sadhana manuscript No.
(will be inserted by the editor)

A compact kernel for the calculus of inductive constructions

A. Asperti · W. Ricciotti ·
C. Sacerdoti Coen · E. Tassi

the date of receipt and acceptance should be inserted later

Abstract The paper describes the new kernel for the Calculus of Inductive Constructions (CIC) implemented inside the Matita Interactive Theorem Prover. The design of the new kernel has been completely revisited since the first release, resulting in a remarkably compact implementation of about 2300 lines of OCaml code. The work is meant for people interested in implementation aspects of Interactive Provers, and *is not self contained.* In particular, it requires good acquaintance with Type Theory and functional programming languages.

Keywords Interactive Theorem Provers · Calculus of Inductive Constructions · Matita · Kernel

Department of Computer Science, University of Bologna (Italy)
E-mail: asperti,ricciott,sacerdot,tassi@cs.unibo.it
The work was written while the first author was on leave at the Microsoft Research-INRIA Joint Center in Orsay, France. He would like to thank the Center for the hospitality and the financial support.

Contents

1 Introduction

We describe in this paper the new kernel for the Calculus of Inductive Constructions (CIC) (see Werner (1994); Paulin-Mohring (1996); Giménez (1998)) implemented inside the Matita Interactive Theorem Prover. The kernel is the most delicate component of interactive provers, in charge of the verification of proofs. Not all interactive provers have a kernel in the above sense, and not all systems adhere to the so called de Bruijn principle, namely the plea for a *small* kernel (see Wiedijk (2006) for a comparison). Still, having a small, clear, well documented, freely inspectable and easily replicable kernel, seems to be the best warranty for the effective reliability of the system.

Somehow surprisingly, even among the developers of systems adopting a kernel-oriented approach, documentation does not seem to be a priority (that is even more surprising considering that the kernel is, usually, a particularly stable part of the system). As far as we know, our paper is the first extensive description of a system kernel, and in particular the first detailed discussion of the implementation of the Calculus of Inductive Constructions. This, even before the technical interest of the paper, is the major novelty of our contribution, promoting a more transparent (and competitive) attitude to the implementation of interactive provers, and hopefully paving the way to a stream of similar contributions.

The new Matita kernel is just the first step of a deep revisitation work of the entire system. Our first implementation of Matita, taking us about 10 man years work, spread over a period of 5 years (see Asperti et al (2006) for an account), although already resulting in a competitive system enabling complex formalization efforts (see e.g. Asperti and Ricciotti (2008)), has been for us mostly an occasion to acquire a direct experience of the field. Taking advantage of this knowledge, and of all the lessons learned during the realization of the first Matita prototype, we feel now ready to restart the work from scratch, and to expose, piece by piece, a coherent and meditated architectural design of the system, and a detailed description of its implementation. The new kernel, described in this paper, is the first piece of this program.

1.1 A software engineering perspective

Reliability is not the only reason for having a small kernel. The point is that most of the data structures of the kernel have a direct impact on the rest of the system. For instance, the structure of the refiner essentially mimics that of the type-checker; hence, a simpler type-checker will likely results in a simpler, more maintainable and hence more adaptable refiner, that in turn could have a beneficial effect on the chaotic realm of tactics. The complex pipeline of transformations between the low level CIC term and the user input syntax is still another aspect of the system where a simplification of the data structures could likely give rise to a sensible simplification of the code.

The point is that interactive provers have an intrinsic complexity that is hard to understand and hence to master. A priori, the basic user interaction loop implemented by these applications seems to be quite trivial: the system must read a user command, execute it, update the system status and present to the user a new view. For this reason, when we completed the implementation of the first Matita prototype, it was for us a sort of surprise to discover that we wrote no less than 70.000 lines of OCaml code[1]. But the point that is really disconcerting is that even a posteriori, and in spite of our attempts to analyze and systematize the code, the reasons for the complexity of these systems (comprising our own!) still remain, for us, quite puzzling. Our conclusion is that it is due to a myriad of *minor* design choices not sufficiently meditated, cumulatively producing a quite aberrant final effect. Trying to correct a posteriori these wrong choices looks like a desperate effort. Better to restart the implementation from scratch, following a more careful methodological design. Our final aim is to reduce the dimension of the system at about 40.000 lines of OCaml code, that is little more than the half of the current dimension of the system. Of course, we are not seeking to have a compact code per se; the point is that having a simpler system would reduce the number of bugs[2], improve the maintainability of the system, reduce the training period for new developers, and make easier to modify the application for experimenting new

[1] That is in any case little more than the half of the code of Coq!

[2] When implementing the new version of the Matita kernel with simpler data structures and invariants, we spotted several bugs in the old kernel version that were hidden by the complexity of its code.

techniques or implementing additional functionalities. All features that are essential for an avant-guard tool like Matita, explicitly meant for experimental prototyping and mostly developed by students.

The paper does not provide a formal description of the CIC calculus and its meta-theory for many reasons. One motivation is that a formal presentation of the calculus and its main properties would escape the software engineering scope of the special issue. Another major motivation is the fact that there is no coherent exposition of the meta-theory of the whole CIC as it is currently implemented in Coq and Matita. Indeed, the CIC calculus is obtained as an extension of the Calculus of Constructions (COC) with a hierarchy of universes, primitive inductive and coinductive types, case analysis operators and constrained recursion. The meta-theory of each one of these extensions has been provided separately by different authors, and the community still deserves a unified view of it and also models for the whole calculus (the interested reader can find some documentation in Paulin-Mohring (1996); Geuvers (1993); Werner (1994); Giménez (1998); Luo (1990); Muoz (1997)). Moreover, many of these extensions are based on syntactic approximations of undecidable problems, like termination of recursive functions, which are clearly heuristics. These heuristics have changed during the years, they have deserved little attention in the literature, and nevertheless their intimate knowledge is necessary for users to exploit the full capabilities of the system.

To conclude, we have decided to accompany the code of Matita with an high level description of the positivity conditions and the termination heuristics for recursive and co-recursive functions. These are given at the beginning of the sections relative to them, together with examples to help the reader to understand the code. Moreover, we recall in App. A a syntax directed version of the type-checking rules for CIC. We do not explain how they are obtained from the standard type-checking rules since this translation perfectly mimics the one for the Calculus of Constructions. The Calculus of Constructions is well known and already documented as a pure type system in Barendregt (1992). A syntax directed presentation of its rules has been already addressed in van Benthem Jutting et al (1994); Pollack (1994) and implemented in at least the COQ Huet et al (1998) and LEGO Pollack (1994) systems. The code of Matita follows this presentation for the PTS part of CIC, which remains a functional and full PTS.

1.2 The overall structure of the kernel

Figure 1 describes the structure of the kernel modules, and their dependencies.

We have 8 modules, whose content will be briefly described here. The rest of the paper contains a detailed documentation of their implementation.

NUri: a small module implementing logical names by means of Uniform Resource Identifiers (URI). We use URIs to locate objects in a library. The untrusted code that maps logical names (URIs) to physical names (URLs) before retrieving and parsing the object is implemented in the NCicLibrary module outside the kernel.

NCicEnvironment: the management of the environment of CIC-objects (Section 4.2).

NReference: the data type descriptor for the different constants of CIC (Section 2.1.3).

NCic: data structures for the Calculus of Inductive Construction, comprising terms, contexts and objects (Section 2). The kernel uses an untrusted module NCicPp to pretty-print CIC terms when reporting errors to the user.

NCicUtils CIC-iterators and main utility functions (Section 3).

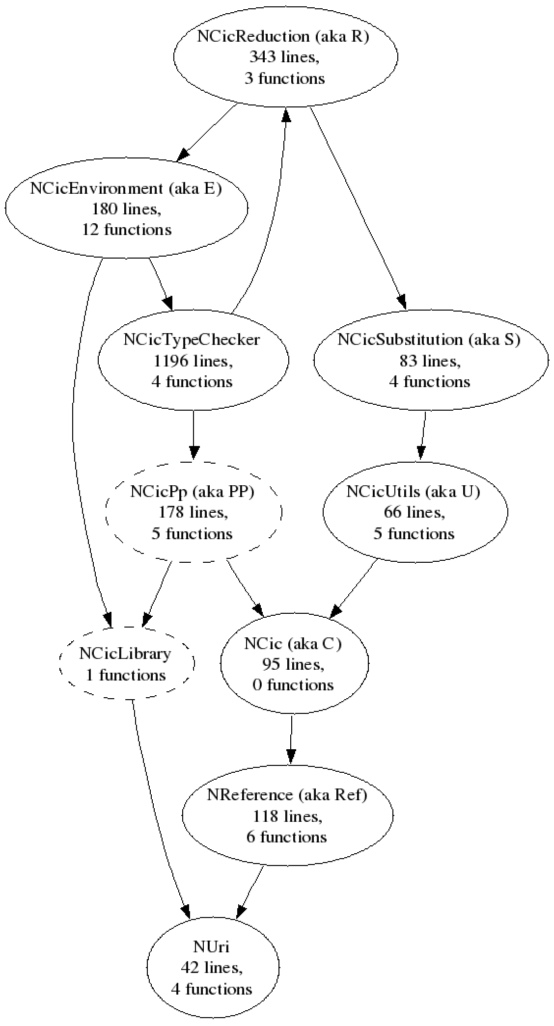

Fig. 1 The solid nodes are the only modules that need to be trusted, and thus belong to the kernel. The dashed nodes are untrusted. In particular, the information retrieved from the library is always type-checked by the kernel before usage. For each module we show the lines of codes required for the implementation and the number of functions exported to other modules in the interface (listed in App. B). Modules names are usually shortened using the acronyms in parentheses.

NCicSubstitution: low-level manipulation of terms in DeBruijn notation: lifting and substitution (Section 3.1).

NCicReduction: the reduction machine for the calculus, and the conversion test (Section 5).

NCicTypeChecker: the type synthesis algorithm (Section 6), comprising in particular positivity conditions for inductive types (Section 6.2) and well guardedness (termination) conditions for recursive functions (Section 6.3).

2 CIC data structures

2.1 Terms

Our description begins with the main data structure we employ for terms of the Calculus. The concrete syntax adopted by Matita for terms closely reflect the internal data structure and is very similar to the one adopted by Coq and described in Bertot and Castéran (2004).

```
type universe = (bool * NUri.uri) list
  (* Max of non-empty list of named universes, or their successor (when true) *)
type sort = Prop | Type of universe
type implicit_annotation = [ `Closed | `Type | `Hole | `Term ]
type lc_kind = Irl of int | Ctx of term list
and local_context = int * lc_kind              (* shift (0 -> no shift),
                                                  subst (Irl n means id of
                                                  lenght n)                *)
and term =
  | Rel      of int                            (* DeBruijn index, 1 based  *)
  | Meta     of int * local_context
  | Appl     of term list                      (* arguments                *)
  | Prod     of string * term * term           (* binder, source, target   *)
  | Lambda   of string * term * term           (* binder, source, target   *)
  | LetIn    of string * term * term * term    (* binder, type, term, body *)
  | Const    of NReference.reference           (* ref: (indtype|constr)no  *)
  | Sort     of sort                           (* sort                     *)
  | Implicit of implicit_annotation            (* ...                      *)
  | Match    of NReference.reference *         (* ind. reference,          *)
     term * term *                             (*  outtype, ind. term      *)
     term list                                 (*  branches                *)
```

A term is either a variable (Rel), a metavariable (Meta), an application (Appl), a product (Prod), a lambda abstraction (Lambda), a let-in construct (LetIn), a defined constant (Const), a sort (Sort), and implicit term (Implicit), or a case construct (Match).

2.1.1 Variables, metavariables and Implicit terms

Variables are encoded by means of direct DeBruijn notation, i.e. as the position of the binder (counting from 1) along the path of the abstract syntax tree leading from the variable occurrence to its binder. Thus, for instance, $\lambda f : \ldots \lambda d : \ldots (f\ (f\ x))$ is represented as Lambda("f",...,Lambda("x",...,Appl ([Rel 2; Appl ([Rel 2; Rel 1])]))) .

A metavariable is a hole in the term. In the Curry-Howard analogy, it also represents a portion of the proof that has still to be filled-in; the types of metavariables are hence the goals yet to be solved by the user. The presence of metavariables in the kernel is one of the distinctive features of Matita w.r.t. Coq: the main advantage is the possibility to check the well typedness of open terms, resulting in a simpler interface with the refiner[3]. The complexity of the management of metavariables is due to the fact that, after being created (in a given context, that we call *canonical context*), due to the possible reduction of the term, they can be duplicated and moved in different contexts (e.g. moved under different binders). This makes extremely difficult to manage their successive instantiation, since the instance must be suitably relocated in the current context of the other occurrences (see McBride (1999); Geuvers and Jojgov (2002)).

[3] The refiner is the component, just outside the kernel, in charge of type inference.

To solve this problem, it is convenient to equip each metavariable with an explicit substitution. In Matita, to avoid the use of explicit names, the explicit substitution is organized as a list of terms, defining a *local context* for the metavariable: when the metavariable will be instantiated with a term, the free variables of the term will be solved in this local context (that hence plays the role of a closure). An additional structure, the metasenv, contains the list of all metavariables, with their canonical contexts and their types. To make an example of how this information is exploited by unification (outside the kernel), suppose to have two instances $M_1[l_1]$ and $M_2[l_2]$ of a given metavariable (with respective local contexts l_1 and l_2), and that M_1 is instantiated with a term t_1. The first step is to find (according to some heuristics) a term t such that $t[l_1] = t_1$, that is to "relocate" t_1 in the canonical context of the metavariable, and then to instantiate M_2 with $t[l_2]$. Let us finally stress that, although we allow the presence of metavariables in the kernel, no unification is running up in it (in particular, nothing in the kernel depends on heuristics).

The implementation of the local context is one of the novelties of the new Matita kernel. In fact, by extensive testing of the system, we realized that in most of the cases the local context (even after duplication) is the identity substitution, while in many others is just a simple shift operation. Hence we decided to make this two frequent subcases explicit, in order to take advantage of this information in the implementation of several functions both in the kernel and outside it.

Implicit terms are a degenerate form of CIC-term (unknown/don't care term) *only meant for extra-kernel usages.* Their degenerate nature is reflected in the fact that no term processed in kernel should (still) contain an Implicit: all kernel functions return an exception, in this case. Typical usages are e.g. in prerefinement phases of parsing, or for expressing paths inside CIC-terms (by "pruning" irrelevant parts).

2.1.2 Lambda terms, types and sorts

Applications, products, lambda abstractions, let-ins and sorts are the basic and well known ingredients of typed lambda calculi (see Barendregt (1992) for an introduction), and there is little to add, here. The type discipline of the Calculus of Inductive Constructions is based on an impredicative sort Prop of propositions, and a predicative hierarchy of Universes (Luo (1990)) $Type_i : Type_{i+1}$ ($Set = Type_0$). For flexibility reasons, it is useful to remove fixed universes (of the form $Type_i$) in favour of universe variables $Type_u$ (where u is an URI) with explicit user-declared constraints between universe variables (Courant (2002)). In Matita we also keep outside the kernel a boolean, associated to universe variables and used during pretty-printing of formulae, to choose the presentational flavor (type vs. formula) for inhabitants of the universe. We plan to let the user dynamically configure these aspects of the type system in the next release of Matita.

In particular, our representation of universes is a major simplification of that of EPECC (Courant (2002)). The user is allowed to define universe variables (Type being just one of them) and constraints among universe variables (e.g. Type < Type1). The boolean value paired with an URI is true when we are taking the successor of a universe variable. A list of universe variables (or they successors) has to be read as the maximum of its elements. An empty list represents the smallest universe and is used to type the sort of propositions. Further informations about this representation of universes will be given in Section 4.3 where we will also present the data structures employed in the kernel to represent constraints between universe variables.

As for the other terms, let us just remark that it is worth to keep applications in "flat" normal form, i.e. without applications in head position of other applications (to be ensured during substitution); we also implicitly assume that the list of arguments of an application has always length greater then 1. Another points concerns the let-in construct; in our first implementation of Matita the type of the named term was optional, and we had to cache the inferred type to avoid duplicate work, for instance during type-checking; a posteriori it is simpler to have it always explicit, at the price of inferring it outside the kernel when the user does not want to give it explicitly.

2.1.3 Constants

The design of constants is the point where the new Matita kernel differs more sensibly from our first implementation and from the current Coq kernel. Under the generic name of constants we are in fact grouping names referring to six different kind of objects: declarations (axioms, variables, ...), definitions (functions, theorems, ...), recursive and co-recursive functions, (co)inductive types and constructors of (co)inductive types. In particular, each constant is equipped with a descriptor of type reference, defined as follows:

```
type spec =
  | Decl
  | Def of int                  (* height *)
  | Fix of int * int * int      (* fixno, recparamno, height *)
  | CoFix of int
  | Ind of bool * int * int     (* inductive, indtyno, leftno *)
  | Con of int * int * int      (* indtyno, constrno, leftno  *)
type reference = Ref of NUri.uri * spec
```

A reference is a couple (u, s) where u is its (long) name, and s is the descriptor of the object, of type spec. The first argument of Fix and CoFix is the number of the function in a block of mutually recursive (resp. corecursive) ones; in addition, the Fix constructor also takes the number of the "recursive parameter", that is of the argument that is supposed to decrease during reduction (used to trigger the reduction of fixpoints, see Section 5). The last parameter of Def and Fix is the height of the constant, expressing the maximum chain of dependencies between the given constant and the library objects it depends upon. The first parameter of Ind makes a distinction between inductive and coinductive types: the information is already in the object declaration, but is replicated in the reference to avoid a few lookups during type-checking. The second parameter of Ind (the first of Con) is the number of the inductive type in a block of mutually inductive ones; the second parameter of Con is the number of the given constructor. In both Ind and Con the number of left parameters (see Sec. 2.3) of the inductive type is replicated to avoid few lookups during reduction.

The major paradigm shift w.r.t the old kernel concerns the decision to move the definition of (co)fixpoints from the level of terms to the level of objects, resulting in a drastic simplification of the kernel code, and with the only drawback of loosing the possibility to declare nested fixpoints (this does not imply a loss of expressiveness, since by the technique of lambda lifting (Johnsson (1985)) we may always move up inner functions).

2.1.4 Matching

The last component of the syntax of terms is **match**. As a first approximation, its usage is similar to the match-with statement of ML-like languages, where the role of constructors in patterns is played by inductive type constructors. A match-term depends on four arguments (u, T, t, pl). t is the matching term, its type must be an inductive type with name (uri) u; pl is a list of bodies for the different branches of the match (as many as the number of constructors of the inductive type); each branch is explicitly abstracted over the input arguments of the corresponding constructor. Finally, T is the so called result type. Due to the dependent type discipline, the type checker would not be always able to guess a uniform result type for all branches, and this must hence be explicitly provided (see Section 6 for a detailed discussion).

2.2 Context, metasenv and substitution

Each term lives in a context containing definitions or declarations for its free variables; moreover, all its free metavariables are either declared in a metasenv, or instantiated by a substitution.

```
type context_entry =                    (* A declaration or definition *)
  | Decl of term                        (* type *)
  | Def  of term * term                 (* body, type *)
type hypothesis = string * context_entry
type context = hypothesis list
type conjecture = string option * context * term
type metasenv = (int * conjecture) list
type subst_entry = string option * context * term * term
type substitution = (int * subst_entry)  list
```

The context_entry type should be clear. An hypothesis is just a named context_entry, where the name is simply a string. A context is a list of hypothesis. A conjecture is a metavariable declaration: it depends on three arguments (s, c, T) where s is an optional name, c is the canonical context discussed in the previous section, and T is the type of the metavariable (obviously, w.r.t. its canonical context). A metasenv is a list of pairs associating to each metavariable (identified by an integer) its declaration.

The natural complement of the metasenv is the substitution, that is a list of pairs associating to a metavariable a declaration subst_entry, that is a tuple (s, c, t, T) where s is an optional name, c is the canonical context, t is the term instantiating the metavariable, and T is the type of t.

2.3 Objects

An object is a possibly incomplete definition, declaration or theorem made of closed terms. Generic attributes referring to all kind of objects are hence its name (uri), its depth (defined as the maximum of the depth of the objects occurring in its definition plus one), a metasenv and a subst (usually empty). Since the depth is mainly used to drive the unfolding of terms during the conversion test, if the object does not have a body (axioms, inductive types and so on) the depth is conventionally set to 0.

```
 (* invariant: metasenv and substitution have disjoint domains *)
type obj_kind =
 | Constant of relevance * string * term option * term * c_attr
 | Fixpoint  of bool * inductiveFun list * f_attr
                 (* true -> fix, funcs, arrts *)
 | Inductive of bool * int * inductiveType list * i_attr
                 (* true -> inductive, leftno, types *)
 (* the int must be 0 if the object has no body *)
type obj = NUri.uri * int * metasenv * substitution * obj_kind
```

We have three main kinds of objects: (non recursive) Constants, blocks of recursive (or corecursive) functions, and blocks of inductive (or coinductive) types.

The simplest kind is a Constant, taking five arguments (r, t, T, a). t is an optional body (for instance, axioms do not have a body), T is its type, and a is a list of attributes, containing information about the object not relevant for the kernel, but useful for many other functionalities of the system. r is a new, experimental argument aimed to integrate *proof irrelevance* (see e.g. Miquel and Werner (2003); Werner (2008)) into Matita: it is a list of boolean parameters expressing, for each of the input arguments of the constant, its "relevance" in view of conversion (see Section 5.2).

```
type relevance = bool list (* relevance of arguments for conversion *)
type def_flavour = (* presentational *)
  [ `Definition | `Fact | `Lemma | `Theorem | `Corollary | `Example ]
type def_pragma = (* pragmatic of the object *)
  [ `Coercion of int
  | `Elim of sort          (* elimination principle; universe is not relevant *)
  | `Projection            (* record projection *)
  | `InversionPrinciple    (* inversion principle *)
  | `Variant
  | `Local
  | `Regular ]               (* Local = hidden technicality *)
type ind_pragma = (* pragmatic of the object *)
  [ `Record of (string * bool * int) list | `Regular ]
  (* inductive type that encodes a record; the arguments are the record
   * fields names and if they are coercions and then the coercion arity *)
type generated = [ `Generated | `Provided ]
type c_attr = generated * def_flavour * def_pragma
type f_attr = generated * def_flavour
type i_attr = generated * ind_pragma
```

A Fixpoint is a block of mutually recursive functions (or mutually corecursive, if the first boolean argument is false). Apart from attributes, it is just a list of (co)recursive functions, defined by a relevance list for arguments, a local name to distinguish functions of a same block, the number of the recursive parameter (i.e. of the argument which is supposed to structurally decrease during reduction, in order to guarantee termination), the body of the function and its type.

```
                         (* relevance, name, recno, ty, bo *)
type inductiveFun = relevance * string * int * term * term
  (* if coinductive, the int has no meaning and must be set to -1 *)
```

An Inductive object, is a block of mutually defined inductive types (resp. coinductive, if the first boolean argument is false). Apart from attributes, and the list of (co)inductive types, it is made of a *leftno* parameter. Actually, each Inductive object is a *family* of types, indexed over a given number of fixed parameters, which are supposed to appear identical as initial sources in all the types of the inductive objects and associated constructors belonging to the block. In lack of an agreed terminology, and due to their initial positions in types, we simply call these arguments *left parameters*. We call *right parameters* the remaining sources in the type of the inductive definition[4]. Hence, *leftno* is simply the number of left parameters of the block. The use of left parameters will be explained in Sections 5 and 6.

```
type constructor = relevance * string * term  (* id, type *)
type inductiveType =
 relevance * string * term * constructor list
 (* relevance, typename, arity, constructors *)
```

Finally, each inductive type is simply defined by a relevance list, a name, a type and its constructor list; a constructor has a relevance list, a name and a type.

2.4 Other changes w.r.t. the old Matita kernel

In this section we discuss some other important changes in the CIC datatype, not directly visible in the new syntax.

The first important modification is in the structure of the contexts. In the old kernel, the hypothesis composing the contexts were *optional*, reflecting the fact that they could no longer be accessible (typically, as a result of an explicit clear command of the user). This decision (that looked quite natural at the time) has in fact a quite nasty impact in several parts of the code. The biggest problem is that the clear operation is performed on a given instance of a metavariable (the current goal), but the clear operation must be reflected in its canonical context in the metasenv, and hence propagated (possibly, on demand) to all other instances. In other words, the optional nature of the context entries, was naturally reflected in the optional nature of the arguments in the local contexts, and the problem was to keep in synch the local contexts of the different instances, both between them and with the canonical context.

With the current implementation, a clear operation involves the creation of a fresh new metavariable with a restricted canonical context, and a unification between this metavariable and the old one.

At the object level, all object declarations were equipped by a list of parameters, which played the role of instantiatable axioms. The invocations of an object had an explicit substitution possibly instantiating these parameters with CIC terms. These explicit substitutions were the Matita analog of Coq sections, and the instantiatable axioms corresponded to Coq section variables. They were mostly introduced in Matita for compatibility purposes. The double role of instantiatable axioms (as axioms and as binders) interfered with the indexing and searching tools of Matita, since axioms had to be indexed when used as axioms by the user, but not when used as abstractions. Moreover, after extensive use of the system, no Matita user has apparently felt the need of using such a feature, and we believe that it should be possible to find better alternatives. Thus we decided to remove it from the new kernel. In general, the management of sections (and palliatives) inside the kernel seems to be a clear mistake: it complicates in a sensible way the management of objects, for purposes which are mostly of an extra-logical (presentational) nature.

[4] The terminology adopted by Dybjer in Dybjer (1997) for what we call left parameters is simply *parameters*, while he uses *indexes* in place of right parameter

Another major change with respect to the previous kernel is the adoption of algebraic universes to represent inferred types, which we have already discussed, and the choice of abandon inference of universes, which used to performed in the old kernel and that has been replaced by checking of user provided constraints in the spirit, and with the same advantages, of EPECC Courant (2002). The new representation also allowed us to drop the special treatment of the CProp sort in the old kernel. The CProp sort was a special type universe inhabited by constructive logical connectives. Notationally, its inhabitants were presented to the user as formulae; semantically, the universe corresponded to a predicative universe of computationally relevant terms, and it was contained in Type_0. In the new kernel we can simply represent CProp as a universe variable, constrained to be strictly smaller than the first universe variable and recognized outside the kernel by the pretty-printer in order to print its inhabitants as formulae. Indeed, we can let the user decide whether inhabitants of declared universe variables are meant to represent formulae or types. This was not possible in the old kernel where each occurrence of Type was associated to a fresh universe variable whose URI was not known in advance and whose constraints were dynamically computed by universe inference.

A minor difference between the old and the new kernel is the management of non dependent binders. In the old kernel, the name carried by binders was optional, resulting in additional case distinctions all over the code, with the only benefit in pretty-printing (done outside the kernel). It is the pretty-printer that is now responsible of detecting anonymous binders (for instance to show non dependent products as implications and not as universal quantifications).

Another minor difference, that greatly simplified and reduced the size of the type-checking rules for pattern matching, is having dropped the non dependent form of pattern matching. In the old kernel, mostly by compatibility with objects from the Coq library, we used to have a simplified representation for matches whose branches could be typed without quantifying on the matched term. However, the alternative representation doubled the number of cases and introduced additional code to infer whose representation was in use and to compare semantically equivalent terms with different representations.

We have also purged casts from the new kernel. A cast is a term constructor made of a term and a type, subject to the typing constraint that requires the provided type to be convertible with the type inferred for the provided term. Casts are sometimes added by the user or by tactics that generate terms outside the kernel in order to drive type inference (also done outside the kernel). In the next version of Matita we plan to use a let-in (that carry an explicitly given type for the definiens) to provide the same functionality.

3 Iterators

One of our motivations for writing a new kernel for the Matita prover was to clean up and factorize the code of the old system, taking advantage of the experience acquired since the first writing. According to this philosophy, we decided to make a more systematic use of generic iterators over the CIC datatype. The general idea for their usage is that the programmer is in charge of writing the interesting cases, delegating then to the iterator the management of the remaining ones. This has the drawback of repeating pattern matching on the input term at each function call, one in the user function and one in the iterator, but this is largely compensated by the compactness and readability of the resulting code.

Our first iterator is a fold function, with the following type

```
val fold :
  (NCic.hypothesis → 'k → 'k) → 'k →
  ('k → 'a → NCic.term → 'a) → 'a → NCic.term → 'a
```

The role of the parameters g, k, f, acc and t is the following: void visit the term t accumulating in acc the result of the applications of f to subterms; k is an input parameter for f and should be understood as the information required by f in order to correctly process a subterm. This information may (typically) change when passing binders, and in this case the function g is in charge to update k.

Here is the formal definition:

```
let fold g k f acc = function
 | C.Meta _ → assert false
 | C.Implicit _
 | C.Sort _
 | C.Const _
 | C.Rel _ → acc
 | C.Appl [] | C.Appl [_] → assert false
 | C.Appl l → List.fold_left (f k) acc l
 | C.Prod (n,s,t)
 | C.Lambda (n,s,t) → f (g (n,C.Decl s) k) (f k acc s) t
 | C.LetIn (n,ty,t,bo) → f (g (n,C.Def (t,ty)) k) (f k (f k acc ty) t) bo
 | C.Match (_,oty,t,pl) → List.fold_left (f k) (f k (f k acc oty) t) pl
```

The fold function returns an exception in the case of a Meta, in order to force the use to re-implement this delicate case. Note, the usage of g in the case of binders, to compute a new value of k in terms of the context entry trespassed.

As an example of its usage, let us consider the function does_not_occur context n nn t, in charge of checking that the term t does not contain variables in the interval $(n, nn]$, under the assumption (precondition) that no variable in $(n, nn]$ occurs in the types of the variables that are declared in context and that hereditary occur in t.[5]

```
let does_not_occur ~subst context n nn t =
  let rec aux k _ = function
    | C.Rel m when m > n+k && m <= nn+k → raise DoesOccur
    | C.Rel m when m <= k || m > nn+k → ()
    | C.Rel m →
        (try match List.nth context (m−1−k) with
           | _,C.Def (bo,_) → aux (n−m) () bo
           | _ → ()
         with Failure _ → assert false)
    | C.Meta (_,(_,(C.Irl 0 | C.Ctx []))) → (* closed meta *) ()
    | C.Meta (mno,(s,l)) →
        (try
           let _,_,term,_ = U.lookup_subst mno subst in
           aux (k−s) () (S.subst_meta (0,l) term)
         with U.Subst_not_found _ → match l with
         | C.Irl len → if not (n+k >= s+len || s > nn+k) then raise DoesOccur
         | C.Ctx lc → List.iter (aux (k−s) ()) lc)
    | t → U.fold (fun _ k → k + 1) k aux () t
```

[5] In other words, t remains well typed in the context obtained dropping the declaration of all variables in $(n, nn]$ and all the following declarations and definitions that refer to them.

```
 in
  try aux 0 () t; true
  with DoesOccur → false
```

The interesting case is that of a variable. Here, we have to check that it is outside the interval $(n, nn]$ taken in input; however, since we passed k binders, the interval must be relocated to $(n+k, nn+k]$. In this case, the number k of binders traversed so far is the only information required by f, and the g function has just to increment it.

A second important iterator is map.

```
val map:
 (NCic.hypothesis → 'k → 'k) → 'k →
 ('k → NCic.term → NCic.term) → NCic.term → NCic.term
```

The role of the parameters g, k, f, and t is the following: void visit the term t mapping subterms via the f function. The function f may depend on an additional parameter k which is supposed to be updated by g when passing binders. The definition of map is similar to fold and we omit it; the only interesting peculiarity of our code is that, due to its frequent use, it has been carefully optimized to preserve sharing of all subterms left untouched by the map.

3.1 Lifting and substitution

Typical examples of use of the map functions are the elementary operations for the management of terms with DeBruijn indices: lift and subst.

As it is well known, adopting a syntax based on de Bruijn indices, we do not have to worry about α-conversion, but

1. we need to introduce a lifting operation to relocate the free variables of a term when moving it in a different context;
2. when substituting the first DeBruijn index in a term M for an argument N, we must: (a) lift each replacement copy of N of the suitable quantity (the depth of the variable occurrence), (b) decrement all free variables of M by 1, to take into account the fact that one binder reference has been solved.

The definition the lifting operation is completely standard.

```
let lift_from k n =
 let rec liftaux k = function
    | C.Rel m as t → if m < k then t else C.Rel (m + n)
    | C.Meta (i,(m,l)) as t when k <= m →
       if n = 0 then t else C.Meta (i,(m+n,l))
    | C.Meta (_,(m,C.Irl l)) as t when k > l + m → t
    | C.Meta (i,(m,l)) →
       let lctx = NCicUtils.expand_local_context l in
       C.Meta (i, (m, C.Ctx (HExtlib.sharing_map (liftaux (k−m)) lctx)))
    | C.Implicit _ → (* was the identity *) assert false
    | t → NCicUtils.map (fun _ k → k + 1) k liftaux t
 in
 liftaux k
let lift ?(from=1) n t =
  if n = 0 then t
  else lift_from from n t
```

The substitution operation is so important that, in the old Matita code we had four almost identical versions of it, independently written in different parts of the code, and by different people: the base case, a parallel substitution operation, another copy to instantiate a term with a local context of a metavariable, and still another copy to solve the result of a computation of the reduction machine w.r.t. its environment (see Section 5). All this versions have been unified by the following function.

```
val psubst :
  ?avoid_beta_redexes:bool → ('a → NCic.term) → 'a list → NCic.term → NCic.term
```

In the call psubst ~avoid_beta_redexes: false map_arg args t the role of arguments is the following: t is the term inside which we must substitute the elements of the args list; these elements are not necessarily CIC terms, hence a map_arg map is used to transform them into the right data type (or to just manipulate them, such as lifting them of some additional value); the optional avoid_beta_redexes parameter (set to false by default) automatically reduces any β-redex obtained by instantiation in the spirit of hereditary substitution. The latter provides some speed-up in the conversion check between an expected user provided type (like a type in a binder) and an inferred type that is obtained by substitution for a variable a λ-abstraction. The reason is that the latter contains β-redexes, which can be deeply nested in the term, whereas the former, being user provided, is likely to be β-redex free. Small deep differences in the two terms are sufficient to block certain optimizations performed during conversion (see Section 5.2).

```
let rec psubst ?(avoid_beta_redexes=false) map_arg args =
 let nargs = List.length args in
 let rec substaux k = function
   | C.Rel n as t →
       (match n with
       | n when n >= (k+nargs) →
          if nargs <> 0 then C.Rel (n − nargs) else t
       | n when n < k → t
       | n (* k <= n < k+nargs *) →
        (try lift (k−1) (map_arg (List.nth args (n−k)))
         with Failure _ | Invalid_argument _ → assert false ))
   | C.Meta (i,(m,l)) as t when m >= k + nargs − 1 →
       if nargs <> 0 then C.Meta (i,(m−nargs,l)) else t
   | C.Meta (_,(m,(C.Irl l))) as t when k > l + m → t
   | C.Meta (i,(m,l)) →
       let lctx = NCicUtils.expand_local_context l in
        C.Meta (i,(0,
          C.Ctx (HExtlib.sharing_map (fun x → substaux k (lift m x)) lctx)))
   | C.Implicit _ → assert false (* was identity *)
   | C.Appl (he::tl) as t →
       (* Invariant: no Appl applied to another Appl *)
       let rec avoid he' = function
          | [] → he'
          | arg :: tl ' as args→
              (match he' with
              | C.Appl l → C.Appl (l@args)
              | C.Lambda (_,_,bo) when avoid_beta_redexes →
                  (* map_arg is here \x.x, Obj magic is needed because
                   * we don't have polymorphic recursion w/o records *)
                  avoid (psubst
                    ~avoid_beta_redexes Obj.magic [Obj.magic arg] bo) tl'
              | _ → if he == he' && args == tl then t else C.Appl (he'::args))
       in
       let tl = HExtlib.sharing_map (substaux k) tl in
```

```
      avoid (substaux k he) tl
  | t → NCicUtils.map (fun _ k → k + 1) k substaux t
 in
  substaux 1
```

A couple of interesting instances of psubst are the following functions:

```
let subst ?avoid_beta_redexes arg = psubst ?avoid_beta_redexes (fun x → x)[arg]
(* subst_meta (n, C.Ctx [t_1 ; ... ; t_n]) t                          *)
(*  returns the term [t] where [Rel i] is substituted with [t_i] lifted by n *)
(*  [t_i] is lifted as usual when it crosses an abstraction             *)
(* subst_meta (n, (C.Irl _ | C.Ctx [])) t | → lift n t                   *)
let subst_meta = function
  | m, C.Irl _
  | m, C.Ctx [] → lift m
  | m, C.Ctx l  → psubst (lift m) l
```

4 Library environment

4.1 Library module

```
exception ObjectNotFound of string Lazy.t
val get_obj: NUri.uri → NCic.obj
```

The library module provides access to a mathematical repository by means of a function from names (URIs) to objects. An exception can be raised when a name does not correspond to any object. The new kernel of Matita relies on the existence of an implementation for a library module. However, it does not trust in any way this implementation: after retrieving an object from the library, the object is type-checked before adding it to the environment.

Currently, we provide two implementations of the library. The first one grant access to a distributed XML repository of objects respecting the format of the new kernel. The second one is a wrapper around the library of the old kernel implementation. Every time an old object is requested, we type-check it using the old kernel and we translate it to the new format. We also exploit memoization to avoid translating the same object multiple times.

The translation performs two major transformations on the terms. The first one transforms the Fixpoint terms of the old kernel into Fixpoint objects on the new one. This requires a λ-lifting technique (Johnsson (1985), see also Peyton-Jones (1987)) to hoist local definitions out of their surrounding context into a fixed set of closed global functions. The main difficulty is due to the fact that equality of CIC-terms is structural, while objects are identified by name. Thus, during λ-lifting, we need to map structurally equal local functions to a single object. We achieve this by means of an heuristics based on a cache of already translated recursive functions. The heuristics may fail when a recursive function definition occurs twice in the library, but in one occurrence some of its free variables have been instantiated. For this reason, we have been able to test the new kernel on most of the 35000 old objects coming from the Coq proof assistant library, but not on all of them (the problem is only due to the heuristics used in the automatic translation, not to a loss of expressiveness of the typing system).

The second major transformation consists in fixing universes in the terms. The old kernel, as the one of Coq, used to perform inference of universes: the user wrote terms containing a single unspecified universe Type; the kernel made a fresh instance of each occurrence of the universe and it inferred a graph of constraints whose nodes were the universes and whose edges were inclusion constraints collected during type-checking. The detection of a cycle in the graph corresponds to the discovery of a logical inconsistency and implies the rejection of the object that induces the cycle.

The main drawback of universe inference is that it can become quite expensive, especially in a system with the granularity of Matita, where each object is a separate entity requiring its own universe graph (in the old kernel, up to 30% of the type-checking time was consumed by the management of universes). Moreover, the code for universe inference, though conceptually simple, is big and error prone. It is also unclear why the kernel should be responsible of universe *inference*, since other kind of inferences are performed outside the kernel. Finally, this technique was invented to leave freedom to the user of building libraries with very complex constraint graphs, but it overkills the problem. Indeed, most of the developments to be found in the Coq and Matita libraries only need 2 universes (and the second one seems to be used mainly to perform reflection, see Barthe et al (1995)). This should not be a surprise, since the logical power of using i universes corresponds to the existence of the first i inaccessible cardinals in Zermelo-Fraenkel set theory (Werner (1997)). Finally, some users like the predicativists of the Formal Topology school want to have a tight control and visibility of the universes used, which is exactly what is made transparent by universe inference.

For all these reasons, the new kernel only performs universe constraints checking in the spirit of Courant (2002). Universe inference is performed once and for all during translation of old objects to the new syntax. Moreover, the graph is collapsed as much as possible by identifying nodes that are allowed to be merged. The result on the library of Matita is just a graph with two nodes.

4.2 Environment module

The environment module is in charge of the management of well typed objects. The environment module and the type-checking module (described in Section 6) need to be mutually recursive: when a reference to an object is found during reduction or type-checking, the object must be retrieved from the environment; when the environment is asked for an object in the library that has not been typed yet, it must ask the type-checker to verify it before adding it to the set of verified objects[6]. To solve the mutual recursion the following function is provided, in order to allow the typechecker to properly set the typecheck_obj function.

```
let typecheck_obj,already_set = ref (fun _ → assert false ), ref false
let set_typecheck_obj f =
 if !already_set then assert false
 else begin typecheck_obj := f; already_set := true end
```

The environment maintains two data structures: the set of already typed objects from the library and the set of objects being typed in this moment. The first set is

[6] This top-down management of the environment, where objects are type-checked only on demand, is a peculiarity of Matita that we keep from the old kernel Sacerdoti Coen (2004b). Less document-centric and less library oriented proof assistants like Coq typecheck objects in bottom-up order, starting checking of an object only after that of all its dependencies.

assumed to be quite large, since it must eventually contain all the objects in the current user development and all the objects they refer to. We choose to represent the set using an hash-table from URIs to objects tagged with `WellTyped or to exceptions tagged with `Exn. The choice of a data structure with an efficient retrieval time is fundamental, since retrieval is performed during reduction and during type-checking almost every time a reference is considered. Insertion in the data structure is performed only once for each object that enters the environment, and only after type-checking that is an expensive operation.

We represent the second set as a stack of pairs (URI,object). Every time the environment retrieves an object from the library, it pushes it on top of the stack (that we call the set of frozen objects, or simply the frozen stack). Then it invokes the type-checker and, after completion, it moves the object from the stack to the set of type-checked objects. When a reference is met during type-checking, the environment verifies that the object referred to is not part of the frozen list. If this happens, the object contains a recursive dependency on itself, which is logically unsound.

The main function of the module is the following get_checked_obj.

```
let get_checked_obj u =
 if List.exists (fun (k,_) → NUri.eq u k) !frozen_list
 then
  raise (CircularDependency (lazy (NUri.string_of_uri u)))
 else
  let obj =
   try NUri.UriHash.find cache u
   with
    Not_found →
     let saved_frozen_list = !frozen_list in
     try
      let obj =
       try NCicLibrary.get_obj u
       with
        NCicLibrary.ObjectNotFound m → raise (ObjectNotFound m)
      in
        frozen_list := (u,obj):: saved_frozen_list ;
       !typecheck_obj obj;
        frozen_list := saved_frozen_list ;
       let obj = `WellTyped obj in
       NUri.UriHash.add cache u obj;
       obj
     with
      | Sys.Break as e →
          frozen_list := saved_frozen_list ;
         raise e
      | Propagate (u',_) as e' →
          frozen_list := saved_frozen_list ;
         let exn = `Exn (BadDependency (lazy (NUri.string_of_uri u' ^
            " depends (recursively) on " ^ NUri.string_of_uri u ^
            " which is not well−typed"))) in
         NUri.UriHash.add cache u exn;
         if saved_frozen_list = [] then exn else raise e'
      | e →
          frozen_list := saved_frozen_list ;
         let exn = `Exn e in
         NUri.UriHash.add cache u exn;
         if saved_frozen_list = [] then exn else raise (Propagate (u,e))
  in
   match obj with
```

```
    ‘WellTyped o → o
  | ‘Exn e → raise e
```

First of all, the function looks for an URI in the frozen list to detect cycles in the library, i.e. objects defined in terms of themselves. If this is not the case, the hash-table is used to retrieve the type-checker verdict, in case the objects was met before. If this fails, the object is retrieved from the untrusted library, frozen, type-checked and unfrozen. If type-checking fails, the frozen object is removed from the list, and the failure is recorded in the cache (but in cases of a Sys.Break[7]). Moreover, if the frozen list is not empty, the failure is propagated by means of a local exception propagate, that allows to have a better diagnostic of the dependency error.

```
exception Propagate of NUri.uri ∗ exn
```

This exception, that cannot escape the function, is caught once for each frozen object, which is removed from the stack before remembering in the hash-table that the object is not well-typed since it depends on a bad object.

The get_checked_obj function may raise one of the three following exceptions:

```
exception CircularDependency of string Lazy.t
exception ObjectNotFound of string Lazy.t
exception BadDependency of string Lazy.t
```

The first exception is raised when an object is defined in terms of itself, either directly or by means of a loop in the library. The second exception re-defines the library module exception with the same name. It is raised when the environment is unable to retrieve an object from the library. The third exception is raised when an object depends on another object that is not well typed. No other exception can be raised by functions in this module.

The cache clearly acts as a data structure to memoize previous invocations of the type-checker. In case of excessive memory consumption, rarely used objects could be removed from the cache, at the price of remembering at least their MD5 sum in order to avoid re-typing them when they get referenced again. The MD5 sum is also necessary in case of modifications to the library, to avoid type-checking an object using two alternative versions of a second object referenced twice.

4.3 Universes hierarchy

Our data type for algebraic universes was presented in Section 2.1.2. An algebraic universe is the representation of a universe as a formula over universe variables built by means of successors of universes (to type an existent universe) and maximum of a list of universes (to type products). This is the representation of algebraic universes in EPECC Courant (2002). However, our data structure only allows to represent a small subset of all algebraic universes. In particular, we can not represent the type of terms containing universes which are not all variables. Concretely, this allows us to represent all types inferred for user provided terms, which can only refer to universe variables, but it does not allow to represent types inferred for already inferred types.

[7] This special exception, which is meant to be caught only at the top level, is raised when the user wants to stop the action currently performed by the system, which is taking too long.

This is not a limitation since the typing rules of the calculus never check the type of an inferred type. The main difference w.r.t. EPECC is, for instance, that we do not allow the user to write Type + 1 explicitly: he must give a name to that universe and force it to be bigger than Type. All benefits of EPECC are preserved: the implementation is rather simple (and much simpler than that of EPECC, see Section 4.2) and good error messages can be presented to the user, since he explicitly declares constraints and the kernel just checks that needed constraints are declared. This implies that a universe inconsistency message is displayed only when the user defines a new type variable, while a missing constraint exception can be reported during type checking, and that constraint will act only on universes the user explicitly declared (or on an algebraic expression involving the maximum and the successor).

Since the user is allowed to declare universe variables, we do not provide any predefined universe. However, since Prop is a sort and since the kernel must be able to infer a type for it, we need an algebraic universe with the property of being included in any user declared universe. We simply represent it with type0, which is the neutral element of the maximum of a list of universes, represented by the empty list.

```
let type0 = []
let le_constraints = ref [] (* constraints: (strict, a, b) *)
```

User provided constraints form the constraints graph, which is simply stored in the list of constraints le_constraints. According to our experience, it is really uncommon to have more that two universe variables and thus a more sophisticated structure is not needed. Constraints are triples, the former component is a boolean defining the constraint (less or equal, strictly less) holding between the second and third components (two universe variables).

```
let rec le_path_uri avoid strict a b =
 (not strict && NUri.eq a b) ||
 List.exists
  (fun (strict',x,y) ->
     NUri.eq y b && not (List.exists (NUri.eq x) avoid) &&
       le_path_uri (x::avoid) (strict && not strict') a x
  ) !le_constraints
let leq_path a b = le_path_uri [b] (fst a) (snd a) b
```

Checking for constraints satisfiability amounts at looking for paths between universe variables walking on constraints. The le_path_uri function uses the avoid accumulator to store already visited universes to avoid loops, while strict is set to true when the path we are looking for has to cross at least one strict constraint. The algorithm is based on the following inductive definitions, where $\cdot_1$ represent the unitary length path.

$$\begin{aligned} a \le b &\iff \exists c.\ a \le c \wedge (c <_1 b \vee c \le_1 b) \\ a < b &\iff \exists c.\ (a < c \wedge c \le_1 b) \vee (a \le c \wedge c <_1 b) \end{aligned}$$

```
let universe_leq a b =
  match a, b with
  | a,[(false,b)] -> List.for_all (fun a -> leq_path a b) a
  | _,_ ->
     raise (BadConstraint
       (lazy "trying to check if a universe is less or equal than an inferred universe"))
let universe_eq a b = universe_leq b a && universe_leq a b
let add_constraint strict a b =
  match a,b with
  | [ false ,a2 ],[ false ,b2] →
      if not (le_path_uri  []  strict  a2 b2) then (
        if  le_path_uri  []  (not strict) b2 a2 then
         (raise (BadConstraint (lazy "universe inconsistency")));
        le_constraints  := ( strict ,a2,b2)  ::  ! le_constraints )
  | _ → raise (BadConstraint
          (lazy "trying to add a constraint on an inferred universe"))
```

User defined constraints are processed with add_constraint, and if the new constraint leads to inconsistency (i.e. Typej < Typei && Typei <= Typej) an exception is raised. The two functions universe_eq and universe_leq are used in the conversion check, see Section 5, and during the type checking of inductive definitions, see Section 6.1

5 Reduction and conversion

The most distinguishing feature of CIC is its computational nature. The logic embeds a schematic but powerful functional programming language, hence programs can not only be encoded in CIC and proved correct, but also run within the logic. This opens the doors to many applications, like for example reflecting decision procedures inside the logic, proving them correct and running them to solve open conjectures (see Barthe et al (1995); Boutin (1997); Werner (2008)). This technique is not only elegant, but also proved to be essential in huge formalization efforts, like the proof of the four color theorem (Gonthier (2005)).

The developers of the Coq system (INRIA's original implementation of the Calculus of Inductive Constructions) have recently pushed this idea up to equip (a version of) the kernel with a proper compiler Grégoire (2003) for CIC-terms. This technique, although debatable when considering the additional non trivial amount of code one has to trust, provides great computational power, allowing to internalize and run bigger programs.

Matita, being born as light tool for experimentations, adopts a more high-level and flexible approach to reduction. It uses a generic environment machine abstracted over the reduction strategy, allowing to define, test and compare different strategies with a minimal programming effort.

5.1 Reduction

Reduction is defined for CIC terms that are closed in a given *environment*, described in Section 4.2, and *context*, defined in Section 2. CIC has several one step reduction rules, with different traditional names that we rapidly recall here:

β-reduction This is the classical reduction step of the λ-calculus:

$$\lambda x : T.M\ N \to M\{N/x\}$$

The application of the function $\lambda x : T.M$ to the argument N results in substituting the argument for the formal parameter x in the body M of the function. In terms of DeBruijn indexes, the variable x has index 1, and the substitution operation is that described in Section 3.1.

δ-reduction This is constant unfolding. It comprises both the expansion of a top level constant (i.e. a CIC object built with the Constant constructor) and the unfolding of a local definition already pushed on the context (i.e. a DeBruijn index pointing to a Def context entry).

ι-reduction It is a pattern matching step involving terms of the form

```
Match (_,_,kj ,[p1;...;pm])
```

To trigger the reduction, the recursive argument kj must be the constructor of an inductive type, possibly applied to some arguments, some instantiating the left arguments followed by some parameters args. A pj branch corresponding to such constructor is then selected in the list [p1;...;pm] and the resulting reduct is the application of pj to args. A justification for dropping the left parameters is given in section 6 when discussing the type checking of pattern matching.

μ-unfolding This reduction step unfolds the body of a recursive function and is triggered only when its recursive argument is a constructor. This is usually considered a constrained δ-reduction step.

ν-unfolding This step unfolds the body of a corecursive function, whenever it occurs as the matching argument of a ι-rule.

The transitive and reflexive closure of these reduction steps is called reduction (and thus t is a reduct of itself) and its symmetric closure is the so called *conversion* equivalence relation. This relation allows the calculus to identify types, and especially dependent types (types/proposition containing occurrences of terms) up to reduction. For instance, with typical arithmetical definitions, the two expressions $x = 2$ and $x = 1 + 1$ would not only be logically equivalent: they would be convertible (i.e. identical).

Since the reduction relation is strongly normalizing on well typed terms, conversion is trivially decidable by reducing both terms to their normal form and syntactically comparing them. Anyway computation is expensive, and it is important to avoid unnecessary computations to obtain good type checking performances. This suggests that, to check if two terms fall in the same equivalence class, a controlled parallel reduction of both may lead to success before reaching the normal form. For example, having defined times in terms of plus, $100 * 100$ and $100 + 99 * 100$ can be judged to be equal only performing few reduction steps on the former term.

The literature suggests (see Crégut (1990); Asperti (1992); Crégut (2007)) that (weak) head normal forms provides good intermediate points to break normalization. We further refine this idea by exploiting the *height* of constants (Sacerdoti Coen (2007)) (statically computed at the moment of their definition): let E be an environment, the height $h(c)$ of a constant c such that $E(c) = t$ is defined by $h(c) = 1 + \max_{c' \in t} h(c')$ where $c' \in t$ if c' occurs in t. If no constant occurs in c the height of c is 0.

An empirical observation confirms that a term Appl [c1;t1;...;tn] whose δ-reduct further reduces to Appl [c2;s1;...;sm] is most of the time[8] characterized by the property $h(c1) > h(c2)$. We can exploit heights for controlled δ-reduction, that will came into hand during conversion.

For instance, consider the two convertible terms $100 * 100$ and $100 + 99 * 100$ that both have a δ-redex in head position. The former term reduces to the latter and since product is defined in terms of addition, we have $h(*) > h(+)$. Thus it is a good idea

[8] This may e.g. fail with highly polymorphic functions that, when applied, can reduce to terms of any type. An example is the polymorphic functions that computes the head of a list.

to perform head reduction on the first term unless a δ-redex in head position of height less or equal than $h(+)$ is found. If the reduced term has height $h(+)$, we can hope that its head is an addition and that the two terms are now convertible. This is indeed the case in our not so artificial example.

We believe that the computation of the weak head normal form up to the unfolding of constants having a given height rarely performs useless reduction steps (not necessary to check the conversion of two terms).

In the following we first describe the machinery to compute the (δ-reduction constrained) weak head normal form of a term, and then we present the conversion algorithm.

5.1.1 Reduction Strategies

To allow the study of different reduction strategies, we used a generic environment abstract machine (called GKAM in Sacerdoti Coen (2004a)), that allows to implement almost every reduction strategy properly instantiating an OCaml functor (higher order module).

A typical environment machine like Krivine's abstract machine (KAM) is made of an environment, a code and a stack. The code is the term to be reduced. Its free variables are assigned values by the environment, that plays the role of an explicit simultaneous substitution. When an application is processed, its argument is moved to the stack together with a pointer to the environment, forming a closure. When a λ-abstraction (part of a β-redex) is processed, the top of the stack is simply moved to the top of the environment. Finally, when a de Bruijn index n is processed, the n-th component of the stack is fetched and becomes the new term to be processed (together with the new environment).

The *GKAM* generalizes the *KAM* in three ways:

1. Reduction of the argument is allowed when it is moved to the stack, to the environment or in code position. In this way, a large variety of reduction strategies can be implemented.
2. The data structures of the elements of the stack and of the environment become parameters. This helps in implementing strategies such as call-by-need. As a consequence, we also introduce as parameters read-back functions from stack and environment items to terms. The last two functions will not be used directly by the reduction machine during its computation, but to reconstruct the CIC-term corresponding to a configuration (i.e. when the reduction stops). This is performed by the following code

```
let rec unwind (k,e,t,s) =
  let t =
    if k = 0 then t
    else
      NCicSubstitution.psubst ~avoid_beta_redexes:true
        (RS.from_env_for_unwind ~unwind) e t
  in
  if s = [] then t
  else C.Appl(t::(RS.from_stack_list_for_unwind ~unwind s))
```

3. The machine is extended to CIC.

The OCaml module system allows to define functors, that are functions from modules to modules. This mechanism allows to define higher order modules, that can be instantiated with every module that satisfies the input signature of the functor.

Every reduction strategy, defines the types of objects living in the stack and in the environment, but their type is not known in advance, thus the generic reduction machine will be only able to manipulate its stack and environment by means of functions provided by the reduction strategy. The signature a reduction strategy module has to inhabit follows.

```
module type Strategy = sig
  type stack_term
  type env_term
  type config = int * env_term list * C.term * stack_term list
  val to_env :
   reduce: (config → config * bool) → unwind: (config → C.term) →
     config → env_term
  val from_stack : stack_term → config
  val from_stack_list_for_unwind :
   unwind: (config → C.term) → stack_term list → C.term list
  val from_env : env_term → config
  val from_env_for_unwind :
   unwind: (config → C.term) → env_term → C.term
  val stack_to_env :
   reduce: (config → config * bool) → unwind: (config → C.term) →
     stack_term → env_term
  val compute_to_env :
   reduce: (config → config * bool) → unwind: (config → C.term) →
     int → env_term list → C.term → env_term
  val compute_to_stack :
   reduce: (config → config * bool) → unwind: (config → C.term) →
     config → stack_term
```

The initial configuration of the reduction machine for the reduction of a term `t` is `(0, [], t, [])`. The first integer is the length of the environment and is used only to speed up some computations.

Given the `unwind` function the following requirements are necessary for the soundness of the reduction strategy (but cannot be expressed by the OCaml type system):

1. $\forall t, e.$ from_stack_list_for_unwind (RS.to_stack t (k,e,t,[])) is a reduct of unwind (k,e,t,[])
2. $\forall s.$ from_env_for_unwind (RS.to_env s) is a reduct of from_stack_list_for_unwind s
3. $\forall e.$ unwind (RS.from_env e) is a reduct of from_env_for_unwind e

The recursive function reduce takes in input a constant height `delta`, a context and a configuration. The context is a CIC context, orthogonal to the one carried in the reduction machine configuration, and fixed for the whole reduction process. The `subst` argument is an optional metavariable substitution. It returns a reduced machine, along with a boolean which is true if the output machine is in weak head normal form (reduction could stop earlier because of the `delta` limit).

```
module Reduction(RS : Strategy) = struct
  let rec reduce ~delta ?(subst = []) context : config → config * bool =
   let rec aux = function
     | k, e, C.Rel n, s when n <= k →
        let k',e',t',s' = RS.from_env (list_nth e (n−1)) in
        aux (k',e',t',s'@s)
  | k, _, C.Rel n, s as config (* when n > k *) →
     let x= try Some (List.nth context (n − 1 − k)) with Failure _ → None in
      (match x with
      | Some(_,C.Def(x,_)) → aux (0,[],NCicSubstitution.lift (n − k) x,s)
      | _ → config, true)
```

The auxiliary function defined inside reduce proceeds by pattern matching on the head term. The first two cases are for DeBruijn indexes. If the index points to a term stored in the machine environment, RS.from_env is called on the n-th element of the machine environment to obtain a new machine configuration whose stack is prepended to the current one. If the length of the environment is less then n the variable has not been substituted by reduction, but it may be bound to a local definition in the context. In that case the definiens is retrieved, and is expanded lifted in the current context (moved under n-k binders). Otherwise the term is in head normal form and the reduction machine stops.

```
  | (k, e, C.Meta (n,l), s) as config →
     (try
        let _,_, term,_ = NCicUtils.lookup_subst n subst in
        aux (k, e, NCicSubstitution.subst_meta l term,s)
      with NCicUtils.Subst_not_found _ → config, true)
```

Metavariables are considered in normal form, unless the input argument subst contains an entry for them. If that entry exists, the substituted term is instantiated in the actual context using the explicit substitution l, and is recursively reduced.

```
  | (_, _, C.Implicit _, _) → assert false
  | (_, _, C.Sort _, _)
  | (_, _, C.Prod _, _)
  | (_, _, C.Lambda _, []) as config → config, true
```

Implicit arguments should never reach the kernel, thus this case is not considered. Sorts are in normal form by definition, thus no step is performed in these cases. The same holds for products and unapplied λ-abstractions (i.e. when the stack is empty).

```
  | (k, e, C.Lambda (_,_,t), p::s) →
      aux (k+1, (RS.stack_to_env ~reduce:aux ~unwind p)::e, t,s)
  | (k, e, C.LetIn (_,_,m,t), s) →
     let m' = RS.compute_to_env ~reduce:aux ~unwind k e m in
      aux (k+1, m'::e, t, s)
```

When a λ-abstraction is encountered with a non empty stack, the first item of the stack is moved to the environment with the RS.stack_to_env function, and the body of the abstraction is recursively processed. When a local definition is encountered, its definendum is moved directly to the environment (thanks to RS.compute_to_env) leaving the stack untouched.

```
  | (_, _, C.Appl ([]|[ _ ]), _) → assert false
  | (k, e, C.Appl (he::tl), s) →
     let tl' =
      List.map (fun t→ RS.compute_to_stack ~reduce:aux ~unwind (k,e,t,[])) tl
     in
      aux (k, e, he, tl' @ s)
```

When an application is encountered, all arguments are pushed on the stack with RS.compute_to_stack and the head is recursively processed. Ill-formed applications are rejected.

```
  | (_, _, C.Const
        (Ref.Ref (_,Ref.Def height) as refer ), s) as config →
      if delta >= height then
        config, false
      else
        let _,_,body,_,_,_ = NCicEnvironment.get_checked_def refer in
        aux (0, [], body, s)
  | (_, _, C.Const (Ref.Ref (_,
      (Ref.Decl|Ref.Ind _|Ref.Con _|Ref.CoFix _))), _) as config →
        config, true
```

A constant can represent very different objects, thus a deep pattern matching is used to exploit all the information carried by the reference. If the reference points to a declaration (i.e. an axiom) reduction stops. If the constant has a body and its height is bigger than the delta parameter it is unfolded and its body is recursively reduced. Note that the choice of storing the information regarding the height of the context not only in the environment but also in the reference allows us to ask the environment the body of the object only when it is strictly necessary, avoiding a possibly expensive lookup.

```
  | (_, _, (C.Const (Ref.Ref
        (_,Ref.Fix (fixno,recindex,height)) as refer) as head),s) as config →
     (match
        try Some (RS.from_stack (List.nth s recindex))
        with Failure _ → None
     with
     | None → config, true
     | Some recparam →
        let fixes,_,_ = NCicEnvironment.get_checked_fixes_or_cofixes refer in
        match reduce ~delta:0 ~subst context recparam with
        | (_,_,C.Const (Ref.Ref (_,Ref.Con _)), _) as c, _
           when delta >= height →
             let new_s =
               replace recindex s (RS.compute_to_stack ~reduce:aux ~unwind c)
             in
             (0, [], head, new_s), false
        | (_,_,C.Const (Ref.Ref (_,Ref.Con _)), _) as c, _ →
             let new_s =
               replace recindex s (RS.compute_to_stack ~reduce:aux ~unwind c)
             in
             let _,_,_,_,body = List.nth fixes fixno in
             aux (0, [], body, new_s)
        | _ → config, true)
```

The most interesting case is when the reference identifies a block of mutual recursive definitions. We can again efficiently check if the height of the constant is greater then the desired one, and if it is the case, we can check if the recursive argument of the function (whose index is stored in the reference too) is a constructor. Only in that case the body of the recursive function pointed by the reference is retrieved from the environment. The body of the recursive function is reduced in a context new_s where its recursive argument (already reduced in weak head normal form to expose a constructor) has been replaced with the reduct, to avoid computing it again.

```
  | (k, e, C.Match (_,_,term,pl),s) as config →
     let decofix = function
       | (_,_,C.Const(Ref.Ref(_,Ref.CoFix c)as refer),s)→
          let cofixes,_,_ = NCicEnvironment.get_checked_fixes_or_cofixes refer in
          let _,_,_,_,body = List.nth cofixes c in
```

```
              let c,_ = reduce ~delta:0 ~subst context  (0,[], body,s) in
              c
           | config → config
         in
         let match_head = k,e,term,[] in
         let reduced,_ = reduce ~delta:0 ~subst context match_head in
         (match decofix reduced with
         | (_, _, C.Const (Ref.Ref (_,Ref.Con (_,j,_ ))),[])  →
             aux (k, e, List.nth pl  (j−1), s)
         | (_, _, C.Const (Ref.Ref (_,Ref.Con (_,j,lno ))),  s')→
           let _,params = HExtlib.split_nth lno s' in
           aux (k, e, List.nth pl  (j−1), params@s)
         | _ → config, true)
  in
    aux
end
```

The reduction of a pattern matching is triggered if the matched term is a constructor or if it is a coinductive function. In the second case the body of the coinductive function is reduced and the guarded_by_constructors check (see Section 6.3) performed by the type checker ensures it will expose a constructor in finite time. The arguments of the constructor are separated from the *left parameters* of its inductive type (see Section 2) with a call of split_nth , and the j-th pattern matching case is reduced pushing on the stack these parameters.

5.1.2 The implemented reduction strategy

As we already highlighted, to achieve good type checking performances it is usually better to avoid unnecessary reduction, thus a call-by-need strategy seems to fit our requirements. When the system is used interactively, and the user asks the system to reduce a term, such a strategy is however too aggressive. Consider for example the following term

```
let primes := sieve 100 in
match primes with
[ nil  ⇒ absurd_from_gt_0 100
| cons _ _  ⇒ pair (last primes) (length primes)]
```

In order to perform ι-reduction, primes is computed, and the second branch is taken. A by-need reduction strategy remembers the weak head normal form of primes, which is the list of prime numbers less than 100 if sieve is implemented using an accumulator. The reduced term is unmanageable: pair (last [2 ; ... ; 97]) (length [2 ; ... ; 97]). What the user probably expects here is head linear reduct Danos and Regnier (2003):

let primes := sieve 100 in pair (last primes) (length primes)

The strategy implemented in Matita carries around both the reduct of a term (to avoid reducing twice the same term in the spirit of the by-need strategy) and the term before reduction, used in the unwind process.

```
module CallByValueByNameForUnwind' = struct
  type config = int ∗ env_term list ∗ C.term ∗ stack_term list
  and stack_term = config lazy_t ∗ C.term lazy_t (∗ cbv, cbn ∗)
  and env_term = config lazy_t ∗ C.term lazy_t (∗ cbv, cbn ∗)
  let to_env ~reduce ~unwind c = lazy (fst (reduce c)),lazy (unwind c)
  let from_stack (c,_) = Lazy.force c
let from_stack_list_for_unwind ~unwind:_ l =
 List.map (function (_,c) → Lazy.force c) l
let from_env (c,_) = Lazy.force c
let from_env_for_unwind ~unwind:_ (_,c) = Lazy.force c
let stack_to_env ~reduce:_ ~unwind:_ config = config
let compute_to_env ~reduce ~unwind k e t =
 lazy (fst (reduce (k,e,t ,[]))), lazy (unwind (k,e,t ,[]))
let compute_to_stack ~reduce ~unwind config =
 lazy (fst (reduce config )), lazy (unwind config)
```

The stack is a list of pairs: the first component is a closure that may be put in head normal form according to the by-name strategy, while the second is the unreduced term obtained unwinding lazily the pristine closure.

The same holds for the environment. The functions used by unwind always use the second component, while on the first component the compute_to function perform reduction. All computation is delayed using the **lazy** OCaml keyword, and is forced only when a configuration is pulled from the stack/environment.

5.2 Conversion

Our statistics show that, when type-checking real world terms, most of the terms checked for conversion are actually identical. This suggests a simple but very effective strategy: two terms are recursively compared, without triggering reduction. If the comparison fails, they are both reduced to weak head normal form and compared again. In our implementation of the conversion check, this check is done in the alpha_eq function.

The first control === done by alpha_eq checks physical equality (identity of memory location) followed by structural equality (both provided by the OCaml runtime). This check is weaker then α-equivalence, even if we are using DeBruijn indexes. The cause is the fact that we store user-provided names inside binders to be able to show the user the same names he gave in input. Anyway the vast majority of calls to are_convertible ends with that comparison.

```
let are_convertible ?(subst=[]) get_exact_relev =
 let rec aux test_eq_only context t1 t2 =
   let rec alpha_eq test_eq_only t1 t2 =
     if t1 === t2 then
       true
     else
       match (t1,t2) with
       | (C.Sort (C.Type a), C.Sort (C.Type b)) when not test_eq_only →
           NCicEnvironment.universe_leq a b
       | (C.Sort (C.Type a), C.Sort (C.Type b)) →
           NCicEnvironment.universe_eq a b
       | (C.Sort C.Prop,C.Sort (C.Type _)) → (not test_eq_only)
       | (C.Sort C.Prop, C.Sort C.Prop) → true
```

If the two terms are not structurally equal, terms are compared recursively. In the tradition of the Extended Calculus of Constructions Luo (1990), the conversion relation is weakened to an order relation called *cumulativity* that takes into account the inclusion of lower universes into higher ones. The test_eq_only parameter is true when comparing the two sources of two products or the arguments of two applications: and in that case the universe of propositions is not considered to be included in any data type universe, and universes are compared by co-inclusion see 4.3.

```
| (C.Prod (name1,s1,t1), C.Prod(_,s2,t2)) →
    aux true context s1 s2 &&
    aux test_eq_only ((name1, C.Decl s1)::context) t1 t2
| (C.Lambda (name1,s1,t1), C.Lambda(_,s2,t2)) →
   aux true context s1 s2 &&
   aux test_eq_only ((name1, C.Decl s1)::context) t1 t2
| (C.LetIn (name1,ty1,s1,t1), C.LetIn(_,ty2,s2,t2)) →
   aux test_eq_only context ty1 ty2 &&
   aux test_eq_only context s1 s2 &&
   aux test_eq_only ((name1, C.Def (s1,ty1))::context) t1 t2
```

Binders are crossed without comparing the name of the abstracted variable and a proper context, needed to eventually trigger reduction, is built.

```
| (C.Meta (n1,(s1, C.Irl i1 )), C.Meta (n2,(s2, C.Irl i2 )))
   when n1 = n2 && s1 = s2 → true
| (C.Meta (n1,(s1, l1 )), C.Meta (n2,(s2, l2 ))) when n1 = n2 &&
   let l1 = NCicUtils.expand_local_context l1 in
   let l2 = NCicUtils.expand_local_context l2 in
   (try List. for_all2
     (fun t1 t2 → aux test_eq_only context
       (NCicSubstitution. lift s1 t1)
       (NCicSubstitution. lift s2 t2))
     l1 l2
   with Invalid_argument _ → assert false) → true
```

The comparison of metavariables is tricky. Since conversion does not perform unification, two metavariables can be considered equal only if they are two occurrences of the same metavariable (the check n1=n2) or if they are instantiated to convertible terms (see next code block). If they are both equipped with an identity local context (lifted by the same amount) they are considered equal without comparing the length of the local context: the equality of i1 and i2 is ensured by the type checking algorithm. If at least one local context is not the identity, both l1 and l2 are expanded into an explicit list of terms (an identity list, in case of Irl) and their elements are pairwise recursively compared after being properly lifted.

```
| C.Meta (n1,l1), _ →
   (try
      let _,_,term,_ = NCicUtils.lookup_subst n1 subst in
      let term = NCicSubstitution.subst_meta l1 term in
       aux test_eq_only context term t2
    with NCicUtils.Subst_not_found _ → false)
| _, C.Meta (n2,l2) →
   (try
      let _,_,term,_ = NCicUtils.lookup_subst n2 subst in
      let term = NCicSubstitution.subst_meta l2 term in
       aux test_eq_only context t1 term
    with NCicUtils.Subst_not_found _ → false)
```

Again, since unification is not performed, the only chance to unify a metavariable with another term is to have a substituted term for that metavariable that is convertible with the other term. This is in general not necessary since reduction to normal form performs the substitution. Anyway, according to our experience, anticipating the application of the substitution leads to success early.

```
| (C.Appl ((C.Const r1) as hd1::tl1), C.Appl (C.Const r2::tl2))
    when (Ref.eq r1 r2 &&
            List.length (NCicEnvironment.get_relevance r1) >= List.length tl1) →
        let relevance = NCicEnvironment.get_relevance r1 in
        let relevance = match r1 with
            | Ref.Ref (_,Ref.Con (_,_,lno)) →
                let _,relevance = HExtlib.split_nth lno relevance in
                  HExtlib.mk_list false lno @ relevance
            | _ → relevance
        in
        (try
            HExtlib. list_forall_default3_var
             (fun t1 t2 b → not b || aux true context t1 t2 )
             tl1 tl2 true relevance
          with Invalid_argument _ → false
             | HExtlib.FailureAt fail →
               let relevance = get_exact_relev ~subst context hd1 tl1 in
               let _,relevance = HExtlib.split_nth fail relevance in
               let b,relevance = (match relevance with
                   | [] → assert false
                   | b :: tl → b,tl) in
               if (not b) then
                 let _, tl1 = HExtlib.split_nth ( fail +1) tl1 in
                 let _, tl2 = HExtlib.split_nth ( fail +1) tl2 in
                   try
                      HExtlib. list_forall_default3
                      (fun t1 t2 b → not b || aux test_eq_only context t1 t2)
                      tl1 tl2 true relevance
                   with Invalid_argument _ → false
               else false )
    | (C.Appl (hd1::tl1),  C.Appl (hd2::tl2)) →
         aux test_eq_only context hd1 hd2 &&
         let relevance = get_exact_relev ~subst context hd1 tl1 in
          (try
            HExtlib. list_forall_default3
             (fun t1 t2 b → not b || aux true context t1 t2)
             tl1 tl2 true relevance
          with Invalid_argument _ → false)
```

In the case of applications, the check should consist of a recursive comparison on subterms. However, as we already mentioned, we implement a proof-irrelevance-aware application comparison: irrelevant arguments do not need to be compared since ideally they do not contribute to the head normal form of closed terms. The type checker will ensure that the irrelevant argument will never be put in head position independently of the context the application will be put in.

How do we compute relevance? We distinguish a static and dynamic inference. The first one is used when comparing two applications whose heads are constants: it depends on the type of the constant alone and is computed when defining that constant. The static inference results in a list of booleans which is best understood as a cache, indicating which argument positions are statically known as irrelevant. The length of the list is equal to the number of products with which the type of the constant begins; it is correct (meaning that positions denoted as irrelevant must always be filled by irrelevant arguments), but possibly incomplete because of two reasons:

- with dependent types the arity of a function is not fixed and can depend on the actual value of arguments; it is thus impossible to completely specify its relevance with a fixed length list;
- the actual sort of an argument can depend on the preceding arguments in subtle ways; suppose for instance you have a constant

```
c: ∀ b. (match f b with [true ⇒nat | false ⇒True]) →...
```

for some function f ranging on booleans: the second arguments of c must be either of type nat (whose sort is Type) or of type True (whose sort is Prop), i.e. relevant or irrelevant, depending on the value of f b which is statically unknown.

The statically inferred relevance list can be obtained by means of a call to the get_relevance function of the environment.

The dynamic inference is used for every application, including applications whose head is a constant, when the relevance list produced by the static inference is not sufficiently accurate. It is performed by the get_exact_relev function defined in the type checker and received as a parameter by are_convertible. This function takes as parameters the head of the application and its arguments and, by substituting the arguments in the type of the head, computes the types of all the arguments and, subsequently, their sorts[9]; it returns a list whose length is equal to the number of arguments and whose elements are false or true depending on the corresponding sort being Prop or not.

When comparing two applications whose heads are both constants, we first check that they are indeed the same constant; then we retrieve from the environment the statically inferred relevance list and by means of list_forall_default3_var we compare in parallel the two list of arguments and the relevance, possibly extending the last with a default value (true) when needed. If the i-th argument is not relevant no check is performed, otherwise the arguments are recursively compared. If the check fails at argument n, a FailureAt n exception is raised: in this case, the check is resumed at argument n with a dynamic relevance inference.

When comparing applications whose heads are not (both) constants, the check is performed only according to the dynamic relevance inference, as we have no relevance cache to exploit.

```
      | (C.Match (Ref.Ref (_,Ref.Ind (_,tyno,_)) as ref1 ,outtype1,term1,pl1),
         C.Match (ref2,outtype2,term2,pl2)) →
          let _,_, itl ,_,_ = NCicEnvironment.get_checked_indtys ref1 in
          let _,_,ty,_ = List.nth itl tyno in
          let rec remove_prods ~subst context ty =
            let ty = whd ~subst context ty in
            match ty with
            | C.Sort _ → ty
            | C.Prod (name,so,ta) → remove_prods ~subst ((name,(C.Decl so))::context) ta
            | _ → assert false
          in
          let is_prop =
            match remove_prods ~subst [] ty with
            | C.Sort C.Prop → true
            | _ → false
          in
          Ref.eq ref1 ref2 &&
          aux test_eq_only context outtype1 outtype2 &&
          (is_prop || aux test_eq_only context term1 term2) &&
          (try List. for_all2 (aux test_eq_only context) pl1 pl2
           with Invalid_argument _ → false)
```

9 Computing the sorts of the arguments of an application by substitution on the type of the head has proven to be on average more efficient than directly computing the sorts of the arguments.

```
      | (C.Implicit _, _) | (_, C.Implicit _) → assert false
      | (_,_) → false
  in
```

In the case of pattern matching, the two terms are compared recursively: however, should the sort of the term being matched be Prop, that term will not be included in the recursive comparison.

Implicit arguments should not reach the kernel. Every other pair fails the α-conversion check.

In case the two terms fail the simple check implemented by alpha_eq we need to feed them to our reduction machine. When the machine stops, we could unwind the resulting state and repeat the α-equivalence test, but this is in general expensive because reduction can exponentially increase terms size. We thus delay as much as possible unfolding, comparing GKAM statuses.

```
   if alpha_eq test_eq_only t1 t2 then true
   else convert_machines test_eq_only (put_in_whd (0,[], t1 ,[]) (0,[], t2 ,[]))
 in
  aux false
```

The small_delta_step function is called on machines whose heads are not convertible. At least one of the machines must not be in weak head normal form. If one machine is in normal form, reduction is performed on the other. If neither machine is in normal form, reduction is performed on the machine whose head (of greater height) is more likely becoming convertible to the other head (of smaller height). When no machine is candidate for reduction (both have the same height) we reduce both to smaller heights.

```
     let small_delta_step
       ((_,_,t1,_ as m1), norm1 as x1) ((_,_,t2,_ as m2), norm2 as x2)
     =
       assert (not (norm1 && norm2));
       if norm1 then
         x1, R.reduce ~delta:(height_of t2 −1) ~subst context m2
       else if norm2 then
         R.reduce ~delta:(height_of t1 −1) ~subst context m1, x2
       else
        let h1 = height_of t1 in
        let h2 = height_of t2 in
        let delta = if h1 = h2 then max 0 (h1 −1) else min h1 h2 in
        R.reduce ~delta ~subst context m1,
        R.reduce ~delta ~subst context m2
```

The conversion check for machine statuses succeeds in the two following cases.

```
     let rec convert_machines test_eq_only
       ((k1,e1,t1,s1),norm1 as m1),((k2,e2,t2,s2), norm2 as m2) =
       (alpha_eq test_eq_only (R.unwind (k1,e1,t1 ,[])) (R.unwind (k2,e2,t2 ,[])) &&
        let relevance = match t1 with Const r → NE.get_relevance r | _ → [] in
        try
         HExtlib. list_forall_default3
           (fun t1 t2 b →
             not b ||
             let t1 = RS.from_stack t1 in
             let t2 = RS.from_stack t2 in
             convert_machines true (put_in_whd t1 t2)) s1 s2 true relevance
        with Invalid_argument _ → false) ||
```

The first case is when the heads of the machines are α-convertible and when all arguments (the stack components) are pairwise convertible. This test is weaker than full conversion, for example when the heads are constant functions. Note that arguments are again compared up to relevance (as it was done in the alpha_eq check).

```
    (not (norm1 && norm2) && convert_machines test_eq_only (small_delta_step m1 m2))
  in
  convert_machines test_eq_only (put_in_whd (0,[], t1 ,[]) (0,[], t2 ,[]))
```

If the first check fails and at least one of the machines is not in weak head normal form, we retry the machine conversion after a reduction step performed by small_delta_step. This step will be repeated until the terms are in weak head normal form, and thus are convertible if and only if the first case of convert_matchines succeeds.

6 Typechecking

The CicTypeChecker module implements the type-checking judgements for CIC terms and objects. Terms are type-checked in a context, a metasenv and a substitution. As explained in Section 2, the context maps the free variables of the term to their local declaration or definition. Metavariables occurring in the term are either declared in the metasenv, or they are defined in the subst. We maintain the invariant that contexts, metasenvs and substitutions passed around in the implementation of the type-checker have already been type-checked. With this invariant, we avoid type-checking again a definiens every time we retrieve it from a context or a substitution. When the programmer invokes the typeof function from outside the kernel, he needs to remember to respect the invariant since the typeof function does not assert it at the beginning. On the other hand, when typecheck_obj is invoked, the invariant is verified for the metasenv and substitution that are found in the object.

Each judgement is implemented by a function that raises an exception when fed with an ill-typed term. The type-checking function on terms also returns the inferred type. The interface of the module is the following one.

```
exception TypeCheckerFailure of string Lazy.t
exception AssertFailure of string Lazy.t
val typeof:
  subst:NCic.substitution → metasenv:NCic.metasenv →
    NCic.context → NCic.term → NCic.term
val typecheck_obj : NCic.obj → unit
```

We analyze first the type-checking functions for contexts, metavariables and substitutions. These functions are not exposed in the interface, but they are used internally by typecheck_obj.

```
let typecheck_context ~metasenv ~subst context =
 ignore
  (List. fold_right
   (fun d context →
     begin
      match d with
         _,C.Decl t → ignore (typeof ~metasenv ~subst:[] context t)
       | name,C.Def (te,ty) →
         ignore (typeof ~metasenv ~subst:[] context ty);
         let ty' = typeof ~metasenv ~subst:[] context te in
```

```
          if not (R.are_convertible ~subst context ty' ty) then
           raise (AssertFailure (lazy (Printf.sprintf (
            "the type of the definiens for %s in the context is not "^^
            "convertible with the declared one.\n"^^
            "inferred type:\n%s\nexpected type:\n%s")
            name (PP.ppterm ~subst ~metasenv ~context ty')
            (PP.ppterm ~subst ~metasenv ~context ty))))
     end;
     d::context
   ) context [])
```

A context is well-typed if the terms in each context entry are closed in the part of the context that follows (in list order). The terms may also contain metavariables that are either declared in the metasenv passed in input, or that are defined in the subst passed in input. As a precondition, we assume the metasenv and the substitution to be well-typed. If the context entry is a definition, we must check that the declared and inferred type for the definiens are convertible. Since reduction and conversion may diverge on terms that are not well-typed, we must take care of always type-checking a term before feeding it to the NCicReduction module.

```
let typecheck_metasenv metasenv =
 ignore
  (List.fold_left
    (fun metasenv (i,(_,context,ty) as conj) →
      if List.mem_assoc i metasenv then
       raise (TypeCheckerFailure (lazy ("duplicate meta " ^ string_of_int i ^
        " in metasenv")));
      typecheck_context ~metasenv ~subst:[] context;
      ignore (typeof ~metasenv ~subst:[] context ty);
      metasenv @ [conj]
    ) [] metasenv)
```

A well-typed metasenv is an ordered list of declarations for metavariables. In order to avoid cycles that do not make sense, the terms in each metavariable declaration may only contains metavariables declared in the part of metasenv that precedes (in list order). Note that it is not possible to have metavariables used in the metasenv but defined in a substitution: an invariant of Matita is that substitutions generated outside the kernel are fully applied to metasenvs before passing objects to the kernel. This allows to easily verify the lack of cyclic dependencies between a variable in the metasenv and a variable in the substitution. A metavariable declaration is given by its name, context and type. Since the type may contain free variables declared in the context, we must remember to type-check the context first, and to pass it to typeof before checking the type.

```
let typecheck_subst ~metasenv subst =
 ignore
  (List.fold_left
    (fun subst (i,(_,context,ty,bo) as conj) →
      if List.mem_assoc i subst then
       raise (AssertFailure (lazy ("duplicate meta " ^ string_of_int i ^
        " in substitution")));
      if List.mem_assoc i metasenv then
       raise (AssertFailure (lazy ("meta " ^ string_of_int i ^
        " is both in the metasenv and in the substitution")));
      typecheck_context ~metasenv ~subst context;
      ignore (typeof ~metasenv ~subst context ty);
      let ty' = typeof ~metasenv ~subst context bo in
       if not (R.are_convertible ~subst context ty' ty) then
        raise (AssertFailure (lazy (Printf.sprintf (
         "the type of the definiens for %d in the substitution is not "^^
         "convertible with the declared one.\n"^^
         "inferred type:\n%s\nexpected type:\n%s")
         i
         (PP.ppterm ~subst ~metasenv ~context ty')
         (PP.ppterm ~subst ~metasenv ~context ty))));
      subst @ [conj]
   ) [] subst)
```

A well-typed substitution is an ordered list of definitions for metavariables. In order to avoid cycles that are logically inconsistent and may lead to divergence, the terms in each substitution definitions may only contains metavariables declared in the metasenv or in the part of the substitution that precedes (in list order). As we did for contexts, we must remember to verify the consistency of the inferred and declared types for the definiens. As we did for metavariables, we must check the context first since the definiens and its type may contain variables declared in the context.

```
let typecheck_obj (uri,height,metasenv,subst,kind) =
 typecheck_metasenv metasenv;
 typecheck_subst ~metasenv subst;
```

The metasenv and substitution must be checked first and in this order before checking the terms in the objects.

```
 match kind with
   | C.Constant (_,_,Some te,ty,_) →
      let _ = typeof ~subst ~metasenv [] ty in
      let ty_te = typeof ~subst ~metasenv [] te in
      if not (R.are_convertible ~subst [] ty_te ty) then
       raise (TypeCheckerFailure (lazy (Printf.sprintf (
        "the type of the body is not convertible with the declared one.\n"^^
        "inferred type:\n%s\nexpected type:\n%s")
        (PP.ppterm ~subst ~metasenv ~context:[] ty_te)
        (PP.ppterm ~subst ~metasenv ~context:[] ty))))
```

A definition or theorem is well-typed when its type and body (definiens) are. Moreover, the type inferred for the body must be convertible with the user declared type.

```
   | C.Constant (_,_,None,ty,_) → ignore (typeof ~subst ~metasenv [] ty)
   | C.Inductive (_, leftno, tyl, _) →
       check_mutual_inductive_defs uri ~metasenv ~subst leftno tyl
```

A declaration (an axiom) is well-typed when the declared type is. A block of mutual inductive types must satisfy a number of conditions that deserve their own test check_mutual_inductive_defs, which will be described in Section 6.2.

```
   | C.Fixpoint (inductive, fl, _) →
      let types, kl =
        List.fold_left
         (fun (types,kl) (_,name,k,ty,_) →
           let _ = typeof ~subst ~metasenv [] ty in
            ((name,C.Decl ty)::types, k::kl)
         ) ([],[]) fl
      in
      let len = List.length types in
```

```
let dfl , kl =
  List . split  (List .map2
    (fun (_,_,_,_,bo) rno →
       let dbo = debruijn uri len  []  bo in
       dbo, Evil rno)
    fl  kl)
in
```

A Fixpoint represents a block of mutual (co)recursive functions, defined in the fl list. Each function definition is given by a name, a type ty and a body bo. Since the functions are mutually defined, a reference to any function in the block may occur in the body of a function in fl. Since we are still type-checking the object, the object is not yet part of the environment (if it is not in the library) or, if it was retrieved from the library, it is in the environment frozen stack (the stack of objects being type-checked, see Section 4.2). In both cases, the environment will raise an exception if we try to resolve a reference to a function in the block.

To solve this problem, we use the debruijn function (see Appendix C) that replaces every recursive reference in a body bo with a DeBruijn index. The URI uri of the block is used to detect recursive references. The *i*-th function in the block will be represented with the (1+len−i)-th DeBruijn index where len is the number of recursively defined functions. In order to maintain the invariant that a term always comes with a context that closes it, we pre-compute the context types that collects all function type declarations. Since we have the invariant that contexts are well-typed, we check every function type before pushing it on top of the environment. At the same time we also collect in the kl list all the recursive indices. The recursive index of a recursive function is the index of the argument the functions performs structural recursion on. The recursive index is undefined (0 in the implementation) for co-recursive functions.

We are now ready to type-check every recursive function definition.

```
List . iter2  (fun (_,name,x,ty,_)  bo →
 let  ty_bo = typeof ~subst ~metasenv types bo in
 if  not (R.are_convertible  ~subst types ty_bo ty)
 then raise (TypeCheckerFailure (lazy (”(Co)Fix: ill−typed bodies”)))
 else
```

For each function definition we also need to perform a different check if the block is made of recursive of corecursive functions. In the first case, we must check totality of the functions by verifying syntactically that the functions are defined by structural recursion on the x-th argument. The check is performed by guarded_by_destructors described in Section 6.3. In the second case we must check syntactically that the function is productive. The check is performed by guarded_by_constructors, also described in Section 6.3.

```
if  inductive  then begin
  let  m, context = eat_lambdas ~subst ~metasenv types (x + 1) bo in
  let  r_uri ,  r_len  =
    let  he =
     match List.hd context with _,C.Decl t → t | _ → assert  false
    in
    match R.whd ~subst (List.tl context) he with
    | C.Const (Ref.Ref (uri,Ref.Ind _)  as ref)
    | C.Appl (C.Const (Ref.Ref (uri,Ref.Ind _)  as ref)  ::  _)  →
        let  _,_, itl ,_,_ = E.get_checked_indtys ref in
          uri ,  List .length  itl
          | _ → assert false
         in
         (* guarded by destructors conditions D{f,k,x,M} *)
         let rec enum_from k =
           function [] → [] | v :: tl → (k,v)::enum_from (k+1) tl
         in
         guarded_by_destructors r_uri r_len
          ~subst ~metasenv context (enum_from (x+2) kl) m
       end else
        match returns_a_coinductive ~subst [] ty with
         | None →
            raise (TypeCheckerFailure
              (lazy "CoFix: does not return a coinductive type"))
         | Some (r_uri, r_len) →
            (* guarded by constructors conditions C{f,M} *)
            if not
            (guarded_by_constructors ~subst ~metasenv types bo r_uri r_len len)
            then
              raise (TypeCheckerFailure
               (lazy "CoFix: not guarded by constructors"))
       ) fl dfl
```

We present now the function typeof that checks if terms are well-typed. Then we will present the most technical ones. The function is defined by structural recursion on the term, and it augments the context when binders are met. Since the metasenv and substitution are constant during the reduction, we perform recursion in an auxiliary function.

```
let rec typeof ~subst ~metasenv context term =
  let rec typeof_aux context =
    fun t → (*prerr_endline (PP.ppterm ~metasenv ~subst ~context t);*)
    match t with
    | C.Rel n →
       (try
         match List.nth context (n − 1) with
         | (_,C.Decl ty) → S.lift n ty
         | (_,C.Def (_,ty)) → S.lift n ty
        with Failure _ → raise (TypeCheckerFailure (lazy "unbound variable")))
    | C.Sort (C.Type [false,u]) → C.Sort (C.Type [true, u])
    | C.Sort (C.Type _) →
        raise (AssertFailure (lazy ("Cannot type an inferred type: "^
          NCicPp.ppterm ~subst ~metasenv ~context t)))
    | C.Sort _ → C.Sort (C.Type NCicEnvironment.type0)
    | C.Implicit _ → raise (AssertFailure (lazy "Implicit found"))
    | C.Meta (n,l) as t →
       let canonical_ctx,ty =
        try
         let _,c,_,ty = U.lookup_subst n subst in c,ty
        with U.Subst_not_found _ → try
         let _,c,ty = U.lookup_meta n metasenv in c,ty
        with U.Meta_not_found _ →
         raise (AssertFailure (lazy (Printf.sprintf
          "%s not found" (PP.ppterm ~subst ~metasenv ~context t))))
       in
        check_metasenv_consistency t ~subst ~metasenv context canonical_ctx l;
        S.subst_meta l ty
    | C.Const ref → type_of_constant ref
```

The type of atoms which are not sorts is retrieved from the context in case of Rels or from the metasenv or the substitution in case of metavariables.

Type universes are typed with their successor (represented by setting theirs first component to true). Since the user is only allowed to write type universe variables, it is impossible that a non singleton universe (or the successor of a universe) is a valid input for the typing function. Moreover, the typing algorithm never types a synthesized type, thus it will never need to typecheck the successor of a type. The impredicative sort Prop is typed with the smallest possible universe type0 (defined as the maximum of the empty set).

In the latter two cases we must remember to relocate the metavariable definiens in the local context, but only after checking the consistency between the local and global metavariable contexts. This is done by the check_metasenv_consistency function (see App. C). In principle, the check is very simple:

1. the local context and the canonical contexts must have the same length (or, equivalently, the local context must instantiate every abstraction in the canonical context)
2. if the n-th entry in the canonical context is a declaration $x_n : T(x_1, \ldots, x_{n-1})$, then the type of the n-th term t_n in the local context must be convertible to $T(t_1, \ldots, t_{n-1})$
3. if the n-th entry in the canonical context is a definition $x_n : T(x_1, \ldots, x_{n-1}) := t(x_1, \ldots, x_{n-1})$, then the n-th term t_n in the local context must be convertible to $t(t_1, \ldots, t_{n-1})$

In practice, the code of the function is quite complex since we exploit the optimized representation of local contexts (see 2.1.1) to speed up the checks as much as possible. The main reason for these optimizations is that automatic proof search may generate hundreds of goals containing many metavariables characterized by potentially long contexts. Thus, even a small speed-up in this test can have a significant effect on the overall time of automatic proof search.

The type_of_constant function checks that data cached in the reference is in coherent with the one stored in the environment and if the constant has never been typed before, it recursively check its type.

```
    | C.Prod (name,s,t) →
       let sort1 = typeof_aux context s in
       let sort2 = typeof_aux ((name,(C.Decl s))::context) t in
       sort_of_prod ~metasenv ~subst context (name,s) (sort1,sort2)
    | C.Lambda (n,s,t) →
       let sort = typeof_aux context s in
       (match R.whd ~subst context sort with
       | C.Meta _ | C.Sort _ → ()
       | _ →
         raise
           (TypeCheckerFailure (lazy (Printf.sprintf
             ("Not well−typed lambda−abstraction: " ^^
             "the source %s should be a type; instead it is a term " ^^
             "of type %s") (PP.ppterm ~subst ~metasenv ~context s)
             (PP.ppterm ~subst ~metasenv ~context sort)))));
       let ty = typeof_aux ((n,(C.Decl s ))::context) t in
         C.Prod (n,s,ty)
    | C.LetIn (n,ty,t,bo) →
       let ty_t = typeof_aux context t in
       let _ = typeof_aux context ty in
       if not (R.are_convertible ~subst context ty_t ty) then
         raise
```

```
          (TypeCheckerFailure
            (lazy (Printf.sprintf
              "The type of %s is %s but it is expected to be %s"
                (PP.ppterm ~subst ~metasenv ~context t)
                (PP.ppterm ~subst ~metasenv ~context ty_t)
                (PP.ppterm ~subst ~metasenv ~context ty))))
      else
        let ty_bo = typeof_aux ((n,C.Def (t,ty)):: context) bo in
        S.subst ~avoid_beta_redexes:true t ty_bo
  | C.Appl (he::(_ :: _ as args)) →
      let ty_he = typeof_aux context he in
      let args_with_ty = List.map (fun t → t, typeof_aux context t) args in
      eat_prods ~subst ~metasenv context he ty_he args_with_ty
  | C.Appl _ → raise (AssertFailure (lazy "Appl of length < 2"))
```

The rules for dependent products, local definitions, λ-abstractions and applications are standard for full PTSs. In particular, the sort_of_prod function (see App. C) computes the sort of a product according to the PTS product formation rules. The eat_prods (also in App. C) function verifies one at a time that the type of the n-th actual parameters is convertible with the declared type for the formal parameter. In order to retrieve the latter information, the head of the application is type-checked and its type is put in weak head normal form and matched against a dependent product: the type of the first actual parameter is the source type of the dependent product. The type of the second parameter is computed from the target of the dependent product where the first bound parameter has been instantiated with the actual parameter. The same procedure is used for the following ones.

Case analysis, the only case left to consider, is also the most involved one. This is partly due to the special management of left (or constant) parameters (see Section 2.1.4) during case analysis. We can informally explain the management on a very trivial example: let's consider the following example (in the concrete syntax of Matita) of a non-mutual inductive type having one left parameter, one right parameter and just one constructor.

```
inductive i (n:nat) : nat → Prop :=
  k: ∀w. i n (f w)
```

Now, let's consider a term u of type (i x y). Its normal form must be (k n w) for some n and w such that the type (i n (f w)) of (k n w) is convertible to (i x y). Since n is a constant parameter, we know that x is n and we can exploit this knowledge in case analysis by avoiding to abstract the case branch on x. On the other hand, the knowledge about y being (f w) does not provide to us a single value for w. Thus, the case branch is only abstracted on the constructor parameter w and case analysis on (k n w) is expressed (in the concrete syntax of Matita) by

```
match k n w return λw1:nat.λx:k n w1.s with [ k w2 ⇒ p ]
```

The output type λw1:nat.λx:k n w.s, where s is a type for each w1 and x, is a schema of types for all the right hand sides of the patterns (in this case λw2:nat.p). The interesting case is when s is a type dependent on the actual value of w1 or x (such as being a natural number if w1 is positive, or an empty type representing an impossible event if w1 is zero). Less artificial examples are given by pattern matching over inductive types with more than one constructor, where it is sometimes useful to return a different type according to the constructor being matched. In this case, the output type is dependent on the value of the actual matched value.

The check to verify that the output type correctly matches the types inferred for each pattern is very technical. We first compute the type ∀w2:nat.i n (f w) of the constructor k applied to the left parameters (here just n); note that the abstraction on w2 exactly matches the abstraction in the pattern, allowing us to consider the context w1:nat in which both i n (f w) and p live. Then we instantiate, in the latter context, the output type schema with the actual right parameters passed to *i* in the computed type, i.e. (f w) and with the constructor applied to the passed parameters, obtaining s [(f w)/w1;(k n w2)/x] which is a sort that is required to type the pattern body (p w2).

To summarize, in OCaml syntax, the same pattern matching example becomes

```
C.Match (i,
 C.Lambda("w2", nat,
  C.Lambda("x",C.Appl [C.Const i; n; C.Rel 1],
   C.Appl [s; C.Rel 1; C.Rel 2])),
 u,
 [C.Lambda("w1", nat, p)])
```

In this expression, u is the term being matched and i is a reference to its type; the second line states the *return type* of the case analysis, and the last line is a list of patterns: since we are matching over a type having just one constructor, the list of patterns contains a single pattern, abstracted over right parameters as described above. As for the return type, it expresses the output type of any branch of a dependently-typed pattern matching and it is abstracted over the right parameters of the type being matched but, similarly to a pattern, not on the left ones; it also abstracted on the actual value being matched.

Now, let's check the code implementing the typechecking in the case of the pattern matching.

```
  | C.Match (Ref.Ref (_,Ref.Ind (_,tyno,_)) as r,outtype,term,pl) ->
     let outsort = typeof_aux context outtype in
     let inductive,leftno,itl,_,_ = E.get_checked_indtys r in
     let constructorsno =
       let _,_,_,cl = List.nth itl tyno in List.length cl
     in
     let parameters, arguments =
       let ty = R.whd ~subst context (typeof_aux context term) in
       let r',tl =
        match ty with
           C.Const (Ref.Ref (_,Ref.Ind _) as r') -> r',[]
         | C.Appl (C.Const (Ref.Ref (_,Ref.Ind _) as r') :: tl) -> r',tl
         | _ ->
            raise
              (TypeCheckerFailure (lazy (Printf.sprintf
                "Case analysis: analysed term %s is not an inductive one"
                (PP.ppterm ~subst ~metasenv ~context term)))) in
       if not (Ref.eq r r') then
        raise
         (TypeCheckerFailure (lazy (Printf.sprintf
           ("Case analysis: analysed term type is %s, but is expected " ^^
            "to be (an application of) %s")
           (PP.ppterm ~subst ~metasenv ~context ty)
           (PP.ppterm ~subst ~metasenv ~context (C.Const r')))))
      else
       try HExtlib.split_nth leftno tl
       with
        Failure _ ->
         raise (TypeCheckerFailure (lazy (Printf.sprintf
           "%s is partially applied"
           (PP.ppterm ~subst ~metasenv ~context ty)))) in
```

So far we have only checked that all the terms occurring in the pattern matching are well typed, and that the type of the matched term is the declared inductive type.

```
      (* let's control if the sort elimination is allowed: [(I q1 ... qr)|B] *)
      let sort_of_ind_type =
        if parameters = [] then C.Const r
        else C.Appl ((C.Const r)::parameters) in
      let type_of_sort_of_ind_ty = typeof_aux context sort_of_ind_type in
      check_allowed_sort_elimination ~subst ~metasenv r context
       sort_of_ind_type type_of_sort_of_ind_ty outsort;
```

This snippet performs a first key check by calling the check_allowed_sort_elimination procedure. Before describing it, we look at the description of the second and last key check, which iterates on all the branches of the case analysis, to verify that the type of the patterns match the output type schema as previously explained.

```
      (* let's check if the type of branches are right *)
      if List.length pl <> constructorsno then
       raise (TypeCheckerFailure (lazy ("Wrong number of cases in a match")));
      let j,branches_ok,p_ty, exp_p_ty =
        List.fold_left
          (fun (j,b,old_p_ty,old_exp_p_ty) p ->
            if b then
              let cons =
                let cons = Ref.mk_constructor j r in
                if parameters = [] then C.Const cons
                else C.Appl (C.Const cons::parameters)
              in
              let ty_p = typeof_aux context p in
              let ty_cons = typeof_aux context cons in
              let ty_branch =
                type_of_branch ~subst context leftno outtype cons ty_cons 0
              in
              j+1, R.are_convertible ~subst context ty_p ty_branch,
              ty_p, ty_branch
            else
              j, false,old_p_ty,old_exp_p_ty
          ) (1,true,C.Sort C.Prop,C.Sort C.Prop) pl
      in
      if not branches_ok then
        raise
         (TypeCheckerFailure
          (lazy (Printf.sprintf ("Branch for constructor %s :=\n%s\n"^^
          "has type %s\nnot convertible with %s")
          (PP.ppterm ~subst ~metasenv ~context
            (C.Const (Ref.mk_constructor (j-1) r)))
          (PP.ppterm ~metasenv ~subst ~context (List.nth pl (j-2)))
          (PP.ppterm ~metasenv ~subst ~context p_ty)
          (PP.ppterm ~metasenv ~subst ~context exp_p_ty))));
      let res = outtype::arguments@[term] in
      R.head_beta_reduce (C.Appl res)
    | C.Match _ -> assert false
 ...
 in
   typeof_aux context term
```

Note that the type inferred for the pattern match is the head beta normal form of the instantiation of the output type on the matched term. Since the output type is always a λ-abstraction, without performing head beta reduction we will always infer a β-redex, that we do not want to present to the user. Moreover, inferring a β-redex is likely to produce a performance loss in reduction, since our heuristics to speed up conversion fail when comparing two types having in deep positions respectively a β-redex and its β-reduct.

We focus now on the check_allowed_sort_elimination that does two jobs at once. The first one is checking that the outtype abstractions match the right parameters of the inductive type with one additional abstraction for the term being eliminated. The check is actually done on the type of the outtype (called outsort) which has a product for each outtype λ-abstraction and ends with the sort of the returned type.

The second job ensures that the elimination of an inductive type can be performed to obtain the output type. The reason not to allow elimination lies in the distinction between computationally relevant parts of a proof (when a term has sort Type) and parts which have no computational content (terms whose sort is Prop). This distinction is crucial for code exportation and proof-irrelevance: the computationally irrelevant subterms are completely forgot during the automatic exportation of code. Thus, eliminating a non-informative type to obtain an informative type must not be allowed, unless there is only one way in which the elimination can be performed.

```
and check_allowed_sort_elimination ~subst ~metasenv r =
 let mkapp he arg =
   match he with
   | C.Appl l → C.Appl (l @ [arg])
   | t → C.Appl [t;arg] in
 let rec aux context ind arity1 arity2 =
  let arity1 = R.whd ~subst context arity1 in
  let arity2 = R.whd ~subst context arity2 in
    match arity1,arity2 with
    | C.Prod (name,so1,de1), C.Prod (_,so2,de2) →
        if not (R.are_convertible ~subst context so1 so2) then
         raise (TypeCheckerFailure (lazy (Printf.sprintf
          "In outtype: expected %s, found %s"
          (PP.ppterm ~subst ~metasenv ~context so1)
          (PP.ppterm ~subst ~metasenv ~context so2)
          )));
        aux ((name, C.Decl so1)::context)
         (mkapp (S.lift 1 ind) (C.Rel 1)) de1 de2
    | C.Sort _, C.Prod (name,so,ta) →
        if not (R.are_convertible ~subst context so ind) then
         raise (TypeCheckerFailure (lazy (Printf.sprintf
          "In outtype: expected %s, found %s"
          (PP.ppterm ~subst ~metasenv ~context ind)
          (PP.ppterm ~subst ~metasenv ~context so)
          )));
        (match arity1, R.whd ~subst ((name,C.Decl so)::context) ta with
          | (C.Sort (C.Type _), C.Sort _)
          | (C.Sort C.Prop, C.Sort C.Prop) → ()
          | (C.Sort C.Prop, C.Sort (C.Type _)) →
              let _, leftno , itl , _, i = E.get_checked_indtys r in
              let itl_len = List.length itl in
              let _,itname,ittype, cl = List.nth itl i in
              let cl_len = List.length cl in
               (* is it a singleton, non recursive and non informative
                  definition or an empty one? *)
```

```
                if not
                 (cl_len = 0 ||
                  (itl_len = 1 && cl_len = 1 &&
                   let _,_,constrty = List.hd cl in
                      is_non_recursive_singleton r itname ittype constrty &&
                      is_non_informative leftno constrty))
                then
                 raise (TypeCheckerFailure (lazy
                  ("Sort elimination not allowed")));
          | _,_ → ())
       | _,_ → ()
  in
   aux
```

The check is performed by the inner recursive function aux, taking the inductive type ind (already applied to the left arguments) that during recursion gets also applied to the variables bound by the products of the outsort. Eventually, it will become the type expected for the matched term, which is the source of the last product of outsort. arity1 is initially the arity of the inductive type applied to the left parameters. An invariant of the recursion is that arity1 is the type of ind. arity2 is initially the outsort.

If the two arities are products we perform the first job by checking if the source of the two products (corresponding to a right parameter) are convertible. Then we proceed recursively on their targets.

If arity1 is a sort and arity2 is a product, then the abstractions on the right parameters are over and we conclude the first job by checking that the outsort ends with a final product over the type ind of the matched term. This is also the case in which the second job is done: if the target of the product in arity2 is Type, we ensure that no informative content escapes a non-informative term, by checking that the eliminated term type of the elimination either has no constructors (is an empty type) or it has just one non-recursive constructor (singleton type) depending only on non-informative types. The procedure is_non_informative checks that the type of the single constructor of an inductive type is abstracted only over types whose sort is Prop and is_non_recursive_singleton ensures that the type of the constructor constrty does no recursive calls to the inductive type.

In any other case, the elimination is admissible.

6.1 Well typedness of inductive definitions

Inductive types are one of key ingredients of CIC, used to model both datatypes and relations. While the termination conditions for recursive functions have been sensibly relaxed over the years, the well typedness conditions for inductive types are a more consolidated subject.

```
and check_mutual_inductive_defs uri ~metasenv ~subst leftno tyl =
  List.iter (fun (_,_,x,_) → ignore (typeof ~subst ~metasenv [] x)) tyl;
  let len = List.length tyl in
  let tys = List.rev_map (fun (_,n,ty,_) → (n,(C.Decl ty))) tyl in
```

The first, easy, check is the well typedness of inductive types arity (the x component of tyl entries), that have to live in an empty context. The next step is to analyze the constructors of the inductive types, that live in a context where all the inductive types are defined (tys). Note that since the arity of inductive types is a closed term lifting them is not needed when generating a context.

```
(List.fold_right
 (fun (_,_,ty,cl) i -> (* i-th ind. type arity and constructors list *)
    List.iter
      (fun (_,name,te) -> (* constructor name and type *)
```

Four different checks have to be performed to accept an inductive type constructor:

1. The left (fixed) parameters have to be used coherently. This implies both that every constructor must abstract the very same left arguments as the inductive type, and that the inductive type occurs applied to these parameters. The following code compares in parallel the initial fragment of the context generated by the binders in the inductive type arity and the binders in the constructor type.

```
let context,ty_sort = split_prods ~subst [] ~-1 ty in
let sx_context_ty_rev,_ = HExtlib.split_nth leftno (List.rev context) in
let te = debruijn uri len [] te in
let context,te = split_prods ~subst tys leftno te in
let _,chopped_context_rev =
 HExtlib.split_nth (List.length tys) (List.rev context) in
let sx_context_te_rev,_ =
 HExtlib.split_nth leftno chopped_context_rev in
(try
  ignore (List.fold_left2
   (fun context item1 item2 ->
     let convertible =
      match item1,item2 with
        (n1,C.Decl ty1),(n2,C.Decl ty2) ->
          n1 = n2 && R.are_convertible ~subst context ty1 ty2
      | (n1,C.Def (bo1,ty1)),(n2,C.Def (bo2,ty2)) ->
          n1 = n2
          && R.are_convertible ~subst context ty1 ty2
          && R.are_convertible ~subst context bo1 bo2
      | _,_ -> false
     in
      if not convertible then
       raise (TypeCheckerFailure (lazy
        ("Mismatch between the left parameters of the constructor " ^
         "and those of its inductive type")))
      else
       item1::context
   ) [] sx_context_ty_rev sx_context_te_rev)
```

The check that all inductive type occurrences are applied to the same left parameters is performed along with check number 4.

2. The type has to be correct. To check that we use the debruijn function to change occurrences of the inductive types to references in the tys context. Here te is the inductive constructor type and context has been previously checked to be well typed and contains not only the inductive types arities but also the left parameters.

```
let con_sort = typeof ~subst ~metasenv context te in
```

3. The universe in which the inductive type lives has to be greater (or equal) to the ones of its constructors.

```
(match R.whd ~subst context con_sort, R.whd ~subst [] ty_sort with
        (C.Sort (C.Type u1) as s1), (C.Sort (C.Type u2) as s2) →
          if not (E.universe_leq u1 u2) then
           raise
            (TypeCheckerFailure
              (lazy ("The type " ^ PP.ppterm ~metasenv ~subst ~context s1^
                " of the constructor is not included in the inductive" ^
                " type sort " ^ PP.ppterm ~metasenv ~subst ~context s2)))
      | C.Sort _, C.Sort C.Prop
      | C.Sort _, C.Sort C.Type _ → ()
      | _, _ →
          raise
           (TypeCheckerFailure
             (lazy ("Wrong constructor or inductive arity shape"))));
```

4. The inductive type must occur only positively in the type of its constructors. This check is far from being trivial, and the whole following section is dedicated to its implementation.

```
      if not
          ( are_all_occurrences_positive ~subst context uri leftno
            (i+leftno) leftno (len+leftno) te)
      then
        raise (TypeCheckerFailure
          (lazy ("Non positive occurrence in "^NUri.string_of_uri uri))))
```

6.2 Positivity conditions

It is well known that in the definition of inductive and coinductive types, a positivity condition on the type of their constructors must be verified, in order to guarantee the logical consistency of the system. With dependent types, this requirement must be strengthened to a property known as *strict positivity.* This notion of positivity is justified by a translation from general inductive definitions (i.e. mutually defined, possibly nested inductive types) to a single inductive type, which is guaranteed to be sound when the input types satisfy the strict positivity condition. The details are both lengthy and involved: the interested reader may check Paulin-Mohring (1996) for a detailed account on the issue of inductive types and positivity. In this section we will just state what strict positivity is and discuss the code implementing the positivity check, occasionally hinting at the reasons behind the check.

Suppose we are checking an inductive type I, whose definition belongs to a block of k mutually defined inductive types $I_1, \ldots, I_k$. We have

$$I : \Pi l_1 : T_1 \ldots \Pi l_m : T_m.\Pi r_1 : U_1 \ldots \Pi r_n : U_n.S$$

where S is a sort, $l_1 \ldots l_m$ and $r_1 \ldots r_n$ are respectively the m left parameters and n right parameters of I ($T_1 \ldots T_m$ and $U_1 \ldots U_n$ being their types). For all constructors c of I, we want to ensure that any occurrence of the types $I_1 \ldots I_k$ in the types of the arguments of c is strictly positive. This is performed by two mutually recursive procedures are_all_occurrences_positive and strictly_positive . The check is triggered by the typechecker invoking are_all_occurrences_positive on the type of any constructor k. This procedure is defined by recursion on the type of the constructor, say

$$c : \Pi x_1 : V_1 \ldots \Pi x_h : V_h.O$$

For all $i = 1, \ldots, h$, if $\Pi x_i : V_i . \cdots$ is a non-dependent product (meaning that x_i does not occur in the target of the product), we will check that I and any other inductive type mutually defined with I occur only strictly positively in V_i (by means of a call to strictly_positive); if on the other hand it is a dependent product, then I and its sibling inductive types must not occur in V_i at all. Finally, the procedure also checks that $O = I\ l_1 \cdots l_m\ a_1 \cdots a_n$, such that I and its sibling inductive types do not occur in $a_1 \ldots a_n$.

The strictly_positive procedure checks the inductive types defined in the current block occur only strictly positively in a type $\Pi x_1 : V_1 \ldots \Pi x_h : V_h . V$. The condition is satisfied if:

1. the inductive types defined in the current block do not occur neither in $V_1 \ldots V_h$ nor in V; or
2. the inductive types defined in the current block do not occur in $V_1 \ldots V_h$ and
   (a) if $V = I'\ t_1 \cdots t_p$ where $I' \in \{I_1, \ldots, I_k\}$, none of these types occurs in $t_1 \ldots t_p$;
   (b) if $V = I''\ t_1 \cdots t_l\ u_1 \cdots u_r$ where I'' is the only inductive type defined in another block with l left parameters, the types defined in the current block do not occur in $u_1 \ldots u_r$ and a call to procedure are_all_occurrences_positive to check that the types defined in the current block occur only positively in the (appropriately instantiated) constructors of $I''\ t_1 \cdots t_l$ succeeds.

The point 2b is particularly involved and deserves to be justified. Basically, we have a priori no knowledge about the way the right parameters of I'' are propagated by constructors in recursive calls, so we do not want the types defined in the current block to occur in $u_1 \ldots u_r$. The situation is radically different if these types occur only in left positions. In this case we can easily track their propagation. For what concerns the call to are_all_occurrences_positive , it is needed because, as we mentioned before, the positivity check is justified by a syntactical transformation which we are now going to show in a simple example. Note that the transformation is not actually performed in the code; it is only used to show its correctness.

Suppose we are defining two types in two different blocks, with the second type mentioning the first one.

```
inductive list (A:Type) : Type :=
| nil : list A
| cons : A → list A → list A.
inductive t : Type :=
| k : list t → t.
```

Matita behaves as if the definition of t were converted in a mutual inductive definition of two types:

```
inductive t' : Type :=
| k' : list_t ' → t'
and list_t ' : Type :=
| nil ' : list_t '
| cons' : t' → list_t ' → list_t '.
```

list_t ' can be understood as the type whose constructors we are feeding to are_all_occurrences_positive . In practice we perform the positivity check inside list_t ' as if it were defined in the same block as t'.

Now let's see the code implementing the positivity check. are_all_occurrences_positive takes as input a substitution and a context (as usual), the uri of the inductive definition we are inspecting, the number of left parameters of the inductive type, the index i of the inductive type whose constructor we are checking (to verify if it really inhabits that inductive type Ref.Ind(uri, i)), the range (n,nn] of indices referring to the types defined in the current block, the type te of the constructor and the number of left arguments indparamsno. The function is defined by recursion on te.

```
(* the inductive type indexes are s.t. n < x <= nn *)
and are_all_occurrences_positive ~subst context uri indparamsno i n nn te =
  match R.whd context te with
  | C.Appl ((C.Rel m)::tl) as reduct when m = i ->
      check_homogeneous_call ~subst context indparamsno n uri reduct tl;
      List. for_all (does_not_occur ~subst context n nn) tl
  | C.Rel m when m = i ->
      if indparamsno = 0 then
       true
      else
        raise (TypeCheckerFailure
         (lazy ("Non-positive occurence in mutual inductive definition(s) [3]"^
          NUri. string_of_uri uri )))
```

The first two cases ensure that occurrences of the *i*-th inductive type are always consistently applied to the left parameters, and that the inductive type itself is not occurring in any additional constructor argument.

```
  | C.Prod (name,source,dest) when
      does_not_occur ~subst ((name,C.Decl source)::context) 0 1 dest ->
       strictly_positive ~subst context n nn indparamsno uri source &&
        are_all_occurrences_positive ~subst
         ((name,C.Decl source)::context) uri indparamsno
         (i+1) (n + 1) (nn + 1) dest
  | C.Prod (name,source,dest) ->
       if not (does_not_occur ~subst context n nn source) then
         raise (TypeCheckerFailure (lazy ("Non-positive occurrence in "^
         PP.ppterm ~context ~metasenv:[] ~subst te)));
        are_all_occurrences_positive ~subst ((name,C.Decl source)::context)
        uri indparamsno (i+1) (n + 1) (nn + 1) dest
```

The next two items check the conditions on the leading products of the type of a constructor. The first one corresponds to the condition on non-dependent products (as checked by the first call to does_not_occur), while the second one implements the check for dependent products.

```
  | _ ->
     raise
      (TypeCheckerFailure (lazy ("Malformed inductive constructor type " ^
        (NUri. string_of_uri uri ))))
```

Finally, we assert that in any other case, the type of the constructor is malformed. The check_homogeneous_call function is in charge of checking that the list of terms tl has a prefix of length indparamsno of variables corresponding to the fixed parameters of the inductive type.

```
let check_homogeneous_call ~subst context indparamsno n uri reduct tl =
 let last =
  List. fold_left
   (fun k x ->
     if k = 0 then 0
     else
```

```
      match R.whd context x with
       | C.Rel m when m = n − (indparamsno − k) → k − 1
       | _ → raise (TypeCheckerFailure (lazy
          ("Argument "^string_of_int (indparamsno − k + 1) ^ " (of " ^
           string_of_int indparamsno ^ " fixed) is not homogeneous in "^
          "appl:\n"^ PP.ppterm ~context ~subst ~metasenv:[] reduct))))
   indparamsno tl
 in
  if last <> 0 then
   raise (TypeCheckerFailure
    (lazy ("Non−positive occurence in mutual inductive definition(s) [2]"^
     NUri.string_of_uri uri )))
```

The actual positivity check is delegated to the function strictly_positive .

```
and strictly_positive ~subst context n nn indparamsno posuri te =
  match R.whd context te with
   | t when does_not_occur ~subst context n nn t → true
   | C.Rel _ when indparamsno = 0 → true
   | C.Appl ((C.Rel m)::tl) as reduct when m > n && m <= nn →
      check_homogeneous_call ~subst context indparamsno n posuri reduct tl;
      List. for_all (does_not_occur ~subst context n nn) tl
   | C.Prod (name,so,ta) →
      does_not_occur ~subst context n nn so &&
        strictly_positive ~subst ((name,C.Decl so)::context) (n+1) (nn+1)
        indparamsno posuri ta
```

The first item in the pattern matching corresponds to the case 1 in the informal description. As for cases 2a and 2b:

– in both cases, the inductive types defined in the current block must not occur in the sources of leading products (as checked by the third item of the pattern matching);
– if the type we are checking ends with an inductive type with no parameters, cases 2a and 2b collapse and the check is vacuously satisfied (this corresponds to the second item of the pattern matching);
– the forth item of the pattern matching corresponds to the remaining condition of case 2a.

```
   | C.Appl (C.Const (Ref.Ref (uri,Ref.Ind _) as r ):: tl ) →
      let _,paramsno,tyl,_,i = E.get_checked_indtys r in
      let _,name,ity,cl = List.nth tyl i in
      let ok = List.length tyl = 1 in
      let params, arguments = HExtlib.split_nth paramsno tl in
      let lifted_params = List.map (S.lift 1) params in
      let cl =
        List .map (fun (_,_,te) → instantiate_parameters lifted_params te) cl
      in
      ok &&
      List. for_all (does_not_occur ~subst context n nn) arguments &&
      List. for_all
       (weakly_positive ~subst ((name,C.Decl ity)::context) (n+1) (nn+1)
         uri indparamsno posuri) cl
```

Finally we have to perform the most complex part of the positivity check, corresponding to case 2b in the informal discussion. The important difference here is that, to check the nested inductive type occurrence, we do not invoke are_all_occurrences_positive , but a specialized function called weakly_positive. The code of weakly_positive is similar to the code of are_all_occurrences_positive since it is used to check the constructors of the nested inductive type, but it differs from it since it is supposed to check positivity with respect to the current block of inductive definitions (which is not the block in which the nested inductive type was defined). Moreover, weakly_positive can exploit the previous knowledge about the fact that the nested type already passed the strictly positive check on its constructors with respect to the block in which it was defined.

The description of strictly_positive is completed by its default case that captures all remaining non strictly positive terms:

```
  | _ → false
```

Last, we see the code of the weakly_positive function, which is very similar to the function are_all_occurrences_positive , but performing a slightly different check. The function is also designed to work on constructor types where recursive parameters are identified not by indices, but by the appropriate constant.

```
(* Inductive types being checked for positivity have *)
(* indexes x s.t. n < x <= nn.                       *)
let rec weakly_positive ~subst context n nn uri indparamsno posuri te =
  (*CSC: Not very nice. *)
  let dummy = C.Sort C.Prop in
  (*CSC: to be moved in cicSubstitution? *)
  let rec subst_inductive_type_with_dummy _ = function
    | C.Const (Ref.Ref (uri',Ref.Ind (true,0,_))) when NUri.eq uri' uri → dummy
    | C.Appl ((C.Const (Ref.Ref (uri',Ref.Ind (true,0,lno))))::tl)
        when NUri.eq uri' uri →
          let _, rargs = HExtlib.split_nth lno tl in
          if rargs = [] then dummy else C.Appl (dummy :: rargs)
    | t → U.map (fun _ x→ x) () subst_inductive_type_with_dummy t
  in
  (* this function has the same semantics of are_all_occurrences_positive
     but the i-th context entry role is played by dummy and some checks
     are skipped because we already know that are_all_occurrences_positive
     of uri in te. *)
  let rec aux context n nn te =
    match R.whd context te with
     | t when t = dummy → true
     | C.Appl (te::rargs) when te = dummy →
        List.for_all (does_not_occur ~subst context n nn) rargs
```

An occurrence of the type i' whose constructors we are checking is weakly positive. To see if a constant refers to i', it is sufficient to check its uri and not its index, since the inductive type i' is not mutually defined with any other type and since i' has already been typechecked in advance (what we are doing now is checking that a reference to i' from inside another inductive type i is admissible, not checking that i' is well typed).

```
     | C.Prod (name,source,dest) when
        does_not_occur ~subst ((name,C.Decl source)::context) 0 1 dest →
         (* dummy abstraction, so we behave as in the anonimous case *)
          strictly_positive ~subst context n nn indparamsno posuri source &&
         aux ((name,C.Decl source)::context) (n + 1) (nn + 1) dest
     | C.Prod (name,source,dest) →
         does_not_occur ~subst context n nn source &&
         aux ((name,C.Decl source)::context) (n + 1) (nn + 1) dest
```

In the case of an arrow or a dependent product, we first replace all the occurrences of i' in the source with a dummy. Then, we check that the indices between n and nn (referring to the inductive types defined in another block) occur only positively (in the case of an arrow) or do not occur (in the case of a dependent product) in the source.

```
  | _ →
    raise (TypeCheckerFailure (lazy "Malformed inductive constructor type"))
```

In any other case, we are checking a malformed constructor type.

6.3 Ensuring termination of recursive functions

We now discuss the guarded_by_destructors algorithm (GD), which is used by the type-checker to ensure termination of functions defined by fixpoint. The algorithm checks that recursive calls are applied to a strictly smaller parameter (in a sense that will be clear later).

In our setting, fixpoint definitions must be top level objects: this means that nested recursive definitions are not allowed and should be replaced by multiple top-level fixpoints. To retain the full expressive power of nested recursive definitions, we are compelled to allow a recursive function to pass itself around as an argument to other recursive functions. Of course, this case needs special care, to make sure that the recursive function passed as an argument is used in a well-founded manner; to ensure termination even in this case, the algorithm must perform a deep analysis, unfolding the fixpoints and keeping track of them too.

Before entering the details of the code we give a precise specification in prose to help the reader understand the termination check.

Let p be a pattern used for matching a term v of inductive type ($I\ largs\ rargs$), and let $x_1, \ldots, x_{n_0}$ be the variables bound by the pattern. Recursively at step k let $x_{n_k+1}, \ldots, x_{n_{k+1}}$ be the variables bound in a pattern used for matching either some x_j or some applications of x_j (with $j \leq n_k$). We say that if x_i has type $\forall z_1 ... z_n . I\ largs'\ rargs'$ then the term bound by x_i applied to n arguments (possibly none) is smaller than v. In this case the typing conditions over I grant that $largs'$ is equal to $largs$, and that this order is well founded. In the implementation the variables x_i are called *safe* and v, which in the interesting case is always a variable, is called *seed*.

In the following example, only the terms bound by variables he and he' are smaller than v, while l, tl ,hd,hd', tl ' are safe.

```
inductive list (T : Type) : Type := nil : list T | cons : T → list T → list T
inductive tree : Type := leaf : tree | node : list tree → nat → tree.
let v := node [ leaf ; node [] 2 ] 3 in
  match v with
  [ leaf ⇒...
  | node l _ ⇒
      match l with
      [ nil ⇒...
      | cons he tl ⇒
          match tl with [ nil ⇒...| cons he' tl' ⇒...]]]
```

A more complex example involving higher order constructors is the following, where terms bound to x, y and terms of the form g n and h n for any n are smaller.

```
inductive ord : Type := Zero : ord | Succ : ord → ord | Lim : (nat → ord) → ord.
let v := Succ (Lim f) in
```

```
 match v with
 [ Zero ⇒...| Succ x ⇒...
 | Lim g ⇒
     match g 4 with [ Zero ⇒...| Succ y ⇒...| Lim h ⇒...]]
```

The desiderata would be to accept any well founded recursion w.r.t. this order relation which is in general undecidable. Thus, we only accept functions that honor the following decidable syntactic approximation. We say that a recursive function $f\ x_1 \dots x_m$ is guarded by destructors on the n-th argument when one of the following hold for each occurrence of f in its definition

- f is applied to at least n arguments $v_1 \dots v_k$ and v_n generates only smaller terms (GST). Ideally a term v_n is in GST if for every possible instantiation with closed terms of its free variables, it reduces to a term smaller than x_n. Because of undecidability, we approximate GST with the following syntactic rules[10]:
  - a term smaller than x_n is in GST.
  - if t reduces to a term in GST then t is in GST.
  - if t is a case analysis end every branch is GST then t is in GST. (†)
- f is passed to another recursive function g (already checked to be terminating) as the k-th argument and the following conditions hold:
  - the first k arguments of g are fixed, that is always passed unmodified to recursive calls. Since we allow only top level fixpoints, the user can not define deep fixpoints fixing some arguments by abstracting them outside the fixpoint.
  - we build a term b representing a generic recursive invocation of g by unfolding g and substituting:
    - all fixed arguments with the actual ones (in which f may occur). Indeed these arguments are the same in every recursive call.
    - the recursive argument with a fresh variable that is considered *safe* if the actual argument passed to g by f is *safe*. The fact that it remains safe at each iteration can be proved by induction on g.
    - all remaining arguments with fresh variables representing generic terms we do not statically know anything about.
    - all occurrences of g applied to its fixed arguments with a fresh variable to avoid unfolding g again.

    The resulting term b has to be guarded by destructors.

The following example is accepted by Matita. The interesting function is count that passes itself to fold.

```
inductive list (T : Type) : Type := nil : list T | cons : T → list T → list T
inductive tree : Type := leaf : tree | node : list (name ∗ tree) → name ∗ nat → tree.
let rec fold T f (l : list T) acc :=
 match l with
 [ nil ⇒acc
 | cons he tl ⇒fold T f tl (f he acc )].
let snd := λt. match t with [ pair _ b ⇒b].
let rec count (t : tree) : nat :=
 match t with
 [ leaf ⇒O
 | node l _ ⇒1 + fold ? (λt,acc.acc + count (snd t)) l O].
```

[10] Clearly our approximation of GST can be improved, for instance considering recursive functions. The current rules proved to be sufficient to accept any recursive definition in the libraries of Matita and Coq.

What is actually checked for guardedness instead of the original call to fold is the following term under the assumption that L is safe:

```
  match L with
  [ nil  ⇒ ACC
  | cons he tl  ⇒ FOLD tl (ACC + count (match he with [ pair _ b  ⇒ b ]))]].
```

Note that the count call is guarded by destructors because of rule (†).

Matita allows mutually recursive function definitions at the top level. Thus we have modified the syntactic termination check explained above to the special case of a block of functions recursive on arguments whose types are also mutually recursive. For instance it is possible to define the mutually recursive types of forests and trees and corecursive functions on them:

```
inductive forest : Type := root : tree  →  forest  →  forest  | empty : forest
with tree : Type := node : nat →  forest  →  tree .
let rec size_forest  (f : forest) : nat :=
  match f with [ root t f  ⇒ size_tree  t + size_forest  f  | empty  ⇒ O ]
and size_tree  (t : tree) : nat :=
  match t with [ node _ f  ⇒ S (size_forest  f)  ].
```

The function that implements the GD check is called guarded_by_destructors and the one for GST is called is_really_smaller . The recursive_args and get_new_safes functions are used in combination to detect safe and smaller terms. Let's see in detail the code of these functions.

All the information regarding the *seed*, *safe* variables and unfolded fixpoints is kept in the recfuns parameter of the guarded_by_destructors function.

```
and guarded_by_destructors r_uri r_len  ~subst ~metasenv context recfuns t =
 let recursor  f k t  = U.fold shift_k  k (fun k () → f k) () t in
 let rec aux (context, recfuns, x as k) t =
  try
  match t with
  | C.Rel m as t when is_dangerous m recfuns →
      raise (NotGuarded (lazy
        (PP.ppterm ~subst ~metasenv ~context t ^
         " is a partial application of a fix")))
  | C.Appl ((C.Rel m)::tl) as t when is_dangerous m recfuns →
     let rec_no = get_recno m recfuns in
     if not (List.length tl > rec_no) then
       raise (NotGuarded (lazy
         (PP.ppterm ~context ~subst ~metasenv t ^
         " is a partial application of a fix")))
     else
       let rec_arg = List.nth tl rec_no in
       if not ( is_really_smaller  r_uri  r_len  ~subst ~metasenv k rec_arg) then
         raise (NotGuarded (lazy (Printf.sprintf ("Recursive call %s, %s is not"
          ^^ " smaller.\ncontext:\n%s") (PP.ppterm ~context ~subst ~metasenv
          t) (PP.ppterm ~context ~subst ~metasenv rec_arg)
          (PP.ppcontext ~subst ~metasenv context))));
       List.iter (aux k) tl
```

First of all, the algorithm checks if the term t to be checked is *dangerous*, i.e. it is a recursive call to one of the functions defined in the current fixpoint block. In this case, it must be applied at least up to the recursive parameter, the actual recursive argument must be *really smaller* and all the arguments must be, recursively, guarded by destructors.

```
  | C.Appl ((C.Rel m)::tl) when is_unfolded m recfuns →
      let fixed_args = get_fixed_args m recfuns in
      HExtlib. list_iter_default2
       (fun x b → if not b then aux k x) tl false fixed_args
```

In the case of an application of a fixpoint which has already been unfolded, and whose body we are traversing, the algorithm checks that any argument which has not been flagged as "fixed" is guarded by destructors.

```
  | C.Rel m →
     (match List.nth context (m−1) with
     | _,C.Decl _ → ()
     | _,C.Def (bo,_) → aux k (S.lift m bo))
  | C.Meta _ → ()
```

In the case of an index bound to a LetIn, we check that the body of the local definition is guarded. As for metavariables, what we should do is to constrain the set of terms with which a metavariable can be instantiated. Such a job would require some rework of our data structures, which we felt would be overkill. Instead, we just state that any metavariable, by itself, is guarded, even if some possible instantiations of it are not. This does not hamper the logical consistency of the system in any way: the only drawback is that, when filling in a subproof, the user does not get any hints on the constraints of well-guardedness, therefore he might provide an unguarded term, which would initially be accepted by the refiner (outside the kernel), but would be rejected when fed to the kernel.

```
  | C.Appl (C.Const ((Ref.Ref (uri,Ref.Fix (i,recno,_))) as r ):: args) →
     if List.exists (fun t → try aux k t;false with NotGuarded _ → true) args
     then
     let fl ,_,_ = E. get_checked_fixes_or_cofixes r in
     let ctx_tys, bos =
       List.split (List.map (fun (_,name,_,ty,bo) → (name, C.Decl ty), bo) fl)
     in
     let fl_len = List.length fl in
     let bos = List.map (debruijn uri fl_len context) bos in
     let j = List. fold_left min max_int (List.map (fun (_,_,i,_,_)→ i) fl) in
     let ctx_len = List.length context in
       (* we may look for fixed params not only up to j ... *)
     let fa = fixed_args bos j ctx_len (ctx_len + fl_len) in
     HExtlib. list_iter_default2
      (fun x b → if not b then aux k x) args false fa;
     let context = context@ctx_tys in
     let ctx_len = List.length context in
     let extra_recfuns =
       HExtlib.list_mapi (fun _ i → ctx_len − i, UnfFix fa) ctx_tys
     in
     let new_k = context, extra_recfuns@recfuns, x in
     let bos_and_ks =
       HExtlib.list_mapi
        (fun bo fno →
         let bo_and_k =
           eat_or_subst_lambdas ~subst ~metasenv j bo fa args new_k
         in
          if
           fno = i &&
           List.length args > recno &&
           (*case where the recursive argument is already really_smaller *)
          is_really_smaller  r_uri  r_len  ~subst ~metasenv k
           (List.nth args recno)
        then
         let bo,(context, _, _ as new_k) = bo_and_k in
         let bo, context' =
          eat_lambdas ~subst ~metasenv context (recno + 1 - j) bo in
         let new_context_part,_ =
          HExtlib.split_nth (List.length context' - List.length context)
           context' in
         let k = List.fold_right  shift_k  new_context_part new_k in
         let context, recfuns, x = k in
         let k = context, (1,Safe)::recfuns, x in
           bo,k
        else
         bo_and_k
      ) bos
  in
   List.iter (fun (bo,k) -> aux k bo) bos_and_ks
```

When we are checking the application of a fixpoint definition, and some of the arguments are not guarded, we perform a deep analysis by unfolding the fixpoint. We keep track of which arguments are used by the fixpoint as “fixed” parameters, and we check recursively that those which are not fixed are guarded. Then we extend the recfuns structure by adding the indices referring to the recursive functions in the unfolded fixpoint, associated to the list of fixed parameters. Finally we check that the body of any recursive function defined in the unfolded fixpoint in appropriately updated context and recfuns.

```
 | C.Match (Ref.Ref (uri,Ref.Ind (true,_,_)),outtype,term,pl) as t ->
    (match R.whd ~subst context term with
    | C.Rel m | C.Appl (C.Rel m :: _ ) as t when is_safe m recfuns || m = x ->
        let ty = typeof ~subst ~metasenv context term in
        let dc_ctx, dcl, start, stop =
          specialize_and_abstract_constrs ~subst r_uri r_len context ty in
        let args = match t with C.Appl (_::tl) -> tl | _ -> [] in
        aux k outtype;
        List.iter (aux k) args;
        List.iter2
          (fun p (_,dc) ->
            let rl = recursive_args ~subst ~metasenv dc_ctx start stop dc in
            let p, k = get_new_safes ~subst k p rl in
            aux k p)
          pl dcl
    | _ -> recursor aux k t)
```

When we encounter a case analysis on a term of an inductive type (say term is the term being matched and ty its type) and term is (an application of) the seed or a safe index, term and the outtype of the case analysis must be guarded (recursively) and for each pattern p = Lambda (x1,t1, (... Lambda (xm,tm,v) ...)) in the pattern list pl, v should be guarded by destructors under an extended list of safe indices, comprising the indices referring to each recursive parameter of the corresponding constructor of ty; this is the case where new safe indices are created, by means of the procedures recursive_args and get_new_safes.

```
 | t -> recursor aux k t
 with
  NotGuarded _ as exc ->
```

```
    let t' = R.whd ~delta:0 ~subst context t in
    if t = t' then raise exc
    else aux k t'
 in
  try aux (context, recfuns, 1) t
  with NotGuarded s → raise (TypeCheckerFailure s)
```

In any other case, the check is propagated through the use of a recursor. In case of failure, as a last resort, the check is repeated on the weak head normal form of the term: this allows to capture a strictly larger set of terms, and it cannot be easily dispensed off. As a major drawback, it accepts also terms that are only weakly normalizing under a call-by-need strategy, since not guarded diverging calls can be passed to affine abstractions. In any case, the system remains logically consistent.

Two key tasks in the procedure are checking that a recursive call receives a smaller argument, and updating the list of safe indices according to each pattern of a Match. The first task is performed by the is_really_smaller procedure, that checks if a term, possibly fed with arguments, will always return smaller arguments.

```
and is_really_smaller
  r_uri r_len ~subst ~metasenv (context, recfuns, x as k) te
=
 match R.whd ~subst context te with
 | C.Rel m when is_safe m recfuns → true
```

If the term is a safe index, then it is really smaller.

```
 | C.Lambda (name, s, t) →
     is_really_smaller r_uri r_len ~subst ~metasenv (shift_k (name,C.Decl s) k) t
 | C.Appl (he::_) →
     is_really_smaller r_uri r_len ~subst ~metasenv k he
 | C.Rel _
 | C.Const (Ref.Ref (_,_,Ref.Con _)) → false
 | C.Appl []
 | C.Const (Ref.Ref (_,_,Ref.Fix _)) → assert false
 | C.Meta _ → true
```

In the case of a lambda abstraction, its target must be really smaller. In the case of an application, then it is sufficient for its head to be really smaller.

Inductive type constructors and non-safe indices are not really smaller. In the case of a metavariable, just like inside guarded_by_destructors, we temporarily accept any metavariable, accepting the possibility that the kernel will reject the subsequent instantiation.

```
 | C.Match (Ref.Ref (uri,Ref.Ind (isinductive,_,_)),outtype,term,pl) →
    (match term with
    | C.Rel m | C.Appl (C.Rel m :: _ ) when is_safe m recfuns || m = x →
        if not isinductive then
          List.for_all (is_really_smaller r_uri r_len ~subst ~metasenv k) pl
        else
          let ty = typeof ~subst ~metasenv context term in
          let dc_ctx, dcl, start, stop =
             specialize_and_abstract_constrs ~subst r_uri r_len context ty in
          List.for_all2
           (fun p (_,dc) →
              let rl = recursive_args ~subst ~metasenv dc_ctx start stop dc in
              let e, k = get_new_safes ~subst k p rl in
               is_really_smaller r_uri r_len ~subst ~metasenv k e)
          pl dcl
    | _ → List.for_all ( is_really_smaller  r_uri  r_len  ~subst ~metasenv k) pl)
 | _ → assert false
```

Pattern matching is really smaller if it always produces a really smaller result in each branch. If the matched term is the seed or it is safe, we extend the list of safe indices as we did in guarded_by_destructors.

The term should always fall in one of the categories above. If it doesn't, it must be a product or a sort (meaning that its type is neither an inductive type, nor a function returning an inductive type) or an implicit (which should never reach the kernel). Therefore, we raise assert false.

The other critical task is to understand which indices should be added to the safe list when matching the seed or another safe index. Informally, for each pattern in the case analysis, we add exactly those indices, corresponding to the arguments of a constructor, whose type is mutually defined with the type of the term being matched.

```
and recursive_args ~subst ~metasenv context n nn te =
  match R.whd context te with
  | C.Rel _ | C.Appl _ | C.Const _ → []
  | C.Prod (name,so,de) →
     (not (does_not_occur ~subst context n nn so)) ::
      ( recursive_args ~subst ~metasenv
        ((name,(C.Decl so))::context) (n+1) (nn + 1) de)
  | t →
     raise (AssertFailure (lazy ("recursive_args:" ^ PP.ppterm ~subst
     ~metasenv ~context:[] t )))
```

The recursive_args procedure takes the type of a constructor and returns a list of booleans whose length is equal to the arity of the constructor: each boolean is true if the corresponding argument is recursive, false otherwise.

```
and get_new_safes ~subst (context, recfuns, x as k) p rl =
  match R.whd ~subst context p, rl with
  | C.Lambda (name,so,ta), b::tl →
      let recfuns = (if b then [0,Safe] else []) @ recfuns in
      get_new_safes ~subst
        ( shift_k (name,(C.Decl so)) (context, recfuns, x)) ta tl
  | C.Meta _ as e, _ | e, [] → e, k
  | _ → raise (AssertFailure (lazy "Ill formed pattern"))
```

The get_new_safes procedure takes a pattern, the booleans obtained from recursive_args, and the current safe list: it returns the body of the pattern (i.e. the pattern stripped of the outer lambdas) and the updated safe list against which the body of the pattern must be checked.

In the case of functions defined by cofixpoint, we want to make sure that even if the function might return a coinductive term constructed by an infinite tree of applications of constructors, the system remains strongly normalizing. This is made possible by a lazy unfolding strategy, performed only when the application of a corecursive function is observed by means of case analysis: provided that a corecursive call appears only inside a constructor, the function can be unfolded only a finite number of times, since the number of observations allowed on a term cannot be infinite, being produced by recursive functions.

Ideally, the guarded by constructors check, which is dual to the guarded by destructors check for recursive functions, should accept only all the terms whose head normal form is obtained in a finite number of steps and is the application of a constructor of some coinductive type. Because of undecidability, we are again obliged to propose a decidable syntactic criterion.

Let f be a corecursive function[11] and t a term in which f occurs. Under the assumption that f reduces to a constructor of a coinductive type, we need to decide if it is certain that t, for each substitution of its free variables with closed terms, will bring the constructor generated by f in head position. We call the set of all such terms t the *certain* (CE) set w.r.t. f and we approximate it by the following rules:

- $f\ t_1\ \dots\ t_n$ is in CE if f does not occur in $t_1\ \dots\ t_n$
- a case analysis whose branches are all in CE is in CE
- an abstraction whose body is in CE is in CE

The set of guarded by constructors (GC) terms w.r.t. a corecursive function f is thus defined by the following rules:

- t is in GC if f does not occurr in t
- a term $k\ t_1 \dots t_n$ where k is a constructor of a coinductive type I with m left parameters is in GC if one of the following holds for each non-left argument t_i (i.e. when $m < i$):
  - f does not occur in t_i
  - t_i is in CE w.r.t. f
- a case analysis whose branches are all in GC is in GC
- an abstraction whose body is in GC is in CG if f does not occur in the type of the abstracted variable

Since the rules for the CE and GC sets are quite similar, we have factorized both checks in the procedure guarded_by_constructors. The h parameter of the auxiliary function aux is used to distinguish between the two semantics. Concretely, it records whether a constructor has been (just) crossed.

```
and guarded_by_constructors ~subst ~metasenv context t indURI indlen nn =
 let rec aux context n nn h te =
  match R.whd ~subst context te with
   | C.Rel m when m > n && m <= nn → h
   | C.Rel _ | C.Meta _ → true
```

We always perform the analysis on the weak head normal form of the term: this captures more recursive functions at the price of accepting terms which are only weakly normalizing under a call-by-need strategy.

When checking an index corresponding to a corecursive definition belonging to the current cofixpoint, we return h, i.e. true if we have crossed a constructor. Other indices and metavariables are always guarded.

```
   | C.Sort _
   | C.Implicit _
   | C.Prod _
   | C.Const (Ref.Ref (_,_,Ref.Ind _))
   | C.LetIn _ → raise (AssertFailure (lazy "17"))
```

Since the term is in weak head normal form, sorts, implicit terms, products, inductive types or local definitions are never considered by the algorithm.

[11] 0-ary function are also accepted

```
  | C.Lambda (name,so,de) →
     does_not_occur ~subst context n nn so &&
     aux ((name,C.Decl so)::context) (n + 1) (nn + 1) h de
```

In the case of a lambda-abstraction, the source must not contain any corecursive call in the source, and the target must be guarded by constructors.

```
  | C.Appl ((C.Rel m)::tl) when m > n && m <= nn →
     h && List.for_all (does_not_occur ~subst context n nn) tl
  | C.Const (Ref.Ref (_,_,Ref.Con _)) → true
```

A corecursive call must appear under a constructor, and all of its arguments must be guarded by constructors. (Co)-inductive constructors are guarded.

```
  | C.Appl (C.Const (Ref.Ref (uri, Ref.Con (_,j,paramsno))) :: tl) as t →
     let ty_t = typeof ~subst ~metasenv context t in
     let dc_ctx, dcl, start, stop =
        specialize_and_abstract_constrs ~subst indURI indlen context ty_t in
     let _, dc = List.nth dcl (j−1) in
     let rec_params = recursive_args ~subst ~metasenv dc_ctx start stop dc in
     let rec analyse_instantiated_type rec_spec args =
      match rec_spec, args with
      | h::rec_spec, he::args →
          aux context n nn h he && analyse_instantiated_type rec_spec args
      | _,[] → true
      | _ → raise (AssertFailure (lazy
        ("Too many args for constructor: " ^ String.concat " "
        (List.map (fun x→ PP.ppterm ~subst ~metasenv ~context x) args))))
     in
     let left, args = HExtlib.split_nth paramsno tl in
     analyse_instantiated_type rec_params args
```

If we are checking an application of a constructor, we ensure that no corecursive calls appear in the left parameters of the constructor and that all the arguments are guarded. In this recursive check, h is set to true if the argument we are considering corresponds to a recursive parameter of the constructor, false otherwise.

```
  | C.Appl ((C.Match (_,out,te,pl))::_)
  | C.Match (_,out,te,pl) as t →
      let tl = match t with C.Appl (_::tl) → tl | _ → [] in
      List.for_all (does_not_occur ~subst context n nn) tl &&
      does_not_occur ~subst context n nn out &&
      does_not_occur ~subst context n nn te &&
      List.for_all (aux context n nn h) pl
```

When considering a case analysis or an applied case analysis, we check that recursive calls do not appear in the term being matched, in the return type or in the possible arguments, and that all the patterns be guarded by constructors.

```
  | C.Const _
  | C.Appl _ as t → does_not_occur ~subst context n nn t
in
  aux context 0 nn false t
```

If the term being checked is any constant or application different from the ones considered above, corecursive calls must not occur in the term.

7 Conclusions

The actual writing of the new kernel was done in four months, by four people, for a total effort of about 10 men months[12].

Half of the code has been rewritten from scratch, while the remaining part has been adapted to the new data structures. In the process, we have improved on a large number of small design decisions that had a major cumulative impact on the complexity of the old kernel. We have also been able to better delimit the trusted parts by abstracting them on code (mostly I/O code) that does not need to be trusted. The trusted and untrusted functionalities were much more interleaved in the old kernel and difficult to separate. This was the consequence of the evolution of the system and the temptation to pollute the kernel with minor additional functionalities operating on CIC-terms, originally meant for extra-kernel usages, but eventually ending up to be invoked in the kernel too.

At the end, we were able to halve the size of the code and the number of functions making up the kernel interface, as the following table shows.

	New kernel	Old kernel	Coq †	Coq
source size	2300	5000	7900	11400
exported functions	38	75	524	647

The last column is the dimension of the kernel of Coq 8.1[13] to which we should also add about 1600 lines of C code implementing an optimized reduction machine. However, the Coq kernel offers additional features (like a module system) that - deliberately - are not implemented in Matita. The column marked with † is our, inevitably rough, attempt to restrict the kernel to the same set of functionalities supported by Matita. The datum is only indicative, but is just meant to give evidence that the new kernel is *really* small.

The main differences between the two versions of CIC implemented in Coq and Matita are:

- Recursive and co-recursive functions that are arbitrary terms in Coq (and in our old kernel) and that we admit only as top level definitions. Thanks to λ-lifting, the expressive power of the calculus is not affected, but more sophisticated termination checks are required.
- Syntactic termination checks, that are more liberal in the new kernel of Matita with respect to our old kernel and the one of Coq.
- Universe inference that is implemented in Coq (and in our old kernel) and that we replaced with universe checking (in the spirit of the Explicit Polymorphic Extended Calculus of Constructions)
- Computationally irrelevant arguments can be associated to constants, avoiding their comparison in the conversion check. Neither Coq nor the old kernel had this feature.

The following table compares the latter functionalities.

	Old kernel	New kernel	Coq
universes	658	35	577
termination check	429	272	514

[12] This is in line with our analysis in Asperti et al (2006), where we estimated in 12 men months the time required to write a CIC-like kernel by a team of trained programmers.

[13] Coq is also written in OCaml, so comparing lines is fair.

We can remark that our old implementation and the one of Coq, which were based on similar data types, had similar sizes. In the case of termination checks, the smaller size of our new kernel can be partially justified by better designed data structures, that allowed us to collapse many similar cases, and by the choice of dropping recursive functions at the term level. Universe checking is much simpler than universe inference despite having much nicer properties (Courant (2002)).

The type-checking performances of a kernel are mostly determined by the effectiveness of the reduction and conversion heuristics (Sacerdoti Coen (2007)) that may avoid unnecessary reduction or that may reduce the size of the terms which are compared. In our tests, the new and old kernel have roughly the same typechecking performances, with some notable exceptions due to the constants height heuristics and to exploitation of proof irrelevance.

The present work is a first step towards the complete reimplementation of Matita. In particular, we expect that the new data structures will have a significant positive effect also on other parts of the code, starting from the refinement code that closely mimics the type-checking code. Indeed, the new representation of objects that have both a metavariable and a substitution have already been thought to simplify the data structures used outside the kernel to represent proof progress via the Curry-Howard isomorphism.

Having reduced the size of the kernel, we would like to explore the possibility of introducing some more complexity by extending the pattern matching construct of CIC to handle nested patterns, and default patterns. These patterns are currently accepted by the system, but they are represented internally by nesting multiple pattern matching constructs and by duplicating patterns. The obtained term can be exponentially larger than the original one, with a major impact on type-checking and conversion performances. Moreover, being quite different from the one inserted by the user, it requires more extra-logical information for pretty-printing it back in a readable form. Thus we feel that it is worthwhile to modify the kernel, even at the price of adding some complexity to the data type used to represent terms.

Another future research direction for the kernel would be to substitute the current syntactic termination checks with a type system based on size types Abel (2004). Although some experiments in this direction has already been performed Barthe et al (2006), some work is still needed to have a usable and user-friendly system for the Calculus of Inductive Constructions.

Finally, a compact kernel paves the way to a formalization of the correctness of the implementation. Experiments in this direction has already been attempted for a subset of the Calculus of Inductive Constructions Barras (1999). Although a complete formalization is surely worthwhile, at present our main interest would be in the formalization of the most error prone checks only, which comprise the termination checks for recursive and co-recursive functions and the positivity conditions for inductive types.

References

Abel A (2004) Termination checking with types. Theoretical informatics and applications 38:277–319

Asperti A (1992) A categorical understanding of environment machines. J Funct Program 2(1):23–59

Asperti A, Ricciotti W (2008) About the formalization of some results by chebyshev in number theory. Invited talk at TYPES'08, Torino, Italy

Asperti A, Sacerdoti Coen C, Tassi E, Zacchiroli S (2006) Crafting a proof assistant. In: Proceedings of Types 2006: Conference of the Types Project. Nottingham, UK – April 18-21, Springer-Verlag, Lecture Notes in Computer Science, vol 4502, pp 18–32

Barendregt H (1992) Lambda Calculi with Types. In: Abramsky, Samson and others (ed) Handbook of Logic in Computer Science, vol 2, Oxford University Press

Barras B (1999) Auto-validation d'un système de preuves avec familles inductives. Thèse de doctorat, Université Paris 7

Barthe G, Ruys M, Barendregt H (1995) A two-level approach towards lean proof-checking. In: Types for Proofs and Programs (Types 1995), Springer-Verlag, LNCS, vol 1158, pp 16–35

Barthe G, Grégoire B, Pastawski F (2006) Type-based termination of recursive definitions in the Calculus of Inductive Constructions. In: Proceedings of the 13th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR'06), Springer-Verlag, Lecture Notes in Artificial Intelligence, to appear

van Benthem Jutting L, McKinna J, Pollack R (1994) Checking algorithms for Pure Type Systems. In: Barendregt, Nipkow (eds) TYPES'93: Workshop on Types for Proofs and Programs, Selected Papers, Springer-Verlag, LNCS, vol 806, pp 19–61, URL http://homepages.inf.ed.ac.uk/rpollack/export/JMPchecking.ps.gz

Bertot Y, Castéran P (2004) Interactive Theorem Proving and Program Development. Texts in Theoretical Computer Science, Springer Verlag, iSBN-3-540-20854-2

Boutin S (1997) Using reflection to build efficient and certified decision procedures. In: Abadi M, editors TI (eds) Theoretical Aspect of Computer Software TACS'97, Lecture Notes in Computer Science, Springer-Verlag, vol 1281, pp 515–529

Courant J (2002) Explicit universes for the calculus of constructions. In: Theorem Proving in Higher Order Logics: 15th International Conference, pp 115–130

Crégut P (1990) An abstract machine for lambda-terms normalization. In: LISP and Functional Programming, pp 333–340

Crégut P (2007) Strongly reducing variants of the krivine abstract machine. Higher-Order and Symbolic Computation 20(3):209–230

Danos V, Regnier L (2003) How abstract machines implement head linear reduction, submitted for publication

Dybjer P (1997) Inductive families. Formal Aspects of Computing 6(4):440–465

Geuvers H (1993) Logics and Type Systems. Ph.D. dissertation, Catholic University Nijmegen

Geuvers H, Jojgov GI (2002) Open proofs and open terms: A basis for interactive logic. In: Bradfield J (ed) Computer Science Logic: 16th International Workshop, CSL 2002, Springer-Verlag, Lecture Notes in Computer Science, vol 2471, pp 537–552

Giménez E (1998) Structural recursive definitions in type theory. In: ICALP, pp 397–408

Gonthier G (2005) A computer-checked proof of the four-colour theorem. Available at http://research.microsoft.com/ gonthier/4colproof.pdf

Grégoire B (2003) Compilation des termes de preuves: un (nouveau) mariage entre Coq et ocaml. Thése de doctorat, spécialité informatique, Université Paris 7, école Polytechnique, France, URL http://www-sop.inria.fr/everest/personnel/Benjamin.Gregoire/Publi/gregoire_these.ps.gz

Huet G, Kahn G, Paulin-Mohring C (1998) The Coq Proof Assistant. A Tutorial

Johnsson T (1985) Lambda lifting: Transforming programs to recursive equations. In: Proc. of Functional programming languages and computer architecture. Nancy, France, Sept 1985.

Luo Z (1990) An Extended Calculus of Constructions. PhD thesis, University of Edinburgh

McBride C (1999) Dependently typed functional programs and their proofs. PhD thesis, University of Edinburgh

Miquel A, Werner B (2003) The not so simple proof-irrelevant model of CC. In: Geuvers H, Wiedijk F (eds) Types for Proofs and Programs: International Workshop, TYPES 2002, Springer-Verlag, Lecture Notes in Computer Science, vol 2646, pp 240–258

Muoz C (1997) A calculus of substitutions for incomplete-proof representation in type theory. PhD thesis, INRIA

Paulin-Mohring C (1996) Définitions inductives en théorie des types d'ordre supŕieur. Habilitation à diriger les recherches, Université Claude Bernard Lyon I, URL http://www.lri.fr/ paulin/habilitation.ps.gz

Peyton-Jones SL (1987) The Implementation of Functional Programming Languages. Prentice-Hall

Pollack R (1994) The theory of lego: A proof checker for the extended calculus of constructions. PhD thesis, PhD thesis, Univ. of Edinburgh

Sacerdoti Coen C (2004a) Mathematical knowledge management and interactive theorem proving. PhD thesis, University of Bologna, technical Report UBLCS 2004-5

Sacerdoti Coen C (2004b) Mathematical libraries as proof assistant environments. In: Andrea Asperti AT Grzegorz Bancerek (ed) Proceedings of Mathematical Knowledge Management 2004, Springer-Verlag, Lecture Notes in Computer Science, vol 3119, pp 332–346

Sacerdoti Coen C (2007) Reduction and conversion strategies for the calculus of (co)inductive constructions: Part i. In: Proceedings of the Sixth International Workshop on Reduction Strategies in Rewriting and Programming, Elsevier, ENTCS, vol 174, pp 97–118

Werner B (1994) Une théorie des Constructions Inductives. PhD thesis, Université Paris VII

Werner B (1997) Sets in types, types in sets. In: Abadi M, editors TI (eds) Theoretical Aspect of Computer Software TACS'97, Lecture Notes in Computer Science, Springer-Verlag, vol 1281, pp 530–546

Werner B (2008) Faire simple pour pouvoir faire compliqué. contributions à une théorie des types pratique. Habilitation à diriger les recherches, Université Paris sud, URL http://www.lix.polytechnique.fr/Labo/Benjamin.Werner/annonceHDR.html

Wiedijk F (2006) The seventeen provers of the world. LNAI 3600

A Syntax-directed type-checking rules

In this section, $\mathcal{I}$ will be short for

$$\begin{array}{l}\Pi x_1 : U_1 \ldots \Pi x_l : U_l.\\ \quad \{\textbf{inductive } I_1 : A_1 := c_1^1 : C_1^1 \ldots C_1^{m_1} : T_1^{m_1}\\ \quad \textbf{with} \ldots\\ \quad \textbf{with } I_n : A_n := c_n^1 : C_n^1 \ldots c_n^{m_n} : C_n^{m_n}\}\end{array}$$

A.1 Environment formation rules (judgement $E \vdash WF$, function typecheck_obj)

$$\frac{}{\emptyset \vdash WF}$$

$$\frac{\begin{array}{c} E \vdash WF \qquad d \text{ undefined in } E \qquad E, \Sigma \vdash WF \qquad E, \Sigma, \Phi \vdash WF \\ E, \Sigma, \Phi, \emptyset \vdash T : S \qquad E, \Sigma, \Phi, \emptyset \vdash S \rhd_{\text{whd}} S' \text{ where } S' \text{ is a sort} \\ E, \Sigma, \Phi, \emptyset \vdash b : T' \qquad E, \Sigma, \Phi, \emptyset \vdash T \downarrow T' \end{array}}{E \cup \langle \Sigma, \Phi, \mathbf{definition}\ d : T := b \rangle \vdash WF}$$

$$\frac{\begin{array}{c} E \vdash WF \qquad d \text{ undefined in } E \qquad E, \Sigma \vdash WF \qquad E, \Sigma, \Phi \vdash WF \\ E, \Sigma, \Phi, \emptyset \vdash T : S \qquad E, \Sigma, \Phi, \emptyset \vdash S \rhd_{\text{whd}} S' \text{ where } S' \text{ is a sort} \end{array}}{E \cup \langle \Sigma, \Phi, \mathbf{axiom}\ d : T \rangle \vdash WF}$$

$$\frac{\begin{array}{c} E \vdash WF \qquad f_1, \ldots, f_n \text{ undefined in } E \qquad E, \Sigma \vdash WF \qquad E, \Sigma, \Phi \vdash WF \\ E, \Sigma, \Phi, \emptyset \vdash T_i : S_i \qquad E, \Sigma, \Phi, \emptyset \vdash S_i \rhd_{\text{whd}} S'_i \text{ where } S'_i \text{ is a sort} \\ E, \Sigma, \Phi, [f_1 : T_1; \ldots; f_n : T_n] \vdash b_i : T'_i \\ E, \Sigma, \Phi, [f_1 : T_1; \ldots; f_n : T_n] \vdash T_i \downarrow T'_i \qquad b_1, \ldots, b_n \text{ guarded by destructors (Sect. 6.3)} \end{array}}{E \cup \langle \Sigma, \Phi, \mathbf{let\ rec}\ f_1 : T_1 := b_1\ \mathbf{and}\ \ldots\ \mathbf{and}\ f_n : T_n := b_n \rangle \vdash WF}$$

$$\frac{\begin{array}{c} E \vdash WF \qquad f_1, \ldots, f_n \text{ undefined in } E \qquad E, \Sigma \vdash WF \qquad E, \Sigma, \Phi \vdash WF \\ E, \Sigma, \Phi, \emptyset \vdash T_i : S_i \qquad E, \Sigma, \Phi, \emptyset \vdash S_i \rhd_{\text{whd}} S'_i \text{ where } S'_i \text{ is a sort} \\ E, \Sigma, \Phi, [f_1 : T_1; \ldots; f_n : T_n] \vdash b_i : T'_i \\ E, \Sigma, \Phi, [f_1 : T_1; \ldots; f_n : T_n] \vdash T_i \downarrow T'_i \qquad b_1, \ldots, b_n \text{ guarded by constructors (Sect. 6.3)} \end{array}}{E \cup \langle \Sigma, \Phi, \mathbf{let\ corec}\ f_1 : T_1 := b_1\ \mathbf{and}\ \ldots\ \mathbf{and}\ f_n : T_n := b_n \rangle \vdash WF}$$

$$\frac{\begin{array}{c} E \vdash WF \qquad I_1, \ldots, I_n, c_1^1, \ldots, c_n^{m_n} \text{ undefined in } E \qquad E, \Sigma \vdash WF \qquad E, \Sigma, \Phi \vdash WF \\ \text{all the conditions in Sect. 6.1 are satisfied} \end{array}}{E \cup \langle \Phi, \Sigma, \mathcal{I} \rangle \vdash WF}$$

A.2 Metasenv formation rules (judgement $E, \Sigma \vdash WF$, function typecheck_metasenv)

$$\frac{}{E, \emptyset \vdash WF}$$

$$\frac{\begin{array}{c} E, \Sigma \vdash WF \qquad ?_i \text{ undefined in } \Sigma \qquad E, \Sigma, \emptyset, \Gamma \vdash WF \\ E, \Sigma, \emptyset, \Gamma \vdash T : S \qquad E, \Sigma, \emptyset, \Gamma \vdash S \rhd_{\text{whd}} S' \text{where } S' \text{ is a sort} \end{array}}{E, \Sigma \cup \langle \Gamma \vdash ?_i : T \rangle \vdash WF}$$

A.3 Subst formation rules (judgement $E, \Sigma, \Phi \vdash WF$, function typecheck_subst)

$$\frac{}{E, \Sigma, \emptyset \vdash WF}$$

$$\frac{\begin{array}{c} E, \Sigma, \Phi \vdash WF \qquad ?_i \text{ undefined in } \Sigma \text{ and in } \Phi \qquad E, \Sigma, \Phi, \Gamma \vdash WF \\ E, \Sigma, \Phi, \Gamma \vdash T : S \qquad E, \Sigma, \Phi, \Gamma \vdash S \rhd_{\text{whd}} S' \text{ where } S' \text{ is a sort} \\ E, \Sigma, \Phi, \Gamma \vdash t : T' \qquad E, \Sigma, \Phi, \Gamma \vdash T \downarrow T' \end{array}}{E, \Sigma, \Phi \cup \langle \Gamma \vdash ?_i : T := t \rangle \vdash WF}$$

A.4 Context formation rules (judgement $E, \Sigma, \Phi, \Gamma \vdash WF$, function typecheck_context)

$$\frac{}{E, \Sigma, \Phi, \emptyset \vdash WF}$$

$$\frac{\begin{array}{c}E,\Sigma,\Phi,\Gamma \vdash WF \qquad x \text{ is undefined in } \Gamma \\ E,\Sigma,\Phi,\Gamma \vdash T : S \qquad E,\Sigma,\Phi,\Gamma \vdash S \rhd_{\text{whd}} S' \text{ where } S' \text{ is a sort}\end{array}}{E,\Sigma,\Phi,\Gamma \cup \langle x : T\rangle \vdash WF}$$

$$\frac{\begin{array}{c}E,\Sigma,\Phi,\Gamma \vdash WF \qquad x \text{ is undefined in } \Gamma \\ E,\Sigma,\Phi,\Gamma \vdash T : S \qquad E,\Sigma,\Phi,\Gamma \vdash S \rhd_{\text{whd}} S' \text{ where } S' \text{ is a sort} \\ E,\Sigma,\Phi,\Gamma \vdash t : T' \qquad E,\Sigma,\Phi,\Gamma \vdash T \downarrow T'\end{array}}{E,\Sigma,\Phi,\Gamma \cup \langle x : T := t\rangle \vdash WF}$$

A.5 Term typechecking rules (judgement $E,\Sigma,\Phi,\Gamma \vdash t : T$, function typeof)

$$\frac{\langle x : T\rangle \in \Gamma}{E,\Sigma,\Phi,\Gamma \vdash x : T}$$

$$\frac{\langle x : T := t\rangle \in \Gamma}{E,\Sigma,\Phi,\Gamma \vdash x : T}$$

$$\frac{}{E,\Sigma,\Phi,\Gamma \vdash \text{Type}_u : \text{Type}_{u+1}}$$

$$\frac{}{E,\Sigma,\Phi,\Gamma \vdash \text{Prop} : \text{Type}_0}$$

$$\frac{\begin{array}{c}\langle \Gamma' \vdash ?_i : T\rangle \in \Sigma \\ \text{check\_metasenv\_consistency } ?_i[lc]\ \Phi\ \Sigma\ \Gamma\ \Gamma'\ lc \text{ ok (Sect. 6)}\end{array}}{E,\Sigma,\Phi,\Gamma \vdash ?_i[lc] : T[lc]}$$

$$\frac{\begin{array}{c}\langle \Gamma' \vdash ?_i : T := t\rangle \in \Phi \\ \text{check\_metasenv\_consistency } ?_i[lc]\ \Phi\ \Sigma\ \Gamma\ \Gamma'\ lc \text{ ok (Sect. 6)}\end{array}}{E,\Sigma,\Phi,\Gamma \vdash ?_i[lc] : T[lc]}$$

$$\frac{\begin{array}{c}\langle \Sigma',\Phi',\textbf{definition } d : T := b\rangle \in E \\ \Sigma' = \emptyset \qquad \Phi' = \emptyset\end{array}}{E,\Sigma,\Phi,\Gamma \vdash d : T}$$

$$\frac{\begin{array}{c}\langle \Sigma',\Phi',\textbf{axiom } d : T\rangle \in E \\ \Sigma' = \emptyset \qquad \Phi' = \emptyset\end{array}}{E,\Sigma,\Phi,\Gamma \vdash d : T}$$

$$\frac{\begin{array}{c}\langle \Sigma',\Phi',\textbf{let rec } f_1 : T_1 := b_1 \textbf{ and } \ldots \textbf{ and } f_n : T_n := b_n\rangle \in E \\ \Sigma' = \emptyset \qquad \Phi' = \emptyset \qquad 1 \leq i \leq n\end{array}}{E,\Sigma,\Phi,\Gamma \vdash f_i : T_i}$$

$$\frac{\begin{array}{c}\langle \Sigma',\Phi',\textbf{let corec } f_1 : T_1 := b_1 \textbf{ and } \ldots \textbf{ and } f_n : T_n := b_n\rangle \in E \\ \Sigma' = \emptyset \qquad \Phi' = \emptyset \qquad 1 \leq i \leq n\end{array}}{E,\Sigma,\Phi,\Gamma \vdash f_i : T_i}$$

$$\frac{\begin{array}{c}\langle \Sigma',\Phi',\mathcal{I}\rangle \in E \\ \Sigma' = \emptyset \qquad \Phi' = \emptyset \qquad 1 \leq k \leq n\end{array}}{E,\Sigma,\Phi,\Gamma \vdash I_k : \Pi x_1 : U_1 \ldots \Pi x_l : U_l.A_k}$$

$$\frac{\begin{array}{c}\langle \Sigma',\Phi',\mathcal{I}\rangle \in E \\ \Sigma' = \emptyset \qquad \Phi' = \emptyset \qquad 1 \leq k \leq n \qquad 1 \leq j \leq m_k\end{array}}{E,\Sigma,\Phi,\Gamma \vdash c_k^j : \Pi x_1 : U_1 \ldots \Pi x_l : U_l.C_k^j}$$

$$\frac{\begin{array}{c} E,\Sigma,\Phi,\Gamma \vdash s : S \\ E,\Sigma,\Phi,\Gamma \vdash S \rhd_{\mathrm{whd}} S' \qquad S' \text{ is a sort or a meta} \\ E,\Sigma,\Phi,\Gamma \cup \langle n : s\rangle \vdash t : T \end{array}}{E,\Sigma,\Phi,\Gamma \vdash \lambda n : s.t : \Pi n : s.T}$$

$$\frac{\begin{array}{c} E,\Sigma,\Phi,\Gamma \vdash s : S \\ E,\Sigma,\Phi,\Gamma \cup \langle n : s\rangle \vdash t : T \\ u = \text{sort\_of\_prod } \Sigma\ \Phi\ \Gamma\ (n,s)\ (S,T) \text{ (Sect. C)} \end{array}}{E,\Sigma,\Phi,\Gamma \vdash \Pi n : s.t : u}$$

$$\frac{\begin{array}{c} E,\Sigma,\Phi,\Gamma \vdash t : T' \\ E,\Sigma,\Phi,\Gamma \vdash T : S \qquad E,\Sigma,\Phi,\Gamma \vdash T \downarrow T' \\ E,\Sigma,\Phi,\Gamma \cup \langle x : T := t\rangle \vdash u : U \end{array}}{E,\Sigma,\Phi,\Gamma \vdash \textbf{let } (x : T) := t \textbf{ in } u : U\left[{}^{t}/{}_{x}\right]}$$

$$\frac{\begin{array}{c} E,\Sigma,\Phi,\Gamma \vdash h : \Pi x : T.U \\ E,\Sigma,\Phi,\Gamma \vdash t : T' \qquad E,\Sigma,\Phi,\Gamma \vdash T \downarrow T' \end{array}}{E,\Sigma,\Phi,\Gamma \vdash h\ t : U\left[{}^{t}/{}_{x}\right]}$$

$$\frac{E,\Sigma,\Phi,\Gamma \vdash (h\ t_1)\ t_2 \cdots t_n\ : T}{E,\Sigma,\Phi,\Gamma \vdash h\ t_1\ t_2 \cdots t_n\ : T}$$

$$\frac{\begin{array}{c} \langle \Sigma',\Phi',\mathcal{I}\rangle \in E \qquad \Sigma' = \emptyset \qquad \Phi' = \emptyset \qquad E,\Sigma,\Phi,\Gamma \vdash t : T \\ E,\Sigma,\Phi,\Gamma \vdash T \rhd_{\mathrm{whd}} I_k\ t_1 \cdots t_l\ t'_1 \cdots t'_q \\ A_k = \Pi y_1 : Y_1 \ldots \Pi y_q : Y_q.s \qquad E,\Sigma,\Phi,\Gamma \vdash ot : os \\ \text{check\_allowed\_sort\_elimination } \Phi\ \Sigma\ I_k\ \Gamma\ (I_k\ t_1 \cdots t_l)\ A_k \left[{}^{t_1 \cdots t_l}/{}_{x_1 \cdots x_l}\right]\ os \text{ ok (Sect. 6)} \\ \text{for all } j = 1,\ldots,m_k\ E,\Sigma,\Phi,\Gamma \vdash p_j : T_j \\ \text{for all } j = 1,\ldots,m_k\ E,\Sigma,\Phi,\Gamma \vdash T_j \downarrow \Delta\{C^k_j \left[{}^{t_1 \cdots t_l}/{}_{x_1 \cdots x_l}\right], ot, (c^k_j\ t_1 \cdots\ t_l)\} \end{array}}{E,\Sigma,\Phi,\Gamma \vdash \textbf{match } t \textbf{ in } I_k \textbf{ return } ot \textbf{ with } [p_1|\ldots|p_{m_k}] : ot\ t'_1 \cdots t'_q\ t}$$

$$\Delta\{T,U,t\} = \begin{cases} U\ t'_1 \cdots t'_n\ t & \text{if } T = I\ t_1 \cdots t_l\ t'_1 \cdots t'_n, \\ & \text{where } I \text{ is an inductive type with } l \text{ left parameters} \\ \Pi x : T_1.\Delta\{T_2,U,t\ x\} & \text{if } T = \Pi x : T_1.T_2\} \end{cases}$$

A.6 Term conversion rules (judgement $E,\Sigma,\Phi,\Gamma \vdash T \downarrow T'$, function are_convertible; $\downarrow_=$ means test_eq_only = true; $\downarrow_\bullet$ means that the current rule must be intended as two rules, one with all the $\downarrow_\bullet$ replaced by $\downarrow$, the other with all the $\downarrow_\bullet$ replaced by $\downarrow_=$)

$$\frac{E,\Sigma,\Phi,\Gamma \vdash T =_\alpha T'}{E,\Sigma,\Phi,\Gamma \vdash T \downarrow_= T'}$$

$$\frac{E,\Sigma,\Phi,\Gamma \vdash T \downarrow_= T'}{E,\Sigma,\Phi,\Gamma \vdash T \downarrow T'}$$

$$\frac{\mathrm{Type}_u \leq \mathrm{Type}_v \qquad \mathrm{Type}_v \leq \mathrm{Type}_u \qquad \text{are declared constraints (Sect. 4.3)}}{E,\Sigma,\Phi,\Gamma \vdash \mathrm{Type}_u \downarrow_= \mathrm{Type}_v}$$

$$\frac{\mathrm{Type}_u \leq \mathrm{Type}_v \qquad \text{is a declared constraint (Sect. 4.3)}}{E,\Sigma,\Phi,\Gamma \vdash \mathrm{Type}_u \downarrow \mathrm{Type}_v}$$

$$\frac{}{E,\Sigma,\Phi,\Gamma \vdash \mathrm{Prop} \downarrow \mathrm{Type}_u}$$

$$\frac{\begin{array}{cc} lc = t_1, \ldots, t_n & lc' = t'_1, \ldots, t'_n \\ \text{for all } i = 1, \ldots, n & E, \Sigma, \Phi, \Gamma \vdash t_i \downarrow_\bullet t'_i \end{array}}{E, \Sigma, \Phi, \Gamma \vdash ?_j[lc] \downarrow_\bullet ?_j[lc']}$$

$$\frac{E, \Sigma, \Phi, \Gamma \vdash T_1 \downarrow_= T'_1 \qquad E, \Sigma, \Phi, \Gamma \cup \langle x : T_1 \rangle \vdash T_2 \downarrow_\bullet T'_2}{E, \Sigma, \Phi, \Gamma \vdash \Pi x : T_1.T_2 \downarrow_\bullet \Pi x : T'_1.T'_2}$$

$$\frac{E, \Sigma, \Phi, \Gamma \cup \langle x : T \rangle \vdash t \downarrow_\bullet t'}{E, \Sigma, \Phi, \Gamma \vdash \lambda x : T.t \downarrow_\bullet \lambda x : T'.t'}$$

In the rule above, no check is performed on the source of the abstractions, since we assume we are comparing well-typed terms whose types are convertible.

$$\frac{\begin{array}{c} E, \Sigma, \Phi, \Gamma \vdash h \downarrow_\bullet h' \\ \text{for all } i = 1, \ldots, n \qquad E, \Sigma, \Phi, \Gamma \vdash t_i \downarrow_= t'_i \end{array}}{E, \Sigma, \Phi, \Gamma \vdash h\ t_1 \cdots t_n \downarrow_\bullet h'\ t'_1 \cdots t'_n}$$

$$\frac{\begin{array}{cc} E, \Sigma, \Phi, \Gamma \vdash t_1 \downarrow_\bullet t_2 & E, \Sigma, \Phi, \Gamma \vdash ot_1 \downarrow_\bullet ot_2 \\ \text{for all } i = 1, \ldots, n & E, \Sigma, \Phi, \Gamma \vdash p_i \downarrow_\bullet p'_i \end{array}}{\begin{array}{l} E, \Sigma, \Phi, \Gamma \vdash \mathbf{match}\ t_1\ \mathbf{in}\ I\ \mathbf{return}\ ot_1\ \mathbf{with}\ [p_1|...|p_n] \downarrow_\bullet \\ \qquad\qquad \mathbf{match}\ t_2\ \mathbf{in}\ I\ \mathbf{return}\ ot_2\ \mathbf{with}\ [p'_1|...|p'_n] \end{array}}$$

$$\frac{E, \Sigma, \Phi, \Gamma \vdash t \rhd_{\text{whd}} t' \qquad E, \Sigma, \Phi, \Gamma \vdash u \rhd_{\text{whd}} u' \qquad E, \Sigma, \Phi, \Gamma \vdash t' \downarrow_\bullet u'}{E, \Sigma, \Phi, \Gamma \vdash t \downarrow_\bullet u}$$

In the previous rule, t' and u' need not be weak head normal forms: any term obtained by t (respectively, u) by reduction (even non head reduction) will do. Indeed, the less reduction is performed, the more efficient the conversion test usually is.

A.7 Term reduction rules

$$E, \Sigma, \Phi, \Gamma \vdash (\lambda x : T.u)\ t \ \rhd_\beta\ u\,[t/x]$$

$$E, \Sigma, \Phi, \Gamma \vdash \mathbf{let}\ (x : T) := t\ \mathbf{in}\ u \ \rhd_\zeta\ u\,[t/x]$$

$$\frac{\langle \emptyset, \emptyset, \mathbf{definition}\ d : T := b \rangle \in E}{E, \Sigma, \Phi, \Gamma \vdash d \ \rhd_\delta\ b}$$

$$\frac{\langle \Gamma' \vdash ?_i : T := t \rangle \in \Phi}{E, \Sigma, \Phi, \Gamma \vdash ?_i[lc] \ \rhd_\delta\ t[lc]}$$

$$E, \Sigma, \Phi, \Gamma \vdash \mathbf{match}\ c_k^i\ t_1 \cdots t_l\ t'_1 \cdots t'_n\ \mathbf{in}\ I_k\ \mathbf{return}\ ot\ \mathbf{with}\ [p_1|...|p_{m_k}] \ \rhd_\iota\ p_i\ t'_1\ ...\ t'_n$$

$$\frac{\langle \emptyset, \emptyset, \mathbf{let\ rec}\ f_1 : T_1 := b_1\ \mathbf{and}\ \ldots\ \mathbf{and}\ f_n : T_n := b_n \rangle \in E}{E, \Sigma, \Phi, \Gamma \vdash f_k\ t_1\ ...(c_j^i\ u_1\ ...\ u_{m_j})...\ t_m \ \rhd_\mu\ b_k\ t_1\ ...(c_j^i\ u_1\ ...\ u_{m_j})...\ t_m}$$

Note that $(c_j^i\ u_1\ ...\ u_{m_j})$ must occur in the position of the recursive argument of f_k. This implies that, for this reduction to be performed, f_k must be applied at least up to its recursive argument.

$$\frac{\langle \emptyset, \emptyset, \mathbf{let\ corec}\ f_1 : T_1 := b_1\ \mathbf{and}\ \ldots\ \mathbf{and}\ f_n : T_n := b_n \rangle \in E}{\begin{array}{l} E, \Sigma, \Phi, \Gamma \vdash \mathbf{match}\ f_k\ t_1\ ...\ t_q\ \mathbf{in}\ I_j\ \mathbf{return}\ ot\ \mathbf{with}\ [p_1|...|p_{m_j}] \rhd_\nu \\ \qquad\qquad \mathbf{match}\ b_k\ t_1\ ...\ t_q\ \mathbf{in}\ I_j\ \mathbf{return}\ ot\ \mathbf{with}\ [p_1|...|p_{m_j}] \end{array}}$$

Note that here q can be zero.

B Kernel interface

The following are the functions exported by the kernel modules. We omit a few uninteresting functions used only once to set up recursion between modules.

```
module type nUri = sig
  val string_of_uri : uri → string
  val name_of_uri: uri → string
  val uri_of_string : string → uri
  val eq: uri → uri → bool
end
module type nReference = sig
  val eq: reference → reference → bool
  val string_of_reference : reference → string
  val reference_of_string : string → reference
  val mk_constructor: int → reference → reference
  val mk_fix: int → int → reference → reference
  val mk_cofix: int → reference → reference
end
module type nCicUtils = sig
  val expand_local_context : NCic.lc_kind → NCic.term list
  val lookup_subst: int → NCic.substitution → NCic.subst_entry
  val lookup_meta: int → NCic.metasenv → NCic.conjecture
  val fold:
    (NCic.hypothesis → 'k → 'k) → 'k →
    ('k → 'a → NCic.term → 'a) → 'a → NCic.term → 'a
  val map:
   (NCic.hypothesis → 'k → 'k) → 'k →
   ('k → NCic.term → NCic.term) → NCic.term → NCic.term
end
module type nCicEnvironment = sig
  val type0: NCic.universe
  val universe_eq: NCic.universe → NCic.universe → bool
  val universe_leq: NCic.universe → NCic.universe → bool
  val add_constraint: bool → NCic.universe → NCic.universe → unit
  val invalidate: unit → unit
  val get_checked_obj: NUri.uri → NCic.obj
  (* specialized versions of get_checked_obj *)
  val get_relevance: NReference.reference → bool list
  val get_checked_def:
    NReference.reference →
      NCic.relevance * string * NCic.term * NCic.term * NCic.c_attr * int
  val get_checked_indtys:
    NReference.reference → bool * int * NCic.inductiveType list * NCic.i_attr * int
  val get_checked_fixes_or_cofixes :
    NReference.reference → NCic.inductiveFun list * NCic.f_attr * int
end
module type nCicSubstitution = sig
  val lift : ?from:int → int → NCic.term → NCic.term
  val subst : ?avoid_beta_redexes:bool → NCic.term → NCic.term → NCic.term
  val psubst :
    ?avoid_beta_redexes:bool → ('a → NCic.term) → 'a list → NCic.term → NCic.term
  val subst_meta : NCic.local_context → NCic.term → NCic.term
end
module type nCicReduction = sig
  val whd :
    ?delta:int → ?subst:NCic.substitution → NCic.context → NCic.term → NCic.term
  val are_convertible :
    ?subst:NCic.substitution → NCic.context → NCic.term → NCic.term → bool
  val head_beta_reduce: ?delta:int → ?upto:int → NCic.term → NCic.term
```

```
end
module type nCicTypeChecker = sig
  val typecheck_obj : NCic.obj → unit
  val typeof:
    subst:NCic.substitution → metasenv:NCic.metasenv →
      NCic.context → NCic.term → NCic.term
end
```

C Auxiliary functions

We do not extensively describe the following minor functions. We already gave intuition about them when we described the code where they are used. The functions are listed in alphabetical order.

```
(* check_metasenv_consistency checks that the "canonical" context of a
   metavariable is consitent − up to relocation via the relocation list l −
   with the actual context *)
and check_metasenv_consistency
  ~subst ~metasenv term context canonical_context l
=
 match l with
  | shift , C.Irl n →
      let context = snd (HExtlib.split_nth shift context) in
       let rec compare = function
        | 0,_,[] → ()
        | 0,_,_::_
        | _,_,[] →
           raise (AssertFailure (lazy (Printf.sprintf
            "Local and canonical context %s have different lengths"
            (PP.ppterm ~subst ~context ~metasenv term))))
        | m,[],_::_ →
           raise (TypeCheckerFailure (lazy (Printf.sprintf
            "Unbound variable −%d in %s" m
            (PP.ppterm ~subst ~metasenv ~context term))))
        | m,t::tl,ct::ctl →
           (match t,ct with
               (_,C.Decl t1), (_,C.Decl t2)
             | (_,C.Def (t1,_)), (_,C.Def (t2,_))
             | (_,C.Def (_,t1)), (_,C.Decl t2) →
                if not (R.are_convertible ~subst tl t1 t2) then
                 raise
                      (TypeCheckerFailure
                         (lazy (Printf.sprintf
                      ("Not well typed metavariable local context for %s: " ^^
                       "%s expected, which is not convertible with %s")
                      (PP.ppterm ~subst ~metasenv ~context term)
                      (PP.ppterm ~subst ~metasenv ~context t2)
                      (PP.ppterm ~subst ~metasenv ~context t1))))
             | _,_ →
              raise
                   (TypeCheckerFailure (lazy (Printf.sprintf
                    ("Not well typed metavariable local context for %s: " ^^
                     "a definition expected, but a declaration found")
                    (PP.ppterm ~subst ~metasenv ~context term)))));
           compare (m − 1,tl,ctl)
       in
        compare (n,context,canonical_context)
  | shift , lc_kind →
```

```
    (* we avoid useless lifting by shortening the context*)
    let l,context = (0,lc_kind), snd (HExtlib.split_nth shift context) in
    let lifted_canonical_context =
      let rec lift_metas i = function
        | [] -> []
        | (n,C.Decl t):: tl ->
            (n,C.Decl (S.subst_meta l (S. lift i t )))::( lift_metas (i+1) tl)
        | (n,C.Def (t,ty )):: tl ->
            (n,C.Def ((S.subst_meta l (S. lift i t )),
                       S.subst_meta l (S. lift i ty )))::( lift_metas (i+1) tl)
      in
        lift_metas 1 canonical_context in
    let l = U.expand_local_context lc_kind in
    try
     List . iter2
     (fun t ct ->
       match (t,ct) with
       | t, (_,C.Def (ct,_)) ->
          (*CSC: the following optimization is to avoid a possibly expensive
                 reduction that can be easily avoided and that is quite
                 frequent. However, this is better handled using levels to
                 control reduction *)
          let optimized_t =
           match t with
           | C.Rel n ->
               (try
                 match List.nth context (n - 1) with
                 | (_,C.Def (te,_)) -> S.lift n te
                 | _ -> t
                 with Failure _ -> t)
           | _ -> t
          in
          if not (R.are_convertible ~subst context optimized_t ct)
          then
            raise
              (TypeCheckerFailure
                (lazy (Printf. sprintf
                  ("Not well typed metavariable local context: " ^^
                   "expected a term convertible with %s, found %s")
                  (PP.ppterm ~subst ~metasenv ~context ct)
                  (PP.ppterm ~subst ~metasenv ~context t))))
       | t, (_,C.Decl ct) ->
           let type_t = typeof_aux context t in
           if not (R.are_convertible ~subst context type_t ct) then
             raise (TypeCheckerFailure
              (lazy (Printf. sprintf
               ("Not well typed metavariable local context: "^^
               "expected a term of type %s, found %s of type %s")
               (PP.ppterm ~subst ~metasenv ~context ct)
               (PP.ppterm ~subst ~metasenv ~context t)
               (PP.ppterm ~subst ~metasenv ~context type_t))))
     ) l lifted_canonical_context
    with
     Invalid_argument _ ->
       raise (AssertFailure (lazy (Printf. sprintf
        "Local and canonical context %s have different lengths"
        (PP.ppterm ~subst ~metasenv ~context term))))
```

```
let debruijn uri number_of_types context =
 let rec aux k t =
  match t with
   | C.Meta (i,(s,C.Ctx l)) →
      let l1 = HExtlib.sharing_map (aux (k−s)) l in
      if l1 == l then t else C.Meta (i,(s,C.Ctx l1))
   | C.Meta _ → t
   | C.Const (Ref.Ref (uri1,(Ref.Fix (no,_,_) | Ref.CoFix no)))
   | C.Const (Ref.Ref (uri1,Ref.Ind (_,no,_))) when NUri.eq uri uri1 →
      C.Rel (k + number_of_types − no)
   | t → U.map (fun _ k → k+1) k aux t
 in
  aux (List.length context)
```

```
let rec eat_lambdas ~subst ~metasenv context n te =
  match (n, R.whd ~subst context te) with
  | (0, _) → (te, context)
  | (n, C.Lambda (name,so,ta)) when n > 0 →
      eat_lambdas ~subst ~metasenv ((name,(C.Decl so))::context) (n − 1) ta
   | (n, te) →
      raise (AssertFailure (lazy (Printf.sprintf "eat_lambdas (%d, %s)" n
        (PP.ppterm ~subst ~metasenv ~context te))))
```

```
let rec eat_or_subst_lambdas ~subst ~metasenv n te to_be_subst args
      (context, recfuns, x as k)
=
  match n, R.whd ~subst context te, to_be_subst, args with
  | (n, C.Lambda (name,so,ta),true::to_be_subst,arg::args) when n > 0 →
      eat_or_subst_lambdas ~subst ~metasenv (n − 1) (S.subst arg ta)
       to_be_subst args k
  | (n, C.Lambda (name,so,ta),false::to_be_subst,arg::args) when n > 0 →
      eat_or_subst_lambdas ~subst ~metasenv (n − 1) ta to_be_subst args
       (shift_k (name,(C.Decl so)) k)
  | (_, te, _, _) → te, k
;;
```

```
let eat_prods ~subst ~metasenv context he ty_he args_with_ty =
  let rec aux ty_he = function
  | [] → ty_he
  | (arg, ty_arg)::tl →
      match R.whd ~subst context ty_he with
      | C.Prod (n,s,t) →
          if R.are_convertible ~subst context ty_arg s then
            aux (S.subst ~avoid_beta_redexes:true arg t) tl
          else
            raise
              (TypeCheckerFailure
                (lazy (Printf.sprintf
                  ("Appl: wrong application of %s: the parameter %s has type"^^
                   "\n%s\nbut it should have type \n%s\nContext:\n%s\n")
                  (PP.ppterm ~subst ~metasenv ~context he)
                  (PP.ppterm ~subst ~metasenv ~context arg)
                  (PP.ppterm ~subst ~metasenv ~context ty_arg)
                  (PP.ppterm ~subst ~metasenv ~context s)
                  (PP.ppcontext ~subst ~metasenv context))))
       | _ →
          raise
            (TypeCheckerFailure
              (lazy (Printf.sprintf
                "Appl: %s is not a function, it cannot be applied"
```

```
                  (PP.ppterm ~subst ~metasenv ~context
                   (let res = List.length tl in
                    let eaten = List.length args_with_ty − res in
                     (C.Appl
                      (he :: List .map fst
                       ( fst (HExtlib.split_nth eaten args_with_ty )))))))))
  in
    aux ty_he args_with_ty
```

```
let fixed_args bos j n nn =
 let rec aux k acc = function
   | C.Appl (C.Rel i::args) when i−k > n && i−k <= nn →
      let rec combine l1 l2 =
       match l1,l2 with
           [],[]  → []
         | he1 :: tl1 , he2 :: tl2 → (he1,he2)::combine tl1 tl2
         | he :: tl , [] → (false,C.Rel ~−1)::combine tl [] (∗ dummy term ∗)
         | [], _ :: _ → assert false
      in
      let lefts , _ = HExtlib.split_nth (min j (List .length args)) args in
       List .map (fun ((b,x),i) → b && x = C.Rel (k−i))
        (HExtlib.list_mapi (fun x i → x,i) (combine acc lefts ))
   | t → U.fold (fun _ k → k+1) k aux acc t
 in
  List . fold_left (aux 0)
   (let rec f = function 0 → [] | n → true :: f (n−1) in f j ) bos
```

```
let rec head_beta_reduce ?(delta=max_int) ?(upto=(−1)) t l =
 match upto, t, l with
   | 0, C.Appl l1, _ → C.Appl (l1 @ l)
   | 0, t , [] → t
   | 0, t , _ → C.Appl (t::l)
   | _, C.Appl (hd::tl ), _ → head_beta_reduce ~delta ~upto hd (tl @ l)
   | _, C.Lambda(_,_,bo), arg:: tl →
      let bo = NCicSubstitution.subst arg bo in
      head_beta_reduce ~delta ~upto:(upto − 1) bo tl
   | _, C.Const (Ref.Ref (_, Ref.Def height) as re ), _
     when delta <= height →
       let _, _, bo, _, _, _ = NCicEnvironment.get_checked_def re in
       head_beta_reduce ~upto ~delta bo l
   | _, t , [] → t
   | _, t , _ → C.Appl (t::l)
```

```
let rec instantiate_parameters params c =
  match c, params with
   | c ,[] → c
   | C.Prod (_,_,ta), he :: tl → instantiate_parameters tl (S.subst he ta)
   | t ,l → raise (AssertFailure (lazy ”1”))
```

```
  and is_non_informative paramsno c =
   let rec aux context c =
     match R.whd context c with
       | C.Prod (n,so,de) →
          let s = typeof ~subst :[] ~metasenv:[] context so in
          s = C.Sort C.Prop && aux ((n,(C.Decl so))::context) de
       | _ → true in
   let context ’,dx = split_prods ~subst :[] [] paramsno c in
    aux context’ dx
  and is_non_recursive_singleton (Ref.Ref (uri,_)) iname ity cty =
     let ctx = [iname, C.Decl ity] in
     let cty = debruijn uri 1 [] cty in
     let len = List.length ctx in
     let rec aux ctx n nn t =
       match R.whd ctx t with
       | C.Prod (name, src, tgt) →
            does_not_occur ~subst:[] ctx n nn src &&
              aux ((name, C.Decl src) :: ctx) (n+1) (nn+1) tgt
       | C.Rel k | C.Appl (C.Rel k :: _) when k = nn → true
       | _ → assert false
     in
     aux ctx (len−1) len cty
```

```
let sort_of_prod ~metasenv ~subst context (name,s) (t1, t2) =
   let t1 = R.whd ~subst context t1 in
   let t2 = R.whd ~subst ((name,C.Decl s)::context) t2 in
   match t1, t2 with
   | C.Sort s1, C.Sort C.Prop → t2
   | C.Sort (C.Type u1), C.Sort (C.Type u2) → C.Sort (C.Type (u1@u2))
   | C.Sort _,C.Sort (C.Type _) → t2
   | C.Meta (_,(_,(C.Irl 0 | C.Ctx []))), C.Sort _
   | C.Meta (_,(_,(C.Irl 0 | C.Ctx []))), C.Meta (_,(_,(C.Irl 0 | C.Ctx [])))
   | C.Sort _, C.Meta (_,(_,(C.Irl 0 | C.Ctx []))) → t2
   | _ →
      raise (TypeCheckerFailure (lazy (Printf.sprintf
        "Prod: expected two sorts, found = %s, %s"
         (PP.ppterm ~subst ~metasenv ~context t1)
         (PP.ppterm ~subst ~metasenv ~context t2))))
;;
```

```
let specialize_and_abstract_constrs ~subst r_uri r_len context ty_term =
  let cl = specialize_inductive_type_constrs ~subst context ty_term in
  let len = List.length context in
  let context_dcl =
    match E.get_checked_obj r_uri with
    | _,_,_,_, C.Inductive (_,_,tys,_) →
        context @ List.map (fun (_,name,arity,_) → name,C.Decl arity) tys
    | _ → assert false
  in
  context_dcl,
  List.map (fun (_,id,ty) → id, debruijn r_uri r_len context ty) cl,
  len, len + r_len
```

```
let specialize_inductive_type_constrs ~subst context ty_term =
  match R.whd ~subst context ty_term with
  | C.Const (Ref.Ref (uri,Ref.Ind (_,i,_)) as ref)
  | C.Appl (C.Const (Ref.Ref (uri,Ref.Ind (_,i,_)) as ref) :: _ ) as ty →
      let args = match ty with C.Appl (_::tl) → tl | _ → [] in
      let is_ind, leftno, itl, attrs, i = E.get_checked_indtys ref in
      let left_args,_ = HExtlib.split_nth leftno args in
      let _,_,_,cl = List.nth itl i in
      List.map
        (fun (rel,name,ty) → rel, name, instantiate_parameters left_args ty) cl
  | _ → assert false
```

```
(* if n < 0, then splits all prods from an arity, returning a sort *)
let rec split_prods ~subst context n te =
  match (n, R.whd ~subst context te) with
   | (0, _) → context,te
   | (n, C.Sort _) when n <= 0 → context,te
   | (n, C.Prod (name,so,ta)) →
       split_prods ~subst ((name,(C.Decl so))::context) (n − 1) ta
   | (_, _) → raise (AssertFailure (lazy "split_prods"))
```

```
  and type_of_branch ~subst context leftno outty cons tycons liftno =
    match R.whd ~subst context tycons with
    | C.Const (Ref.Ref (_,Ref.Ind _)) → C.Appl [S.lift liftno outty ; cons]
    | C.Appl (C.Const (Ref.Ref (_,Ref.Ind _)):: tl) →
        let _,arguments = HExtlib.split_nth leftno tl in
        C.Appl (S.lift liftno outty::arguments@[cons])
    | C.Prod (name,so,de) →
        let cons =
         match S.lift 1 cons with
          | C.Appl l → C.Appl (l@[C.Rel 1])
          | t → C.Appl [t ; C.Rel 1]
        in
         C.Prod (name,so,
           type_of_branch ~subst ((name,(C.Decl so))::context)
            leftno outty cons de (liftno+1))
    | _ → raise (AssertFailure (lazy "type_of_branch"))
```

```
and type_of_constant ((Ref.Ref (uri,_)) as ref) =
 let error () =
  raise (TypeCheckerFailure (lazy "Inconsistent cached infos in reference"))
 in
  match E.get_checked_obj uri, ref with
  | (_,_,_,_,C.Inductive(isind1,lno1,tl,_)),Ref.Ref(_,Ref.Ind (isind2,i,lno2))→
      if isind1 <> isind2 || lno1 <> lno2 then error ();
      let _,_,arity,_ = List.nth tl i in arity
  | (_,_,_,_,C.Inductive (_,lno1,tl,_)), Ref.Ref (_,Ref.Con (i,j,lno2)) →
      if lno1 <> lno2 then error ();
      let _,_,_,cl = List.nth tl i in
      let _,_,arity = List.nth cl (j−1) in
      arity
  | (_,_,_,_,C.Fixpoint (false,fl,_)), Ref.Ref (_,Ref.CoFix i) →
      let _,_,_,arity,_ = List.nth fl i in
      arity
  | (_,h1,_,_,C.Fixpoint (true,fl,_)), Ref.Ref (_,Ref.Fix (i,recno2,h2)) →
      let _,_,recno1,arity,_ = List.nth fl i in
      if h1 <> h2 || recno1 <> recno2 then error ();
      arity
  | (_,_,_,_,C.Constant (_,_,_,ty,_)), Ref.Ref (_,Ref.Decl) → ty
  | (_,h1,_,_,C.Constant (_,_,_,ty,_)), Ref.Ref (_,Ref.Def h2) →
     if h1 <> h2 then error ();
     ty
  | _ → raise (AssertFailure (lazy "type_of_constant: environment/reference"))
```