. Courant in its thesis [22]. But his system, more ambitious than the one of Ocaml, seemed to be too complex for being implemented over the existing Coq code. Instead, a system à la Ocaml was finally implemented by J. Chrząszcz [18, 19], taking as a starting point the the modular module system of X. Leroy [52].

In Coq, we can now find modules, signatures and functors. For example, an equivalent of our preceding toy module can be written this way in Coq :

```
Module OneModule.
 Inductive oneType (A:Set) : Set := OneConstructor : A → oneType A.
 Definition oneFunction (A:Set)(a:oneType A) : A :=
  match a with OneConstructor x ⇒ x end.
End OneModule.
```

Unlike Ocaml, the Coq modules are *interactive*. Whereas in Ocaml, a module must be provided in one whole declaration, Coq allows to build it step by step, by successive declarations between the starting command (`Module OneModule`) and the ending one (`End OneModule`). This need for interactivity rises from the possible presence of theorems, and thus of proof, among the declarations present in one module. Except for the lucky ones whose native language is λ-calculus, it is then illusory to try to directly give a not-trivial proof. This difference, even if significant from the user's point of view, does not influence the extraction, which always proceeds after the end of the module declarations.

The signatures and the functors are defined in the same manner:

```
Module Type OneSignature.
 Parameter oneType : Set → Set.
 Parameter oneFunction : ∀A:Set,(oneType A) → A.
End OneSignature.

Module OneFunctor (M:OneSignature).
 Definition oneOtherType := fun A ⇒ list (M.oneType A).
 Definition oneOtherFunction := fun A l ⇒ List.map (M.oneFunction A) l.
End OneFunctor.
```

Concerning the contents of a signature, one moves away somewhat from the parallel with Ocaml signatures, because of the differences between the two systems, and in particular the lack of distinction between terms and types in Coq. Let us recall that in Ocaml a term is necessarily abstract because of the declaration `val`, whereas a `type` declaration can be abstract or not.

- For an abstract declarations in a Coq interface, the generic keyword[1] is `Parameter`, and can be used for a function like `oneFunction` or a type `oneType`.
- Writing a concrete type in a interface Coq leads no surprise: we can still declare a new inductive type via `Inductive`, or define an alias via `Definition`.
- the principal innovation is the possibility of placing one concrete term in an interface. Thus, `Definition x:=0` in a interface will oblige any implementation of this interface to contain a constant `x` being worth `0`.

The use of Coq interfaces are made delicate by two practical details. First of all, only a type `oneType` defined as a constant (via `Definition` for example) is accepted for realization of the parameter `oneType`, and *not* a type defined by `Inductive`. Thus, for Coq, `OneSignature` is *not* a valid signature of `OneModule`. The solution is then to place an alias on the level of the module `OneModule`: we rename `oneType` in `oneType'` in the inductive definition, then the following declaration is added:

```
Definition oneType := oneType'.
```

In addition, it should be noted that if a signature is imposed to our module (for example via `Module OneModule : OneSignature`), and if this signature contains abstract declarations `Parameter`, then the body of our function `oneFunction` is hidden: one can never again make use of it out of the module, for example during a computation. That semantics differs from Ocaml : the values of the constants are actually hidden in the interfaces by the construction `val`, but we obviously use these values during the execution of program! The Coq typing relation "`:`" between modules and interface is then very constraining, and we often prefer to use instead the weaker form "`<:`", that only checks if the interface *could* be a valid signature for this module, but without setting up the associated restrictions.

Just as for the presentation of the Ocaml modules, much remains to be said concerning the advanced possibilities of the Coq modules. And once again, the chapter 7 presents some more realistic examples using these modules and functors.

4.1.3 The extraction of modules

Let us now see what becomes the extraction in presence of these Coq modules. First of all, let us announce that only the extraction to Ocaml can currently treat the Coq modules. Indeed, there is no direct equivalent in Haskell of such a module system, but rather a system of *classes*, a rather different concept. Perhaps it is possible to use these classes to express the extracted Coq modules, but the feasibility of this translation was not explored.

We now consider the extraction of the Coq modules towards Ocaml. At first glance, everything works simply: a Coq signature becomes a Ocaml signature, a Coq module becomes a Ocaml module and similarly for functors.

[1] In fact, `Axiom` is also appropriate, it is even a synonym here.

When we want to go further than these obvious points, the delicate point is the translation of the internal declarations inside these modular structures. Indeed one wish to preserve during the extraction the typing relations between modules and signatures. To translate a Coq declaration in a corresponding Ocaml declaration, we use the extraction functions $\mathcal{E}$ and $\widehat{\mathcal{E}}$ of chapters 2 and 3, which respectively produces terms and types.

We now detail the various possible situations. In fact, there are only four types of possible declarations in Coq :

- the declaration of inductive via `Inductive`.
- the declaration of a constant, for example via `Definition`. All the declarations `Lemma`, `Theorem`, `Fixpoint` are indeed only variants of `Definition` allowing an easier build for the body of the constant being defined.
- the specification of an object in a signature via `Parameter`.
- the declaration of an axiom in a module, via `Axiom`. We ignore this case for the moment, see section 4.4.2 for the treatment of axioms by the extraction.

The first two cases can occur both inside a module (or functor) or inside a signature.

First, all declarations of objects of sort `Prop`, both inductive types and constants, will be purely and simply removed by the extraction. Indeed no extracted declaration can later depend on such a logical declaration. If we take for example the declaration of a constant logical `c` of sort `Prop`, any following occurrence of `c` will be placed by function $\mathcal{E}$ under a constant $\square$. And if `c` appears in a term which is a type and is thus treated by the function $\widehat{\mathcal{E}}$, then any reference to `c` also disappears, because `c` is not a type. As for a logical inductive type `I`, the function $\mathcal{E}$ make it disappear like all types, and the function $\widehat{\mathcal{E}}$ transforms it into a degenerated type $\square$.

Then, a simple case is the declaration of an informative inductive type. Indeed, we then quite simply output the associated declaration generated by the function $\widehat{\mathcal{E}}$. And this is valid both in a signature or in a module.

Let us now consider an informative declaration `Parameter x:T` in a signature, which thus claims the existence of an object `x` of type `T` in any module fulfilling this signature. Ocaml, unlike Coq, distinguishes types and values. This declaration can thus correspond in Ocaml either to the declaration of a value `val x:`$\widehat{\mathcal{E}}$`(T)`, or either to the declaration of an abstract type `type X`. And this choice is done according to the shape of `T`: if `T` is a type like `nat`, this leads to declaration of a constant `val x:nat`. On the contrary, if `T` is a sort, or more generally an arity, `x` is a type (resp. a type scheme), and its natural translation is a type declaration in Ocaml.

The last case concerns the declaration of a constant, typically via `Parameter x:T:=t`. We then re-use the distinction between type schemes and other terms. For the former, we generate a type declaration by using the function $\widehat{\mathcal{E}}$, and return `type X = `$\widehat{\mathcal{E}}$`(t)`. For the latter, we generate a declaration of an Ocaml term. And this declaration is either a concrete form if we are inside a module or a functor, namely `let x = `$\mathcal{E}(t)$, or either an abstract form `val x : `$\widehat{\mathcal{E}}$`(T)` if we are in a signature.

We can summarize all these possible situations in the following table:

	Prop sort	type scheme	standard case
`Inductive i := C:T`	∅	//////////////	`type i = C of` $\widehat{\mathcal{E}}$`(T)`
`Definition x:T:=t`	∅	`type X =` $\widehat{\mathcal{E}}$`(t)`	Module: `let x =` $\mathcal{E}$`(t)` Signature: `val x :` $\widehat{\mathcal{E}}$`(T)`
`Parameter x:T`	∅	`type x`	`val x :` $\widehat{\mathcal{E}}$`(T)`

Even if this summary table does not show it in order to remain simple, all Ocaml type declarations, both those coming from inductive types and type schemes, can contain type variables `'a`, as we have seen in chapter 3.

Once described this "shunting" of the Coq objects towards either Ocaml terms or types, one must then check that this shunting is quite coherent. In fact, for any Coq object named `c` who finds himself placed on the term level by the extraction, all the uses of `c` in the objects extracted later on must also be on the term level. And the same for an object placed at the type level. This property is not completely immediate. Let us take for example the polymorphic identity:

```
Definition id := fun (X:Type)(x:X) ⇒ x.
```

As it is not a type scheme, our extraction chooses to transform it into an Ocaml function:

```
let id _ x = x
```

But `id` can certainly be also used in a Coq type:

```
Definition nat_bis := id Set nat.
```

A possible extraction of this type `nat_bis` would be then the application `nat id`, with `id` referring here to a type constant `id` which could be defined by:

```
type 'x id = 'x
```

However our extraction does not define this type constant, but only the constant `id` at the term level. In fact everything works correctly here, thanks to the rules chosen for the extraction $\widehat{\mathcal{E}}$ of types in section 3.3.5. Indeed, the extraction $\widehat{\mathcal{E}}$`(nat_bis)` is not `nat id`, but `nat`, because `id` is not a type scheme. This constant `id` is thus reduced before extracting, according to the rule (cst2) defining $\widehat{\mathcal{E}}$.

More generally, our "shunting" is coherent with both functions $\mathcal{E}$ and $\widehat{\mathcal{E}}$ because these three operations divide with respect to the same stratification criterion, namely the fact of being or not a type scheme. Thus a type constant `c` cannot appear in a extracted term: since it is a type scheme, it ends under a □. On the opposite, a term constant as `id` cannot remain in an extracted type according to the rules (cst1) (cst2) and (cst3) of $\widehat{\mathcal{E}}$.

4.1.4 Current limitations of the extraction of modules

This extraction of the modules Coq is still to be considered as experimental. First of all, a thorough theoretical study has not been done yet for lack of time. This would suppose in particular the integration in our theoretical system (chapters 1 and 2) of the typing rules specific to Coq modules. However these rules currently occupy 6 pages in the Reference Manual [78] (chapter 5).

The implementation, which is thus more advanced than the theory concerning these modules, gives in practice satisfactory results on the first real developments using these modules. One can for example refer to the chapter 7 for a case study about finite sets. But we have in same time identified two extreme situations which can put the current code at fault.

A first typing problem

As we mentioned previously, an essential property of the extraction of modules is the conservation of the typing relations between a module and a signature during the extraction. This is for example critical in order for a functor application to remain legal after extraction. At first sight, this property seem to be a simple consequence of the following result connecting the functions $\mathcal{E}$ and $\widehat{\mathcal{E}}$:

$$t : T \Rightarrow \mathcal{E}(t) : \widehat{\mathcal{E}}(T)$$

Unfortunately, this result speaks only about concrete types, and the presence of abstract types in a signature can disturb this situation. In fact, it is currently possible to build a module Mod and a signature Sig contradicting our expected conservation property for module typing. For that, it is enough to combine abstract types and type schemes whose extraction will be an approximation:

```
Module Type Sig.
 Parameter t : nat → Set.
 Parameter x : t 0.
End Sig.

Module Mod : Sig.
 Definition t := F.
 Definition x := 0.
End Mod.
```

The type scheme F comes from page 91: (F N) is the type of the integer functions of arity n, and thus (F 0) is the type of the integer functions with 0 arguments, that is nat. However the extraction of Mod and Sig gives:

```
module type Sig =
 sig
  type t
  val x : t
 end

module Mod =
 struct
```

.../...

.../...

```
  type t = 𝕋
  let x = O
 end
```

Indeed, the type scheme `F` defined by fixpoint gives the most general extracted type $\mathbb{T}$, and at the abstract level `t O` is approximated in `t` since one cannot reduce the abstract type `t`. It is then visible that after extraction, `Sig` is not any more a valid signature for `Mod`.

To correct this problem, one could imagine to make $\mathbb{T}$ really become an universal type for Ocaml, and require that `t:`$\mathbb{T}$ for any term Ocaml `t`, and in particular `O:`$\mathbb{T}$. But this would suppose a modification of the type system of Ocaml. Otherwise, one solution is here to insert a `Obj.magic` around the `O` in `Mod`. The problem with this insertion of `Obj.magic` is that it is no longer done according to a local analysis (the type of `x` inside the module), but according to a global analysis, for example knowing if `Mod` is used later on with the signature `Sig` in an functor application. One could also use a encapsulation at the time of the functor application, like:

```
module Mod_bis =
 include Mod
 let x = Obj.magic x
end
```

None of these solutions have been implemented for the moment. Anyway, it is time to moderate the importance of this problem. It is certainly one exception in our goal of the "100% of Coq constructions extractable in a well-typed way". But until now, no realistic development has felt into this precise case, namely the combination of abstract types and type schemes in a signature. And even if the currently low number of such modular developments makes an extrapolation difficult, it is really far from probable that the problem will occur one day in practice.

A second typing problem

In fact it is also possible to induce typing problems between modules and signature by using the cumulativity :

```
Module Type Sig2.
 Parameter t:Type.
 Parameter x:t.
End.

Module Mod2 : Sig2.
 Definition t:Prop:=True.
 Definition x:True:=I.
End.
```

The signature is naturally translated into:

```
module type Sig2 =
 sig
  type t
  val x : t
 end
```

On the other hand the extracted module Mod2 is empty, because Mod2.t and Mod2.x are respectively a type and a logical value. And we made previously the choice of removing such declarations that are never useful in later declarations. However, here, the typing of Mod2 by Sig2 requires the presence of fields t and x in Mod2. But this typing problem is less awkward than it first appears. After all, nothing prevents us from reconsidering our choice and stopping the complete elimination of logical declarations, for getting here:

```
module Mod2 =
 struct
  type t = 𝕋
  let x = 𝕋
 end
```

But the extraction would then leave quantity of parasite logical declarations, never useful except in the event of a cumulativity use between a module and its signature. It is then better to continue to purge modules and signatures from the logical declarations and to cure the highly exceptional situations of cumulativity via a encapsulation similar to solution of the previous problem. Moreover, the correctness of these two situations with problems will be studied without doubt in a joint way.

In fact, our preoccupations about module typing present large similarity with the typing problems on the term level. In the same way that we need untyped coercion functions Obj.magic, it would be necessary to have coercion functions at the module level, for example at the time of a functor application to a module. But unlike for terms, such functions do not exist at the module level.

Let us finish nonetheless this discussion about typing of extracted modules by two positive remarks:

- The current extraction fulfills at least a weak form of conservation for the module typing: if a module M admits S as its most general signature (the one inferred by the system), then the extraction of M still admits the extraction of S as its most general signature. And more generally, in the absence of abstract types and of cumulativity between module and signature, then everything works well.
- Pragmatically, if the Ocaml type-checker is satisfied by the result of an extraction containing modular structures, then all is for best. In particular the preceding correctness results for the execution or the semantics of an extracted term are still valid. After all, modules and functors are only a way of re-using code. And since the functors applications are known statically, one can obtain equivalent code with no functors thanks to a process named defunctorization. And this process has been in particular implemented for Ocaml by J. Signoles [77].

A naming problem

Another problem, syntactic this time, can also lead to extracted modules being refused by Ocaml. Coq, with its interactive modules, is more tolerant than Ocaml concerning the possibilities of object naming. In particular one can refer to a module even during its built, whereas this is illegal in Ocaml. The following example is valid in Coq :

```
Module M.
 Definition t := 0.
 Module N.
  Definition t := 1.
  Definition u := M.t
 End N.
End M.
```

On the other hand in Ocaml one cannot use the qualified name `M.t` inside `M`, and the simple name `t` is incorrect due to the presence in the local name space of the `t` corresponding to `N.t`. Since renaming can be difficult because of the possible signatures to respect, a reasonable solution, proposed by J. Signoles, is then to add a local renaming module:

```
module M =
 struct
  let t = O
  module AdHoc = struct let t = t end
  module N =
   struct
    let t = S O
    let u = AdHoc.t
   end
 end
```

Detecting the need for such modules and adding them properly in all the possible cases (terms and types) is very complex and is not implemented yet. But as for the preceding problems, it is a priori far from probable to fall into such a situation in a realistic development.

4.2 Co-inductive types and extraction

4.2.1 From inductive types to co-inductive types

The inductive types that we handled up to now are designed to only deal with *finite* objects. More precisely these objects cannot contain more than a finite number of constructors of this type[2]. This finiteness, or good foundation, implies the existence of induction

[2]By the way, one can note that an Ocaml inductive type like `list` do *not* fulfill the same property of finiteness as the type `list` of Coq. Indeed, Ocaml authorizes infinite cyclic objects like `let rec l = 0::l`. And obviously, if one applies this list to a extracted function like `map`, computation will never end. This

principles associated with each of these inductive types. These induction principles are in fact generated automatically by Coq, such as for example nat_rec for nat. Now, thanks to the works of E. Giménez, there also exist in Coq some types similar to the inductive types, but for which there is no finiteness constraint. These are the co-inductive types (see for example [37]). The typical example of such a type is the stream type:

```
CoInductive Stream (A:Set) : Set := Cons : A → Stream A → Stream A.
```

When an co-inductive type is defined, the first difference with declaration of an inductive type is that Coq cannot generate any associated induction principle, since these principles are not valid for infinite objects.

On an co-inductive object, the case analysis works as for an inductive object. Here is for example the way to reach the head of a stream:

```
Definition hd (A:Set) (x:Stream A) :=
 match x with
  | Cons a _ ⇒ a
 end.
```

On the other hand the induction is quite different in the co-inductive world. First of all, one speaks rather of co-induction, and the Coq keyword is CoFixpoint instead of Fixpoint[3]. Then there is no concept of decreasing argument as for the Fixpoint. One can thus write:

```
CoFixpoint from (n:nat) : Stream nat := Cons n (from (S n)).
```

Or, without any argument:

```
CoFixpoint zero_stream : Stream nat := Cons 0 zero_stream.
```

But any definition is necessarily authorized, because it is at the very least delicate to give a meaning to a definition such as:

```
CoFixpoint dummy : Stream nat := dummy.
```

The rule is to authorize only co-inductions which indeed build a new co-inductive object. More precisely, any co-recursive call must be located under at least a co-inductive constructor, and that is not the case for our dummy example.

Concerning the reduction of co-fixpoints, it also follows a particular rule, in order not to break the strong normalization property of the system. This reduction is *lazy*: one co-fixpoint can be unfolded and replaced by its body only if this co-fixpoint and its possible arguments appear in head position of a pattern matching. For example (from 0) is in normal form, whereas in (hd (from 0)) it is not, since hd contains a pattern matching. In the latter case, the from can unfold, which gives after simplification 0 for normal form.

does not invalidate the theoretical results of the chapter 2, since l cannot be put in correspondence via $\widehat{\texttt{list}}$ with a list l' of Coq.

[3] There exists also a cofix anonymous construct similar to the fix.

4.2.2 The extraction of co-inductive types

The case of Haskell

The co-inductive types form the first and only feature of Coq which was initially taken into account by the extraction towards Haskell before being integrated into the Ocaml extraction. The reason is of course the laziness of the Haskell evaluation, which makes obvious the extraction of co-inductive types towards this language. Thus, there is no differences between the extraction of the finite lists and that of the streams except the presence or not of the base case `Nil`.

```
data List a = Nil
            | Cons a (List a)

data Stream a = Cons a (Stream a)
```

Then the extraction of a co-fixpoint gives naturally a recursive function:

```
from n =
 Cons n (from (S n))

zero_stream =
 Cons O zero_stream
```

As long as `zero_stream` is not necessary to a later computation, this constant will never be unfolded. And even then, there will never be superfluous unfolding. Thus, if one asks the printing of `(hd (tl (tl zero_stream)))`, there will be only three unfoldings, leading to a result of `O`.

Historically, the extraction of co-inductive types towards Haskell was already usable in the old extraction. We only maintained this possibility. As an example, one can refer to the user's contribution named `Rocq/MUTUAL-EXCLUSION`. E. Giménez studies in this development the mutual exclusion of two processes via the Petersson's algorithm. A small graphical interface using the Fudgets library of Haskell allows to visualize the run of the algorithm in an interactive way.

The case of Ocaml

During the implementation of our new extraction, we have added the possibility of extracting from co-inductive type towards Ocaml. This is done via an encoding, because a direct and naive translation of our preceding examples would be incorrect, due to the strict evaluation of Ocaml. If one takes:

```
type 'a stream = Cons of 'a * 'a stream

let rec from n = Cons n (from (S n))
```

Then the computation of `(hd (from O))` starts an infinite loop of calls to `from`.

Fortunately, there exists in Ocaml a mechanism to introduce lazy objects. First, `(lazy x)` is a stopped version of the computation of the object `x`. And if this `x` has a type `a`, then `(lazy x)` has type `a Lazy.t`. Finally the function `Lazy.force: 'a Lazy.t → 'a` makes it possible to force the resumption of a computation.

We thus use this mechanism to encode the co-inductive types extracted from Coq. Each type `t` is in fact extracted into two types `t` and `__t`, mutually defined, the first being the type of objects on standby, and the second the type of freed objects. In the case of streams, that gives us:

```
type 'a stream = 'a __stream Lazy.t
and 'a __stream = Cons of 'a * 'a stream
```

Then we simulate the Coq reduction in the following way. A co-inductive constructor produces a frozen object, and is thus surrounded by the keyword `lazy`. On the other hand, a pattern matching on a co-inductive object needs to reach the head structure of this object. We thus force a level of computation by inserting a `Lazy.force` around the matched object.

This gives us the following Ocaml extraction for our examples:

```
let hd x = match Lazy.force x with
  | Cons (a,s) ⇒ a

let rec from n =
  lazy (Cons (n,(from (S n))))

let rec zero_stream =
  lazy (Cons (O,zero_stream))
```

This encoding is inspired by the style "even, with difficulty" presented in [81]. The naming "even" refers to the fact that any co-inductive constructor is associated to a `lazy`, thus doubling the number of syntactic constructions. And the "with difficulty" announces that this style is opposed to another style, simpler but being able to carry out superfluous evaluations of elements in a stream. But even this chosen style for extraction can pose problems of superfluous evaluation and thus of effectiveness, as we will see now.

co-inductive types, Ocaml and efficiency

The use of `lazy` and `Lazy.force` constructions by the extraction does nothing but delay the evaluation of co-inductive constructions. But this is not enough for transforming Ocaml into a completely lazy language. In particular, strict aspects of Ocaml can re-appear, in particular during the evaluation of function arguments, and lead to significant differences with the Haskell evaluation of our examples containing co-inductive types.

Let us consider for example a function `iter` which, starting from a function `f` and an initial point `a`, calculates the stream made up of `a`, `(f a)`, `(`f^2` a)`, etc.

```
CoFixpoint iter (A:Set)(f:A→A)(a:A): Stream A := Cons a (iter A f (f a)).
```

Its extraction is then:

```
let rec iter f a = lazy (Cons (a,(iter f (f a))))
```

But then asking for the evaluation of `(hd (iter f a))` leads to the superfluous evaluation of `(f a)`, even if the recursive call `(iter f (f a))` is indeed blocked by the `lazy` at the head of `iter`. This superfluous evaluation, not very natural, can be annoying if `f` leads to long computations.

At the same time, these superfluous evaluations cannot be considered as correctness problem for the extraction. Indeed, if the reduction done in Ocaml starts from `(hd (iter f a))`, one can simulate this reduction by a similar reduction at the Coq level, in a same way that for the theoretical results of the chapter 2. That means that Coq could also choose to normalize `(f a)` when reducing `(hd (iter f a))`. In practice Coq does not do it, since its strategy is rather lazy by default.

We thus enter, with this example, in the field of the efficiency questions for the extracted code. We detail this field in the following section, but we can already announce that very often this field has only imperfect solutions.

Here, in the current example, one can choose to move the `lazy` construct in order to block the evaluation of the arguments for the recursive call:

```
let rec iter f a = Cons (a,lazy (iter f (f a)))
```

The function `iter` does not build any more a `stream` but a `__stream`. But in addition to the small types adjustments that it implies, this solution corresponds to the "odd" style of the article [81]. And this style also suffers from problems of superfluous evaluation. One can also try to block the evaluation of `iter` at two places :

```
let rec iter f a = lazy (Cons (a,lazy (Lazy.force (iter f (f a)))))
```

In a more general way, it seems to be sometimes interesting to insert additional blocking points `lazy (Lazy.force (...))` in the extracted code around the sub-expressions of co-inductive types. This is not made automatically yet, but adding such blocking points manually can be made without risk, because this does not modify the correctness of the code.

A situation close to that of the preceding `iter` was met during the test of the Ocaml extraction upon the contribution `Rocq/MUTUAL-EXCLUSION`. A blocking point has been added via one diverted use of the command `Extract Constant`, which will be presented at the end of this chapter. The reader interested by more details can consult directly the files of this contribution.

Lastly, let us remark that for supporting co-fixpoint without arguments that does not start with a constructor, it would be necessary to insert a `lazy (Lazy.force (...))` around the body of this co-fixpoint. It's true that Ocaml accepts our fixpoint without argument `zero_stream`. But it will refuse more complex bodies, starting for example with one `if`. Hiding these overly complex bodies under a `lazy` would allow to circumvent the difficulty.

4.3 Extraction and code optimizations

We now describe a certain number of transformations that the extraction carries out on the extracted code in order to try to improve its efficiency, or sometimes simply its readability.

4.3.1 Removal of some logical arguments

The first of these transformations is intended to make disappear as much as possible the logical residues left by the extraction function $\mathcal{E}$ of chapter 2. These residues[4] take the form of anonymous abstractions `fun _ → ...` and constants □. We have already seen that in Haskell, □ can be implemented by an error (see page 81). On the other hand, in Ocaml, the possibility of having to reduce `□ x` into □ forces us to use a complex definition (see page 103):

```
let __ = let rec f _ = Obj.repr f in Obj.repr f
```

It is then obviously desirable to remove as much as possible these constants □, for both a better efficiency and readability of the extracted code.

Let us take the example of a division `div` with pre-condition, but without post-condition, whose Coq type is `∀a b:nat, b≠0 → nat`. The extraction seen until now produces a function `div` of type `nat → nat → □ → nat` . And any later use of this function to compute a division will have the form `(div a b □)`.

Now, if we must keep such a logical residue, this comes from the evaluation in Ocaml of the partial applications that could otherwise lead to abnormal situations (see 2.1). The logical residual argument of `div` allows here to ensure that `(div a b)` is a closure, blocked until it finds its third argument.

But the large majority of the applications met by the extraction are in fact total. It is thus preferable to treat these total applications of function as simply and naturally as possible, even if it means to weigh down the writing of partial applications.

Let us try then to return to our extracted function `div` its natural type `nat → nat → nat`. Concerning the definition of `div`, this does not pose any problem. Modulo η-expansion, one can indeed to suppose that this definition of `div` starts with `fun a b _ →...`. It is then enough to remove the third abstraction. Once this definition of `div` is simplified, it is obviously necessary to also adapt the later calls to this function:

- Let us first consider a total application `(div a b □)`. To adapt to our new purified function `div`, it is enough to to throw the third argument □. In this case, we find back the code produced by the old extraction: only informative arguments remain whereas the logical arguments disappear completely. And this is not done to the detriment of safety: if we unfolds `div` in this application before and after the transformation, it is seen that one goes from:

```
(fun a b _ → ...) a b □
```

  to the new form:

```
(fun a b → ...) a b
```

  These two forms are clearly equivalent from the point of view of evaluation.

[4] In fact, these residues can also come from type schemes and not only from logical parts, but we will merge here these two situations by simply speaking of "residues".

- now Let us now take in Coq a partial application `(div a0 b0)`, and thus an extraction producing originally a partial application like `(div a b)`. For using our new improved version of `div`, we must imperatively maintain the "blocked" aspect of this partial application. For that, we just adapt it in `(fun _ → (div a b))`. Once again, unfolding `div` shows that we have not modified the semantics of the application:

```
(fun a b _ → ...) a b
```

  becomes

```
(fun _ → ((fun a b → ...) a b))
```

  The only consequence is to delay the evaluation of `a` and `b`. One could imagine an even more faithful adaptation, namely:

```
let x = a and y = b in fun _ → (div x y)
```

  But in practice using this version has not seemed necessary.

- In the case of applications that are even more partial, like `(div a)` and finally `div` without any argument, we proceeds in the same way: `(div a)` becomes `(fun b _ → (div b))` and `div` only becomes `(fun a b _ → (div a b))`.

Of course, the method described here for `div` can be generalized to the transformation of any function declaration having an arbitrary number of arguments, and logical arguments also at arbitrary positions. Thus the declaration of a function `f` of type $\mathtt{t}_1 \to \ldots \to \mathtt{t}_n \to \mathtt{t}$ will be transform into the declaration of a function of type $\mathtt{t}_{i_1} \to \ldots \to \mathtt{t}_{i_p} \to \mathtt{t}$ where the $\mathtt{t}_{i_k}$ are the $\mathtt{t}_i$ different from $\square$.

The only exception to this method relate to the functions having only logical arguments, such as for example `False_rec` (see 2.1). Removing all the arguments and hence transforming the extraction of `False_rec` into a constant would not be correct, since here `False_rec` corresponds to an abnormal situation, and thus induces the raising of an exception via `assert false` in Ocaml. The following declaration:

```
let false_rec = assert false
```

would then stop the final program as soon as its launch. This problem is corrected by keeping always at least one argument to our functions, which gives us here:

```
let false_rec _ = assert false
```

The transformation presented here does not obviously allow to remove all logical residues. For example a pattern matching having some of its branches as informative and soon as logical will continue to produce $\square$ during the extraction. In addition, this method only acts at the first level: we remove or not some arguments of the treated functions, but we do not modify at all type of these arguments. Thus a function of type $\mathtt{nat} \to \square \to \mathtt{nat}$ will have as new type $\mathtt{nat} \to \mathtt{nat}$, but a function of type $(\mathtt{nat} \to \square \to \mathtt{nat}) \to \mathtt{nat}$ will remain unchanged.

Extending this transformation to allow the simplification of the type interior has been considered for one moment, then abandoned: this would indeed imply the use of increasingly complex term manipulations for each additional treated nesting level.

In practice, the currently implemented transformation, although limited, allows already to eliminate most of the logical residues. This new extraction is then very frequently compatible with the old extraction, even on quite complex examples, whilst still keeping its correctness property, even on the most pathological examples.

Let us finally announce that in Haskell this elimination could be much more advanced, because this language has no need for logical residues in order to block reductions. For the moment, this was not done, the Haskell and Ocaml extraction currently share a broad common base for simplicity reasons.

4.3.2 Optimizations of inductive types

We show now how the elimination of the logical arguments presented previously for the functions also apply to the constructors of inductive types. Then in a second time we present two two particular cases of inductive types that are subject to particular treatments by the extraction, that is the singleton types and the records.

Constructors and elimination of logical arguments

Just as the arguments of type □ for functions, the arguments of type □ for inductive constructors can also be removed, because they are superfluous. Let us take for example the informative inductive type `sig` initially presented page 28. As a reminder, the type `(sig A P)` expresses the existence of an object `x` of type `A` fulfilling the predicate `P`, a fact that we also notes with the syntax `{x:A|P x}`. And the single constructor of `sig` admits four arguments: `(exist A P x p)` has type `(sig A P)` when `x` is the required witness and `p` is a proof of `(P x)`. The raw version of the extraction of `sig` is then[5]:

```
type 'a sig0 = Exist of □ * □ * 'a * □
```

In fact, the first two arguments `A` and `P` of `exist` are `parameters` of the inductive type `sig`. The typing rules of Cic ensure that these parameters cannot vary during the the definition of `sig` constructors. These parameters thus cannot bring new calculative contents, and are systematically removed from inductive definitions by the extraction. Here, in fact, that does not change anything, because these parameters are types, which would thus have been removed by the mechanism we are about to see. But even one informative parameter, for example of type `nat`, would have been removed by the extraction.

Since the third argument is to be kept, it only remains to study fourth argument, which is a logical term. But it is in fact immediate to free the extracted definition from this useless □ :

```
type 'a sig0 = Exist of 'a
```

[5] `sig` is renamed into `sig0` by the extraction since `sig` is an Ocaml keyword.

One must then adapt the uses of this type:

- On the level of an application to the constructor `Exist`, one must suppose the presence of the four arguments. Indeed, the uncurryfied syntax of Ocaml requires it, with the result that the extraction builds when needed the missing arguments by η-expansion. The last step is then to filter the arguments, and here only keep the third one.
- On the level of a matching like `match e with Exists(a, p, x, q) → t`, the properties of the extraction ensure that `a`, `p` and `q`, who have all $\Box$ as type, will not appear in `t`. We thus replaces this matching by `match e with Exists(x) → t`.

And what was presented here for `sig` is generalized to all inductive types, whatever are its number of constructors, parameters and arguments. One can notice that there is no need here for a particular treatment when all the arguments of a constructor are removed during the process: this constructor is simply a constant one afterward.

Simplification of informative singleton types

In the particular case of the extraction of the inductive type `sig`, one can go one step further in the simplifications. Indeed, it is noticeable that there remains only one argument to `Exist` after the previous transformation. Since `sig` has one argument, its extracted version is now a simple encapsulation. It is then preferable to remove this layer of encapsulation. So we simply convert the type `(a sig0)` into `a`, and the term `Exists(t)` into `t`. Finally we need to represent a pattern matching on an object of type `sig`: the term `match e with Exists(x) → t` corresponds now to `let x = e in t`.

This simplification extends to the types known as informative singletons, that is having one constructor, and whose single constructor has only one argument after extraction[6].

This treatment of informative singleton types allows to gain in memory occupation, in computation times and also in readability. In addition to that, we then obtain the awaited behavior for the extraction with respect to the informative existential quantification: starting with a Coq proof whose statement has the form `∀x:t, P x → ∃y:u, Q y`, we now obtain a function of type `t → u`, and not of type `t → U sig0`.

Records

The other category of inductive types that receives a particular treatment during extraction are the inductive types defined via the declaration `Record` of Coq. For example:

```
Record paire (A B:Set) : Set := { gauche : A ; droite : B }.
```

At the internal level, these Coq records are not primitive, but translated into inductive types. Our example is thus stored by Coq under the form:

```
Inductive paire (A B:Set) : Set := Build_paire : A → B → paire A B.
```

[6] To be completely accurate, one should also check that the type of this unique remaining argument does not mention the original inductive type, otherwise this simplification is erroneous.

The advantage of the `Record` declaration is that Coq generates automatically[7] from this declaration two projection functions `gauche: (pair A B) → B` and `droite: (pair A B) → B`.

During the extraction, it is absolutely possible to stay unaware of the fact that the type `paire` has been defined as a record. This type would then be extracted like a standard inductive type:

```
let ('a,'b) paire = Build_paire of 'a *'b
```

And the associated projections would then be matchings, for example:

```
let gauche = function
 | Build_paire (x, y) → x
```

At the same time, it is preferable to benefit from the possibilities of Ocaml [8], by using its primitive syntax for the records. This allows in particular to reach directly a record field via the "dotted notation", here for example `p.gauche`, which is slightly more effective than a projection by matching. Moreover, the readability of the extracted code is also improved. Let us illustrate the change for the primitive records of Ocaml with our small example:

- The extracted type is now:

```
type ('a,'b) paire = { gauche : 'a; droite: 'b }
```

- The projection functions are then provided only for compatibility, in case they are used without argument:

```
let gauche x = x.gauche
```

- And for each projection coming with its argument, rather than using the functional form `(gauche p)`, we now produce `p.gauche`.
- Each use of the constructor `Build_paire`, as in `(Build_paire G d)`, becomes a record `{gauche=g; droite=d }`. Note that during the preceding transformations of inductive ones, any partial application of this constructor has already been initially η-expanded.
- Lastly, any pattern matching on a type that becomes a record is adapted as follows:

```
match p with Build_paire (x,y) → ...
```

becomes:

```
match p with {gauche=x; droite=y} → ...
```

[7] It is necessary to note that the generation of some projection functions is sometimes impossible for typing reasons.

[8] There also exists in Haskell some primitive records, but the extraction does not use them yet.

Originally, the need of an improved extraction of the records was felt during the first extraction tests of the C-CoRN project (see chapter 6 for more details). Indeed, this development uses abundantly algebraic structures defined via records. And these structures are connected by coercions which are in fact some projections. At the level of the Coq development, these coercions remain implicit, thus invisible. But on the level of the extracted code, the least addition of two reals in this formalism makes then intervene 7 projections. Any profit of effectiveness and readability at this level is then appreciable.

4.3.3 Unfolding the body of some functions

The transformations presented up to now could modify the types of the extracted objects. We now go on a second category of transformations, that preserve the type of the extracted objects. The first of these transformations consists in replacing some names of functions by their bodies. In other words, during the extraction time, we anticipate the δ-reduction of these functions. Obviously, such unfoldings are not to be performed systematically, otherwise the returned code would become gigantic. But we will see that in certain precise cases, these unfoldings are crucial for the efficiency of the extracted code in Ocaml. In fact there was already such a mechanism of unfolding in the old extraction. We used this previous mechanism as inspiration whereas modifying some of its criteria. Moreover this former mechanism has never been documented, up to our knowledge.

Let us first study the induction principle[9] `bool_rect` associated to the boolean type. This function is automatically generated by Coq at the time of the definition of the type `bool`. Its type is:

```
∀P:bool→Type, P true → P false → ∀b:bool,P b
```

And its extraction is:

```
let bool_rect f f0 = function
  | True → f
  | False → f0
```

In this definition, `f` corresponds to the proof of `(P true)` and `f0` to the proof of `(P false)`. One can naturally note that according to the value of the third boolean argument applied to this function, only one of the terms `f` or `f0` will really be used. However any application `(bool_rect a a' b)` leads in Ocaml to the evaluation of `a` and `a'`, because of the strict evaluation strategy of the arguments in this language. Of course, that can be trivial if `a` and `a '` are simple values like 0, or functional closures whose evaluation stops immediately. But if `a` and `a'` carry out expensive computations, that can be annoying.

A solution to avoid these superfluous evaluations is then to replace `bool_rect` by its definition, and then evaluate symbolically its arguments. This then will have for effect of to push the arguments `a` and `a'` under a matching, in branches that are mutually excluded:

[9] In fact, there are normally three such principles, one for each sort of Coq. But we will consider only the one over `Type`, named `bool_rect`. The two others, `bool_rec` and `bool_ind`, are in fact only alias.

```
bool_rect a a' b
```

becomes:

```
(fun f f0 b = match b with True → f | False → f0) a a' b
```

that we immediately simplify into:

```
match b with True → a | False → a'
```

In induction principles with more than two constructors, there always exists such arguments that will be reduced by Ocaml even if some of them are in fact useless. But one can also meet this situation in several other locations. The strategy of the extraction towards Ocaml is then the following one:

- All the principles of induction are unfolded, even for inductive with less than two constructors. Indeed, in addition to the interest in term of efficiency, this leads experimentally to an extracted code which is closer to what an human would have written, and thus more readable. In particular, the tactics `elim` and `induction` use systematically the induction principles and never the underlying fixpoint and matchings.
- For the other functions, one carries out an unfolding when the two following conditions are met:
  - the body of the function is not too large, *i.e.* in practice of size lower than a arbitrary limit fixed at ten syntactic constructions.
  - certain arguments are detected as potentially useless, for example present under only one branch of a matching.

Of course, this is an heuristic trade-off between opposed requirements, which can certainly be still improved. Until more complete investigation of this subject, we have at least given to the user the possibility of controlling these unfoldings more finely:

- With the command `Set` (resp. `Unset`) `Extraction AutoInline`, the user can activate (resp. deactivate) our automatic mechanism of unfolding.
- It is also possible to force the unfolding or the non-unfolding of a particular object `t` with `Extraction Inline t` or `Extraction NoInline t`.

To illustrate the critical aspect of these unfoldings for the efficiency, we now study a concrete problem appeared one day where an unfortunate modification in the extraction source code has completely deactivated the automatic unfolding. An extracted example[10], which normally runs for a few seconds, was still running this particular day even after ten hours. The faulty function was a function comparing two Peano integers, `lt_eq_lt_dec`. At that time, this function was built by a double induction on its two arguments[11]. This

[10] In fact the contribution Rocq/COC.

[11] In fact, the second induction is useless here and has since been replaced by a simple case analysis. And this change in the proof solves almost completely the problem of efficiency of the non-unfolded extracted version. But who is concerned by such details when the first goal is to finish a proof?

leads after extraction to a term containing two levels of `nat_rect`, which is the induction principle associated with the type `nat`:

```
let lt_eq_lt_dec n m =
  nat_rect
    (fun m0 → nat_rect (Inleft Right) (fun m1 iHm → Inleft Left) m0)
    (fun n0 iHn m0 → nat_rect Inright (fun m1 iHm → iHn m1) m0)
    n m
```

The details of this code are not so relevant, only this double level of `nat_rect`. Here is the same function after the automatic unfolding of functions:

```
let rec lt_eq_lt_dec n m =
  match n with
    | O → (match m with
              | O → Inleft Right
              | S n0 → Inleft Left)
    | S n0 →
        (match m with
           | O → Inright
           | S n1 → lt_eq_lt_dec n0 n1)
```

If we study this unfolded version, it is clear that the comparison between two integers of Peano `n` and `m` will involve a number of recursive calls that is `min(n, m)`. On the other hand, the not-unfolded previous version has a quite different behavior. That can seem strange, because even if `nat_rect` evaluates its arguments, they are only functions here. But if one looks in detail the sequence of the function calls (with `#trace` by example), one realizes that these unnecessarily evaluated functions will nevertheless meet some arguments, be reduced, and finally make the complexity explode.

The recursivity being hidden behind the function `nat_rect`, we count here the number of calls to this function. Here is the table recapitulating the number of these calls with respect to the value of the input integers `n` and `m`:

m \ n	0	1	2	3	4	5	6	7	8	9	10
0	**2**	2	2	2	2	2	2	2	2	2	2
1	2	**3**	4	5	6	7	8	9	10	11	12
2	2	3	**5**	8	12	17	23	30	38	47	57
3	2	3	5	**9**	16	27	43	65	94	131	177
4	2	3	5	9	**17**	32	58	100	164	257	387
5	2	3	5	9	17	**33**	64	121	220	383	639
6	2	3	5	9	17	33	**65**	128	248	467	849
7	2	3	5	9	17	33	65	**129**	256	503	969
8	2	3	5	9	17	33	65	129	**257**	512	1014
9	2	3	5	9	17	33	65	129	257	**513**	1024
10	2	3	5	9	17	33	65	129	257	513	**1025**

One notes in particular that to comparing an integer `n` with itself involve $2^{n} + 1$ calls to `nat_rect`. In fact, if we name $\mathcal{N}_{\mathtt{n}}^{\mathtt{m}}$ these call numbers, one can note that $\mathcal{N}_{\mathtt{n}}^{\mathtt{m}} - \mathcal{N}_{\mathtt{n}}^{\mathtt{m-1}} = C_{\mathtt{n}}^{\mathtt{m}}$. The columns of this table are thus partial sums of $C_{\mathtt{n}}^{\mathtt{m}}$ starting from 2, and one hence finds again the particular case of the diagonal. We do not justify more these formulas giving $\mathcal{N}_{\mathtt{n}}^{\mathtt{m}}$, but it is not (too much) difficult to establish for example $\mathcal{N}_{\mathtt{n}}^{\mathtt{n}} = 2^{\mathtt{n}} + 1$ starting from the code of `lt_eq_lt_dec`.

In any case, comparing this way 30 and 32 generates more than one billion calls of `nat_rect`! The unfolding of `nat_rect` transforms here an exponential function into a linear one.

4.3.4 Code improvements: optimization or deceleration?

The conclusion of the preceding example is that it is very delicate to predict which will be the form of the extracted code when one looks at the original proof script made of long sequences of tactics like `induction` or `auto`. And very often the raw extracted code does not correspond at all to what an human would have written. In addition to the readability problem, this can implies efficiency problems, as we have just seen. And this is particularly obvious with a strict language like Ocaml.

In addition to its problem with `nat_rect`, the preceding function `lt_eq_lt_dec` also illustrates a case of `stupid` extracted code. Without one of the extraction optimizations, the extraction of this function would contain instead of the recursive call `(lt_eq_lt_dec n0 n1)` a subterm being worth:

```
match lt_eq_lt_dec n0 n1 with
 | Inleft x →
   (match x with
     | Left → Inleft Left
     | Right → Inleft Right)
 | Inright → Inright
```

Obviously, deconstructing an object to rebuild it identically has no interest, even if it is hardly expensive here. And no programmer will ever write such a code spontaneously. The reason of these surprising matchings is to be sought in the logical parts, erased during the extraction. Indeed, the constructors `left`, `right` and `inright` of the inductive types `sumbool` and `sumor` carry at first a logical argument each. And the initial matchings in Coq are in fact useful to modify these logical arguments. The extraction then tries to simplify such matchings.

More generally, the extraction applies a certain number of transformations aimed at giving the produced code a more usual aspect. Here the detail of these transformations:

0. (code=1) Elimination of internal logical residues.
1. (code=2) Improvement of the printing for functions defined by fixpoint.
2. (code=4) When a matching is at the head of another matching, one permutes the order of these two matchings if some simplifications occur in all new branches.
3. (code=8) A matching on `bool`, `sumbool` or `sumor` building again what it just has deconstructed is removed

4. (code=16) This optimization extends the previous one to all types of matchings.
5. (code=32) A matching in which all branches produce the same result is replaced by this result.
6. (code=64) If all the branches of matchings begin with one abstraction, one moves this abstraction in front of the matching.
7. (code=128) One propagates the arguments of one matching inside its branches.
8. (code=256) One permutes the applications and let-in.
9. (code=512) An abbreviation `let x=u in t` is unfolded if `x` appears at most once in `t`.
10. (code=1024) a β-redex `((fun x → t) u)` is reduced if `x` appears at most once in `u`.

If one wishes to deactivate all these transformations, it is enough to use the command `Unset Extraction Optimize`. If one wishes on the contrary to return to the default situation where (almost) all transformations are activated, one can use the command `Set Extraction Optimize`. Between these two extremes, version 8.0 of `Coq` makes it possible to refine the preferences via the command `Set Extraction Flag N`, where `n` is an integer between 0 and 2047, corresponding to the sum of the binary codes of the transformations the user wants to activate. For example, if one wish all to be activated except the transformation n° 4, of code 16, one use `n` $= 2047 - 16 = 2031$. By the way, this value correspond in fact to the default situation enforced by `Set Extraction Optimize`, and we will see why in a moment.

One can wonder why such a mechanism "à la carte" for controlling the behavior of the extraction. The problem is that these transformations are not always optimizations, since they rely on some heuristics which were not yet the subject of deep studies, and can perfectly appear harmful in certain situations. Worse, a transformation recently turned out to be perilous for the good typing of the extracted terms.

Let us see now in detail which benefit and concern one can wait from each transformation.

0. These internal eliminations of logical residues are a priori always beneficial.
1. This embellishment of fixpoint can sometimes alter the arguments of the internal recursive calls, making these calls less effective.
2. This permutation of two matchings can increase the size of the code, or decrease it, depending of further simplifications that can be applied on each new branch.

3&4. We delete here the previously mentioned "stupid" matchings. This is a priori benign, but let us take the following example:

```
type 'a t = T

let f a = match a with T → T
```

It is then very tempting to change `f` into `let f a = a`, but its type would be then `'a t →'a t` and not any more `'a t →'b t`. A fast remedy for this problem was to split this transformation in two, with on one side a sure part acting only on known types like `sumbool`, and on the other side a not-sure part deactivated by default. In practice the optimization n° 3 is enough for simplifying the usual situations of "stupid" matchings. In addition, optimization n° 4 is not so dangerous: an user activating this option will likely obtain correctly typed code, which is then completely correct.

5. This optimization is normally without disadvantage.
6. Moving up λ-abstraction in front of a matching can lead to multiple evaluations of the head of this matching, but can also improve the deletion of the logical residues by other transformations.
7. This option duplicates code, but can also help to propagate simplifications.
8. A term as `((let x = t in u) v)` is little natural, and hides the fact that `v` is morally an argument of `u`, which prevents from possibly applying others simplifications. It seems benign to permute the application and the let-in, in order to obtain `let x = t in (u v)`. Unfortunately, we have met once a situation where this permutation, combined with η-expansions intended to overcome some limitations of Ocaml, produced in fact a permutation between a function and a let-in, which corresponds to the next case, and is not always desirable.
9. This unfolding of a let-in, even linear, is debatable, considering that a computation initially factorized can then be found in one function, and thus been carried out several times. But this computation can also be moved in a branch which is never evaluated. In addition, functions generated by tactics contain very often partial applications like `let g = f x in g y`. This can lead to the computation of many useless intermediate closures, and prevents the Ocaml compiler from optimizing the total call `f x y` as it should.
10. The problems are the same as in the previous case.

These transformations are then to be used with great precautions. All the difficulty is that the Coq code to be extracted can as well come from proof by tactics or from functions written directly in the functional language of Coq. In the first case, the difficulty in controlling the precise underlying proof term implies that one has to expect many transformation before obtaining some reasonable extracted code. But in the second case, the best thing to be done is to leave the term intact to respect the choices of the user. In any event, it is illusory to always hope to find the most effective code, hence the interest in allowing the user to make his choices among the suggested transformations.

The extraction currently makes the choice to privilege the improvement of code coming from proofs, to the possible detriment of the directly written code. All possible optimizations are thus activated, except the one endangering the typing, that is the n° 4.

This current situation of optimizations is finally far from satisfactory:

- No correctness guarantee other than a visual inspection of the transformation rules.
- Improvement of the efficiency and readability on average, but presence of harmful particular situations.
- Possibility of a manual adjustment, but which remains rather coarse, and on the level of a entire file.

This part would thus deserve a thorough study. Finally, the extraction is here only a automatic generator of code. Perhaps it is necessary to move the problem of optimization to the level of the compiler or or to the level of a generic tool for preprocessing source code. At the same time, the extracted code has a certain number particular features that a generic tool would perhaps neglect, like the absence of imperative parts, or the importance of the partial applications. Moreover, it would undoubtedly be interesting to apply a search for dead-code in the extracted code since the informative parts not pruned by the extraction can still contain some.

4.4 Current state of the actual implementation

4.4.1 Description of the code

The implementation of the new extraction of Coq made during this thesis began in 2001 with the initial assistance of J.-C. Filliâtre. This implementation has been improved gradually, from version 7.0 of Coq until the current 8.0. It is currently in a reasonable state of stability and of functionality, even if for example the preceding section shows that a certain number of optimizations are to be re-examined.

This implementation is located in the sub-directory `contrib/extraction` of the sources of Coq, and is currently composed of 3 700 lines of Ocaml. The figure 4.1 presents the dependency graph of the files contained in this implementation. More precisely:

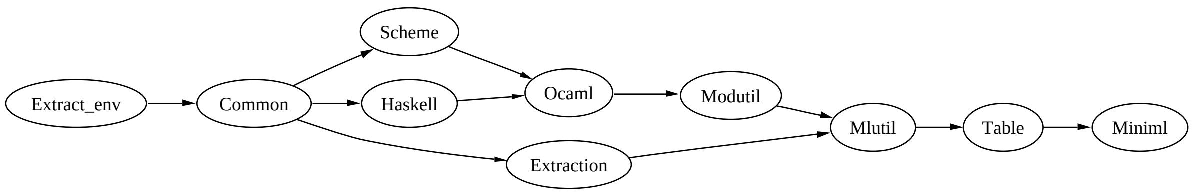

FIG. 4.1: Dependency graph of the implementation

- The interface `Miniml` defines the abstract syntax trees for terms and types of a MiniML language which is used as common target by the extraction.
- The module `Table` is the "memory" of the extraction, it allows for example to find the extraction of inductive type already met.
- The module `Mlutil` is a toolbox for handling terms and types of MiniML. There is defined for example the substitution over terms, or the unification over types, or various optimizations over terms.
- As for `Modutil`, it contains auxiliary functions to handle the module structures of MiniML
- The module `Extraction` is the heart of the extraction. It is there that a Coq term is translated into a MiniML term or type.

- The three modules `Ocaml`, `Haskell` and `Scheme` are used to translate the MiniML objects into concrete syntax of one target languages.
- The module `Common` factorizes some functions of renaming common to the three target-languages.
- Lastly, `Extact_env` allows to determine which minimal environment of Coq objects will have to be extracted to satisfy the request of a user.

4.4.2 Small user's manual

Naturally, this small handbook will be less detailed than the entire chapter present in the Coq Reference Manual [78]. But since the present manuscript is meant to be a self-contained and complete review of the extraction in Coq, here comes a brief outline of Coq commands related with extraction.

The extraction itself

```
Coq < Extraction plus.
Coq < Recursive Extraction plus minus.
Coq < Extraction "myfile" plus minus.
```

The first order is the simplest, it only prints on the screen the extraction of an object, here `plus`. The second, on the other hand, prints in addition every dependency of the required object(s). In our example, one will also see the extraction of the type `nat`. Finally the third order behaves exactly as the second, except that it writes to a file and not on the screen. Normally, since this file contains all the needed dependencies, it is directly compilable. Let us note that when the selected target-language is Ocaml, one obtains at the same time a file *.ml and an interface *.mli. Lastly, these orders also accept one module `M` in the place of an object like `plus`.

```
Coq < Extraction Library Peano.
Coq < Recursive Extraction Library Peano.
```

These two somewhat obsolete orders are intended to extract in one pass the complete contents of a Coq library, i.e. a file *.v compiled and charged via a `Require`. In the case of the library `Peano.v`, one obtains then `peano.ml` and `peano.mli`. In the second alternative, the extraction produces also the *.ml and *.mli files for all the libraries on which `Peano.v` depends.

How to control the extraction

One can first modify the target language:

```
Coq < Extraction Language Ocaml.
```

.../...

.../...

```
Coq < Extraction Language Haskell.
Coq < Extraction Language Scheme.
```

We also encounter again the commands controlling the "optimizations", already met in previous section 4.3:

```
Coq < Set Extraction Flag n.
Coq < Set Extraction Optimize.
Coq < Unset Extraction Optimize.
```

Taking into account the current state of these optimizations, it is sometimes preferable to use for the moment the `Unset` version.

Here finally come the commands controlling the unfolding of functions, also described already in section 4.3:

```
Coq < Unset Extraction AutoInline.
Coq < Set Extraction AutoInline.
Coq < Extraction Inline f g.
Coq < Extraction NoInline h k.
Coq < Reset Extraction Inline.
```

The axioms

It is in fact possible to use the extraction in conjunction of axioms in a development. If a logical axiom is used, the extraction only prints a warning message recalling that the correctness of the extracted code rely on the validity of this logical axiom. And when a development uses an informative axiom `f`, the extracted code which depends on it then contains an exception announcing that some missing code have to be filled:

```
let f = failwith "AXIOM TO BE REALIZED"
```

The user must then provide code indeed realizing this axiom. This can be done manually in the extracted file, or in Coq via a special command:

```
Coq < Extract Constant f ⇒ "my_realizing_code".
```

This command is only some syntactic convenience, it does not check the contents of the character string provided in realization of the axiom, but only generates:

```
let f = my_realizing_code
```

It may be happens that an axiom is a type scheme. By example, for `t : Set → Set → Set` , the following realization:

```
Coq < Extract Constant t "'a" "'b" ⇒ "'a * 'b".
```

generates the type declaration:

```
type ('a,'b) t = 'a * 'b
```

Even if it does not concerns axioms, a related feature is the possibility of replacing an inductive type by another at the the extraction time. For example, the Coq type `sumbool` with its two constructors `left` and `right`, presented p.29, is isomorph to `bool` after extraction. One can then identify these two types as follows:

```
Coq < Extract Constant sumbool ⇒ "bool" [ "true" "false" ].
```

4.5 A first complete example

We now give as illustration the complete extraction of the function `div` of page 31. This function, with pre- and post-conditions, is defined by using the well-foundedness of the usual order over the Peano integers. If we extract the function named `well_founded_induction` used in the definition by tactics for our function `div`, one obtains a fixpoint combinator à la `Y`, without trace of logical parts put aside an anonymous lambda:

```
let rec well_founded_induction f a =
  f a (fun y _ → well_founded_induction f y)
```

And this constant `well_founded_induction` will be automatically unfolded in `div`, leaving finally the desired non-structural fixpoint.

In addition, `div` uses an auxiliary function `le_lt_dec` used to compare two integers. This function, close to the function `lt_eq_lt_dec` already met, works with objects of type `sumbool`. As announced right before, it is convenient to replace `sumbool` by `bool`:

```
Coq < Extract Inductive sumbool ⇒ bool [ true false ].
```

The final step is of course the extraction itself:

```
Coq < Extraction "div.ml" div.
```

The obtained interface `div.mli` is then:

```
type nat =
  | O
  | S of nat

type 'a sig0 = 'a
  (* singleton inductive, whose constructor was exist *)

val minus : nat → nat → nat

val le_lt_dec : nat → nat → bool
```

.../...

.../...

```
val div : nat → nat → nat sig0
```

In this signature, the `sig0` is nothing more that an alias recalling that the type of `div` was previously finishing by one existential type. That put aside, one obtains the type `nat→nat→nat` expected for `div`.

Here comes finally the file `div.ml`:

```
type nat =
  | O
  | S of nat

type 'a sig0 = 'a
  (* singleton inductive, whose constructor was exist *)

(** val minus : nat → nat → nat **)

let rec minus n m =
  match n with
    | O → O
    | S k → (match m with
              | O → S k
              | S l → minus k l)

(** val le_lt_dec : nat → nat → bool **)

let rec le_lt_dec n m =
  match n with
    | O → true
    | S n0 → (match m with
               | O → false
               | S n1 → le_lt_dec n0 n1)

(** val div : nat → nat → nat sig0 **)

let rec div x b =
  match le_lt_dec b x with
    | true → S (div (minus x b) b)
    | false → O
```

Chapitre 5

Examples of extraction

Since the beginning of the works by C. Paulin on the extraction in Coq, we must admit that no application of industrial scale has been completely certified via this method. Despite this fact, several examples of significant size have been carried out. We try here to draw up a brief panorama of these examples. Among these examples, we have selected four of them who have benefited from our new extraction or could not exist without it. These practical cases that we have chosen to illustrate are:

- the contribution `Lannion` of J.-F. Monin,
- the contribution `Rocq/HIGMAN` of H. Herbelin,
- the contribution `Nijmegen/C-CoRN` by the team of H. Barendregt,
- the contribution `Rocq/FSets` by J.-F. Filliâtre and the author.

The first two contributions are detailed in this chapter. The third is covered by the following chapter, dedicated to the extraction of a constructive formalization of real numbers. Lastly, the fourth contribution is described in the chapter 7.

5.1 The standard library of Coq

If one search Coq developments for proofs to extract, the first accessible source is the standard library of Coq, which is directly provided with the system. A good number of functions given as illustration up to now come from there or are derived from functions available there. We have set up a test consisting in extracting systematically all contents of this standard library. This test came be found in the directory `contrib/extraction/test` of Coq sources. Once in this directory, the command `make tree; make` allows to launch the test.

When we look in detail at the functions extracted from this standard library, the interesting objects are sparse. First of all, the majority of the results in this library are logical properties, and thus do not appear after extraction. Then, the more significant component of this library, namely the `Reals`, are out of the field of the extraction. This is due to the axiomatic approach followed during this formalization of the real numbers, starting with:

```
Parameter R : Set.
Parameter R0 : R. (* one *)
Parameter R1 : R. (* zero *)
Parameter Rplus : R → R → R.
```

This is not inevitably crippling for the extraction, which can work with axioms as we have just seen in the previous chapter. But some of these axioms correspond to a non-constructive vision but rather classical of the reals, and thus cannot be realized accurately. For example, this is the case for:

```
Axiom total_order_T : ∀r1 r2:R,{r1 < r2} + {r1 = r2} + {r1 > r2}.
```

It is well known indeed that no exact real arithmetic can have a decidable equality. Even more awkward, the last of the axioms, `completeness`, requests the constructive existence of a smaller upper bound for any non-empty upper-bounded set of reals. However a set is here a logical object of type `R→Prop`. This axiom is thus non-realizable.

Finally, it remains after extraction only elementary functions dealing with data structures like Peano natural numbers, binary lists or binary integers. The latter constitute the most interesting part of these examples, with divisions functions, square roots, etc. One can also find a sorting algorithm for lists in the directory `theories/Sorting`. Finally the directory `theories/IntMap` contains a formalization of finite sets indexed by integers. Amongst all these informative functions, one finally find relatively little functions defined by tactics, the only ones really interesting from the extraction point of view. And the extraction behaves reasonably well on these functions, which all are sufficiently standard not to need `Obj.magic` to type-check.

5.2 The user contributions

The main examples base for extraction is today the developments proposed the users of Coq. These *user contributions* are gathered and classified by Coq development team (by geographical origin and topic). They are also maintained up to date with respect to each modification of Coq and published for each new version of Coq. It is possible to consult the list of the 88 current contributions on the site `http://coq.inria.fr/contribs-eng.html`.

Obviously, this data base of Coq developments is certainly not exhaustive, since it relies on a declaration on behalf of the users. And some extremely interesting Coq developments with respect to the extraction are currently not part of these contributions proposed by the users. Among such developments, one can quote for example a decision procedure dedicated to propositional intuitionistic logic, by K. Weich [82], and also a static programs analyzer by abstract interpretation, by D. Cachera, T. Jensen, D. Pichardie and V. Rusu [17]. And there exists doubtless other works which would deserve to be quoted here except that we are not aware of them.

Let us return now to these user contributions. First, several do not concern the extraction, because they only establish results in `Prop`. Nevertheless, about thirty of them are relevant for the extraction. The complete list of these extractable contributions can be found in Appendix A.

If we try to somewhat classify these multiple examples, we can first distinguish a certain number of decision procedures. Several of them concern the boolean formulas: `Dyade/BDDS`, `Suresnes/BDD` and `Sophia-Antipolis/Stalmarck` [56]. As for `Rocq/GRAPHS`, it treats of linear (in)equalities over Z. `Nancy/FOUnify` and `Lannion` solve the problem of first order term unification. And `Rocq/COC` proposes an type-checker for the calculus of constructions.

In addition, several contributions are centered on data structures, and can thus be used as basic library for other work. We can in particular find some definitions of rational numbers, `Nijmegen/QArith` and `Orsay/QArith`, of constructive real numbers in `Nijmegen/C-CoRN`, of tree structures in `Bordeaux/SearchTrees`, `Bordeaux/dictionnaries` and `Orsay/FSets`. Lastly, formalizations of already quoted BDDs can also to included in this category.

On the other hand, the large variety of the contributions prevents from pushing much further such a classification. And the subjects of study are not the only variable aspect in these contributions. First, the size of these contributions can go from a simple file of 300 lines, as for `Bordeaux/Exceptions`, up to more than one hundred files and 75 000 lines for `Nijmegen/C-CoRN`.

The age is also very variable. Among the oldest contributions, one finds `Rocq/Higman` made around 1992 with `Coq` version 5.x. On the opposite, some contributions benefit from recent features of `Coq`, like the new module system for `Bordeaux/dictionnaries` and `Orsay/FSets`.

The authors' attitude with respect to extraction differs also from contributions to contributions. In certain cases, the authors have planned from the beginning to extract their results. For example, B. Barras has integrated in the `Makefile` of `Rocq/COC` an extraction followed by the compilation of the extracted file and the launch of a test for this extracted program. On the opposite, the initial developers of `Nijmegen/C-CoRN`, although conscious of the theoretical possibility of extracting their work, have not considered it as feasible. In several contributions, we have emphasized the informative aspects by adding extractions, compilations and automatic tests in the spirit of `Rocq/COC`. This has been done during a manual inventory of the contributions. Perhaps it still remains some extractable examples not yet located in these contributions.

Lastly, the difficulty of extraction varies largely. For example, in `Lyon/Firing-Squad`, the main extracted object is a purely informative transition function, written via a `Fixpoint`. The extraction consists then of a simple translation into the syntax of the target language. The program finally obtained is nonetheless interesting because it allows to visualize the evolution of the states under the action of this transition function, thanks to a small graphical interface. But the interest is very limited from the pure extraction point of view. By the way, let us note that this case corresponds to an indirect possible use of the extraction, namely for simulating and illustrating `Coq` functions. Concerning the task of the extraction, we note that there is few extractions of functions defined by tactics. In addition, the majority of the situations were working or could have worked with the old extraction.

All the preceding points explain why there remain little contributions allowing to study the contributions of our new extraction. in particular, there exist only four contributions which are extracted into untyped code, requiring the usage of some `Obj.magic` : these four contributions are `Lyon/Circuits`, `Lannion`, `Nijmegen/C-CoRN` and `Rocq/Higman`. And we have already announced that only two contributions use the new module system. Lastly, we have only tested our extraction of co-inductive types towards Ocaml in the situation of the contribution named `Rocq/Mutual-Exclusion`. We now study more in detail some of these contributions benefiting from our new extraction.

5.3 Exceptions by continuations in Coq

We start by studying the works by J.-F. Monin concerning the certification of functional programs with exceptions. These works [60, 61] were carried out between 1995 and 1997. Historically, this is one of the two concrete situations where the old extraction of Coq was generating non-typable code. The second of these concrete situations corresponds to the contribution `Rocq/Higman` which we study just afterward.

5.3.1 Formalization of the exceptions in Coq

To model exceptions in Coq, J.-F. Monin uses a translation by continuation (CPS, for "continuation passing style"). We enter directly in the core of this method, the reader eager to find an more gradual introduction to these works can refer to [61]. Here comes, first of all, the type scheme used to express the result type of a function that can raise an exception:

```
Definition Mx (C:Prop)(A:Set) :=
  ∀(X:Set)(P:Prop), (P→X) → (C → P) → (A → X) → X.
```

Here, the informative type `A` is the "usual" return type for the function that we consider, whereas `C` is a logical condition implying the raising of an exception. The presence of the quantification on `X` corresponds to the translation by continuation[1]. Lastly, `P` and the following argument `e` of type `P→X` represents the manner of building the result `X` of the continuation in case of an exception. Finally, the type `(Mx C A)` can be seen as a sum type `A∨C`, representing the two possible exits, standard or exceptional, of our function.

Let us now see how to manufacture values in this type:

```
Definition Mx_unit (C:Prop)(A:Set)(a:A) : Mx C A :=
  fun X P e i k ⇒ k a.

Definition Mx_raise (C:Prop)(A:Set)(c:C) : Mx C A :=
  fun X P e i k ⇒ e (i c).
```

The first function corresponds to the normal result: if one succeeds in building an object `a` of type `A`, it is enough to apply it to the current continuation `k` of type `A→X`. On the contrary, the second function deals with the raising of an exception: if one obtains a proof `c` stating that one is in the exceptional situation `C`, one uses the mechanism of exception generation, namely the functions `i` then `e` of the respective types `C→P` and `P→X`.

[1] Without the exceptions, this translation would have consisted in changing `A` into `∀X, (A→X)→X`.

Here come now two functions allowing to treat the exceptions. The first corresponds to an case analysis:

```
Definition Mx_try (C:Prop)(A:Set)(m:Mx C A)(X:Set)(k:A→X)(e:C→X): X :=
  m X C e (fun p ⇒ p) k.
```

In other words, if one can always generate a result of type `X` starting as well from a normal value of type `A` or from an exception of type `C`, then one can always build a result in `X` starting from an object `m` in the sum type `(Mx C A)`. In addition, one can pass from a type with exceptions to another type with exceptions by the means of the following function:

```
Definition Mx_bind
  (A A':Set)(C C':Prop)(m:Mx C A)(f:A→Mx C' A')(j:C→C') : Mx C' A' :=
  fun X P e i k ⇒ m X P e (fun c ⇒ i (j c)) (fun a ⇒ f a X P e i k).
```

This operator allows in practice to compose two functions being both able to raise exceptions.

5.3.2 Impredicativity and typing of the extracted functions

The contribution `Lannion` uses the impredicativity of `Set` (cf page 23) for the type `Mx`. Indeed, this type, which contains a universal quantification on `X:Set`, should a priori be in `Type`. But if we activate the impredicativity of `Set`, `Mx` can then be of type `Set`. As a consequence, this allows to take `X=(Mx C A)` inside `(Mx C A)`. This situation occurs for example when one builds a result of type `Mx C A` via a `Mx_try`. Such an impredicative situation is a strong indicator of the presence of typing errors in the raw extracted code. J.-F. Monin notices moreover p.48 of [61] that problems occur for fixpoint (translated into continuations) of the shape:

```
let rec f x = ... try ... f y ... with ...
```

We will see thereafter that the impredicativity alone does not involve necessarily the need for `Obj.magic` in the extracted code, and also that `Obj.magic` can appear without any impredicative objects. Historically anyway, the need for `Obj.magic` in the extracted code was felt initially in `Lannion` and `Rocq/Higman`, two contributions using the impredicativity, and that is not by chance.

5.3.3 The extraction of this development

If we stick to the extraction of types exposed in chapter 3, the type `Mx` is translated in a very approximate way, because of its two universal quantifications, the second being logical in addition:

```
type 'a mx = __ → __ → (__ → __) → __ → ('a → __) → __
```

On the other hand, in the type of `Mx_unit` and `Mx_raise`, this type scheme `Mx` is used in head position, and its universal quantifications ends on the first level of types. Improvement of types described in section 3.5.4 gives then a type much more standard to these two functions:

```
(** val mx_unit : 'a1 → (__ → 'a2) → ('a1 → 'a2) → 'a2 **)

let mx_unit a e k = k a

(** val mx_raise : (__ → 'a2) → ('a1 → 'a2) → 'a2 **)

let mx_raise e k = e __
```

We can see in particular that the extraction of these two functions do not require the use of `Obj.magic`, even if their later use can need some. On the other hand, the situation gets worse for `Mx_try`, since `Mx` is then found in the type of one of the arguments of this function, and not any more in head position.

```
(** val mx_try : 'a1 mx → ('a1 → 'a2) → (__ → 'a2) → 'a2 **)

let mx_try m k e = Obj.magic m __ __ e __ k
```

We see that an `Obj.magic` appears. For example, the argument `e` of `Mx_try` has a type `C→X` on the Coq level, which becomes `__→' a2` on the Ocaml level since `C` is logical and since we try to transform `X` into a type variable according to the improvement 3.5.4. But the object `m` of type `(Mx C A)` expects on the Ocaml level a third argument, of type `__→__`. The use of an `Obj.magic` is thus necessary if one wishes to keep the type of `e` as generic as possible. In fact, if we would not have applied the improvement 3.5.4, the type of `mx_try` would have been made up primarily of `__`, and this function would have been typable without `Obj.magic`. But proceeding this way would do nothing but delay the problem, here at the time of the use of `mx_try`.

Lastly, for `Mx_bind`, the situation is similar to that of `Mx_try`:

```
(** val mx_bind :
    'a1 mx → ('a1 → 'a2 mx) → (__ → 'a3) → ('a2 → 'a3) → 'a3 **)

let mx_bind m f e k =
  Obj.magic m __ __ e __ (fun a → Obj.magic f a __ __ e __ k)
```

5.3.4 Usage of these exceptions

The contribution `Lannion` contains a certain number of applications of these exceptions to practical cases. We now describe the extraction of these various cases.

Unification of first order terms

In directory `Lannion/continuations/FOUnify_cps`, J.-F. Monin extends the study by J. Rouyer of a decision procedure for the unification of first order terms (cf `Nancy/FOUnify`). Simply, in the event of an unification failure for two subterms, an exception is now raised instead of finishing to treat all remaining computations.

In this simple case (only one exception, without contents), J.-F. Monin use in fact a simplified version of the exceptions presented previously, without polymorphism related to the ∀X. Instead of Mx, the type of the exceptions is:

```
Definition Nx (X:Set)(P:Prop)(e:P→X)(C:Prop)(A:Set) :=
  (C → P) → (A → X) → X.
```

And the fact of having X, P and e in parameters rather than in universally quantified variables largely improves the accuracy of the extracted type:

```
type ('x, 'a) nx = __ → ('a → 'x) → 'x
```

And the last remaining __ corresponds to the logical argument of type C→P. Finally, the extracted unification program is directly typable without Obj.magic, a fact that J.-F. Monin has already noted in [61]:

> *This example only includes one* try... with *which is outside the call to a recursive function. The impredicativity of* Mx *is thus not put at contribution, the extracted program is typable in ML.*

And our extraction does not generate here any superfluous Obj.magic.

Traversal of trees with exceptions

Here comes the examples of the directory Lannion/continuations/weight. The goal is now to compute the "weight" of a binary tree, namely the sum of all integers present at its leafs. And the use of exception is gradually introduced in order to answer the following (arbitrary) constraints:

- One stops the traversal if the current partial sum exceeds a certain preset quantity.
- One also stops the traversal if one meets a zero in one of the leafs.

In practice, these examples use exceptions of the same kind as the preceding example of unification, i.e. with a parameter X fixed in advance instead of an universal quantification on X. During extraction, one can note that these examples are more complex that the previous unification: some Obj.magic appear indeed. But these Obj.magic are used here just to ensure the conformity of the type of each subterm with the choices of our extraction of types. And in fact the extracted code remains typable here if these Obj.magic are removed.

Huffman's algorithm

The directory Lannion/polycont contains the first true example using the impredicative situation identified by J.-F. Monin, namely the use of exceptions in a recursive loop. As reminder, the Huffman's encoding algorithm associates a path[2] in a tree t to an object a that we wish to encode. Here is a manually coded version:

[2] that is a list of directions L or R

```
let encode a t =
  let rec lookup = function
    | Leaf → raise Not_found
    | Node(t1,b,t2) →
      if a=b then []
      else try L::lookup t1 with Not_found → R::lookup t2
  in lookup t
```

And here comes the version finally extracted from the Coq proof of J.-F. Monin:

```
(** val lookup : 'a1 → 'a1 tree → (__ → 'a2) →
                 (direction list sig0 → 'a2) → 'a2 **)

let lookup a t e x =
  let rec lookup0 = function
    | Leaf → (fun _ _ x0 _ x1 → x0 __)
    | Node (t1, b, t2) →
        (match eg a b with
           | true → (fun _ _ x0 _ x1 → x1 Nil)
           | false →
               mx_try (Obj.magic lookup0 t1) (fun h _ _ x0 _ x1 →
                 x1 (Cons (L, h))) (fun _ _ _ x0 _ x1 →
                 mx_bind (Obj.magic lookup0 t2) (fun l2 _ _ x2 _ x3 →
                   x3 (Cons (R, l2))) x0 x1))
  in lookup0 t __ __ e __ x

(** val encode : 'a1 → 'a1 tree → direction list sig0 **)

let encode a t =
  mx_try (fun _ _ x _ x0 → lookup a t x x0) (fun x → x) (fun _ → Nil)
```

It is thus noticeable that this version is very complex, and contains two `Obj.magic` (plus three hidden behind `mx_try` and `mx_bind`). In fact, as already remarked by J.-F. Monin, only one `Obj.magic` in front of `mx_try` is sufficient.

In addition, the presence of the `Obj.magic` seems to interact badly with the optimizations normally carried out by the extraction, which usually allows to remove the majority of the logical residues `_` and `__`. We intend to improve that in the future.

In any case, the generated code, without being elegant, has the advantage of being generated automatically, including the `Obj.magic`, and of being directly compilable.

Maximal sharing of common subterms

We will not develop the last example of this contribution, namely a tree transformation with maximum sharing of subtrees. Indeed the behavior of this example with respect to the extraction is similar to the one of Huffman.

5.4 Higman's lemma

We now study the extraction of another contribution taking advantage of the impredicativity, namely `Rocq/Higman`. This contribution is a constructive proof of Higman's lemma in the case of words on an two letter alphabet, implemented in Coq by H. Herbelin around 1992.

Higman's lemma is a combinatorial result which stipulates that for any sequence w_n of words on a finite alphabet, there exists two indices $i < j$ such that w_i is a sub-word of w_j, in the sense that it is enough to remove some letters of w_j to obtain w_i. We note $w_i \trianglelefteq w_j$ the latter relation on words. This lemma also has a formulation in term of well quasi-orders, this time on alphabets that are not necessarily finite any more. Since the alphabet considered here has only two letters, we do not use this generalized formulation.

This result gave place to a lot of works in the intuitionistic community, aiming at building a constructive proof of which the algorithm derived by extraction is the most effective possible. Instead of recalling here the complete history of these works, we prefer to refer the interested reader to the very good manuscript of M. Seisenberger [74], and in particular to the synoptic table p.47. For replacing the formalization of H. Herbelin in its context, let us just mention that an elegant classical proof of Higman's lemma has been found by Nash-Williams in the sixties, i.e. ten years after the work of Higman. This proof is based on an reasoning known as of "minimal bad sequence". Then several years later, in 1990, C. Murthy has formalized this classical proof in Nuprl in order to obtain from it a constructive version by using the A-translation of Friedman. Independently, H. Herbelin formalized around 1992 an A-translation "when required" in Coq, introducing a minimum of double-negations.

But it quickly appeared that the algorithm corresponding to this initial constructive proof had very bad computational properties, as we will see thereafter. This is why many other constructive proofs of this lemma has been proposed later on, such as for example [63] or [21]. According to H. Herbelin:

> *The motivation of T. Coquand was not so much the extraction, but the understanding of the share of impredicativity needed in the proof, in fact induction on trees with infinite branching, as well as the symmetrization of the proof, that is abstracting oneself to the need of ordering the alphabet. I should also add his very strong quest for elegance, and his concern of accessing the very core of mathematical theorems.*

The proof of [21] is in fact a reformulation of the one by Nash-Williams, in which the reasoning by contradiction has been made positive via the use of some ad hoc inductive types. This proof suggested by T. Coquand and D. Fridlender has since been formalized in Minlog by M. Seisenberger [74] and in Isabelle by S. Berghofer [13, 14]. Lastly, S. Berghofer has recently adapted his Isabelle proof into Coq. This new Coq formalization of Higman's lemma, which forms now a user contribution named `Muenchen/Higman`, is particularly concise while providing at the same time an extracted code excessively more effective than the code extracted from `Rocq/Higman`.

5.4.1 Higman and the impredicativity

We first illustrate the mechanism of the proof formalized by H. Herbelin, which is originally due to Nash-Williams for its classical part, then transformed of constructive proof by A-translation. In particular we now detail how this formalization relies strongly on impredicativity, and how this fact implies the presence after extraction of type non expressible in ML.

The proof is done by reasoning on an hypothetical “minimal bad sequence”. In this context, a “bad sequence” is a succession of words which would invalidate Higman’s lemma. Here sequences are formalized via informative relations[3] (*i.e.* on `Set`) connecting integers and words:

```
Definition seq := nat → word → Set.

Section Bad_sequence.

 Variable f : seq.

 Definition exi_im := ∀n, (∀x, f n x → A) → A.

 Definition uniq_im := ∀n x y, f n x → f n y → (x = y → A) → A.

 Definition cex := ∀i j x y, f i x → f j y → i < j → x ⊴ y → A.

 Inductive bad : Set := bad_intro : exi_im → uniq_im → cex → bad.

End Bad_sequence.
```

The properties `exi_im` and `uniq_im` stipulate that our relation `f` is in fact functional, by requiring respectively the existence and the uniqueness of the image of an integer `n` by `f`. On the way, one can note the effects of the A-translation. Intuitively, `A` plays the role of `False`, but will be finally used to contain the result of the constructive algorithm. For example, `(∀x, f n x → A)→A` is to be read informally as `¬(∀x, ¬(f n x))`, which is classically equivalent to `∃x, f n x`. Then, the property `cex` express the fact that `f` invalids indeed Higman’s lemma. And finally, a sequence `f` is “bad” if we have `(bad F)`, that is the three preceding conditions are met.

Once specified what is a “bad sequence”, let us pass now to the definition of a “minimal bad sequence”:

```
(* To be equal on the n-1 first terms *)

Definition eqgn (n : nat) (h h' : seq) :=
  ∀i, i < n → ∀s t, h i s → h' i t → (s = t → A) → A.

(* To be minimal on the nth term *)

Definition Minbad (n : nat) (h : seq) (y : word) :=
```

.../...

[3]H. Herbelin told us that this use of relations, and as a consequence the use of impredicativity, allows to avoid the use of the description axiom necessary to the proof via functions.

.../...

```
  ∀h' : seq, bad h' → eqgn n h h' → ∀z, h' n z → z ≺ y → A.

(* To be minimal on the first n-1 terms *)

Definition Minbadns (n : nat) (h : seq) :=
  ∀p, p < n → (∀y, h p y → Minbad p h y → A) → A.

(* The minimal (bad) counter-example *)

Definition Mincex (n : nat) (x : word) :=
  ∀C : Set,
   (∀h : seq, bad h → Minbadns n h → h n x → Minbad n h x → C) → C.
```

Let us detail mostly the last definition. One recognizes there in fact the impredicative encoding of an universal quantification, hence `Mincex` means informally:

```
Definition Mincex (n : nat) (x : word) :=
  ∃h:seq, bad h ∧ Minbadns n h ∧ h n x ∧ Minbad n h x.
```

One can also interpret `Mincex` as an infinite union over all sequences. And the impredicative character of this encoding comes of course from the quantification over all the types `C:Type`, while at the same time we wish `Mincex` to be a sequence, and thus has a result in `Set`.

After having built this smaller bad sequence in an abstract way, the proof then establishes that, if there is a bad sequence (not necessarily minimal), then `Mincex` is also a bad sequence (*i.e.* we have `bad Mincex`). The next step is to derive from any bad sequence another smaller bad sequence. By examining the sequence thus derived starting from `Mincex`, we finally obtain a contradiction (that is a proof of `A`) by taking into account the minimality properties of `Mincex`. Choosing `A` carefully allows to conclude, here to get that `∃i j, i<j → f i ⊴ f j`.

5.4.2 The extraction of Higman

Historically, this formalization was extracted by H. Herbelin around 1992, by using a version 5.x of Coq. This was one of the first versions of the Coq extraction, implemented then by C. Paulin and B. Werner. The target languages for extraction during these first experiments were Caml Lourd and also LazyML, a lazy version of ML, which one still finds today as base for the hbc compiler for Haskell.

In addition to the interest of lazyness, close to the reduction strategy of Coq, LazyML presented as other interest at that time to have an compilation option deactivating completely the type-checking.

In fact, we have observed that there is only one location in all the extracted code which requires a workaround for getting ML typability. This is rather surprising compared to the central role of `Mincex` in the formalization and its ML-incompatible type. The current extraction retrieve indeed automatically this problematic point, and inserts there an `Obj.magic` :

```
let snake badf f0 f_0_f0 a h n h0 h1 =
  xa_elim a n h1 (fun y h3 →
    Obj.magic h3 __ (fun _ h5 h6 h7 h8 →
      h8 __ (badfx (badMin badf f0 f_0_f0) a h n (Leastp_intro (h1,h0)))
        (fun x _ x0 x2 x3 x4 x5 →
        eqgn_fxMin_h badf f0 f_0_f0 a n h5 h6 x x0 x2 x3 x4 x5) y (Fxtl
        (n,(Leastp_intro (h1,h0)),h3)) __))
```

On the other hand, just as for Lannion, some other "superfluous" `Obj.magic` are also added. They are used to make coincide the approximate type chosen for `Mincex` and the types of the functions handling such `Mincex` objects. If the goal is only to compile the file `higman.ml` created by the extraction, then these 14 additional `Obj.magic` can be removed. But then the obtained file is not any more compatible with the interface `higman.mli` automatically generated.

Let us return now to the study of the obtained program. The extracted code is relatively short (435 lines) but incomprehensible even with a good preliminary knowledge of the initial proof, as shown by the previous excerpt. Concerning the execution of this program, it is possible, at least for small examples:

```
./higman 101 0110 01010
f(0)=101
f(1)=0110
f(2)=01010
f(3)=...
==> f(0) is included in f(2)
```

This display has been printed by the standalone program obtained by combining the extracted code with a small manual interface allowing a simple input of a word sequence prefix to test (see the contents of contribution `Rocq/Higman`). This interface also allows to randomly choose words to test the behavior of the algorithm when it must explore longer sequences and/or larger words. But these tests quickly encountered an ineffective behavior of the program. Indeed, as soon as the prefix to manipulate contains more than a dozen elements, it becomes frequent to obtain a fast but disappointing answer:

```
./higman --random
Fatal error: exception Stack_overflow
```

We have also tested an extraction to Haskell [4]. As the extraction does not generate yet any `unsafeCoerce` (the functions equivalent to `Obj.magic` in Haskell), we have inserted one manually in the function `snake`. We then obtain a program with a different behavior: no more stack overflows, but really slow computation times. For example, we easily exceed one minute of computation on a recent machine for a sequence for which the prefix to be

[4] Unlike the extraction to Ocaml, the corresponding files of this experimentation are not yet part of `Rocq/Higman`. The interested reader can still contact the author.

considered is made of less than ten elements. And these computation times grow very quickly when the length of the prefixes increases, in an apparently exponential way.

Our experiments thus join the former analysis announcing disastrous effectiveness for the algorithm obtained by (partial) A-translation. Moreover, taking into account the complexity of the produced code, it seems difficult to analyze a posteriori this code for understanding how to improve it.

5.4.3 The new proof of Higman

We now evoke a new proof of Higman's lemma in Coq, still in the version restricted to an alphabet of two letters. This new version, that dates back to 2003, is due to S. Berghofer, and is now named `Muenchen/Higman` as a contribution.

This section is intended for to illustrate the progresses carried out since 1993 in the search of an effective constructive proof of Higman's lemma. On the other hand, contrary to all the others examples detailed starting from this chapter, the extraction of this new formalization is not of great interest from the strict point of view of the Coq extraction. Indeed, the types handled here are very simple, without recourse to the impredicativity or to quantifications of an higher nature. Finally, one obtains some extracted code perfectly typable in ML, free from any `Obj.magic`, and that the old extraction could have managed.

A new approach in the proof

As mentioned in our introduction to Higman's lemma, the concept of this new formalization is due to T. Coquand and D. Fridlender [21]. This is a use of a principle named "bar induction", which is a form of induction on trees with infinite branching. This induction principle allows to turn in a positive way the reasoning by contradiction of Nash-Williams. In fact, the current implementation in Coq is only an adapted version of the implementation in Isabelle made by S. Berghofer. In addition, a similar proof was also implemented in Alf by D. Fridlender and in Minlog by M. Seisenberger [74]. We do not describe here in detail the proof, very well described in [14], p. 104. Let us simply announce that instead of relying on an hypothetical smaller counterexample `Mincex`, one now defines a predicate bar on the finite sequences of words:

```
Inductive L (v : word) : list word → Set :=
  | L0 : ∀w ws, w ⊴ v → L v (w::ws)
  | L1 : ∀w ws, L v ws → L v (w::ws).

Inductive good : list word → Set :=
  | good0 : ∀ws w, L w ws → good (w::ws)
  | good1 : ∀ws w, good ws → good (w::ws).

Inductive bar : list word → Set :=
  | bar1 : ∀ws, good ws → bar ws
  | bar2 : ∀ws, (∀w, bar (w::ws)) → bar ws.
```

This predicate `bar` thus expresses either that one already found the two overlapping words $w \trianglelefteq v$ in the current finite prefix, that is to say that all prolongation of this prefix leads later to the validation of the lemma. This is in in fact a constructive formulation of the good foundation of $\trianglelefteq$. The key point of the demonstration is then to establish `(bar nil)`. For that we use two other inductive predicates:

```
Inductive R (a : letter) : list word → list word → Set :=
  | R0 : R a nil nil
  | R1 : ∀vs ws w, R a vs ws → R a (w::vs) ((a::w)::ws).

Inductive T (a : letter) : list word → list word → Set :=
  | T0 : ∀b w ws zs, a ≠ b → R b ws zs → T a (w::zs) ((a::w)::zs)
  | T1 : ∀w ws zs, T a ws zs → T a (w::ws) ((a::w)::zs)
  | T2 : ∀b w ws zs, a ≠ b → T a ws zs → T a ws ((b::w)::zs).
```

And with these five predicates (plus $\trianglelefteq$), we have finished the tour of all the structures used by this formalization! The remainder is only a succession of small lemmas aiming at establishing `(bar nil)` in a first time, then at extracting from that the two expected indices `i` and `j`. On the whole, this gives us a proof of 220 lines of `Coq`, which is remarkably short[5] compared to the 1085 lines of the proof obtained by A-translation from Nash-Williams's proof.

The extraction

After extraction, one obtains 165 lines of code, which hence gives us an extremely low ratio (`Coq` script)/(extracted code), ratio which is usually nearer to 10 (see for example `Orsay/FSets` in chapter 7). In addition, this extracted code is extremely simple. Here is for example the largest and most complex extracted function:

```
let rec prop2 a b xs h ys x zs x0 x1 =
  match h with
    | Bar1 (ws,g) → Bar1 (zs,(lemma3 ws zs a x0 g))
    | Bar2 (ws,b0) →
        let rec f l0 b1 zs0 h2 h3 =
          match b1 with
            | Bar1 (ws0,g) → Bar1 (zs0,(lemma3 ws0 zs0 b h3 g))
            | Bar2 (ws0,b2) → Bar2 (zs0,(fun w →
                match w with
                  | Nil → prop1 zs0
                  | Cons (l1,l2) →
                      (match letter_eq_dec l1 a with
```
.../...

[5] In fact, we have re-worked the proof of S. Berghofer in order to use as much as possible the automatic tactics of `Coq`, which reduces appreciably the size of this contribution, whereas this have not been done in the case of `Rocq/Higman`.

…/…

```
                                  | Left →
                                      prop2 a b (Cons (l2, ws))
                                        (b0 l2) ws0 (Bar2 (ws0, b2)) (Cons ((Cons
                                        (a, l2)), zs0)) (T1 (l2, ws, zs0, h2)) (T2
                                        (a, l2, ws0, zs0, h3))
                                  | Right →
                                      f (Cons (l2, ws0)) (b2 l2) (Cons ((Cons (b,
                                        l2)), zs0)) (T2 (b, l2, ws, zs0, h2)) (T1
                                        (l2, ws0, zs0, h3)))))
          in f ys x zs x0 x1
```

Even if the two internal branches are not very readable due to an imperfect pretty-printing, one recognizes at least the algorithmic structure of this function, which was very difficult with `Rocq/Higman`.

But the major progress with respect to `Rocq/Higman` is the gain in execution speed. The program obtained can now handle prefixes of more than one thousand of words in a few tens of seconds, each word having a random length ranging between 20 and 80.

Chapitre 6

Constructive reals and extraction

In this chapter, we present some works that we carried out in collaboration with several others European researchers. The goal of these works is obtaining a certified library of exact real arithmetic, and this by extraction from constructive formalizations of real analysis in Coq.

The first of these works relates to the C-CoRN project (previously known as FTA) from the university of Nijmegen in the Netherlands. The study of the extraction for certain parts of this project was started by two members of the Nijmegen team, namely L. Cruz-Filipe and B. Spitters. In particular, they have presented this first study in a publication, whose reference is [25]. We see in the first part of this chapter that a lot of progresses have been accomplished since the idea of extracting FTA was first launched, but also that a great number of difficulties remain, in particular concerning the effectiveness of the extracted code. This first part of the chapter is very close to chapter 6 to [24], since they both report the same work, simply approached under slightly different angles.

More recently, we had the opportunity to have long discussions with H. Schwichtenberg concerning the extraction of constructive real numbers during the summer school 2003 in Marktoberdorf. And these discussions were prolonged by a visit of one week in Munich in September 2003. We then realized together the outline of a formalization of constructive reals in Coq. This mini-formalization is obviously without common scale with C-CoRN from the point of view of the size, but its interest is to be right from the beginning thought in term of extraction. And this study has brought new insights to the difficulties encountered with the extraction of C-CoRN.

6.1 The extraction of the C-CoRN project

6.1.1 Description of the FTA/C-CoRN project

FTA is the abbreviation of the Fundamental Theorem of Algebra. This theorem stipulates that any not-constant polynomial with complex coefficients has at least a root in $\mathbb{C}$. In 1999 and 2000, the group directed by H. Barendregt at the university of Nijmegen has formalized in Coq a constructive proof, due to Kneser, of this result [35]. This formalization is really impressive, the final version of FTA being made of 40 000 lines of Coq scripts (see http://www.cs.kun.nl/gi/projects/fta/).

The algebraic structures of FTA

The principal reason of such a width is the construction from scratch of a whole stack of constructive algebraic structures, since the standard library of Coq do not comprise such algebraic structures adapted to the need for this formalization. These structures consist in particular of:

- CSetoids
- CSemiGroups
- CMonoids
- CGroups
- CRings
- CPolynomials
- CFields
- COrdFields
- CReals
- CComplex

It should be noted that the CSetoids are based on a calculative relation of difference, or “apartness”, rather than on a relation of equality, which could not be constructive for fields like real numbers.

In addition, these structures are nested the ones in the others via the use of Coq coercions. For example, here is the definition of semigroups:

```
Definition is_CSemi_grp (A:CSetoid)(unit:A)(op:CSetoid_bin_op A) :=
  Associative op.

Record CSemi_grp : Type :=
  { csg_crr   :> CSetoid;
    csg_unit  :  csg_crr;                     (* non-empty *)
    csg_op    :  CSetoid_bin_op csg_crr;
    csg_proof :  is_CSemi_grp csg_crr csg_unit csg_op
  }.
```

A CSemi_grp is thus composed of a CSetoid which is additionally required to be non-empty (since it is equipped at least of an element), and must also contain an associative operation on the other hand. Without going more into the details, let us just note that the coercion :>ensure that any CSemi_grp is also visible as a CSetoid. And similarly for the other algebraic structures: each one is based on a preceding structure, and adds new objects and/or properties. One thus obtains a linear chain of coercions going from a CComplex down to a CSetoid. The polynomials form the only structure that deviates from this continuous chain.

The definition of these structures and their basic properties deserve between one third and one half of the 40 000 lines previously mentioned, including a big part devoted to the basic properties of polynomials. In fact, once proved the intermediate value theorem[1] and the existence of n-th roots[2] in $\mathbb{R}$ and $\mathbb{C}$, there only remain approximately 3 000 lines devoted to Kneser's lemma and FTA itself[3].

Some axiomatized reals, later realized

Let us mention another crucial point of the FTA architecture. The real numbers have been used in an "axiomatic way" [36]. Indeed, after defining in `CReals.v` the Coq type of the algebraic structures that are isomorph to the real numbers, all the remainder of the FTA proof is done with respect to such a particular structure. In fact, for technical convenience [4], this structure is posed as an axiom:

```
Axiom IR : CReals.
```

This way, any particular representation of the constructive real numbers can replace this axiom `IR`. And such a particular representation `Concrete_R` was developed later on by M. Niqui, based on the Cauchy sequences. The transformation of the previous axiom in the following definition should normally not modify the validity of the remainder of FTA :

```
Definition IR : CReals := Concrete_R.
```

From FTA to C-CoRN

Since the completion of the FTA proof itself, this formalization was gradually reorganized in a more ambitious project. The proof of the fundamental theorem of the algebra is now just one of the the facets of this new C-CoRN project (for Constructive Coq Repository At Nijmegen). This project is aimed at becoming a wide library of constructive mathematical results based on the hierarchy of algebraic structures described previously. C-CoRN contains already, in addition to FTA, some extended results concerning series, usual transcendent functions, and especially a part named FTC (for Fundamental Theorem of Calculus). L. Cruz-Filipe has proved there that integration and derivation are two reciprocal processes [23].

6.1.2 The first extraction attempts

Our first contact with FTA dates back to October 2001. In a mail, M. Niqui reported his first attempt at extracting FTA : "*after 24 hours my machine ran out of memory*", the mentioned machine being quite reasonable.

[1] See the file `IVT.v`.
[2] See the files `NRootIR.v` and `NRootCC.v`.
[3] See the files `KeyLemma.v`, `MainLemma.v`, `KneserLemma.v`, `FTAreg.v` and `FTA.v`.
[4] At that time, the modules and functors did not exist in Coq.

It should be said that FTA have not been initially conceived with the extraction in mind. Of course, as this development is constructive, its authors were conscious of the theoretical possibility of extracting a program from it. But this possibility, considered to be unrealistic at the time, has not influenced the initial design choices. In particular, all the development has been placed initially in the sort `Prop` of the objects precisely ignored by the extraction. For his first extraction attempts, M. Niqui then simply replaced all the occurrences of `Prop` by `Set`. Instead of ignoring everything, the extraction now keeps and translates everything. Since this development of 40 000 lines of Coq script generates complex and bulky proof terms, it is understandable that this extraction is excessively more demanding than the one of other examples met up to that point, hence the problems of computing time and memory occupation of M. Niqui.

In fact, it appeared thereafter that the extraction of this version of FTA using `Set` as universe was not so unfeasible after all, provided that one uses slightly more recent hardware and in addition corrects the extraction by removing one heuristic "optimization" more than hazardous, which as a consequence the replacement of too many constants by their definitions. In the particular case of FTA, that was leading to an amazing increase in the code size. Here what L. Cruz-Filipe reported in February 2003:

> *You might find it interesting to know that we actually managed to extract the *original* FTA version (e.g. the one currently in the Coq library, with all the logic in Set) after you fixed the inlining bug in the extraction routine. We still almost ran out of resources (on a 2GHz machine with 1Gb RAM memory and 2Gb swap), but the extracted code is "only" around 13 Mb.*

This figure of 13 Mo of source code seems really aberrant for a program which, let us recall, is just supposed to compute approximations of roots for complex polynomials. One can nevertheless relativize somewhat these 13 Mo. They are indeed due in great part to two unpleasant, but simple phenomena:

- the first problem is due to the attempts made by the extraction to embellish the display (or "pretty-print"). This printing is done by means of display boxes, vertical or horizontal, provided by the Ocaml language. However this mechanism works with a fixed width of line, which is 80 columns by default. In the case of the disproportionate functions of FTA, respecting at the same time the indentation and this width limit implies to frequently use only the last quarter of the lines, sometimes even less. Finally more than half of the extracted file is made up of white spaces at the beginning of lines.
- Looking quickly at this enormous extracted file, the other point immediately noticeable is the omnipresence of projections resulting from coercions. For example, for a structure like that the real numbers, the addition is done by considering $\mathbb{R}$ as a semi-group `CSemi_grp`, thanks to a succession of coercions, then by using the operator inside the field `csg_op` of this `CSemi_grp` structure. In fact, the Coq user does not have to be concerned by these coercions, since they are implicit. If `IR` is the structure of real numbers, one can form `IR.csg_op` directly. But these coercions are stored explicitly in the proof term, and are thus found also in the final extracted term. Thus the extraction of the addition over reals is:

```
iR.crl_crr.cof_crr.cf_crr.cr_crr.cg_crr.cm_crr.csg_op
```

One can recognize here the successive projections into the substructures of ordered field, then field, then ring, then group, then monoid, and finally semi-group, before the final projection into the field containing the addition operator. Finally, one needs almost a line to write this addition, who appears several hundreds of times in the extracted file. And the same applies to other elementary objects like `0`, `1` and the remaining operations.

It is of course possible to factorize the produced code in order to avoid, at least partially, such chains of projections. It would thus be enough to define once and for all a constant `iR_plus` equal to the previous object. But what could be easily done for a fixed structure like `iR`, is much more complicated and less effective to do it in first half of the extracted file, where one reasons on unknown structures, given as argument of functions.

In fact, even after manually factorizing these spaces and these coercions chains, the source code still weight several megabytes. It thus seems obvious that a substantial part of this code has in fact no interest at the algorithmic level.

6.1.3 Distinction between logical parts and informative parts

To allow the extraction to at least partially eliminate the dead code from these extracted terms, L. Cruz-Filipe and B. Spitters have then modified again the initial development, and tried to identify the parts that could remain in `Prop`. This work is described in detail in [25] and in the beginning of chapter 6 of [24].

It should be well understood that FTA is an atypical development with respect to the distinction between informative and logical parts. Usually, it is rather easy to see the distinction between informative operations like `1+2` and logical assertions like `n=0`$\vee$`0<n`. But in FTA, it is crucial for the relation `<` over reals to be informative: behind `x<y` is in fact hidden a strictly positive rational below `y-x`. And this rational is indeed used in practice in operations, for example when computing the inverse of a non-null real (see for example p. 148 of [24]). In the same way, difference between two real has some computational contents.

On the opposite, the equality and the relation $\leq$ are not decidable over reals and can thus remain logical. In fact, these two relations are defined as the respective negations of the apartness and strict order `<` relations. This gives them a logical status, since `False`, used in negations, always remains in `Prop`.

These decisions for basic relations then influence the status of operators like disjunctions or conjunctions. For example, one will not be able to use the usual logical disjunction `or` in the example `n=0`$\vee$`0<n`, since the right part is informative. In particular, this expression is not equivalent constructively to `0`$\leq$`n`, which is completely logical.

One sees that using the `Prop`/`Set` distinction in the case of FTA is not obvious, which explains the initial choice of "everything logical", and then the switch to the converse in a second time. L. Cruz-Filipe and B. Spitters have in particular had to alter certain portions of the initial development, similarly to the previous `or` replaced by an ad hoc disjunction. But this task was worth it, since one obtains a gain of a factor 10 on the size of the extracted code.

In addition to this distribution between `Prop` and `Set` for all constructions of FTA, L. Cruz-Filipe and B. Spitters also noted that it was possible to decrease even more the size of the extracted terms thanks to some clever modifications of the proofs relating `<` and $\leq$. First of all, they have exploited the fact that the two following formulations of the Cauchy property are equivalent:

$$\forall \varepsilon > 0 \ \ \exists N : \texttt{nat} \ \ \forall m, n > N \ \ |x_m - x_n| < \varepsilon$$

and

$$\forall \varepsilon > 0 \ \ \exists N : \texttt{nat} \ \ \forall m, n > N \ \ |x_m - x_n| \leq \varepsilon$$

Using the second alternative then results in making disappear from the extracted terms the major part of these Cauchy properties, and only keeping the essential part, namely the part building indeed the bound N starting from a given ε.

Another example of optimization concerns the proofs of properties of the form $a < b$. Initially, these proofs were frequently done by successive reasoning as in $a < x_1 < x_2 < x3 < b$. In fact, only one of these stages really requires a strict order: it is enough for example to establish $a < x_1 \leq x_2 \leq x_3 \leq b$, and only this first stage of reasoning will remain in the extraction.

Finally, thanks to such transformations of the FTA proof, and in particular the redesign of the division part, L. Cruz-Filipe and B. Spitters have reduced the size of the extracted code down to a little more than 200 KB, that is to say almost 100 times less than during the first successful extraction. Moreover, this extraction is now carried out in a few seconds, even if the speed of extraction has never been yet a major design goal for our implementation . Lastly, this size of 200 KB still includes multiple redundancies related to coercions/projections, which are not solved yet in a satisfactory way. On the other hand, reducing the size of the extracted code largely attenuates the problems of indentation for this extracted code.

6.1.4 Compilation of the extracted code

After bringing the extracted code from FTA on a more reasonable scale of a few thousands of lines, the following challenge consists in trying to build a program which can be run and return some useful result. In fact, during the first successful extractions of FTA, we encountered immediately an important obstacle, namely the non-typability of this extracted code. And we were far from the situation of contributions like `Lannion` and `Higman` as in the previous chapter, where it was still possible to insert manually one or two `om`. Here, the typing conflicts amount to hundreds, which almost impose the use of an automatic method. Fortunately, we have in the meantime implemented an automatic insertion of `Obj.magic` at the locations of these typing conflicts (see chapter 3). Here, approximately 400 of these `Obj.magic` are inserted in the extracted code. And this code is indeed accepted by the compiler Ocaml without any additional modification.

If we look more closely at these typing conflicts, we find some of the situations evoked in chapter 3. For example, the first problem that appears is related to the `CSetoid` structure:

```
Record CSetoid : Type :=
  { cs_crr   :> Set;
    cs_eq    :  (Relation cs_crr);
    cs_ap    :  (Crelation cs_crr);
    cs_proof :  (is_CSetoid cs_crr cs_eq cs_ap)
  }.
```

A `CSetoid` is thus an unspecified type provided with two equality and apartness relations. In fact, this example is similar to the inductive type `any` of page 92. In both cases, one embed a type in a structure, and subsequent fields of this record depend over this type. This structure does not have counterparts[5] in ML, and the extraction produces the following approximation, in which `cs_crr` was replaced by the unknown type $\mathbb{T}$ (or `__`):

```
type cSetoid = { cs_ap : __ crelation; cs_proof : (__,__) is_CSetoid }
```

The use of the fields `cs_ap` or `cs_proof` then leads frequently to untypable situations, which force the use of `Obj.magic`.

6.1.5 The execution of the extracted program

Once settled the question of compilation, we then designed a small input-output interface in order to be able to use this code and to test its efficiency. As FTA handles real numbers as Cauchy sequence, our interface just asks for an rank n and a predefined real x to approximate, and then returns the rational number x_n constituting the n-th element of the Cauchy sequence x. More exactly, this interface *tries* to return x_n. Very quickly, it has appeared that this extracted program suffers from huge efficiency problems. In particular, in the case of the "canonic" example of the computation of $\sqrt{2}$ seen as a root of the polynomial $x^2 - 2$, the program seems to need a few centuries, except in the case of the first approximation ... which is worth zero. As L. Cruz-Filipe has said once, "the only good news was that this program required very little memory to run".

In fact, to obtaining an efficient extracted program is a challenge at least as difficult as the first task of giving it a reasonable size. And the resolution of this new challenge is still in progress today even if much has already be done. The remaining of this section reports the various transformations and optimizations that we tried on FTA, and the progresses thus realized. It should be noted that this part corresponds to a phase of very active collaboration with the Nijmegen team and in particular with L. Cruz-Filipe, whereas our personal contribution to this study of FTA had been previously limited to only correct the dysfunction of the extraction leading to too many unfoldings of constants.

[5] In fact, Ocaml 3.07 allows in records a certain form of abstraction over types. We can write `type setoid = {eq : 'a.'a→'a→bool}`. But this is not really appropriate for our needs here, because one particular equality `eq_int: int→int→bool` cannot to be used to build such a `setoid`.

Improving the datatypes

Our first try for improving the extracted program has been to hunt and remove ineffective datatypes. Indeed, in other development, this sole stage had sometimes allowed to produce reasonable programs starting from developments not initially planned to be extracted, such as for example the contribution `Bordeaux/Additions` of P. Castéran. In the case of FTA, a first modification has been related to the rational numbers used by M. Niqui to model real numbers via Cauchy series. These rational numbers were indeed at first records where the first field was an integer representing the numerator of the fraction, and the second field was a natural number coding the denominator. But as much the integers are effective since coded in a binary way in the Coq type `Z`, as much the natural numbers used at first were `nat`, coded in a unary way. Our first contribution to FTA has thus been to replace in the denominator the type `nat` by the type `positive` of strictly positive integers, encoded in binary notation. Not only has this largely accelerated the operations on the rational numbers, but it has even simplified the formalization: there is no need any more to be concerned with the case of a null denominator.

A redesign of the real number model

In fact, the previous modification was rather disappointing. Of course, this improvement of the rational numbers deserved to be made, but it did not induce a visible effect on the first tests: computations which were diverging continued to do it. Worse, the extracted code persisted, in spite of this change, to contain hundredths of values in `nat`, including several constants up to 48. In fact, this kind of constants was useful primarily during the construction of a model of reals by the means of Cauchy sequences. And L. Cruz-Filipe finally understood how to alter completely this part, so that a gigantic proof like `Rmult_is_extensional`[6], long of 1800 lines of Coq script, is now obtained in ten lines. And the size of the extracted code still decrease, the majority of these bulky constants of type `nat` disappear ... but however the effectiveness is still not improved concerning the computation of $\sqrt{2}$ via FTA.

Some less ambitious tests

In front of such difficulties, we then tried to better identify the causes of inefficiencies via more progressive tests. It then appeared that all arithmetical computations on numbers rational are almost instantaneous, even when these rationals are seen as reals[7] or when these computations are done via the use of polynomials. On the other hand, the computation of $\sqrt{2}$ as reciprocal of 2 by the function $\lambda x.x^2$ diverges, while at the same time this way of obtaining $\sqrt{2}$ is normally much simpler than the use of FTA on the polynomial $x^2 - 2$. By seeking examples of simple real numbers that have nevertheless non-constant Cauchy sequence, L. Cruz-Filipe then suggested to consider limits of series, and in particular $e = \sum_{n=0}^{\infty} \frac{1}{n!}$ and to a lesser extent $\pi = 4 \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1}$. For the first time, we then obtained finishing not-trivial

[6] This result stipulates that for all reals such as `a*b`$\neq$`a'*b'`, then `a`$\neq$`a'` or `b`$\neq$`b '`. This is the dual of the usual statement on the equality, which states that two multiplications of equal numbers produce equal numbers

[7] We then considers constant Cauchy sequence.

computations. Here come for example the fractions returned as first terms of the Cauchy sequence representing the constant e. Without surprise, they are partial sums of the previous series.

rank	fraction	value
0	0/1	0.000000000
1	1/1	1.000000000
2	2/1	**2**.000000000
3	5/2	**2**.500000000
4	32/12	**2**.666666666
5	780/288	**2.7**08333333
6	93888/34560	**2.71**6666666
7	67633920/24883200	**2.718**055555
8	340899840000/125411328000	**2.7182**53968
9	13745206960128000/5056584744960000	**2.7182**78769
10	4987865758275993600000/1834933472251084800000	**2.718281**525
11	18099969098565397826764800000/6658606584104736522240000000	**2.7182818**01

In fact, we were unable of obtaining all these fractions during the first tests, since computing the approximation of rank 7 took already more than one hour, for experimentally giving only three correct digits. The reason of such a limited effectiveness lies in the manner of computing the terms $\frac{1}{\mathtt{n}!}$ of the series. Indeed the real number $\mathtt{n}!$ was initially calculated in type `nat` via the factorial `fac : nat→nat`, then only in a second time injected into reals via the function `nring : ∀R:CRing, nat→R` plus some more coercions. And this last function just transforms a unary integer in a succession of additions $1 + \ldots + 1$ in the considered ring `R`, here the one of reals. This method thus implies a huge number of computations on the factorial numbers with a *unary* coding of numbers. We then proposed to place the computation of factorial in the type `positive`, followed by the use of a function `pring : ∀R:CRing, positive→R`, which injects integers into a ring by now transferring their binary codings.

Unfortunately, this improvement did not have the expected effect. After some investigations, it appeared that the next bottleneck was the inversion of $\mathtt{n}!$ in $\frac{1}{\mathtt{n}!}$. In fact, in FTA, the division is not a binary operator `a/b` but an ternary operator `a/b//h`, where `h` is a non-nullity proof for `b`. And as we have already mentioned previously, the computational content of `h` is indeed used for constructing the division. In the case which interests us, the non-nullity proof of $\mathtt{n}!$ was given by a term `(fac_ap_zero n)`, and the structure of this term was initially isomorph to $\mathtt{n}!$ coded in unary. But these proofs were of the form $0 < 1 < \ldots < \mathtt{n}!$. The utilization of the previously mentioned techniques allows us to pass to a proof of the form $0 < 1 \leq \ldots \leq \mathtt{n}!$. The new estimated complexity of the computation of $\frac{1}{\mathtt{n}!}$ is then close to the size of the writing in base two of `n!`, that is to say $\ln(\mathtt{n}!) \sim_\infty \mathtt{n} \ln(\mathtt{n})$. And indeed, this version allows to obtain in a few hours all the fractions presented above, which was impossible beforehand.

During these experiments, we noted that the computation of the approximation of rank $k+1$ requests approximately ten times longer than the computation for the rank k. In the same time, the accuracy is also increased by a ten factor: on average, a new correct decimal digit is obtained at each step. This shows that the extraction is able to handle fractions of consequent size, but that the speed is still not perfect: at that pace, one should be ready for a computation of several month in order to obtain ten correct digits.

The issue of the computation duplications

In order to more finely understand these computations, we have then used some "profiling" techniques on the executions. And we noted a few curious things. For example, the computation for the approximation of rank 7 generated 14 calls to a function named `e_series`, which computes for a given `n` the value of $\frac{1}{n!}$. As the required approximation was only supposed to use the first 7 terms of the series, we deduce that the computations of these terms were duplicated. At first, we suspected that this computation redundancy was coming from the use of `e_series` in the function `e_series_conv: (convergent e_series)`, which proves the convergence of the series whose terms are given by `e_series`. But this was a false track, since `e_series_conv` only appears in a record field that is never used. In fact, we finally identified the duplication origin, located in the following definition:

```
Definition LimR_CauchySeq (a:CauchySeq R_COrdField') :=
  Build_CauchySeq F
    (fun m ⇒ CS_seq F (CS_seq _ a m) (T (CS_seq _ a m) m))
    (CS_seq_diagonal a).
```

The type `R_COrdField'` is the set of Cauchy sequence built on an archimedian ordered field. And the construction `LimR_CauchySeq` allows to build the limit of a Cauchy sequence of such sequences, by means of an diagonal argument. The problem here is the repetition of `(CS_seq _ a m)`. If one wants to avoid a double computation when the sequence `a` is instantiated by `e_series`, it is necessary to perform a factorization:

```
Definition LimR_CauchySeq (a:CauchySeq R_COrdField') :=
  Build_CauchySeq F
    (fun m ⇒ let b := CS_seq a m in CS_seq F b (T b m))
    (CS_seq_diagonal a).
```

Here, the two occurrences of this duplicated subterm were in the same context. One can thus imagine an automatic tool able to identify this redundancy and to factorize it. But in addition to its cost, such a factorization stage will have to take difficult decisions in more complex situations. For example, when these multiple occurrences can perfectly not been evaluated, must one then risk to cause additional computations by creating a "let-in"?.

In FTA, one also meets very frequent duplications due to the dependent types. For example, to build the fraction $\frac{1}{5}$ in a field `F`, one uses the ternary division `One/(pring R 5)//(pring_ap_zero F 5)`. However the proof part of non-nullity, `(pring_ap_zero F 5)`, of type `(pring F 5)`$\neq$`Zero` almost undoubtedly contains the term `(pring F 5)`, which is thus at least computed twice during this division. Similarly, our new proof of `fac_ap_zero` in the previous section still contains an occurrence of `n!`, even if this proof is now of the form $0 < 1 \leq \ldots \leq n!$.

This situation is really common in FTA, whereas in a more standard development the proof part would be purely logical, and would disappear without causing redundant computations. Sometimes, fortunately, the duplication is potentially present in FTA, but skipped, as for `e_series` above. But these duplications can also change dramatically the complexity, as soon as they intervene within recursive functions. This probably explains to a large extend the inefficiency of the program extracted from FTA. And correcting automatically this kind of redundancies during the the extraction would be really delicate, because the problem can be distributed between several functions, as with the division, `pring` and `pring_ap_zero`. Currently, the only answer to these duplications is a manual analysis a posteriori, by `profiling`, which is long and painful, and only allows to locate the most obvious problems.

A too constraining axiomatization of reals

When we think of it, it is really disconcerting to have to devote so many efforts in order to obtain an effective solution for computing a real number like $\frac{1}{\mathtt{n!}}$, which is after all only a rational. Why not make directly this computation in the structure of rational numbers, and inject it into the reals only in a second time?

Unfortunately for the extraction, it is not possible to directly do so. The problem is located at the distinction in FTA between the abstract structure of the real numbers and its concrete counterpart. In fact, all the parts using real numbers, such as for example the proof of FTA itself, use an abstract real structure, axiomatized:

```
Axiom IR : CReals.
```

And this type `CReals` is only specified as being an archimedian ordered field where all Cauchy sequence admit a limit. In this framework, one has only access to a minimal number of primitive objects and basic properties, resulting from the underlying structures.

In particular, the only known primitive reals are 0 and 1, which are respectively the elements `csg_unit` and `cr_one` of the semi-group and ring structures `IR`. Instead of injecting directly rational numbers in `IR`, one must rebuild them by using 0, 1 and the operations $+$, $*$ and $/$, this last operation requiring moreover a proof of non-nullity as third argument. Such proofs often involve in fact some proofs of strict positivity or negativity, which one must also build from a restricted core of basic properties. Here come for example the only properties known initially concerning the order $<$ over `IR`:

- the antisymmetry
- the transitivity
- the compatibility with respect to the addition: $x < y$ implies $x + z < y + z$
- the conservation of positivity by multiplication: $0 < x$ and $0 < y$ imply $0 < x * y$
- the dichotomy: $x \neq y$ implies $x < y$ or $y < x$
- the archimedian property, stipulating that any real number can be bounded by the injection in `IR` of an adequate integer.

A property as basic as $0 < 1$ is hence not primitive, but derived from the preceding properties. And it is obviously the same for more complex proofs as $0 < \mathtt{n}!$.

On the other hand, FTA also provides a concrete model named `Concrete_R` of the real numbers, built on the Cauchy sequences of rational numbers. And in this model, it is immediate to inject a rational `q` into `Concrete_R` via a function `inject_Q : Q→Concrete_R`. It is indeed enough to take the Cauchy sequence where all the terms are worth `q`. In the same way, a proof of the form `a<b` can be much more direct in `Concrete_R`, since we can access now the *definition* of `<`, that claims the existence of a strictly positive rational Δ and a rank `N` such that all terms of rank greater than `N` in the two Cauchy sequences being compared are always apart by more than Δ. In particular, for two rational `q<q'`, we immediately obtain `(inject_Q q)<(inject_Q q')` in `Concrete_R`, by taking `Δ=q'-q` and `N=0`.

This separation between abstract reals and concrete reals is without doubt beneficial at the mathematical level, because it ensures that a proof made at the abstract level is independent of the particular representation selected at the concrete level. One can later on change this concrete model with no risk for the abstract proofs. On the other hand, from the programming point of view, we are here in the presence of two modules interacting via an interface way too minimalist, which obliges the upper level module to frequently reinvent the wheel, and moreover in an ineffective way. What would be said of an integer arithmetic module whose interface would not export the multiplication, under the justification that one can simulate it by repeated additions?

To confirm that this distinction between the concrete and abstract levels constituted indeed a bottleneck for program extraction, we have added to the abstract level a few axioms, which we then realized at the concrete level. Then we ordered the extraction to replace the axioms by their concrete realizations. This experiment had a spectacular effect on the test computing the approximations of Euler's constant. Instead of painfully obtaining the approximation of rank 11 in more than one hour, we can now compute that of rank 100 in 77 seconds. The fraction obtained fills two screens, and gives 157 correct digits. As for complexity, it seems only to double every ten ranks, instead of being multiplied by ten for each additional rank. Here come some details on the code allowing that. First of all, we add a certain number of definitions at the concrete level:

```
(* Direct injection of factorial in Concrete_R via inject_Q *)
Definition concrete_fact (n:nat) : Concrete_R :=
  inject_Q Q_as_COrdField (inject_Z (pos_fact n)).

Lemma concrete_fact_ap_zero : ∀n:nat, (concrete_fact n)[#]Zero.
 intros; red; simpl; unfold R_ap. (* back to the definition of ≠ *)
 right; unfold R_lt. (* back to the definition of < *)
 exists O. (* the rank N *)
 exists (inject_Z 1). (* a str. positive rational between 0 and n! *)
 .... (* the rest is in Prop *)

(* The link between the old factorial and the new one *)
Lemma concrete_fact_pos_fact :
 ∀n:nat, (pring Concrete_R (pos_fact n))[=](concrete_fact n).
```

Then, we add some axioms at the abstract level, the one of `IR`:

```
Axiom concrete_fact' : nat → IR.
Axiom concrete_fact_ap_zero' : ∀n:nat,(concrete_fact' n)[#]Zero.
Axiom concrete_fact_pos_fact' :
  ∀n:nat,(pring IR (pos_fact n))[=](concrete_fact' n).

Definition concrete_e_series :=
  fun n ⇒ One[/]?[//](concrete_fact_ap_zero' n).

Lemma concrete_e_series_conv : convergent concrete_e_series.
...
(* The proof is done as before,by using the equality
   concrete_fact_pos_fact' for going back to the previous case. *)

Definition concrete_E := series_sum ? concrete_e_series_conv.
```

Lastly, it is necessary to announce to the extraction what it should do with these axioms:

```
Extract Constant concrete_fact' ⇒ concrete_fact.
Extract Constant concrete_fact_ap_zero' ⇒ concrete_fact_ap_zero.
```

And there is no need to declare the third axiom, since it is logical.

It is interesting to note by the way that our formalization of $\mathbb{Q}$ as `Z*positive` pairs works correctly, at least for our current needs. Of course, one can to still find better, since Maple or Mathematica is able to return even the big fraction of rank 100 in a few tenths of a second. But these rational computations are certainly not a bottleneck for FTA.

Let us announce finally a last experimentation, which tried to find a third way between doing everything in `IR` and doing everything in `Concrete_R`. Indeed, as we have already mentioned, this distinction between `IR` and `Concrete_R` has its own interests, and anyway switching completely to `Concrete_R` would quickly become painful. We thus tried to replace only certain critical proof of `IR` by an equivalent in `Concrete_R`. But our attempts on `fac_ap_zero` brought only tiny gains compared to the initial (lack of) speeds, without common measurement with the gains brought by `concrete_E`.

6.2 Some alternative reals dedicated to the extraction

Our last attempt at improving the extraction of FTA has consisted in analyzing the computational behavior of $\sqrt{2}$ seen as reciprocal of 2 via the square function. But due to lack of time, we have only determine that inefficiency comes from certain sub-functions dealing with polynomials. We will detail that later, but first let us present how a small formalization of constructive reals, independent of FTA, has enabled us to identify some critical points explaining the efficiency or inefficiency of such a computation of $\sqrt{2}$.

This small formalization of constructive reals was realized in collaboration with H. Schwichtenberg, after some long discussions we had during the summer school of Marktoberdorf 2003, about his lecture of constructive analysis [73] and about Coq extraction. This study was then prolonged by a visit of week in Munich in September 2003.

Without being made to compete with FTA/C-CoRN, this study aims at considering constructive reals immediately from the point of view of the extraction, unlike FTA where this idea of extraction came a posteriori. To justify this new study, here comes immediately its main result:

```
1850738521931038153706479986072768566074474889952922673412495088628037078498
212257925892008186006084221171975185924353893529607482952 7 / 1308669759060
6049824350852503626336293843757278081792174783812611032824335644861742036169
5749987134911710575859986086592962928589138 67
```

This long fraction is an approximation of $\sqrt{2}$ with more than 140 correct binary digits, or 42 correct decimal digits. And we obtained this result in approximately 3 minutes, by using the same principle than during our unfruitful tests with FTA, namely the seek of a reciprocal value of 2 via the square function.

It should be specified immediately that this small development does not establish any general results like FTA, but on the contrary is specialized on the computation of $\sqrt{2}$. Moreover, as we will see, some parts remains unfinished and posed as axioms. This is thus not a complete formalization, but rather a proof of concept This being said, these few hundreds of lines are quite full of lessons for the extraction of reals.

6.2.1 The development method

The goal was clearly to be able to test as fast as possible the the extracted code. We thus formalized the first pages of the course notes of H. Schwichtenberg [73], or more precisely the exact concepts needed to the definition of $\sqrt{2}$. Here for example the definition of the real numbers:

```
(* First, the Cauchy property. *)

Definition Is_Cauchy (f : nat → Q) (mo : nat → nat) :=
  ∀k m n, mo k ≤ m → mo k ≤ n → let e:=(f m - f n)*2^k in -1≤e≤1.

(* A real is given by a Cauchy sequence, a modulus sequence *)
(* and a proof of the Cauchy property of these sequences. *)

Record R : Set := {
  cauchy : nat → Q;
  modulus : nat → nat;
  is_cauchy : Is_Cauchy cauchy modulus }.
```

In fact, even by limiting ourself to the "useful" concepts, this would have needed more than one week or two of work. We have then chosen to focus only on the informative portions of the terms being defined. And the required logical parts were then systematically posed as axioms, waiting for a later completion. Actually, rather than posing multiple axioms, which is tiresome, we have in fact "cheated", by posing only one:

```
Axiom Falsum: False.
```

…/…

.../...

```
Ltac fed_up := elim Falsum.
```

And the use of the tactic `fed_up` then allows us to get rid of any "boring" end of proof. Here is for example the initial definition of the addition for two real numbers:

```
Definition Rplus : R → R → R.
 intros x y.
 apply (Build_R (fun n ⇒ cauchy x n + cauchy y n)
               (fun k ⇒ max (modulus x (S k)) (modulus y (S k)))).
 fed_up.
Defined.
```

In this case, the use of the `fed_up` allows to avoid the complete proof that our new sequence representing `x+y` is indeed a Cauchy sequence. In this particular case, this `fed_up` have been later on replaced by a Coq script of about thirty lines.

Of course, as long as there remain some `fed_up` in our development, we cannot affirm for sure that the big fraction above is indeed an approximation of $\sqrt{2}$ with the previously mentioned accuracy. On the other hand, the extraction of a proof containing `fed_up` is exactly identical to the extraction of the same complete proof, as long as these `fed_up` are used in portions of sort `Prop`. We must nevertheless be very careful with this "magic" tactic:

- If we use it in an informative part, then an exception is placed in the program (see the extraction of `False_rec` page 79).
- If we use it to force the proof of an erroneous logical proposition, this can lead the extracted program to a false result but also possibly to execution errors or to not-termination (see the examples of chapter 2).

6.2.2 The rational numbers

As we were already knowing quite well the rational numbers defined in FTA, we reused again these rationals. But we added two improvements. The first of these improvements relates to the proofs concerning these rationals. When we started with to replace the several `fed_up` by true proofs, we have indeed noted that the proofs of rational arithmetic are extremely painful. The problem comes from the chosen representation, which is not canonical: the fraction $\frac{1}{2}$ is not equal to the fraction $\frac{2}{4}$ when using the usual equality of Coq. One must then define an ad hoc equality for this datatype, and this forbids us a priori to use a certain number of tools that work only with the equality of Coq, such as for example of the tactics `rewrite` or `ring`[8]. Fortunately Coq has recently been extended by a mechanism due to C. Renard, that allows to work more easily on such structures, known as "setoids" (see [78]). We have thus equipped our type `Q` with such a structure of setoid, which gives us access to tactics like `setoid_rewrite`. And in addition, we have helped to finalize the extension of the tactic `ring` to support these setoids. This way, the rational numbers start to be of

[8] This automatic tactic is able to solve in a ring the equalities deducible from associativity, commutation and distributivity of `+` and `*`.

a reasonably practical use in proofs, even if a certain number of automatic tools are still missing, such as for example the tactics `field` and `fourier`.

The second improvement relates more directly to the extraction. We have indeed noticed that the operations on `Q` were never reducing the fractions in canonical forms. Consequently, during our computations of Euler's constant in FTA, the fractions obtained were growing very quickly, while for example finishing by a certain number of zeros in the numerator and in the denominator. And the approximation of rank 100, which fills out initially two screens, can in fact be reduced to an irreducible fraction of only four lines. We were at first quite reluctant to the idea of frequently simplifying the fractions, because this has also a cost. But placing some simplifications in this experimental development showed without possible ambiguity the enormous computation speed-up induced by these simplifications: instead of three quickly accessible decimal digits, we can know reach several hundreds. In details, this addition of simplification was done via a function `Qred: Q→Q`, which computes the gcd of the numerator and denominator, before dividing them by this gcd. We have also proved that any fraction returned by `Qred` is indeed equivalent to the input fraction. We have then inserted a `Qred` in the principal loop of computation for $\sqrt{2}$. Perhaps would it be better to place some at each elementary operation, or on the contrary less frequently? Only more complete tests can answer that question. In any case, it seems obvious that the computation of FTA would also gain to integrate such simplifications. Finally, this improved library of rational numbers was gathered in a new contribution `Orsay/Qarith`.

6.2.3 The Cauchy sequences

Let us return one moment to the definition of `R` given previously. This definition follows the formulation of H. Schwichtenberg [73], which differs slightly from the formulation used in FTA. The latter, more usual, is `∀k, ∃N, ∀n>N, ∀n>N,|f(n)-f(m)|`$\leq 2^{-k}$, whereas in [73] the bound `N` is given explicitly as a function over `k`, this last function being named *modulus* of the Cauchy sequence. These two formulations are in fact equivalent from the constructive point of view, since one can find the modulus `N(k)` starting from the proof of `∀k, ∃N,...` Nevertheless, the more explicit formulation of the modulus encourages to choose it carefully, and we endeavored to choose it as accurately as possible. As a consequence, during the computation of an approximation of $\sqrt{2}$ by the extracted program, one directly obtains an upper error margin for the result: it is enough to ask `(sqrt2.cauchy (sqrt2.modulus 140))` to be sure to obtain a result within 2^{-140} of the limit which is $\sqrt{2}$ or, said otherwise, to get 140 correct binary digits. In practice, we even get some additional correct digits because of approximations in the computation of the modulus for the sequence, but the order of magnitude is the correct. In comparison, during the computations of the `e` approximations in FTA, it is experimentally clear that the approximation of rank `k`, that is the `k`-th term of the Cauchy sequence, will provide us `n` correct decimal digit, but the relation between `k` and `n` is not explicit. And even if it would be made explicit, it is not sure that this relation would be very accurate, due to the choices sometimes naive in the bounds `N` in FTA. The errors estimates for computations in FTA were thus carried out a posteriori with Maple.

6.2.4 The continuous functions

The way of defining the continuous functions differ also appreciably between [73] and FTA. In FTA, a continuous function is a function $\mathbb{R} \to \mathbb{R}$ plus some associated properties. According to the definition of $\mathbb{R}$ as set of of Cauchy sequences, a continuous function in FTA thus corresponds mainly to a function of type $(\mathbb{N} \to \mathbb{Q}) \to \mathbb{N} \to \mathbb{Q}$. This function as argument is in fact not desirable, because it makes the extracted code more complex, more delicate to analyze and potentially less effective. The alternative is then to use a family of rational functions which converge to the desired real function:

```
Record continuous [i:itvl] : Set := {
   cont_h : Q → nat → Q;
   cont_α : nat → nat;
   cont_w : nat → nat;
   cont_cauchy: ∀a:Q, Is_Cauchy (cont_h a) cont_α;
   cont_unif : ∀a b n k, n≤(cont_α k) → a∈i → b∈i →
       -1 ≤ (a-b)*2^((cont_w k)-1) ≤ 1 →
       -1 ≤ (cont_h a n - cont_h b n)*2^k ≤ 1 }.
```

In this definition, `itvl` is the type of the intervals delimited by two rationals. The true function in this definition is `cont_h`, of type $\mathbb{Q} \to \mathbb{N} \to \mathbb{Q}$. Then follow two modulus functions and their properties:

- “pointwise” Cauchy property when the first argument of `cont_h` is fixed.
- “uniform” continuity when one fixes the second argument of `cont_h`.

In particular, for the function $X \mapsto X^2 - 2$ which interests us, we take the following values:

```
Definition sqr2_h := fun (x:Q)(_:nat) ⇒ x*x-2.
Definition sqr2_α := fun _:nat ⇒ 0.
Definition sqr2_w := fun k:nat ⇒ 2+k.
```

6.2.5 The intermediate value theorem

Finally, the last major difference between this development and FTA relates to the method used to find the reciprocal of a given value via a function. There are indeed two possible versions for the intermediate value theorem (IVT):

- The first version is the theorem 3.12 of [73]. This is the most general version of the two, the only condition on the considered function is to be continuous. But the ransom of such a wide scope is an expensive algorithm which divides the current interval into a multitude of sufficiently small subintervals, and which are inspected sequentially before stopping in one of them. Incidentally, it is also noticeable that this version allows to find reciprocal as finely approximate as one wants, but not exact.
- The second version is proposition 3.13 of [73]. This version requires an additional assumption on our function, which must be *locally not-constant*: for any subinterval and any value, one should be able to exhibit a point where the function differs from this

value. Equipped with this additional information, one can then proceed by "trichotomy", an alternative of the dichotomy adapted to constructive logic. And an exact reciprocal is obtained this time.

Taking into account the advantages of the second version in term of effectiveness and exactitude, both FTA and our development use it. On the other hand, a distinction is done concerning the proof of local not-constancy. FTA shows in fact that any polynomial of non-null degree is locally not-constant, via a general proof based on complex factorizations of the polynomials. But for a particular case like ours, one can proceed way simpler: the strict growth of $X \mapsto X^2 - 2$ on the interval which interests us is enough to imply that this function is locally not-constant. And this algorithm using the strict monotony produces the goods results which we have seen above.

On the opposite, L. Cruz-Filipe has identified later on that the bad computational behavior of $\sqrt{2}$ using the IVT in FTA was due to the phase of computation of local not-constancy via polynomials. By also using strict monotony, he managed obtain a computation of $\sqrt{2}$ that terminates. But the effectiveness of this computation is still far from the one of the small experimentation presented here. This fact is certainly due to fractions not being factorized and computations on terms of proof made in the abstract part of FTA reals.

6.3 Conclusion

Arrived at this point, it is quite difficult to be satisfied by the current state of the program extracted from FTA. Truly, we have succeeded with L. Cruz-Filipe with to understand and optimize some computations of series limits using the FTA formalism, but this study has highlighted many limitations in the initial extracted program, and these limitations were removed only by some manual interventions, sometimes very complex, in the initial Coq code.

A mathematical development like FTA, not thought at the origin in term of extraction, can thus induce a huge amount of rewriting work before being able to generate a reasonable program. The extraction is thus not a magic button undoubtedly creating interesting programs starting from any proof. The situation is not so different from the more usual software development methods: a program fulfilling its specification is not necessarily thought a good program. There are just here two additional difficulties:

- An analysis a posteriori as the one done here for FTA is very delicate. In particular the extraction induces an additional distance between the initial code to modify and the program to to analyze, for example by profiling.
- The concept of "good proof" is less precise than the one of "good program". From the point of view of the extraction, a good proof is obviously a proof whose extraction is effective. On the other hand, for the user, a good proof can be simply a completely finished proof. And for the mathematician, a good proof can be an elegant or very abstract proof as those done in IR. Conversely, a very specialized development, made for the extraction as the one we have outlined, will undoubtedly be described as less elegant.

The use of the code extracted from the principal theorem of FTA is certainly a failure for the moment. But first of all, it is not a final failure, since the understanding of this extracted code has enormously progressed, and that many ideas of improvements exist, such as for example the simplification of the fractions in FTA. Then, it is advisable to relativize this failure, and in particular to not see that as the failure of the extraction as a methodology. The example of FTA is indeed hardly generalizable:

- First, it is a *premiere*: no developments of comparable size have ever been extracted. And as with a lot of premieres, it is hardly astonishing to have to endure initial problems. FTA constitutes currently a kind of Mount Everest for the extraction.
- It is also an *exception*: FTA is the only development met by the extraction that have such an atypical use of the logic of Coq.

Finally, it is rather reassuring to see that one can obtain quickly effective extracted programs in the same domain as FTA, even if that supposes to go back really far away, and especially to work from the beginning with the extraction as ultimate goal.

Chapitre 7

A formalization of finite sets

The development presented in this chapter has been realized in collaboration with J.-C. Filliâtre. All files of this development, both the the Coq *sources and the extracted files, are available on the site* `http://www.lri.fr/~filliatr/fsets`*. These files constitute now the contribution named* `Orsay/FSets`*. This development is also described in the joint article [34], with a slightly different point of view, less centered on the extraction.*

When a programmer wishes to obtain certified non-trivial programs by using the methodology of the program extraction, the first need is the existence of a certified basic library, sufficiently rich not to have to reinvent the wheel at each program line. For that purpose, we have chosen to inspect the standard library coming with Ocaml. Is it possible to develop a certified version of it?

Unfortunately, this standard library of Ocaml contains only a few purely functional data structures. And in our extraction paradigm, the study of the imperative modules like `Array` is not feasible. To treat these imperative cases, one should rather turn to tools like Why [33] developed by J.-C. Filliâtre, which supersedes the old tactic `Correctness` [32]. Some other modules, without being imperative, would be delicate to formalize such as for example those using the hardware integers.

It thus only remains in our scope the modules `List`, `Set`, `Map` and to a lesser extent `Sort` and `Stream`. Our study was finally limited to the module `Set` of the finite sets in Ocaml. At the same time, this choice of `Set` is far from being an uninteresting choice. This module presents indeed the following characteristics:

- The need for its use is felt very frequently, obviously by the programmer, but also by mathematician. We will see for example how these sets can be used to establish a result of graph theory.
- Its interface, relatively simple, allows to highlight the new capacities of Coq in term of modular organization. (see section 4.1).
- The library `Set` thus allows to build modules fulfilling this interface, thanks to a functor `Make` taking as input a module containing at least a type and a comparison function on this type. However there exist numerous manners of coding this functor `Make`, ranging from the most naive to the most advanced one. For example, the implementation currently used by Ocaml is based on AVL trees [2]. During this formalization, we

carried out a first implementation using of sorted lists, and J.-C. Filliâtre has built two others more effective implementation based on AVL trees and on Red-Black trees [41].

The general outline of our development is as follows:

Coq		Caml
interface `OrderedType`	extraction $\dashrightarrow$	interface `OrderedType`
interface `S` for sets	$\dashrightarrow$	interface `S` for sets
functor `Make` : `OrderedType` $\rightarrow$ `S`	$\dashrightarrow$	functor `Make` : `OrderedType` $\rightarrow$ `S`

A simple manner to obtain the Coq interfaces consists in taking the Ocaml interfaces `OrderedType` and `S`, and adding in it only the specification part for the functions. Ideally, the extraction of these Coq interfaces would then give again the initial Ocaml interfaces. We have first followed this approach, in spite of one certain number of difficulties that we detail in the following section:

- First of all, some incompatibilities between the Ocaml types and Coq one prevents the interface produced by the extraction to be exactly equal to the initial interface of Ocaml.
- Then, the specification of higher order functionals as `fold` appeared particularly delicate, and several versions were necessary before coming to a satisfactory result.
- Finally this first approach in which specifications are separated from the signature of the functions is not inevitably the the most natural style in Coq. Indeed Coq also allows to incorporate the specification in the type, via the use of dependent types. In fact, we propose our Coq interface under these two versions, as well as translation functors between these versions.

In the same way, one can imagine to import[1] into Coq the current code of the functor `Make` of Ocaml, and in a second time prove that these purely informative functions fulfill indeed their separate specifications. We followed this approach for the certification of a naive version of `Make` based on sorted lists. On the other hand J.-C. Filliâtre preferred to follow the interface based on dependent types, and directly defined the algorithms and their justifications interleaved together. The same methodology was used to obtain an certified implementation of `Make` based on Red-Black trees.

Finally, the programmer has now four Ocaml implementations compatible with the extraction of our Coq interface for finite sets:

- the non-certified initial implementation using AVL, via a slight wrapping for precisely corresponding to our interface;

[1] For lack of better mechanism, this importation is currently to be done manually.

- a simple and certified but ineffective implementation based on sorted lists;
- two implementation certified and effective, via Red-Black trees and AVL trees.

7.1 The Coq interface

This interface corresponds to the file `FSetInterface.v`. Let us see now more in details the points that Coq allows to import directly from Ocaml, and those that need an adaptation.

7.1.1 The ordered types

First of all, the Ocaml interface of `Set` starts by the definition of a signature representing a type equipped with an order relation, this type being meant to become the support type for the future sets.

```
module type OrderedType =
  sig
    type t
      (* The type of the set elements. *)
    val compare : t → t → int
      (* A total ordering function over the set elements.
         This is a two-argument function [f] such that
         [f e1 e2] is zero if the elements [e1] and [e2] are equal,
         [f e1 e2] is strictly negative if [e1] is smaller than [e2],and
         [f e1 e2] is strictly positive if [e1] is greater than [e2].
  end
```

A first problem appears immediately: for efficiency reasons, the comparison of Ocaml returns an hardware integer `int`. However these integers do not exist in Coq, and even though, would not be convenient to perform logical reasoning. More in accordance with the use in Coq, our interface is based on logical relations, i.e. on functions of type `t→t→Prop`. These logical relations are the equality `eq` and the strict order `lt`. They come together with five `Axiom` that require that `eq` and `lt` fulfill theirs usual elementary properties.

```
Inductive Compare (X : Set) (lt eq : X → X → Prop) (x y : X) : Set :=
  | Lt : lt x y → Compare lt eq x y
  | Eq : eq x y → Compare lt eq x y
  | Gt : lt y x → Compare lt eq x y.

Module Type OrderedType.
  Parameter t : Set.

  Parameter eq : t → t → Prop.
  Parameter lt : t → t → Prop.

  Axiom eq_refl : ∀x, eq x x.
```

.../...

.../...

```
  Axiom eq_sym : ∀x y, eq x y → eq y x.
  Axiom eq_trans : ∀x y z, eq x y → eq y z → eq x z.
  Axiom lt_trans : ∀x y z, lt x y → lt y z → lt x z.
  Axiom lt_not_eq : ∀x y, lt x y → ¬ eq x y.

  Parameter compare : ∀x y, Compare lt eq x y.
End OrderedType.
```

In fact, the Ocaml and Coq versions are not so different, at least from an informative point of view, the one of the extraction. Indeed, `eq` and `lt` are ignored by the extraction since placed in sort `Prop`, the one of logical propositions. In the same way, the five properties of `eq` and `lt` are also in `Prop`, thus ignored by the extraction. Thus the only remaining parts after extraction are the informative parts placed in `Set`, here the type `t` and the function `compare`. The latter function, similar to the `compare` of Ocaml, allows to discriminate according to the respective positions of two elements of type `t`, and returns a result in the ternary inductive type `Compare`. From the logical point of view, `compare` states the decidability of `eq` and `lt`. It should indeed be noticed that `compare` returns not only the desired position information (via the used constructor `Lt`, `Eq`, or `Gt`), but also a logical proof certifying the current situation. This logical part is also forgotten by the extraction, and that gives us for this signature `OrderedType` the following extracted version:

```
type 'x compare =
  | Lt
  | Eq
  | Gt

module type OrderedType =
 sig
  type t
  val compare : t → t → t compare
 end
```

It should be noted that we can easily write manually wrappers between this type `compare` and the type `int` used in the initial Ocaml interface as a three-value type. One can thus adapt the functor `Make` provided by Ocaml in order that it works with our interface.

7.1.2 The signature of the sets

Now, in Ocaml, an set structure is created via the following functor:

```
module Make (Ord : OrderedType) : S with type elt = Ord.t
(* Functor building an implementation of the set structure
   given a totally ordered type. *)
```

And here comes now the beginning of this signature `S` of the set structure:

```
module type S =
  sig
    type elt
    (* The type of the set elements. *)

    type t
    (* The type of sets. *)

    val empty: t
    (* The empty set. *)

    val is_empty: t → bool
    (* Test whether a set is empty or not. *)

    val mem: elt → t → bool
    (* [mem x s] tests whether [x] belongs to the set [s]. *)

    val add: elt → t → t
    (* [add x s] returns a set containing all elements of [s],
       plus [x]. If [x] was already in [s], [s] is returned unchanged. *)

    [...]
  end
```

One finds there, in addition to the types `elt` and `t` and the constant `empty`, 22 elementary functions over sets. It should be noted that each function comes with an informal specification in comment. It is in fact starting from these informal specifications that we have built our formalization. Our Coq signature thus begin in the same manner as the Ocaml signature, namely by the type declaration for the set operators:

```
Module Type S.
  Declare Module E : OrderedType.
  Definition elt := E.t.

  Parameter t : Set.

  Parameter empty : t.

  Parameter is_empty : t → bool.

  Parameter mem : elt → t → bool.

  Parameter add : elt → t → t.

  [...]
End S.
```

Some function signatures are not adaptable so simply:

- `compare : t → t → int`
  One finds again the same problem as with `OrderedType`: what to do with `int`? The answer is similar. First, we add in the signature `S` two relations `eq` and `lt` on sets, of

type t→t→Prop, without equivalents in Ocaml, and ignored by the extraction. And beside that, S requires the presence of an informative function compare of type ∀s∀s', (Compare eq lt s s'). This way, we still have in Coq the following interesting property of Ocaml : a module fulfilling S can also be seen as a OrderedType, which allows to build sets of sets.

- iter: (elt → unit) → t → unit
  This is the only occurrence of an imperative function in all the module, untranslatable in a purely functional world, and thus omitted.

- cardinal: t → int
  Once again, int is not directly usable in Coq. We have chosen to declare cardinal: t → nat, where nat is the Coq type of Peano integers. This debatable choice has only the interest of providing simple induction principles over the size of a set. But this is done at the detriment of the efficiency. Among the other alternatives, one can also use the binary integers Z of Coq, or an axiomatized abstract type, which one then extracts manually to int (see section 4.4.2). In any case, the other versions of cardinal can easily be written thanks to the generic function fold.

- min_elt, max_elt and choose: t → elt
  All these three functions are supposed to raise the exception Not_found if their argument is empty. A natural manner of encoding this behavior in Coq is the use of the type option.

At the head of the Coq interface, one can notice the presence of a declaration E: OrderedType. This declaration, lacking in Ocaml, allows in particular to name E.t, E.lt and E.eq in the specifications. The Coq type of the Make modules will then be:

```
Module Make (X:OrderedType) : S with Module E := X.
```

It is possible to avoid the use of this internal sub-module E by replacing it with two Parameter eq and lt in S, and by providing three "with Definition..." instead of only one "with Module...".

The specification part, with is the second half of the signature S of Coq, is centered around a logical membership relation In : elt→t→Prop. This relation is abstract: each module implementing our finite set interface should provide one. The only property required for this In is the compatibility with respect to the equality of E:

```
Parameter In_1: E.eq x y → In x s → In y s.
```

This property must be understood with a implicit universal quantification over the variables x, y and s, as authorizes by the mechanism of Section (see page 29). The same is true in the following examples.

All the set function specifications are now expressed with respect to this predicate In. One writes for example:

```
(** Specification of [mem] *)
```

.../...

…/…

```
Parameter mem_1: In x s → mem x s = true.
Parameter mem_2: mem x s = true → In x s.

(** Specification of [add] *)
Parameter add_1: In x (add x s).
Parameter add_2: In y s → In y (add x s).
Parameter add_3: ¬ E.eq x y → In y (add x s) → In y s.
```

7.1.3 The case of higher order functions

Among these set functions, those taking a function as argument require a little more attention. This concerns `fold`, `filter`, `for_all`, `exists` and `partition`. For example the abstract specification of `fold` is:

```
val fold: (elt → 'a → 'a) → t → 'a → 'a
(* [fold f s a] computes [(f xN ... (f x2 (f x1 a))...)], where
   [x1 ... xN] are the elements of [s]. The order in which elements
   of [s] are presented to [f] is unspecified. *)
```

Our first attempts at specifying `fold` were based on the formalization of the two following equations:

```
fold f empty i = i
fold f (add x s) i = f x (fold f s i)
```

But this approach appeared to be extremely hard to finalize. Two problems have arisen in particular:

- To take in account the unspecified character of the order of computations, while speaking of a meaningful final result, it was necessary to add assumptions on `f`, like the commutativity: `f x (f y a) = f y (f x a)`.
- Moreover, what happens if `f` returns two distinct values for two elements `x` and `y` that are equal modulo `E.eq`? In a preliminary version, we did not treat this case correctly, which would have allowed to prove `false = true` starting from a hypothetical module having for interface this version `S`.
- Finally the use of the usual equality "`=`" of `Coq` in these equations is too restrictive for certain uses. For example if one wishes to rebuild a set via a `fold`, one would then write `(fold add s empty)`. But the usual equality is not appropriate with our sets parameterized by an equality `E.eq`. We can then accept this case by using an additional equality `eqA` over the output type, but that becomes really heavy.

Finally, we have chosen a specification at the same time simpler and more expressive, by relying on a previously defined datatype, namely the lists, and more precisely on the function `fold_right` defined on these lists. The final specification is rather close to the informal version given in comment above. Instead of saying "`x1... xN` is the elements of the set `s`", we affirm "`l` is a list without redundancy containing all the elements of `s` and only those". Of course, the membership of our list and the not-redundancy are expressed modulo

E.eq. For that we have defined two specialized predicates InList and Unique parameterized by one equality. Here come this final specification of fold:

```
Parameter fold_1 : ∀(A : Set)(i : A)(f : elt → A → A), ∃l : list elt,
  Unique E.eq l ∧
  (∀x, In x s ↔ InList E.eq x l) ∧
  fold f s i = fold_right f i l.
```

This formulation does in particular not assume any pre-condition on the function f. If this function f does not check the commutativity of computations, or is not invariant with respect to E.eq, then several lists containing the elements of s will produce different results by fold_right. But at least one of these lists ends in the same result as the fold. In fact, if we adds these pre-conditions on f, one can replace ∃l by ∀l in the specification.

For specifying the other higher order functions, that is filter, for_all, exists and partition, we could also have established a parallel with the versions of these functions working on lists. In fact, these cases are appreciably simpler than the example of fold, since there is no problem of computations order or equality on the output type. We have thus used a direct specification, such as for example:

```
Parameter filter_1 : compat E.eq f → In x (filter f s) → In x s.
Parameter filter_2 : compat E.eq f → In x (filter f s) → f x = true.
Parameter filter_3 :
  compat E.eq f → In x s → f x = true → In x (filter f s).
```

The condition compat then requires the invariance of f with respect to E.eq.

7.1.4 A alternate signature containing dependent types

Our signature is thus divided into two, with one one side the purely informative functions, and on the other side the specifications in the form of purely logical axioms. This approach is not the only possible one in Coq. Thanks to the dependent types, one can indeed gather both parts in only one expression, whose general outline is, for a function with an argument, ∀x,P(x)→∃y,Q(x,y) with P and Q logical predicates expressing respectively the pre- and post-conditions.

We have then written a second version of the set signature, named Sdep, by using this style of “dependent types”. Here comes an excerpt:

```
Module Type Sdep.
  Declare Module E : OrderedType.
  Definition elt := E.t.

  Parameter t : Set.

  Parameter In : elt → t → Prop.
  Definition Empty s := ∀a, ¬ In a s.
```

.../...

.../...

```
  Definition Add (x:elt)(s s':t) := ∀y, In y s' ↔ E.eq y x ∨ In y s.
  [...]
  Parameter empty : {s : t | Empty s}.
  Parameter is_empty : ∀s,{Empty s}+{¬ Empty s}.
  Parameter mem : ∀x s,{In x s}+{¬ In x s}.
  Parameter add : ∀x s,{s' : t | Add x s s'}.
  [...]
End Sdep
```

The parameter In is now needed at the very beginning to express the specifications. Then follow a certain number of shortcuts such as Empty and Add, expressing logical properties based on In. The function add is now build on the pre- and post-condition model: the pre-condition is here always true, and the post-condition (Add x s s') expresses the fact that the new set s' contains the same elements as the old s, plus x. The case of the functions is_empty and mem is slightly different: instead of saying "there exists a boolean such that...", we directly use an inductive type with two values, sort of enriched boolean type, allowing to express what happens in both cases. This type is sumbool, presented p. 29.

Once again, the functions that are the most difficult to specify are the higher order functions. Here is for example fold, which now contains the property fold_1 in post-condition:

```
Parameter fold : ∀(A : Set)(f : elt → A → A)(s : t)(i : A),
  {r : A | ∃l : list elt,
    Unique E.eq l ∧
    (∀x, In x s ↔ InList E.eq x l) ∧
    r = fold_right f i l}.
```

7.1.5 Two functors to choose the signature style

The new system of module of Coq then allows us to avoid a choice between the two possible signatures. Indeed one can easily write a functor which transforms a module of type S into a new module of type Sdep and another functor doing the converse work. This is actually done in the file FSetBridge.v. This way, all new implementation of the sets only needs to be done for one version, whichever one is chosen. According to the taste of the programmer, the two implementations based on sorted lists and on red-black trees were build over different signature, one in accordance to S, the other in accordance to Sdep. And conversely, any user has the choice of the version which he prefers to use. He can even, and it is really appreciable in practice, use the two interfaces simultaneously.

7.1.6 Extraction of the set signatures

What happen to the extracted versions from these two interfaces? They are in fact extremely close, and can even be made equal.

Concerning the interface `S`, the extraction is quite simple. Indeed, the pure signatures do not contain any logical part, and there is no use of advanced Coq types like sorts, pattern matchings or fixpoints. The extraction is then a simple translation to Ocaml. Concerning the specifications, since they are completely logical, they are just forgotten. We just go back to the initial Ocaml interface modulo the slight adaptations mentioned previously. Here comes its beginning:

```
module type S =
 sig
  module E : OrderedType

  type elt = E.t

  type t

  val empty : t

  val is_empty : t → bool

  val mem : elt → t → bool

  val add : elt → t → t
  [...]
 end
```

Things get more complicated in the case of the interface `Sdep`. First of all, the possible logical arguments corresponding to pre-conditions are eliminated. Then the inductive type corresponding to the post-conditions of the form `{y: Y| ... }`, that is `sig`, is recognized as being a "singleton informative" type (see p. 131), and is thus translated into the identity: `type 'a sig0 = 'a`. The type extracted from the dependent version of `add` is thus `elt → t → t sig0`, which is then convertible to the expected `elt → t → t`. In the other possible situation, namely the use of `sumbool` as in `is_empty` or `mem`, the extraction of `sumbool` consists in forgetting the logical decorations of this inductive, which gives:

```
type sumbool =
  | Left
  | Right
```

This extracted type `sumbool` is isomorph with the boolean type, but not equal. If one wishes to force the equality, in order to really obtain the same signature in both cases, it is enough to use the replacement mechanism for the extracted inductive (see p. 142):

```
Extract Inductive sumbool ⇒ bool [true false].
```

Here is the beginning of the interface extracted from `Sdep`, with no replacement of `sumbool` by `bool`:

```
module type Sdep =
 sig
```

.../...

.../...

```
  module E : OrderedType
  type elt = E.t
  type t
  val empty : t sig0
  val is_empty : t → sumbool
  val mem : elt → t → sumbool
  val add : elt → t → t sig0
  [...]
 end
```

At this point, we have a formal interface in Coq, as well as an extraction of this interface in Ocaml. We can then right now provide a not-formal implementation of this extracted interface thanks to the module Set of Ocaml, with the help of a manual encapsulation of certain functions, in particular the comparisons.

```
(* c2i : 't compare → int *)
let c2i = function Lt → -1 | Eq → 0 | Gt → 1

(* i2c : int → 't compare *)
let i2c i = if i<0 then Lt else if i=0 then Eq else Gt

(* i2n : int → nat,tail recursive *)
let i2n =
  let rec acc p = function 0 → p | n → acc (S p) (n-1)
  in acc O

module Make(X:OrderedType) : S with module E = X and type elt = X.t =
 struct
  module E = X
  module M = Set.Make(struct
                       type t = X.t
                       let compare x y = c2i (X.compare x y)
                      end)
  include M
  let compare s s' = i2c (compare s s')
  let cardinal s = i2n (cardinal s)
  let max_elt s = try Some (max_elt s) with Not_found → None
  let min_elt s = try Some (min_elt s) with Not_found → None
  let choose s = try Some (choose s) with Not_found → None
 end
```

Please note that this only works if we have replaced beforehand certain extracted inductive types by their primitive equivalent in Ocaml. This concerns the boolean, the pairs and the lists. For boolean, that can be done via an `Extract Inductive`, but the two other cases require an external intervention, for example the use of a syntactic script of improvement of the extracted code. This script, written in Camlp4, is available at the following address: `http://www.lri.fr/~letouzey/download/pp_extract.ml`.

7.2 A implementation based on sorted lists

Before going further, one may wonder why do we really need to code in Coq an implementation of our set interfaces. After all, using a pragmatic attitude, one can indeed choose to regard the module `Set` of Ocaml as being sufficiently tested to be correct with respect to its informal specification written in comments. And concerning our small manual wrapping functor `Make` in the previous paragraph, its small size and its simplicity leave little place to errors. If we accept these two points, then we can perfectly carry out a certified development in Coq using our sets only via their interface, and obtain nevertheless a complete program thanks to this functor `Make`.

In fact, we will see later in the description of our implementation based on AVL trees that the original implementation of Ocaml was not so correct after all, since we have found there an error. The formalization of this implementation was thus not vain.

And from the point of view of a Coq user interested in our finite set library, stopping this library with only the Coq interface presents two disadvantages. First of all, such an abstract vision excludes any computation in Coq. In particular, one can build a proof that `(is_empty empty)` is worth `true`, but not compute/execute/simplify `(is_empty empty)` into `true`. This is only possible for a particular implementation of the sets. Maybe a future introduction of primitive rewriting in Coq will one day change this fact. And providing an effective Coq implementation in which we can compute can also lead to the use of these sets in tactics based on reflexion (see for example [16]).

The other disadvantage is the risk of inconsistency of our interface, which is by itself only an axiomatization. This risk is to be taken seriously. For example we have already mentioned the fact that an earlier version of `FSetInterface.v` was allowing to deduce `False` from some particular `OrderedType`, due to a bad specification of the higher order functions like `fold`. This nasty surprise cannot happen any more, since at least an implementation allows, from any `OrderedType`, to build a module fulfilling `S`, and this without using any axiom.

7.2.1 Description of the module `FSetList`

The goal of this module is to provide as quickly as possible an implementation to our Coq interface for sets, mainly in order to to check its coherence. Efficiency was hence not a preoccupation during its creation. As a consequence, the first idea was an implementation using unspecified lists. But contrary to the common ideas, the set operations for unspecified lists are not so obvious to write. In particular the function `remove` must parse all the list to detect possible doubled items. Similarly, the function `fold` must deal with these duplicated items. A solution is then to maintain an invariant of not-redundancy. Since an invariant is to be maintained, it is hardly more difficult to require directly that the lists are sorted. And this way the efficiency is much better.

For this implementation, we have followed the not-dependent interface `S`. But it is not that simple to work with lists associated with an invariant. In particular, for each operation producing a list, it is immediately necessary to check that this invariant is preserved. We have preferred to split the work in several phases.

A first functor `Raw`, taking a `OrderedType` as argument, defines set functions over the datatype `t = (list elt)`, implicitly assuming that these lists are sorted Thus the union corresponds to the classical `merge` algorithm:

```
Fixpoint union (s : t) : t → t :=
  match s with
   | [] ⇒ fun s’ ⇒ s’
   | x :: l ⇒
     (fix union_aux (s’ : t) : t :=
        match s’ with
          | [] ⇒ s
          | x’ :: l’ ⇒
            match E.compare x x’ with
              | Lt _ ⇒ x :: union l s’
              | Eq _ ⇒ x :: union l l’
              | Gt _ ⇒ x’ :: union_aux l’
            end
        end)
  end.
```

Note here the usual trick consisting in using an anonymous internal `fix` for enabling the call `(union_aux l’)`, not structurally decreasing compared to the first argument.

In a second time, the functor `Raw` proves the properties expected by the signature `S`, except that we add at the top of the lemmas the assumption that the input lists are initially sorted.

```
Lemma union_1 : ∀(s s’ : t)(Hs : Sort s)(Hs’ : Sort s’)(x : elt),
 In x (union s s’) → In x s ∨ In x s’.
```

We also prove that our operations always produce sorted lists when their arguments are sorted. For example:

```
Lemma union_sort :
 ∀(s s’ : t) (Hs : Sort s) (Hs’ : Sort s’), Sort (union s s’).
```

Consequently, we then have all the pieces to define a second functor named `Make`, which this time really produces a module of signature `S`. In this module, the datatype is now the well sorted lists, defined by:

```
Record sorted_list : Set := { this :> Raw.t ; sorted : sort E.lt this }.
Definition t := sorted_list.
```

The remainder of the module is only a long sequence of wrapping/unwrapping, which thanks to the implicit arguments and to the coercion `t :> Raw.t` are done without problem. For example:

```
Definition union (s s' : t) :=
 Build_sorted_list (Raw.union_sort (sorted s) (sorted s')).
Definition union_1 (s s' : t) := Raw.union_1 (sorted s) (sorted s').
```

7.2.2 Extraction of `FSetList`

The extraction of the whole cannot be simpler. Concerning the functor `Raw`, the pure functions are extracted into themselves, and their properties are forgotten. Our example of the union gives:

```
let rec union s x =
  match s with
    | Nil → x
    | Cons (x0,l) →
        let rec union_aux s' = match s' with
          | Nil → s
          | Cons (x',l') →
              (match E.compare x0 x' with
                 | Lt → Cons (x0,(union l s'))
                 | Eq → Cons (x0,(union l l'))
                 | Gt → Cons (x',(union_aux l')))
        in union_aux x
```

And in the functor `Make`, the type `sorted_list` is recognized as being isomorph to `(list elt)` as soon as the logical part `sorted` is removed (one more informative singleton inductive type). All the remainder is thus only definitions of aliases. For example:

```
let this s = s
let union s s' = Raw.union (this s) (this s')
```

7.2.3 The tail recursivity

It should be noted that our functions on lists were written in a direct recursive style, and are thus almost never tail recursive. This is a priori not a problem, because this module has no claim of effectiveness. Except that...

It indeed appeared during the realization of the effective implementation based on Red-Black trees, that a certain number of operations, such as for example the union and the intersection, could be in fact advantageously "subcontracted" to the module `FSetList`:

- On the efficiency level, one can transform a Red-Black tree, which is a particular form of binary search tree, into a sorted list via a simple linear traversal. And the reciprocal transformation, even if less simple, can also be done in linear time. Finally, we obtain this way an union on trees whose complexity is in the worst case the sum of the sizes of his arguments. This complexity is theoretically optimal, and quite far from being obvious to obtain by direct analysis of the trees.

- On the level of the correctness proof for these "subcontracted" functions, there is an obvious gain: we only need to prove once and for all the correctness of the two functions of conversion between Red-Black trees and sorted lists, and then the correctness of the tree functions done this way is obtained by direct translation of the results already proved for the lists.

Consequently, it would be interesting to carry out an alternative of `FSetList` that consume no stack. A simple way to reach that point would be undoubtedly to define the tail recursive alternatives, and to prove immediately that they return the same result as their simpler counterparts.

Concerning the implementation based on AVL tree, we made a point of being as faithful as possible to the original Ocaml code of the library `Set`. We thus have not used this method of flattening, merging, then rebuilding, but rather the method used in `Set`, more effective in practice.

7.3 A implementation based on Red-Black trees

The corresponding file is `FSetRBT.v`. Let us remind that Red-Black trees (RBT) are binary search trees for which one limits the maximum imbalance by associating two colors to the nodes, and by controlling the locations of these colors. More precisely:

(i) all the paths from the root to a leaf should contain exactly the same number of black nodes.

(ii) a red node cannot have red child.

(iii) the leaves are considered black.

Our implementation of sets based on RBT is thus effective. For example, the search for an element has a logarithmic maximal cost. But this effectiveness is paid during the correctness proofs, which are much more complex than in the preceding implementation. For example, the correctness proof of the `add` consists of more than 350 lines. Fortunately, only `add`, `remove` and `of_list` (the building of a RBT starting from a sorted list) have presented real difficulties. The other functions were either simpler, or "subcontracted" to the list functor as seen in the previous paragraph.

Our intention is here not to describe the details of this implementation. Moreover the main functions like `add` and `remove` having been realized by J.-C. Filliâtre. One can nevertheless note that the interface used is `Sdep`, the one containing dependent types. The functions are thus defined by tactics instead of directly providing terms, and these tactics allow to build at the same time the structure of the underlying algorithm and the proofs of the intermediate needed properties, invariants and post-conditions. This module thus constitutes a good test bench for the extraction, which have to distinguish informative parts from logical parts. In practical, the extracted functions are each time close to what one would have manually written, except for syntactic details like the shape of pattern matchings.

And as shows in a following section, the effectiveness of the extracted functions is indeed as awaited.

Let us consider for example the function `of_list` that constructs a RBT from a sorted list. It relies on a function `of_list_aux` with three informative arguments:

- the height `k` of the tree to be manufactured, initialized in `of_list` with `N_digits`, that is a base two logarithm.
- the size `n` of the tree to be manufactured.
- the list of the elements, that can become larger than `n` during the internal recursive calls. So this function `of_list_aux function` also returns the extra elements.

Here come shortened versions of `of_list_aux` and `of_list`. We have removed all the invariants propagated in these functions, and have only kept the treatment of the various situations. But even this way, this Coq script remains highly indigestible, and is given only as an illustration.

```
Definition of_list_aux :
 ∀k : Z, 0 ≤ k →
 ∀n : Z, two_p k ≤ n + 1 ≤ two_p (Zsucc k) →
 ∀l : list elt, sort E.lt l → n ≤ Zlength l →
 {rl' : tree * list elt | ... }.
Proof.
 intros k Hk; pattern k; apply natlike_rec3; try assumption.
 intro n; case (Z_eq_dec 0 n).
 (* k=0 n=0 *)
 intros Hn1 Hn2 l Hl1 Hl2; exists (Leaf,l); [...].
 (* k=0 n>0 (in fact 1) *)
 intros Hn1 Hn2.
 assert (n = 1). [...]
 rewrite H.
 intro l; case l.
  (* l = [],absurd case. *)
  intros Hl1 Hl2; unfold Zlength,Zlt in Hl2; elim Hl2; trivial.
  (* l = x::l' *)
  intros x l' Hl1 Hl2; exists (Node red Leaf x Leaf,l'); [...]
 (* k>0 *)
 clear k Hk; intros k Hk Hrec n Hn l Hl1 Hl2.
 rewrite <- Zsucc_pred in Hrec.
 generalize (power_invariant n k Hk).
 elim (Zeven.Zsplit2 (n - 1)); intros (n1,n2) (A,B) C.
 elim (C Hn); clear C; intros Hn1 Hn2.
 (* 1st recursive call : (of_list_aux (Zpred k) n1 l) gives (lft,l') *)
 elim (Hrec n1 Hn1 l Hl1).
 intro p; case p; clear p; intros lft l'; case l'.
```

.../...

.../...

```
  (* l' = [], absurd case. *)
  intros o; elimtype False.  [...]
  (* l' = x :: l'' *)
  intros x l'' o1.
  (* 2nd rec. call : (of_list_aux (Zpred k) n2 l'') gives (rht,l''') *)
  elim (Hrec n2 Hn2 l''); clear Hrec.
  intro p; case p; clear p; intros rht l''' o2.
  exists (Node black lft x rht, l'''). [...]
Defined.

Definition of_list : ∀l : list elt, sort E.lt l →
 {s : t | ∀x : elt, In x s ↔ InList E.eq x l}.
Proof.
 intros.
 set (n := Zlength l) in *.
 set (k := N_digits n) in *.
 assert (0 ≤ n). [...]
 assert (two_p k ≤ n + 1 ≤ two_p (Zsucc k)). [...]
 elim (of_list_aux k (ZERO_le_N_digits n) n H1 l); auto.
 intros (r, l') o.
 assert (∃n : nat, rbtree n r). [...]
 exists (t_intro r (olai_bst o) H2). [...]
Defined.
```

And here comes now the extraction of these two functions:

```
(** val of_list_aux : z → z → elt list → (tree, elt list) prod sig0 **)

let rec of_list_aux x n l =
  match x with
    | ZERO →
        (match z_eq_dec ZERO n with
           | Left → Pair (Leaf, l)
           | Right →
               (match l with
                  | Nil → assert false (* absurd case *)
                  | Cons (x0, l') → Pair ((Node (Coq_red, Leaf, x0,
                      Leaf)), l')))
    | POS p →
        let Pair (n1, n2) = zsplit2 (zminus n (POS XH)) in
        let Pair (lft, l1) = of_list_aux (zpred (POS p)) n1 l in
        (match l1 with
           | Nil → assert false (* absurd case *)
           | Cons (x0, l2) →
               let Pair (rht, l3) = of_list_aux (zpred (POS p)) n2 l2 in
```

.../...

.../...

```
              Pair ((Node (Coq_black, lft, x0, rht)), l3))
    | NEG p → assert false (* absurd case *)

(** val of_list : elt list → t sig0 **)

let of_list l =
  let n = zlength l in
  let Pair (r, l') = of_list_aux (n_digits n) n l in
  r
```

The proof structure of `of_list_aux` can be better understood by reading the extracted code rather than the proof itself, even with the accompanying notes. This readability of the extracted code does not come for free, and is the result of many optimizations (cf section 4.3). Here, one notices an induction on the integer `k≥0`, thanks to an ad hoc induction principle named `natlike_rec2`, whose type is:

```
Lemma natlike_rec2 :  ∀P : Z → Type,
  P 0 →
  (∀z : Z, 0 ≤ z → P z → P (Zsucc z)) →
  ∀z : Z, 0 ≤ z → P z
```

In the case `k=0`, we have either `n=0` and then we builds an empty tree or `n=1` and then we build a final node of red color. And when `k>0`, we divide `n-1` into two halves, (`n1` and `n2`), and we call recursively twice the function to build the left and right parts of the tree.

7.4 A implementation based on AVL trees

We have also realized a third implementation of a functor `Make` taking an `OrderedType` and returning a module of signature `Sdep`. After those based of sorted lists and on Red-Black trees, this third implementation is based on AVL trees. It has been carried out by J.-C. Filliâtre by following as closely as possible the initial implementation of the functor `Set.Make` available in the standard library of Ocaml. In fact, the code which we obtain by extraction of this Coq implementation is sufficiently close to the original manual code to be able to affirm reasonably that we have formalized and certified this Ocaml library.

Except for the printing details, the principal difference between two codes concerns the arithmetic used. AVL Trees are indeed trees in which the depth difference between any two sub-trees does not exceed a certain fixed[2]. quantity Δ. And it is essential to store the depth of the tree in the chosen data structure, otherwise we would recompute this depth unceasingly. However a depth is an hardware integer `int` in the initial Ocaml code, and we have replaced this in Coq by an integer of type `Z`.

This difference in the integer representation has an important influence on the speed of the two codes, the manual Ocaml version going approximately four times faster than the extracted version, as shown by the figures presented in [34]. The Coq integers, encoded via

[2] In the literature, Δ is often 1, whereas 2 was chosen in the Ocaml implementation as trade-off between the advantages of having very balanced trees and the costs related to re-balancing.

inductive types, even in binary format, cannot compete with hardware integers. It would then have been interesting to have a tool allowing to substitute a representation by another during the extraction, generalizing the command `Extract Inductive`, too limited. In theory, this replacement is not sure, because the type `int` is subject to overflow problems that cannot happen with the type `Z` of Coq. But this danger is here quite hypothetical, since it would be quite utopian to try to handle balanced binary trees with depth higher than 2^{30}, since that would mean that that they would have about $2^{2^{30}}$ elements! More generally, one can imagine to replace `Z` by `big_int`, which gives arbitrary precision while doing a maximum of computation via the hardware integers.

At the technical level, this implementation is finally quite similar to the one via Red-Black trees, except for a great quantity of reasoning on the integers, and thus a strong use of the automatic tactic `omega`. Instead of giving here too many details, we refer to [34] for (a little) more information. The major surprise has been the discovery of an error in the initial Ocaml code. Certain functions could indeed return trees which were not correctly balanced any more. This problem was not critical since the trees were remaining correct (*i.e.* still contained the good elements), but on the other hand the efficiency could be strongly affected. For example the logarithmic complexity for the search for an element, advertised in a comment, was not guaranteed any more. We informed X. Leroy of this problem, and he immediately corrected the Ocaml source code.

7.5 An example of use in a mathematical context

The example which follows now is included in our contribution `Orsay/FSets`, in the sub-directory `PrecedenceGraph`. This result was initially an exercise posed during some practical sessions about theory of operating systems [9]. Since the official solution of this exercise was incorrect, we have then searched a correct proof, and formalized this proof in Coq to be convinced of its correctness once and for all. This formal proof has in fact been at the origin of our interest in finite sets in Coq. We have modified it later so that it now uses our sets à la Ocaml. It is now a good example of use of these sets within a mathematical context.

The result deals with the precedence graphs. Such a graph is a representation without redundancy of a strict order: more precisely, if $<$ is a strict order, the associated graph is defined by $a \to b$ iff $a < b$ and $\forall c, \neg(a < c < b)$. In practice, we can also see a precedence graph as an acyclic directed graph whose transitive edges have been removed (i.e. $a \to ... \to b$ implies $\neg(a \to b)$). The result then states that:

$$E \leq \frac{N^2}{4}$$

with E being the edge number of a precedence graph, and N its node number.

The proof is done by induction over N, by removing at each step a carefully chosen node and the edges which are associated to this node. This proof outline has made us choose to the following representation for the graph[3]

[3] We have chosen to label our nodes with integers. In fact, any `OrderedType` could have worked.

```
Record Graph : Set := {
  nodes:> t;
  to: nat → nat → bool }.
```

with `t` being a finite set of integers. This way, when we withdraw a node, we just have to update `nodes`, whereas the transition function `to` can remain invariant. Indeed, the only edges that are counted are those whose both ends are in `nodes`. The function `filter` then allows to define the sets of successors and predecessors of a node, which in combination to `cardinal` allows to define the number of edges.

The proof proceeds then in four stages:

```
Theorem edges_remove : ∀G:Graph,∀n:elt, In n G → to G n n = false →
 nb_edges G = (nb_edges (node_remove G n))+(arity G n).

Theorem get_init : ∀G:AcyclicGraph, 0 < nb_nodes G →
 {p:nat| In p G ∧ nb_pred G p = 0}.

Theorem low_arity: ∀p:nat,∀G:PrecGraph, 0 < nb_nodes G →
 nb_nodes G < 2*(p+1) →
 {k:nat| In k G ∧ nb_linked G k ≤ p}.

Theorem TheBound : ∀G:PrecGraph,
 (nb_edges G)*4 ≤ (nb_nodes G)*(nb_nodes G).
```

The first theorem establishes that the edges number of $G \setminus \{n\}$ is the one of G less the number of successors and predecessors of n. Then, for an acyclic graph, we show the existence of an initial element. The third theorem is crucial: it affirms that in any precedence graph whose size is strictly lower than $2(p+1)$, we can find a node with less than p neighbors. And finally this particular node is adequate for our purpose: we can remove it and finish the induction reasoning, which enables us to establish the final theorem.

7.6 A final word

This case study shows that it is perfectly possible to specify and to implement functional and efficient data structures in Coq, while remaining in connection with an interface and an effective Ocaml implementation by the means of the extraction.

This method can obviously be reused for other structures, such as those presented by C. Okasaki in [64]. In particular all the present work could normally allow us to obtain a module `Map` at low cost. One can, indeed, see a `Map` structure as being a `Set` where the `OrderedType` is a pair `index * value` and where the comparison only accesses the first component. The only new function to write is then `find`, which is anyway only an alternative of `mem`.

Conclusion

Arrived at the end of this work, it is now time for taking stock, by asking at least the two following questions:

- Is this new extraction better than the old one? We do think so.
- Now, is our is extraction still perfectible? Of that we are sure.

The realizations

First, the critical correctness issues that we have evoked in introduction have been solved. A syntactic proof, enabling to compare the reduction of a extracted term with the reduction of an initial Coq term, ensures indeed that potential execution errors, like those from which the old extraction suffered, cannot occur any more at the present time, whether it is with the strict evaluation à la Ocaml or with the lazy evaluation à la Haskell.

In addition we have finalized a second correctness proof, inspired by realizability, which guarantees that the semantic properties of the initial Coq terms are indeed preserved during extraction. This proof has been made in a system as close as possible to the Cic currently employed in Coq, which has implied a very consequent increase in complexity for this semantic proof when comparing to the original works by C. Paulin. This proof, although manual and thus partly unsatisfactory, must be seen as a first stage towards a internal correctness proof, formalized in Coq, that we will evoke among the prospects.

Another realization of this thesis is the solution of the typing problem for the extracted terms. Our solution consists in the use of untyped coercion functions: this is certainly ad hoc, but works very well in practice. This allows now to explore all the range of the Coq terms with no more fear of ending on terms without extraction or with untyped extraction.

Lastly, a substantial effort has been made on the implementation level, in order to make of extraction a real platform of certified code generation, and not any more an experimental tool. In particular, this point, one can note the profits in readability for the extracted code, and especially the progresses concerning the integration of extracted code with broader developments, thanks to the generation of interfaces, and especially thanks to the extension of the extraction to the new module system of Coq.

The prospects

First of all, it is obvious that one can still improve the safety of this mechanism of code generation. Of course, as shown by K. Thompson in the article [79], our final program will be worthy of confidence only if each link in the production chain is, and in particular the compiler Ocaml or Haskell used, the operating system, the processor, etc[4]. In practice such a level of verification is never reached. But one of the weak spot in this safety chain seems to be today the correctness of our extraction. It is indeed hardly satisfactory that all this complex mechanism relies only on a proof in LaTeX. There are then two possible approaches to cure that:

- One can first add to the extraction mechanism an extension that, for each extracted term, produces the Coq term proving the correctness of this extracted term. Consequently, at each extraction, the acceptance by the type-checker of Coq of this Coq proof term will guarantee the correctness of the given extracted term. This approach corresponds to systems like Minlog or Isabelle. And the realization of such an extension of the extraction seems within reach, thanks to our study of section 2.4.
- There also exists an much more ambitious alternative, but also less realistic in the immediate future. It would be the complete formalization in Coq of our correctness proof for the extraction. That would require much more work than the preceding approach, since it would be a Coq formalization instead of an extension in Ocaml. At the same time we would then obtain a total guarantee, instead of building a particular proof at each extraction. With regard to this possible complete formalization, one can imagine to start from the formalization of Coq in Coq by B. Barras [8], and also re-use one of the works about the semantics of ML, as for example [29]. And why not then try to extract this formalization in order to build an extraction made of extracted code, at least for its central part? This would take naturally place into the utopian project of "bootstrapping" the core of Coq defended by B. Barred.

Concerning the applications of the extraction, the work on the extraction of C-CoRN deserves to be continued. Of course, many problems still persist about efficiency. These problems can be located at several levels. First the algorithms employed are sometimes the most general possible, at the detriment of the efficiency. In addition, the proof scripts are still largely improvable, for example to avoid any redundant computation. Finally the code generated by the extraction is certainly not without defect. All this explains that it appears difficult to hope to compute approximations of roots of polynomials via the extraction of FTA within the next months. At the same time, if one looks at the progress already achieved, it would be a shame to stay there. Let us recall that some time ago, even the extraction stage out of FTA seemed unrealistic...

It would be also interesting to try to extend the field of application of the extraction methodology. Indeed, for the moment, the programs adapted as candidates to certification by extraction are those for which only the result of the execution imports. On the other hand, in the situations where it is of primary importance to terminate quickly or without consuming more than a certain memory quantity, then the extraction is currently not a good methodology. This immediately excludes any embedded program, whether real-time, or with limited memory, or both. On the contrary, the current field of application for the extraction

[4] Since the writing of [79], the situation is even more complex, since the majority of the modern systems use now dynamic libraries (`.so` or `.dll`). It is thus theoretically possible to pervert the behavior of a program *after* its creation without ever touching it.

is highlighted by the main examples: tautology checkers, type checkers [8], program analyzers [17]... Indeed the time and the resources does not matter much when you check a tautology or search for a counterexample, when you typecheck or analyze a program, as long as the result is correct. Currently begins around J.-P. Jouannaud some work on the topic of the time complexity evaluation of the extracted programs, with as reference some work done in Nuprl [11]. Let us hope that this work will allow to extend the applicability of the extraction.

Another possible extension of the field of application for the extraction would consist of an internalization of the results of extracted programs inside Coq. After all, if the extraction is sure, why Coq would not trust a decision procedure proved and then extracted? There undoubtedly matter of interaction with the internal compiler of B. Grégoire [40].

A last domain currently out of reach by the Coq extraction is the extraction of programs starting from classical proofs. This is an extremely active field of research, in particular around Minlog. It should be said that Coq was not, until recently, a good platform for such studies, because the addition of a classical axiom in `Prop` does not change anything for the extraction, whereas the same addition in `Set` makes the system incoherent when `Set` is impredicative. However `Set` is now predicative by default, which may allow a classical extraction in Coq in the future.

Lastly, a last topic of possible improvement is to make this methodology more pleasant to use. A current lack is, for example, the impossibility of importing in Coq some already existing ML code. Of course, one can always translate it by hand, prove the expected properties, then extract, and finally check that the differences between the original ML code and the extracted code are tiny. We have proceeded this way in our certification of the finite sets of Ocaml (see chapter 7), the original manual code being in fact available with Ocaml. This importation of ML code is related with the work by C. Parent around the tactic `Program` [65]. Unfortunately, the implementation of this tactic have not been adapted to the versions 7.0 and following of Coq. It should be stressed that this importation of ML code is far from obvious. For example the importation of a non-structural recursive function will immediately require the proof justifying its good foundation. This would undoubtedly be interesting to combine this importation with the work of A. Balaa and Y. Bertot that facilitates the definition of such recursive functions [6, 5].

It is thus clear that the research possibilities still open at the end of this thesis are multiple, even if much have been done since the beginnings of the extraction in Coq fifteen years ago. Arrived at this point, let us wish that this methodology for program development named extraction can continue its rise, in particular in direction of the industrial world. May this thesis have contributed to that...

Appendix

A User contributions using extraction

- Bordeaux/Additions
- Bordeaux/dictionaries
- Bordeaux/EXCEPTIONS
- Bordeaux/NewSearchTrees
- Bordeaux/SearchTrees
- Dyade/BDDS
- Lannion
- Lyon/CIRCUITS
- Lyon/FIRING-SQUAD
- Marseille/CIRCUITS
- Muenchen/Higman
- Nancy/FOUnify
- Nijmegen/C-CoRN
- Nijmegen/QArith
- Orsay/FSets
- Orsay/QArith
- Rocq/ARITH/Chinese
- Rocq/ARITH/ZChinese
- Rocq/COC
- Rocq/GRAPHS
- Rocq/HIGMAN
- Rocq/MUTUAL-EXCLUSION
- Sophia-Antipolis/Buchberger
- Sophia-Antipolis/Bertrand
- Sophia-Antipolis/Huffman

- Sophia-Antipolis/RecursiveDefinition
- Sophia-Antipolis/Stalmarck
- Suresnes/BDD

Bibliographie

[1] J.-R. Abrial. *The B-Book, assigning programs to meaning.* Cambridge University Press, 1996.

[2] G. M. Adel'son-Vel'skiĭ and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics–Doklady*, 3(5):1259–1263, September 1962.

[3] A. Amerkad, Y. Bertot, L. Rideau, and L. Pottier. Mathematics and proof presentation in pcoq. In *Proceedings of Proof Transformation and Presentation and Proof Complexities (PTP'01)*, 2001. Software available at `http://www-sop.inria.fr/lemme/pcoq`.

[4] D. Aspinall. Proof general: A generic tool for proof development. In *Proceedings of TACAS'2000*, volume 1785. Lecture Notes in Computer Science, 2000. Software available at `http://proofgeneral.inf.ed.ac.uk`.

[5] A. Balaa. *Fonctions récursives générales dans le calcul des constructions.* Thèse d'université, Nice Sophia-Antipolis, November 2002.

[6] A. Balaa and Y. Bertot. Fix-point equations for well-founded recursion in type theory. In Harrison and Aagaard [43], pages 1–16.

[7] H. Barendregt. Lambda Calculi with Types. Technical Report 91-19, Catholic University Nijmegen, 1991. In Handbook of Logic in Computer Science, Vol II.

[8] B. Barras. *Auto-validation d'un système de preuves avec familles inductives.* Thèse de doctorat, Université Paris 7, November 1999.

[9] J. Beauquier and B. Bérard. *Systèmes d'exploitation: concepts et algorithmes.* McGraw Hill, 1990.

[10] M. J. Beeson. *Foundations of Constructive Mathematics.* Springer-Verlag, 1980.

[11] R. Benzinger. Automated complexity analysis of Nuprl extracted programs. *Journal of Functional Programming*, 11(1):3–31, January 2001.

[12] S. Berardi. Pruning simply typed λ-calculi. *Journal of Logic and Computation*, 6(2), 1996.

[13] S. Berghofer. A constructive proof of Higman's lemma in Isabelle. In S. Berardi and M. Coppo, editors, *Types for Proofs and Programs, International Workshop, TYPES 2003*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

[14] S. Berghofer. *Proofs, Programs and Executable Specifications in Higher Order Logic.* PhD thesis, Institut für Informatik, Technische Universität München, 2003.

[15] L. Boerio. Extending pruning techniques to polymorphic second order λ-calculus. In *Proceedings ESOP'94*, volume 788. Lecture Notes in Computer Science, 1994.

[16] S. Boutin. Using reflection to build efficient and certified decision procedure s. In Martin Abadi and Takahashi Ito, editors, *TACS'97*, volume 1281. Lecture Notes in Computer Science, 1997.

[17] D. Cachera, T. Jensen, D. Pichardie, and V. Rusu. Extracting a Data Flow Analyser in Constructive Logic. In D. Schmidt, editor, *European Symposium on Programing, ESOP'2004*, volume 2986 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[18] J. Chrząszcz. Implementing modules in the system Coq. In *16th International Conference on Theorem Proving in Higher Order Logics*, University of Rome III, September 2003.

[19] J. Chrząszcz. *Modules in Type Theory with generative definitions*. PhD thesis, Warsaw University and Université Paris-Sud, 2003.

[20] T. Coquand. An analysis of Girard's paradox. In *Proceedings of the First Symposium on Logic in Computer Science*, Cambridge, MA, June 1986. IEEE Comp. Soc. Press.

[21] T. Coquand and D. Fridlender. A proof of Higman's lemma by structural induction. Technical report, Chalmers University, November 1993. Unpublished draft, available at `ftp://ftp.cs.chalmers.se/pub/users/coquand/open1.ps.Z`.

[22] J. Courant. A Module Calculus for Pure Type Systems. In *Typed Lambda Calculi and Applications 97*, Lecture Notes in Computer Science, pages 112 – 128. Springer-Verlag, 1997.

[23] L. Cruz-Filipe. A constructive formalization of the fundamental theorem of calculus. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs, TYPES 2002*, volume 2646 of *LNCS*, pages 108–126. Springer–Verlag, 2003.

[24] L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications*. PhD thesis, University of Nijmegen, April 2004.

[25] L. Cruz-Filipe and B. Spitters. Program extraction from large proof developments. In D. Basin and B. Wolff, editors, *Theorem Proving in Higher Order Logics, 16th International Conference, TPHOLs 2003*, volume 2758 of *LNCS*, pages 205–220. Springer–Verlag, 2003.

[26] H. B. Curry and R. Feys. *Combinatory Logic*, volume 1. North-Holland, Amsterdam, 1958.

[27] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th Annual ACM Symposium on Principles of Programming Languages*, pages 207–212. ACM Press, 1982.

[28] G. Dowek, G. Huet, and B. Werner. On the Definition of the Eta-long Normal Form in the Type Systems of the Cube. Informal Proceedings of the Workshop "Types", Nijmegen, 1993.

[29] C. Dubois. Typing Soundness of ML within Coq. In Harrison and Aagaard [43], pages 127–144.

[30] H. Benl et al. Proof theory at work: Program development in the Minlog system. In Wolfgang Bibel and Peter H. Schmidt, editors, *Automated Deduction: A Basis for Applications. Volume II, Systems and Implementation Techniques.* Kluwer Academic Publishers, Dordrecht, 1998.

[31] S. Peyton Jones et al. *Haskell 98, A Non-strict, Purely Functional Language*, 1999. Available at `http://haskell.org/`.

[32] J.-C. Filliâtre. *Preuve de programmes impératifs en théorie des types.* Thèse de doctorat, Université Paris-Sud, July 1999.

[33] J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003.

[34] J.-C. Filliâtre and P. Letouzey. Functors for Proofs and Programs. In D. Schmidt, editor, *European Symposium on Programing, ESOP'2004*, volume 2986 of *Lecture Notes in Computer Science.* Springer-Verlag, 2004.

[35] H. Geuvers, F. Wiedijk, and J. Zwanenburg. A Constructive Proof of the Fundamental Theorem of Algebra without Using the Rationals. *LNCS*, 2277:96–111, 2001.

[36] Herman Geuvers and Milad Niqui. Constructive Reals in Coq: Axioms and Categoricity. *LNCS*, 2277:79–95, 2001.

[37] E. Giménez. An application of co-Inductive types in Coq: verification of the Alternating Bit Protocol. In *Workshop on Types for Proofs and Programs*, number 1158 in LNCS, pages 135–152. Springer-Verlag, 1995.

[38] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types.* Cambridge Tracts in Theoretical Computer Science 7. Cambridge University Press, 1989.

[39] B. Grégoire and X. Leroy. A compiled implementation of strong reduction. In *International Conference on Functional Programming 2002*, pages 235–246. ACM Press, 2002.

[40] B. Grégoire. *Compilation des termes de preuves : un (nouveau) mariage entre Coq et OCaml.* PhD thesis, Université Paris 7, 2003.

[41] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science*, pages 8–21, Ann Arbor, Michigan, 16-18 October 1978. IEEE.

[42] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Conference record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 123–137, Portland, OR, January 1994.

[43] J. Harrison and M. Aagaard, editors. *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science.* Springer-Verlag, 2000.

[44] S. Hayashi and H. Nakano. PX, a Computational Logic. Technical report, Research Institute for Mathematical Sciences, Kyoto University, 1987.

[45] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, 1969.

[46] W.A. Howard. The formulae-as-types notion of constructions. In J.P. Seldin and J.R. Hindley, editors, *to H.B. Curry : Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, 1980. Unpublished 1969 Manuscript.

[47] G. Huet, G. Kahn, and Ch. Paulin-Mohring. *The Coq Proof Assistant - A tutorial - Version 8.0*, February 2004. Available at `http://coq.inria.fr/`.

[48] R. Kelsey, W. Clinger, and J. Rees (eds.). *Revised^5 Report on the Algorithmic Language Scheme*, 1998. Available at `http://www.scheme.org/`.

[49] S. C. Kleene. *Introduction to Metamathematics*. North-Holland, Amsterdam, 1952.

[50] C. Kreitz. *The Nuprl Proof Development System, Version 5*. Cornell University, Ithaca, NY, 2002. Available at `http://www.nuprl.org`.

[51] O. Lee and K. Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):707–723, July 1998.

[52] X. Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.

[53] X. Leroy, J. Vouillon, D. Doliguez, J. Garrigue, and D. Rémy. *The Objective Caml system – release 3.07*, September 2003. Available at `http://caml.inria.fr/`.

[54] P. Letouzey. Exécution de termes de preuves: une nouvelle méthode d'extraction pour le Calcul des Constructions Inductives. Université Paris VI, 2000. Available at `http://www.lri.fr/~letouzey/download/rapport_dea.ps.gz`.

[55] P. Letouzey. A New Extraction for Coq. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002*, volume 2646 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[56] P. Letouzey and L. Théry. Formalizing Stålmarck's algorithm in Coq. In Harrison and Aagaard [43], pages 387–404.

[57] Z. Luo. *Computation and Reasoning; a type theory for Computer Science*, volume 11 of *International Series of Monographs in Computer Science*. Oxford Science Publication, 1994.

[58] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

[59] R. Milner. A theory of type polymorphismn programming. *Journal of Computer and System Sciences*, 17, 1978.

[60] J.-F. Monin. Extracting Programs with Exceptions in an Impredicative Type System. In B. Möller, editor, *Mathematics of Program Construction*, volume 947 of *LNCS*. Springer Verlag, 1995.

[61] J.-F. Monin. *Contribution aux méthodes formelles pour le logiciel*. Mémoire d'habilitation à diriger des recherches, Université de Paris Sud, avril 2002.

[62] D. Monniaux. Réalisation mécanisée d'interpréteurs abstraits. Rapport de DEA, Université Paris VII, 1998.

[63] C. Murthy and J.R. Russell. A constructive proof of Higman's lemma. In *Proc. 5th IEEE Symp. Logic in Computer Science, Philadelphia*, pages 257–267, 1990.

[64] C. Okasaki. *Purely functional data structures.* Cambridge University Press, 1998.

[65] C. Parent. *Synthèse de preuves de programmes dans le Calcul des Constructions Inductives.* thèse d'université, École Normale Supérieure de Lyon, January 1995.

[66] C. Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, Austin, January 1989. ACM Press.

[67] C. Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions.* Thèse d'université, Paris 7, January 1989.

[68] C. Paulin-Mohring. *Définitions Inductives en Théorie des Types d'Ordre Supérieur.* Habilitation à diriger les recherches, Université Claude Bernard Lyon I, December 1996.

[69] C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15:607–640, 1993.

[70] L. Pottier. Extraction dans le calcul des constructions inductives. In *Journées Francophones des Langages Applicatifs*, 2001.

[71] D. Prawitz. *Natural Deduction, A Proof-Theoretical Study.* Almqvist & Wiksell, 1965.

[72] F. Prost. *Interprétation de l'analyse statique en théorie des types.* PhD thesis, École Normale Supérieure de Lyon, december 1999.

[73] H. Schwichtenberg. Constructive analysis with witnesses. Technical report, Ludwig-Maximilians-Universität, München, 2003. Proceedings of Marktoberdorf '03 Summer School.

[74] M. Seisenberger. *On the Constuctive Content of Proofs.* PhD thesis, Ludwig-Maximilians-Universität München, Fakultät für Mathematik, Informatik und Statistik, 2003.

[75] M. Serrano and P. Weis. Bigloo: a portable and optimizing compiler for strict functional languages. In *2nd Static Analysis Symposium (SAS)*, pages 366–381. Lecture Notes in Computer Science, 1995.

[76] P. Severi and N. Szasz. Studies of a theory of specifications with built-in program extraction. *Journal of Automated Reasoning*, 27(1), 2001.

[77] Julien Signoles. Calcul statique des applications de modules paramétrés. In *Journées Francophones des Langages Applicatifs*, 2003.

[78] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version 8.0*, February 2004. Available at `http://coq.inria.fr/`.

[79] K. Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, Aug 1984.

[80] A. S. Troelstra and D. van Dalen. *Constructivism in Mathematics: an Introduction.* Studies in Logic and the Foundations of Mathematics. North-Holland, 1988.

[81] P. Wadler, W. Taha, and D. MacQueen. How to add laziness to a strict language, without even being odd. In *Workshop on Standard ML*, Baltimore, September 1998.

[82] K. Weich. Decision procedures for intuitionistic propositional logic by program extraction. *Lecture Notes in Computer Science*, 1397:292, 1998. see `http://www.mathematik.uni-muenchen.de/~weich/`.

[83] B. Werner. *Méta-théorie du Calcul des Constructions Inductives.* Thèse d'université, Univ. Paris VII, 1994.

Résumé

Nous nous intéressons ici à la génération de programmes certifiés corrects par construction. Ces programmes sont obtenus en extrayant l'information pertinente de preuves constructives réalisées dans l'assistant de preuves Coq.

A telle traduction, ou « extraction », des preuves constructives en programmes fonctionnels n'est pas nouvelle, elle correspond à un isomorphisme bien connu sous le nom de Curry-Howard. Et l'assistant Coq comporte depuis longtemps un tel outil d'extraction. Mais l'outil précédent présentait d'importantes limitations. Certaines preuves Coq étaient ainsi hors de son champ d'application, alors que d'autres engendraient des programmes incorrects.

Afin de résoudre ces limitations, nous avons effectué une refonte complète de l'extraction dans Coq, tant du point de vue de la théorie que de l'implantation. Au niveau théorique, cette refonte a entraîné la réalisation de nouvelles preuves de correctness de ce mécanisme d'extraction, preuves à la fois complexes et originales. Concernant l'implantation, nous nous sommes efforcés d'engendrer du code extrait efficace et réaliste, pouvant en particulier être intégré dans des développement logiciels de plus grande échelle, par le biais de modules et d'interfaces.

Enfin, nous présentons également plusieurs études de cas illustrant les possibilités de notre nouvelle extraction. Nous décrivons ainsi la certification d'une bibliothèque modulaire d'ensembles finis, et l'obtention de programmes d'arithmétique réelle exacte à partir d'une formalisation d'analyse réelle constructive. Même si des progrès restent encore à obtenir, surtout dans ce dernier cas, ces exemples mettent en évidence le chemin déjà parcouru.

Mots clés. Preuve de programmes. Programmation fonctionnelle. Extraction. Théorie des types. Isomorphisme de Curry-Howard. Calcul des Constructions Inductives. Système Coq.

Abstract

This work concerns the generation of programs which are certified to be correct by construction. These programs are obtained by extracting relevant information from constructive proofs made with the Coq proof assistant.

Such a translation, named "extraction", of constructive proofs into functional programs is not new, and corresponds to an isomorphism known as Curry-Howard's. An extraction tool has been part of Coq assistant for a long time. But this old extraction tool suffered from several limitations: in particular, some Coq proofs were refused by it, whereas some others led to incorrect programs.

In order to overcome these limitations, we built a completely new extraction tool for Coq, including both a new theory and a new implementation. Concerning theory, we developed new correctness proofs for this extraction mechanism. These new proofs are both complex and original. Concerning implementation, we focused on the generation of efficient and realistic code, which can be integrated in large-scale software developments, using modules and interfaces.

Finally, we also present several case studies illustrating the capabilities of our new extraction. For example, we describe the certification of a modular library of finite set structures, and the production of programs about real exact arithmetic, starting from a formalization of constructive real analysis. These examples show the progress already achieved, even if the situation is not perfect yet, in particular in the last study.

Keywords. Proof of programs. Functional programming. Extraction. Type theory. Curry-Howard isomorphism. Calculus of Inductive Constructions. Coq system.