Reduction and conversion strategies for the Calculus of (co)Inductive Constructions: Part I

Claudio Sacerdoti Coen[1]

Department of Computer Science
University of Bologna
Bologna, Italy

Abstract

We compare several reduction and conversion strategies for the Calculus of (co)Inductive Constructions by running benchmarks from the library of the Coq proof assistant. All the strategies have been implemented in an independent verifier for the proof objects of Coq that is part of the Matita proof assistant.

Keywords: reduction strategy, conversion, calculus of inductive construction, abstract machine

1 Introduction

According to the Poincaré principle simple facts that can be automatically verified by means of computation should not be proved explicitly with deduction steps. In theorem provers based on the Curry-Howard correspondence the Poincaré principle is implemented by the conversion typing rule: every proof term for P is also a proof term for Q whenever P is convertible with Q ($P \simeq Q$). The *conversion* relation must be a decidable equivalence relation, and it is usually defined as the symmetric reflexive closure of a transitive and contextual *reduction* relation. Usually the latter includes at least β-reduction and δ-reduction (the substitution of a definiens with its definiendum), and it may become larger when sigma types or primitive inductive types are introduced in the calculus.

Most of the time the amount of reduction required by conversion in the type-checking of a term is quite limited, consisting only of a few unfolding of definitions (δ-reduction steps) and β-reduction steps to instantiate general properties (e.g. symmetry) to specific arguments (the relation that is symmetric). However, careless implementations of the conversion check risk to perform more reduction than required and to analyse reduced terms that become larger than expected. Conversion

[1] Email: sacerdot@cs.unibo.it

strategies can be used to control and minimise the amount of reduction performed. Orthogonally, we can choose reduction strategies that differ both in reduction time and in the shape and size of the reduced term (e.g. call-by-value weak head normal forms differ from call-by-name weak head normal forms).

This paper is about performance evaluation of several combinations of conversion and reduction strategies for the Calculus of (Co)Inductive Constructions (CIC). The goal is to assess their degree of independence and to suggest improved strategies or combinations. To isolate the strategies from the rewriting rules employed, we adopt a reduction machine parameterised over the reduction strategy and we pick a set of rules for conversion that define a conversion algorithm also parameterised over conversion strategies. In the second part of this paper we will choose a different set of conversion rules and we will analyse again the same combinations of strategies on the new rules. This way we will assess the degree of independence of the strategies from the rules.

Although reduction can be avoided most of the time, there are cases where a long chain of reductions is unavoidable and should be performed efficiently. This happens when two level reasoning (also called reflection) is exploited. The two level reasoning approach [5] really sticks to the Poincaré principle: an algorithm to verify a property for an internalised form of a formula is implemented in the calculus and a proof that the algorithm is correct is also provided; the proof of the property for a given formula is just the application of the proof of correctness to the internalised formula (that is computed in the meta-language). Type-checking the proof requires a conversion check that involves running the algorithm on the formula. The latter computation can be arbitrarily complex. For instance, recently the two level approach has been exploited to check in the Coq system the four colour theorem [4], that requires several days to be type-checked.

In this paper we look for strategies that behave reasonably both in the common situation and when two level reasoning is employed. However, better results could probably be achieved exploiting different strategies according to the terms to be tested for convertibility. This research direction is mostly unexplored for CIC and is left as future work.

To perform the benchmarks we have implemented the strategies in the kernel of the Matita interactive theorem prover (`http://matita.cs.unibo.it`) that is compatible with the proof terms exported from the Coq proof assistant [11]. So we can run the benchmarks from the library of Coq, that comprises about 40,000 theorems and definitions, several of them proved with two level reasoning. In the paper we will use the syntax of Stratego [12] to formally define several conversion strategies. The reader is supposed to know the syntax to understand the formal definition, but an informal description of the strategies is always provided anyway. Notice that the strategies for the benchmarks are not implemented in Stratego, but in the OCaml programming language in which Coq and Matita are implemented.

In Sect. 2 and App. A we describe the syntax, reduction and convertibility rules of CIC. In Sect. 3 we describe the basic conversion algorithm whose performance is evaluated in Sect. 4 in combination with several conversion and reduction strategies. Sect. 5 and Sect. 6 describe two conversion strategies. Reduction strategies for a parametric reduction machine described in Sect. 7 are discussed in Sect. 8.

2 The Calculus of (co)Inductive Constructions

We present now the Calculus of (co)Inductive Constructions with de Bruijn indexes, and its reduction and extended conversion rules. We omit the typing rules, the pragmatics and also the meta-theory of the calculus since they are not relevant to this work. The presentation we give is adapted from the author's PhD. thesis [10] and is as close as possible to the calculus implemented in the proof assistant Coq, version 8.0. It is also a strict subset of the calculus implemented in the proof assistant Matita under development at the University of Bologna.

2.1 The Syntax

With a small notational abuse, all the sequences indexed from 1 to n will also comprise the empty sequence, unless stated otherwise. Moreover, for the sake of readability, we adopt the compressed syntax $\overrightarrow{\phi[\alpha]}$ for the list $\phi[1] \dots \phi[n]$ where each list item $\phi[i]$ is obtained by syntactically replacing α with i. The dual notation $\overleftarrow{\phi[\alpha]}$ stands for the same list in reverse order. If two compressed expressions indexed by the same Greek variable are nested, each variable is meant to be bound in the innermost compressed list. E.g.: $\overrightarrow{t_\alpha\{\overleftarrow{f_\alpha/\alpha}\}}$ stands for $t_1\{f_n/n\ ;\ \dots\ ;\ f_1/1\}\ ;\dots\ ;\ t_n\{f_n/n\ ;\ \dots\ ;\ f_1/1\}$.

In the rest of the paper we reserve c for constant names, i for inductive type names, $k, k_1, \dots, k_n$ for inductive constructor names, s for sorts of the form $Set, Prop, Type(j)$ for j being a natural number, t, f, u, T, N, M for well formed terms and j, n, m, l, r for positive integers.

Well formed terms are inductively defined in Table 1. We use parentheses to disambiguate expressions and we assume the usual convention that application is

Table 1
CIC terms syntax

$t ::= n$	de Bruijn index, $n \in [1, +\infty)$
$\mid\ c$	constant
$\mid\ i$	(co)inductive type
$\mid\ k$	(co)inductive constructor
$\mid\ Set \mid Prop \mid Type(j)$	sort
$\mid\ t\ t$	application
$\mid\ \lambda : t.t$	λ-abstraction
$\mid\ \lambda := t.t$	local definition
$\mid\ \Pi : t.t$	dependent product
$\mid\ \langle t\rangle_h t\{\overrightarrow{t}\}$	case analysis
$\mid\ \mu_l\{\overrightarrow{t : t/n_\alpha}\}$	well-founded mutually recursive definition
$\mid\ \nu_l\{\overrightarrow{t : t}\}$	mutually co-recursive definitions

left associative and λ-abstraction and local definition are right associative.

Note that, in CIC, there is no syntactic distinction between terms and types. For the sake of clarity, we will write unknown terms as T if meant to be types and as f if meant to be function bodies.

Only the last three constructors deserve an explanation, all others coming from the theory of Pure Type Systems [1] and from the Extended Calculus of Constructions [8]. We just remind that local definition, λ-abstraction and dependent product bind a single variable in their second argument. The first argument is the type of the bound variable for λ-abstractions and dependent products; it is the definiens for local definitions.

The case analysis constructor $\langle T\rangle_h t\{\overrightarrow{f_\alpha}\}$ performs pattern matching on the term t, that must inhabit an inductive type family with h parameters. One branch f_α is associated to each constructor of the inductive type, with the intended meaning that, if t reduces to the application $(k_i\ t_1\ \ldots\ t_h\ t'_1\ \ldots\ t'_n)$, the whole expression will reduce to the branch f_i applied to $t'_1 \ldots t'_n$. The arguments $t_1\ \ldots\ t_h$ are dropped since they are used only for typing purposes to identify the family instance. Finally, the term T can be thought of as the dependent type of each branch and is also used for typing purposes only.

The case analysis constructor is often used within the $\mu_l\{\overrightarrow{t_\alpha : T_\alpha/n_\alpha}\}$ constructor that simultaneously defines a block of mutually recursive functions and returns the l-th function in the block. The α-th function in the block has body t_α and type T_α and is defined by structural recursion over his n_α-th argument. Each defined function is bound in the bodies, but not in the types.

Example 2.1 Let `Tree` and `Forest` be two families of mutually inductive types indexed by one parameter `A` that is the type of the elements of the trees. The types are constructed by the constructors `Node`, `Empty` and `Cons` according to the following definition in Matita syntax.

```
Inductive Tree (A:Type(i)) : Type(j) :=
    Node : A -> Forest A -> Tree A
with Forest (A:Type(i)) : Type(h) :=
    Empty : Forest A
  | Cons: Tree A -> Forest A -> Forest A
```

The types of the defined types and constructors are abstracted over the family parameter. E.g. `Tree` has type $Type(i) \to Type(j)$ and `Node` has type $\Pi A : Type(i). A \to \text{Forest } A \to \text{Tree } A$. The function that checks if some element in a forest satisfies a property `P` is defined in Matita syntax as follows:

```
let rec check_forest (A:Type) (P:A -> bool) (f: Forest A) : bool :=
   match f with
      Empty => false
    | Cons(t,f) => and (check_tree A P t) (check_forest A P f)
with check_tree (A:Type) (P:A -> bool) (t: Tree A) : bool :=
   match t with
      Node(x,f) => and (P x) (check_forest A P f)
in check_forest
```

In CIC syntax with de Bruijn indexes the same term becomes

$$
\begin{aligned}
&\mu_1\{\lambda : \mathit{Type}.\lambda : 1 \to \texttt{bool}.\lambda : \texttt{Forest}\ 2.\\
&\qquad \langle \lambda : \texttt{Forest}\ 3.\texttt{bool}\rangle_1 1\{\texttt{false}; \lambda : \texttt{Tree}\ 3.\lambda : \texttt{Forest}\ 4.(\texttt{and}\ (6\ 5\ 4\ 2)\ (7\ 5\ 4\ 1))\}\\
&\quad : \Pi : \mathit{Type}.\Pi : 1 \to \texttt{bool}.\Pi : \texttt{Forest}\ 2.\texttt{bool}/3;\\
&\quad \lambda : \mathit{Type}.\lambda : 1 \to \texttt{bool}.\lambda : \texttt{Tree}\ 2.\\
&\qquad \langle \lambda : \texttt{Tree}\ 3.\texttt{bool}\rangle_1 1\{\lambda : 3.\lambda : \texttt{Forest}\ 4.(\texttt{and}\ (4\ 2)\ (7\ 5\ 4\ 1))\}\\
&\quad : \Pi : \mathit{Type}.\Pi : 1 \to \texttt{bool}.\Pi : \texttt{Tree}\ 2.\texttt{bool}/3\}
\end{aligned}
$$

Notice that the mutually defined functions are bound in both bodies and thus are normally accessed using de Bruijn indexes. The subscript 1 of the μ symbol selects the first of the two mutually defined bodies, thus encoding the "`...in check_forest`" bit of the previous syntax.

Let T_1 be the previous term and let T_2 be the same term where the subscript 1 of μ has been changed to 2 (i.e. in T_1 `check_forest` is selected; in T_2 `check_tree` is). Let us consider the application of T_1 to the forest made of one single tree with just one node that contains the integer 0:

$$(T_1\ \texttt{nat}\ \texttt{even}\ (\texttt{Cons}\ \texttt{nat}\ (\texttt{Node}\ \texttt{nat}\ 0\ (\texttt{Nil}\ \texttt{nat}))\ (\texttt{Nil}\ \texttt{nat})))$$

The reduction of the term starts as expected by selecting the body of `check_forest` and unfolding the mutually recursive definitions (that are duplicated in the body as both T_1 and T_2, doubling the term size); then three β-reduction steps are performed, yielding

$$
\begin{aligned}
&\langle : \texttt{Forest}\ \texttt{bool}.\texttt{bool}\rangle_1\ \ (\texttt{Cons}\ \texttt{nat}\ (\texttt{Node}\ \texttt{nat}\ 0\ (\texttt{Nil}\ \texttt{nat}))\ (\texttt{Nil}\ \texttt{nat}))\\
&\{\texttt{false}; \lambda : \texttt{Tree}\ \texttt{nat}.\lambda : \texttt{Forest}\ \texttt{nat}.(\texttt{and}\ (T_2\ \texttt{nat}\ \texttt{even}\ 2)\ (T_1\ \texttt{nat}\ \texttt{even}\ 1))\}
\end{aligned}
$$

Then the case analysis operator is reduced matching the argument with the second branch, yielding

$$(\texttt{and}\ (T_2\ \texttt{nat}\ \texttt{even}\ (\texttt{Node}\ \texttt{nat}\ 0\ (\texttt{Nil}\ \texttt{nat})))\ (T_1\ \texttt{nat}\ \texttt{even}\ (\texttt{Nil}\ \texttt{nat})))$$

Notice that the actual family parameter nat of the constructor Cons — that is present only for type-checking purposes — has been discarded during matching.

The $\nu_l\{\overrightarrow{t_\alpha : T_\alpha}\}$ constructor defines a block of mutually co-recursive functions and picks the l-th defined function; it is syntactically very similar to $\mu_l\{\overrightarrow{t_\alpha : T_\alpha/n_\alpha}\}$, but for the third argument n_α of each recursive function that is missing in the co-recursive case.

2.2 Lifting and Substitution

Since we have adopted a syntax based on de Bruijn indexes, we do not have to worry about α-conversion, but: 1) we need to introduce a lifting operation to move terms under one or several binders; 2) we need to define a substitution operation to replace the first de Bruijn index with a term that avoids capturing on the substituted term

Table 2
Reduction

$$E[\Gamma] \vdash (\lambda : T.M)N \rhd_\beta M\{N/1\} \qquad \beta\text{-reduction}$$

$$E[\Gamma] \vdash \lambda := t.M \rhd_\zeta M\{t/1\} \qquad \zeta\text{-reduction}$$

$$E[\Gamma] \vdash c \rhd_\delta t \quad \text{if } E(c) = t \qquad \delta\text{-reduction}$$

$$E[\Gamma] \vdash n \rhd_\delta \uparrow^n t \quad \text{if } \Gamma(n) = t \qquad \delta\text{-reduction}$$

$$E[\Gamma] \vdash \langle T\rangle_h(k_j\ t_1\ \dots\ t_h\ t'_1\ \dots\ t'_{n_j})\{\overrightarrow{f_\alpha}\} \rhd_\iota (f_j\ t'_1 \dots t'_{n_j}) \qquad \iota\text{-reduction}$$

$$E[\Gamma] \vdash \mu_j\{\overrightarrow{f_\alpha : T_\alpha/n_\alpha}\}\ t_1\ \dots\ t_{n_j-1}\ (k\ t'_1\ \dots\ t'_m)$$
$$\rhd_\mu f_j\{\overrightarrow{\mu_\alpha\{\overrightarrow{f_\alpha : T_\alpha/n_\alpha}\}/\alpha}\}\ t_1\ \dots\ t_{n_j-1}\ (k\ t'_1\ \dots\ t'_m) \qquad \mu\text{-unfolding}$$

$$E[\Gamma] \vdash \langle T\rangle_h(\nu_j\{\overrightarrow{f_\alpha : T_\alpha}\})\{\overrightarrow{t_\beta}\} \rhd_\nu \langle T\rangle_h(f_j\{\overrightarrow{\nu_\alpha\{\overrightarrow{f_\alpha : T_\alpha}\}/\alpha}\})\{\overrightarrow{t_\beta}\} \qquad \nu\text{-unfolding}$$

$\rhd$ is the reflexive, transitive and contextual closure of
$(\rhd_\beta \cup \rhd_\zeta \cup \rhd_\delta \cup \rhd_\iota \cup \rhd_\mu \cup \rhd_\nu)$

and also decrements all the free de Bruijn indexes that are not substituted. The definition of the two functions, that is completely standard, can be found in App. A.

2.3 Reduction

Reduction is defined only for CIC terms that are closed in a given *environment* and *context*.

An *environment* E associates constant definitions to constant names, declarations of mutually (co)inductive types to inductive names and declarations of constructors of mutually (co)inductive types to constructor names. For the sake of reduction only, an environment can be seen as an abstract data type with only one lookup operation $E(c)$ that returns the definiens of c in E.

A *context* Γ is a stack of anonymous declarations $(: T)$ or definitions $(:= t)$. As usual, we write the stack as a list whose rightmost element is the topmost element of the stack. If the de Bruijn index i occurs free in a term, it is supposed to be an occurrence of the i-th "constant" declared or defined in Γ. We write $\Gamma(i) = t$ to say that the i-th entry in Γ from the top of the stack is a definition whose definiens is t.

In Table 2 we have collected all the one step reduction rules of CIC. The formulation of β-, ζ-and δ-reduction is the standard one when de Bruijn indexes are used. Definiens coming from the environment E are not lifted during δ-reduction since the typing rules grant that every definiens in E is a closed term. On the contrary, a term in the stack Γ can depend on the terms occurring below it. Thus the need for the lifting.

An example for ι-reduction and μ-unfolding has already been given in Sect. 2.1. The μ- and ν-unfolding rules, usually also called ι-reduction rules, are restricted forms of the usual unfolding rule given by (co)fixpoints. In particular, a recursive function definition can be unfolded only when applied to a constructor (possibly applied to some arguments) and, dually, a co-recursive function definition can be unfolded only when it is the argument of a destructor (here called case analysis).

Table 3
Convertibility and cumulativity

The convertibility equivalence relation $\simeq$ is the symmetric closure of $\triangleright$.
$E[\Gamma] \vdash T_1 \preceq T_2$ (i.e. T_1 is a "subtype" of T_2 up to universe cumulativity) iff

- $T_1 =_{\beta\delta\iota\zeta} T_2$ or
- $T_1 = \mathbf{Type}(i)$ and $T_2 = \mathbf{Type}(j)$ and $i \leq j$ or
- $T_1 = \mathbf{Prop}$ and $T_2 = \mathbf{Type}(i)$ or
- $T_1 = \mathbf{Set}$ and $T_2 = \mathbf{Type}(i)$ or
- $T_1 = \Pi x : S_1.T_1'$ and $T_2 = \Pi x : S_2.T_2'$ and $S_1 \simeq S_2$ and $T_1' \preceq T_2'$

Together with additional typing restrictions, this is sufficient to grant strong normalisation [13,3] for the well-typed terms of the calculus (supposing E and Γ also well-typed). Notice that the constraint on the unfolding of co-recursive functions forces a call-by-name strategy for the co-recursive fragment: unfolding is allowed only when the function is in head position.

2.4 *Convertibility and Cumulativity*

Convertibility is defined in Table 3. Since the reduction relation is strongly normalising, convertibility is trivially decidable by reducing both terms to their normal form and syntactically comparing them. Conversion and reduction strategies to decide convertibility avoiding unnecessary computation are the topic of this paper and will be explored in further sections.

In the tradition of the Extended Calculus of Constructions [8], the convertibility relation is weakened to an order relation called *cumulativity* that takes into account the inclusion of lower universes into higher ones and that is also defined in Table 3.

Cumulativity plays the role of a subtype relation in the typing rules of CIC. Since any algorithm that decides convertibility can be easily adapted to decide cumulativity, we will speak of conversion strategies including also strategies to decide cumulativity. Moreover, in the rest of the paper we will consider only algorithms that decide convertibility.

3 The Basic Conversion Algorithm

In this section we present the idea behind a simple algorithm to test convertibility that is easily adapted to cumulativity. We call it the *basic conversion algorithm*. The algorithm will be presented as an almost syntax directed judgement that, seen as a rewriting system, presents critical pairs that must be solved using strategies. Moreover, the judgement is parameterised over a class of reduction algorithms that leave great freedom in the choice of the reduction strategy. In the following sections we will evaluate a few strategies.

The basic conversion algorithm can be regarded as folklore for calculi simpler than CIC. In [9] it is presented for the pure Calculus of Constructions. It consists of intertwining weak head reduction steps with α-conversion steps, according to the observation that two terms in weak head normal form (w.h.n.f.) can be equivalent only if their fixed heads are so.

$$\frac{}{E[\Gamma] \vdash t \downarrow t} \quad t \in \{n, c, i, k, s\} \qquad \frac{E[\Gamma] \vdash T \downarrow T' \quad E[\Gamma; (:T)] \vdash t \downarrow t'}{E[\Gamma] \vdash \lambda : T.t \downarrow \lambda : T'.t'}$$

$$\frac{E[\Gamma] \vdash t_1 \downarrow t_1' \quad E[\Gamma] \vdash t_2 \downarrow t_2'}{E[\Gamma] \vdash (t_1\ t_2) \downarrow (t_1'\ t_2')} \qquad \frac{E[\Gamma] \vdash t \downarrow t' \quad E[\Gamma] \vdash T \downarrow T' \quad \overrightarrow{E[\Gamma] \vdash t_\alpha \downarrow t_\alpha'}}{E[\Gamma] \vdash \langle T\rangle_h t\{\overrightarrow{t_\alpha}\} \downarrow \langle T'\rangle_h t'\{\overrightarrow{t_\alpha'}\}}$$

$$\frac{\overrightarrow{E[\Gamma] \vdash T_\alpha \downarrow T_\alpha'} \quad \overrightarrow{E[\Gamma\ \overrightarrow{(:T_\alpha)}] \vdash f_\alpha \downarrow f_\alpha'}}{E[\Gamma] \vdash \mu_j\{\overrightarrow{f_\alpha : T_\alpha / n_\alpha}\} \downarrow \mu_j\{\overrightarrow{f_\alpha' : T_\alpha' / n_\alpha}\}} \qquad \frac{E[\Gamma] \vdash t_1 \downarrow t_1' \quad E[\Gamma; (:= t_1)] \vdash t_2 \downarrow t_2'}{E[\Gamma] \vdash \lambda := t_1.t_2 \downarrow \lambda := t_1'.t_2'}$$

$$\frac{\overrightarrow{E[\Gamma] \vdash T_\alpha \downarrow T_\alpha'} \quad \overrightarrow{E[\Gamma\ \overrightarrow{(:T_\alpha)}] \vdash f_\alpha \downarrow f_\alpha'}}{E[\Gamma] \vdash \nu_j\{\overrightarrow{f_\alpha : T_\alpha}\} \downarrow \nu_j\{\overrightarrow{f_\alpha' : T_\alpha'}\}} \qquad \frac{E[\Gamma] \vdash T \downarrow T' \quad E[\Gamma; (:T)] \vdash t \downarrow t'}{E[\Gamma] \vdash \Pi : T.t \downarrow \Pi : T'.t'}$$

$$\textsc{Conv-Whd-l}\ \frac{E[\Gamma] \vdash t_1 \rhd_h t_1' \quad E[\Gamma] \vdash t_1' \downarrow t_2}{E[\Gamma] \vdash t_1 \downarrow t_2} \qquad \textsc{Conv-Whd-r}\ \frac{E[\Gamma] \vdash t_2 \rhd_h t_2' \quad E[\Gamma] \vdash t_1 \downarrow t_2'}{E[\Gamma] \vdash t_1 \downarrow t_2}$$

Fig. 1. Almost syntax directed convertibility judgement, parameterised over weak head progress $\rhd_h$

Instead of presenting the algorithm in its usual form, in Fig. 1 we provide a new judgement $E[\Gamma] \vdash t_1 \downarrow t_2$ that can be proved [10] to be equivalent to $E[\Gamma] \vdash t_1 \simeq t_2$, but that is more direct to implement since it is almost syntax directed. The last two rules are parameterised over another judgement $E[\Gamma] \vdash t \rhd_h t'$ that must satisfy the following property.

Definition 3.1 [Weak head progress] A judgement $E[\Gamma] \vdash t \rhd_h t'$ satisfies weak head progress (or, abusing the standard terminology, is a *weak head reduction*) if whenever the judgement holds we have that either t has no redex in head position and t and t' are the same term, or t has a redex in head position, $E[\Gamma] \vdash t \rhd t'$ and the redex in head position in t has been reduced in the latter reduction.

Definition 3.2 [Redex in head position (r.h.p.)] A subterm r is the redex in head position of a term t iff r is t and t is a redex or r is the redex in head position of t' and t' is the subterm in head position of t.

Definition 3.3 [Subterm in head position] Only the following terms have a subterm in head position:
$(\mu_l\{\overrightarrow{t_\alpha : T_\alpha / n_\alpha}\}\ t_1'\ \ldots\ t_{n_l-1}'\ \underline{u})$
$(\underline{u}\ t)$ when u is not a well-founded mutually recursive definition
$\langle T\rangle_h \underline{u}\{\overrightarrow{t_\alpha}\}$
The subterm in head position is either the subterm in head position in the underlined subterm $\underline{u}$ (if it exists) or the subterm $\underline{u}$ itself.

The previous definition clearly shows that the calculus is not elegant because subterms in head position may occur deeply in the term structure and because of lack of symmetry: we have subterms in head position that correspond to β-redexes, ι-redexes and μ-unfolding, but we have none for ν-unfolding. Moreover, an application can form either a β-redex (where it acts as a logical destructor) or a μ-redex (because the definition of well-founded mutually recursive definitions is artificially split in CIC from the application of one recursive definition to some arguments).

To obtain an elegant formulation of the calculus the most promising possibility is to develop the calculus that is to CIC what the $\bar{\lambda}$-calculus [6] is to the λ-calculus (see [7], in particular slide 35 where non-mutually recursive definitions are defined by the $\tilde{\nu}$ evaluation context and non-mutually co-recursive definitions are defined by the perfectly dual ν term).

In that calculus the terms are actually states of a reduction machine and the subterm in head position is clearly separated from its context, leading to a cleaner and more elegant definition of the reduction rules and the weak head normal forms. Those terms are also actually used in the kernel of Coq to implement lazy reduction and, of top of it, the convertibility and cumulativity checks. However, this representation is way less understandable to the user, requiring a transparent translation back and forth to the syntax adopted in this paper. The conversion algorithm adopted in the second part of the present paper will work on those terms and will consist of a different set of rewriting rules. However, we will test it using the same combinations of conversion and reduction strategies. The interest in evaluating the performances of these combinations on both algorithms is motivated by the goal of assessing whether the relative performance of the strategies is actually independent from the term format and the exact shape of the rewriting rules.

A first important observation on weak head reduction judgements is that they do not need to be implemented in the usual way, that consists of performing the usual call-by-name computation of the w.h.n.f. of the input. Indeed, only one head reduction step is required and moreover reduction in non-head position is allowed. The latter observation tells us that we can employ any reduction strategy we want, such as call-by-name and call-by-need, and any reduction technology such as reduction machines or term rewriting systems.

The second important observation is that the new convertibility judgement is not completely syntax directed, since most of the rules form a critical pair with the Conv-Whd-l or the Conv-Whd-r rule any time one of the two terms is not in w.h.n.f. However, there exists one and only one complete strategy that does not employ backtracking. It is the strategy that always applies one of the Conv-Whd-* rules, unless the two terms are already in w.h.n.f. Since reduction on well typed terms is strongly normalising, the two rules cannot be applied for ever. We call this the *simplest convertibility strategy*. Adopting the syntax of Stratego, the simplest and backtracking free convertibility strategy can be defined as follow:

Strategy 3.4 (Simplest convertibility strategy) *The strategy is parameterised over the reduction strategy* `r`*.*

```
ss(r) = redex-head-position-l < conv-whd-l(ss(r),r) +
        redex-head-position-r < conv-whd-r(ss(r),r) +
        recur(ss(r))
```

where 1) the `redex-head-position-*` fails if the left (right) term has no redex in head position; 2) the basic strategies `conv-whd-l(s,r)` and `conv-whd-r(s,r)` implement the corresponding rules in Fig. 1 where the parameter `r` is the reduction strategy; 3) the strategy `recur(s)` implements the deterministic choice between all the other non overlapping rules of the same figure. We show only the implementation of the rules `conv-whd-l(s,r)` and the one for application, all others being similar:

Strategy 3.5 (`conv-whd-l(s,r)`)

```
conv-whd-l(s,r) =
  ?(t1,t2); <redex-head-position < r => t1' ; <s> (t1',t2) + fail> t1
```

Strategy 3.6 (`recur(s)` **traversal strategy)**

```
recur(s) = ?(Appl(t1,t2),Appl(t1',t2')); <s> (t1,t1') ; <s> (t2,t2')
```

Any other strategy can be completed by means of backtracking: every time one of the two terms is not yet in weak head normal form and the strategy prefers a different rule over the appropriate Conv-Whd-* rule, in case of failure the rule Conv-Whd-* rule must be immediately applied before continuing as before. In Stratego syntax:

Strategy 3.7 (Completing a strategy)

```
complete(s,r) =
   s <+ (redex-head-position-l < conv-whd-l(complete(s,r),r) +
         redex-head-position-r < conv-whd-r(complete(s,r),r) +
         fail)
```

In the average case a completed strategy can be more efficient than the simplest strategy `ss(r)` when `s` is able to test often the convertibility of the two terms without performing full reduction to head normal form.

We evaluate now the performance of several reduction and conversion strategies for the basic convertibility algorithm.

4 Performance Evaluation on Well Typed Terms

The simplest convertibility strategy allows to quickly detect non convertible terms without performing full reduction. Instead, if the two terms are convertible, no computation is avoided. Interactive theorem provers that record proof terms that are certified by a trusted kernel use convertibility and type-checking in two different places: inside the kernel (to check the correctness of the terms produced outside) and outside the kernel, for instance in the implementation of tactics. Since the kernel is supposed to check well typed terms, for the first usage we expect the two terms to be always convertible. Thus, at least in this case, the simplest convertibility strategy is not very satisfactory, as shown by our benchmarks (first line, table 4).

The benchmarks show the effect of convertibility and reduction strategies on type-checking time. The Improved Strategy and the Further Improved Strategy will be discussed in Sect. 5 and Sect. 6 respectively. The benchmarks have been performed running the kernel of the Matita interactive theorem prover on a subset of the library of the Coq proof assistant. All the tests have been run on a Pentium IV 2.5 GHz with 1GB of RAM. Matita is written in OCaml.

Conversion strategy	Reduction Strategy	Standard library	Skipped theorems	Heavy theorems	Heaviest theorem
SS	call-by-name	1285.71s	375	170	29.6s
IS	call-by-name	246.76s	1	15	6.9s
IS	call-by-value, read-back by name	279.58s	1	23	13.0s
IS	call-by-value	422.51s	1	36	9.7s
IS+	call-by-name	199.26s	1	2	2.2s
IS+	call-by-need	201.71s	1	3	1.5s
IS+	call-by-value, read-back by name	220.54s	1	9	10.1s
IS+	call-by-value	391.36s	0	19	11.8s
Coq		40.87s	0	2	2.5s

SS = Simplest strategy
IS = Improved Strategy (Sect. 5)
IS+ = Further Improved Strategy (Sect. 6)

Table 4
Benchmarks

The "standard library" of Coq is the library developed by the authors of Coq and distributed with the system. It is made of 5904 theorems and definitions.

Skipped theorems are theorems that require more than 30s to be type-checked. The overall and mean type-checking times shown in the table do not take in account the first 30s spent on skipped theorems.

Heavy theorems are non skipped theorems whose type-checking type requires more than 1s. Since we consider 1s to be an acceptable type-checking time for a single theorem, a satisfactory choice of strategies should produce no skipped theorems and no heavy theorems.

The last line of the table shows the time required by Coq when run on the same machine. Coq does not employ the basic conversion algorithm, but the one that will be described in the second part of this work. Moreover, constants can be marked in Coq automatically or by the user as "opaque", preventing their δ-expansion and seriously speeding up conversion in some frequent situations (-33% on the standard library). Our implementation does not exploit opaque constants since opacity is an information that is not available in the library exported from Coq.

Since the type-checker implementation of Coq is not the same of Matita, we cannot say if the better performances are all due to the conversion algorithm and to the opacity trick or if they are partly due to a most performant implementation of type-checking. The benchmarks planned for the second part of this work will be based on the type checker of Matita, allowing a direct comparison without biases of the different conversion and reduction strategies and algorithms.

Since the simplest convertibility strategy is not satisfactory, we will now explore two alternative convertibility strategies.

5 An Improved Convertibility Strategy

We assume now that the two terms whose convertibility must be checked are almost always convertible. One special case of convertibility is α-convertibility, that reduces to a check for identity when de Bruijn indexes are employed. Identity is recognised by our judgement when we use the strategy that never applies the rules Conv-Whd-l and Conv-Whd-r. We call this (incomplete) strategy the α-convertibility strategy (even if we employ de Bruijn indexes).

Strategy 5.1 (α-convertibility strategy) `alpha = recur(alpha)`

When type-checking real world terms, most of the terms checked for convertibility are actually identical. This suggests a simple but very effective strategy: the reduction rules Conv-Whd-* are used only to make the α-convertibility strategy complete, as explained in Sect. 3. More concretely, the two terms are recursively compared using every rule but Conv-Whd-*; if the comparison fails, they are both reduced to w.h.n.f. and compared again. A second failure grants that the two terms are not convertible. We call this strategy (parameterised over the reduction strategy) the Improved Strategy.

Strategy 5.2 (Improved strategy `is(r)`) `is(r) = complete(alpha,r)`

The benchmarks (table 4, first two lines) show a remarkable improvement over the simplest convertibility strategy. We observe that the improvement does not derive only from the reduced reduction time: since weak head normal forms are usually larger than the input terms, checking convertibility of their normal forms requires a significantly larger amount of time that is saved in the new strategy.

6 Further Improvement of the Convertibility Strategy

Let us consider now the improved convertibility strategy of the previous section. The strategy is optimal for identical terms. Let t_1 and t_2 be the two smallest non identical subterms in corresponding position (i.e. in two identical contexts). To check their convertibility, both terms are reduced to their w.h.n.f., possibly performing other reduction steps as well according to the reduction strategy.

In practice, quite often one term can be reduced to the other one without computing the w.h.n.f. This is for instance the case when, during a proof, the user unfolds a definition (performs a δ-reduction step). In this case the two terms to be compared will be the δ-redex and its δ-reduct and not their probably larger normal forms. As before, we would like to avoid unnecessary computation but also the additional time spent in checking convertibility of large w.h.n.f.

The way to improve the situation is: 1) to be able to detect in advance which one of the two terms is more likely to reduce to the second one; 2) reduce it only until the second one or a w.h.n.f. is reached.

Empirical observations suggest that, in CIC, long chains of β-reduction steps are rare and that either the bound variable occurs linearly or the substituted terms are small (i.e. they are formulae, but not long proof terms). Moreover real computations exploited by the user correspond to long chains of ι- and μ- or ν-reduction steps and are unlikely to be avoidable during convertibility. The conclusion is that δ-reduction steps are the important ones to address, since they often produce large reducts that can even start long reduction chains. The same idea is exploited in Coq to speed up convertibility. Here we will address only δ-reduction steps of constants for implementation reasons.

To control δ-reduction steps we propose a strategy that behaves as the one in the previous paragraph until the first comparison fails. In this case it performs weak reduction of both t_1 and t_2 avoiding δ-steps for constants, until a normal form is reached. There are now three possibilities: 1) both terms are in w.h.n.f. and the algorithm proceeds with the second pass; 2) one term has a head δ-redex and the other one is a w.h.n.f.: the first term is reduced to w.h.n.f. before proceeding with the second pass; 3) both terms have head δ-redexes: we use a heuristic to decide what term (or terms) must be head reduced and when reduction should stop.

Strategy 6.1 (Further improved strategy `isp(h,r)`) *The* `h(s,r)` *parameter is the heuristic that reduces the terms using* `r` *before applying* `s`*. The* `r` *parameter is the reduction strategy.*

```
isp(r,h) = alpha <+ conv-whd-both(isp(r,h),r,h)

conv-whd-both(s,r,h) =
      both-delta-redexes < h(conv-whd-both(s,r,h),r) +
      redex-head-position-l < conv-whd-l(conv-whd-both(s,r,h),r) +
      redex-head-position-r < conv-whd-r(conv-whd-both(s,r,h),r) +
      fail
```

The heuristic we propose is based on the following metric.

Definition 6.2 [Height of a constant] Let E be an environment. The height $h(c)$ of a constant c such that $E(c) = t$ is defined by $h(c) = 1 + \max\{h(c')|c' \in t\}$ where $c' \in t$ if c' occurs in t. If no constant occurs in c the height of c is 1.

We claim that in practice it is often the case that given a δ-redex $(c\ t_1 \ldots t_n)$ whose δ-reduct reduces to $(c'\ t'_1 \ldots t'_m)$ we have $h(c) > h(c')$.

For instance, consider the two convertible terms $(*\ 1000\ 100)$ and $(+\ 100\ (*\ 999\ 100))$ that have both 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 to perform head reduction on the first term until a δ-redex in head position of height less or equal to $h(+)$ is found. If the reduced term has height $h(+)$ and its head is an addition the arguments could be convertible. This is indeed the case in our not so artificial example.

As a counterexample to the property above, if c is defined as the identity function then $(c\ c') \rhd_h c'$ but for most c' we have $1 = h(c) \not\geq h(c')$. Notice that this is quite a rare case: it can occur only if the argument of a constant can occur in head position during reduction.

According to the previous metric, the heuristic of our new strategy is defined as follows: the height h_1 and h_2 of the two head δ-redices are compared; if one (say h_1) is greater than the other, its term is head reduced until it becomes a w.h.n.f. or a δ-redex of height less or equal to h_2; otherwise both terms are head reduced until they become w.h.n.f. or a δ-redex of height less than h_1. Then the algorithm proceeds with the second pass.

Strategy 6.3 (Heuristic reduction strategy `h`) `<cmin>(h1,h2)` *returns* `h1-1` *if* `h1=h2` *or the minimum of* `{h1,h2}` *otherwise.* `compute-height` *returns the height of* `t` *if* `t` *is a δ-redex, $+\infty$ if* `t` *is neither a δ-redex, nor it is in weak head normal* *form, and 0 if* `t` *is in weak head normal form.*

```
h(s,r) =
   ?(t1,t2); <compute-height> t1 => h1; <compute-height> t2 => h2;
   <cmin> (h1,h2) => hmin; <reduce-to-height(r)> (t1,hmin) => t1';
   <reduce-to-height(r)> (t2,hmin) => t2'; <s> (t1',t2')

reduce-to-height(r) =
   ?(t,hmax);
   repeat(where(tall-redex-head-position) ; <r> t => t';
          <reduce-to-height(r)> (t',hmax))

tall-redex-head-position =
   ?(t,hmax); <compute-height> t => ?h; <gt> (h,hmax)
```

The benchmarks in table. 4 show that the proposed improvement is really effective in decreasing type-checking time, independently from the reduction strategy it is associated with. The improvement could also be applied to the code of the Coq proof assistant, that right now delays δ-conversion as we suggest, but does not exploit any heuristic similar to ours both to choose which term must be reduced when two δ-redexes are met and to guide the reduction avoiding intermediate usually useless convertibility checks.

7 A Parametric Reduction Machine

Having considered three different convertibility strategies, we want now to compare their behaviour when combined with different reduction strategies. To make the comparison, in our PhD. thesis [10] we have described GKAM_{CIC}, a generic reduction machine, based on the abstract machine of Krivine, that is parameterised over the reduction strategy.

The state of the 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 is processed, the n-th component of the stack is fetched and becomes the new term to be processed (together with the new environment). As an example, consider the following reduction of the identity function applied to a closed term t:

$$\langle\emptyset,(\lambda.1\ t),\emptyset\rangle \rhd \langle\emptyset,\lambda.1,\langle\emptyset,t\rangle.\emptyset\rangle \rhd \langle\emptyset.\langle\emptyset,t\rangle,1,\emptyset\rangle \rhd \langle\emptyset,t,\emptyset\rangle$$

Since the argument t is never reduced before reaching the weak head position, the machine implements a call-by-name strategy.

The final state of the reduction is a stuck state that must be read-back to a term by a function $\mathcal{T}(\_,\_,\_)$. E.g.: $\langle\emptyset.\langle\emptyset,3\rangle,\lambda.2,\emptyset\rangle$ is read back as $\lambda.4$ by: 1) recursively reading back each closure in the environment and in the stack interpreting it as a machine state with an empty stack (e.g. $\mathcal{T}(\emptyset,3,\emptyset)=3$); 2) applying the read-back environment (now a simultaneous substitution) to the term (e.g. $\lambda.2$

becomes $\lambda.4$ since 2 now refers to the term 3 in the read back environment); 3) forming an application of the term to the read back arguments in the stack if any (e.g. $\lambda.4$ remains $\lambda.4$ since applied to the empty stack $\emptyset$). As another example $\mathcal{T}(\emptyset.\langle\emptyset, 3\rangle, 1, \emptyset.\langle\emptyset, 2\rangle) = (3\ 2)$.

The GKAM_{CIC} (Generalised KAM for CIC) generalises the KAM in three ways. 1) Arbitrary reductions of the argument of an application are now allowed when the argument is moved to the stack, to the environment or in code position. This way any kind of reduction strategy 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 $\mathcal{T}_s(\_)$ and $\mathcal{T}_e(\_)$ from stack and environment items to terms. The read-back functions are used when the machine becomes stuck to map the machine state to the corresponding computed term. 3) The machine is extended to CIC.

We describe now the generic reduction machine, that is parameterised over a few functions and datatypes that are instantiated by each reduction strategy. The machine is actually implemented as an ML functor whose input is a module that collects the parameters instantiation describing a reduction strategy.

Definition 7.1 (Parameters of the reduction machine a.k.a. strategies) The reduction machine is abstracted over the following parameters. Any instantiation of the parameters is called a *strategy*. The first two parameters are the abstract datatypes used for the environment items and the stack items:

$$\text{abstract datatype EItem, SItem}$$

Environment and stacks are defined as list of items. Closures and machine configurations (or states) are defined in terms of them.

$$\begin{aligned}
\text{Environment} &\stackrel{def}{=} \text{EItem List}\\
\text{Stack} &\stackrel{def}{=} \text{SItem List}\\
\text{Closure} &\stackrel{def}{=} \text{Environment} * \text{Term}\\
\text{Configuration} &\stackrel{def}{=} \text{Environment} * \text{Term} * \text{Stack}
\end{aligned}$$

There must exist a strict and well-founded order $<_E$ on environments such that

$$\forall \xi : \text{ Environment } .\forall \alpha : \text{ EItem } .\xi <_E \xi.\alpha$$

The next three parameters are functions to build stack items, move stack items to the environment and fetch machine configurations from the environment:

$$\text{to\_stack} : \text{Closure} \to \text{SItem}$$

$$\text{to\_env} : \text{SItem} \to \text{EItem}$$

$$\text{from\_env} : \text{SItem} \to \text{Configuration}$$

The final two parameters are read-back functions for the abstract datatypes:

$$\begin{aligned}
\mathcal{T}_s &: \text{SItem} \to \text{Term}\\
\mathcal{T}_e &: \text{EItem} \to \text{Term}
\end{aligned}$$

In order to grant correctness and liveness properties for the reduction machine, reduction strategies (i.e. instantiation of the machine parameters) must satisfy the following correctness and liveness conditions.

Definition 7.2 (GKAM$_{\text{CIC}}$ correctness conditions for strategies)

(i) $\forall(\xi,t)$: Closure $.\mathcal{T}(\xi,t,\emptyset) \rhd \mathcal{T}_s(\text{to\_stack }(\xi,t))$

(ii) $\forall\alpha$: SItem $.\mathcal{T}_s(\alpha) \rhd \mathcal{T}_e(\text{to\_env }\alpha)$

(iii) $\forall\alpha$: EItem $.\mathcal{T}_e(\alpha) \rhd \mathcal{T}(\text{from\_env }\alpha)$

Definition 7.3 (GKAM$_{\text{CIC}}$ liveness condition for strategies) $\pi_1 \circ \text{from\_env } \circ \text{hd} \circ \pi_1$ must be strictly decreasing according to the ordering $<_E$. π_n stands for the n-th projection of a tuple.

Page 17 shows the transition rules for the GKAM$_{\text{CIC}}$. The initial configuration of the machine to reduce a term t is $(\emptyset, t, \emptyset)$. The final configurations of the machine are the special configurations $(-,(\xi,t,S),-)$. We write $\mathcal{R}(\xi,t,S) = (\xi',t',S')$ if the machine reduces in many steps the configuration (ξ,t,S) to the configuration $(-,(\xi',t',S'),-)$.

We define the following read-back function from machine configurations and closures to terms:

Definition 7.4 (Read-back function $\mathcal{T}(\_,\_,\_)$) The function is parameterised over $\mathcal{T}_e$ and $\mathcal{T}_s$ that must be instantiated in the strategy.

$$\mathcal{T}(-,(\xi,t,S),-) \stackrel{def}{=} \mathcal{T}(\xi,t,S)$$

$$\mathcal{T}([\alpha_1,\ldots,\alpha_n],t,[\beta_1,\ldots\,\beta_m]) \stackrel{def}{=}$$

$$(t\{\ \mathcal{T}_e(\alpha_1)/1\ ;\ \ldots\ ;\ \mathcal{T}_e(\alpha_n)/n\}\ \mathcal{T}_s(\beta_1)\ \ldots\ \mathcal{T}_s(\beta_m)))$$

where the simultaneous substitution $\{\sigma\}_m$ maps every de Bruijn index in its domain to its image.

The following theorems are proved in [10]. Together with the strong normalisation property of CIC for well typed terms they show that the reduction machine implements strongly normalising reduction on well typed terms when instantiated on correct and lively strategies. Moreover, weak head progress is granted.

Theorem 7.5 ***(Correctness)*** *For each strategy that respects the correctness and liveness properties and for all machine states (ξ,t,S) and (ξ',t',S'), if (ξ,t,S) evolves into (ξ',t',S') then $\mathcal{T}(\xi,t,S) \rhd^* \mathcal{T}(\xi',t',S')$.*

Theorem 7.6 ***(Liveness)*** *For each strategy that respects the correctness and liveness properties there is no infinite sequence of GKAM$_{\text{CIC}}$ states $(\xi_0,t_0,S_0),(\xi_1,t_1,S_1),\ldots$ such that for each index i (ξ_i,t_i,S_i) moves into $(\xi_{i+1},t_{i+1},S_{i+1})$ and $\mathcal{T}(\xi_i,t_i,S_i) = \mathcal{T}(\xi_{i+1},t_{i+1},S_{i+1})$*

Theorem 7.7 ***(Weak head progress)*** *For each well typed term t with a redex in head position, if $\mathcal{R}(\emptyset,t,\emptyset) = (\xi,t',S)$ and $\mathcal{T}(\xi,t',S) = t''$ then $t \rhd^* t''$ and the redex in head position in t has been reduced in t''.*

Before			After		
Env.	Code	Stack	Env.	Code	Stack
ξ	i (where $i \leq \lvert\xi\rvert$)	S	π_1(from_env ξ_i)	π_2(from_env ξ_i)	π_3(from_env ξ_i)$.S$
ξ	i (where $i > \lvert\xi\rvert$ and $\Gamma(i - \lvert\xi\rvert) = b$)	S	$\emptyset$	$\uparrow^{i-\lvert\xi\rvert} b$	S
ξ	i (otherwise)	S	-	(ξ, i, S)	-
ξ	c (where $E(c) = b$)	S	$\emptyset$	b	S
ξ	c (otherwise)	S	-	(ξ, c, S)	-
ξ	i	S	-	(ξ, i, S)	-
ξ	k	S	-	(ξ, k, S)	-
ξ	s	$\emptyset$	-	$(\emptyset, \emptyset, s, \emptyset)$	-
ξ	$\Pi : T.t$	$\emptyset$	-	$(\xi, \Pi : T.t, \emptyset)$	-
ξ	$\lambda : T.M$	$\alpha.S$	ξ.(to_env α)	M	S
ξ	$\lambda : T.M$	$\emptyset$	-	$(\xi, \lambda : T.M, \emptyset)$	-
ξ	MN	S	ξ	M	(to_stack (ξ, N))$.S$
ξ	$\lambda := M.N$	S	ξ.to_env (to_stack(ξ, M))	N	S
ξ	$\langle T\rangle_h t\{\overrightarrow{t_\alpha}\}$ (when †1 holds)	S	ξ	t_i	$I'_1 \ldots I'_l.S$
ξ	$t_0 = \langle T\rangle_h t\{\overrightarrow{t_\alpha}\}$ (otherwise)	S	-	(ξ, t_0, S)	-
ξ	$\mu_l\{\overrightarrow{f_\alpha : T_\alpha/n_\alpha}\}$ (when †2 holds)	S	$\xi.\overleftarrow{\text{to\_env(to\_stack}(\xi, \mu_\alpha\{\overrightarrow{f_\alpha : T_\alpha/n_\alpha}\}))}$	f_i	S''
ξ	$t_0 = \mu_l\{\overrightarrow{f_\alpha : T_\alpha/n_\alpha}\}$ (otherwise,)	S	-	(ξ, t_0, S'')	-
ξ	$t_0 = \nu_l\ \{\overrightarrow{f_\alpha : T_\alpha}\}$	S	-	(ξ, t_0, S)	-

†1: Let $\mathcal{R}(\xi, t, S) = (\xi_1, t_1, S_1)$. If $t_1 = \nu_l\{\overrightarrow{f_\alpha : T_\alpha}\}$ then let $\mathcal{R}(\xi_1.\overleftarrow{\text{to\_env(to\_stack}(\xi_1, \nu_\alpha\{\overrightarrow{f_\alpha : T_\alpha}\}))}, f_l, S_1) = (\xi_2, t_2, S_2)$. Otherwise let $(\xi_2, t_2, S_2) = (\xi_1, t_1, S_1)$. The rule is fired only if $(t_2, S_2) = (k_i, I_1 \ldots I_h.I'_1 \ldots I'_l)$.

†2: Let $\mathcal{R}(\text{from\_env}(\text{to\_env}(S_{n_l}))) = (\xi', t', S')$ and let S'' be equal to S but for the n_l-th entry that is replaced with to_stack$(\emptyset, \mathcal{T}(\xi', t', S'))$. The rule is fired if t' is a constructor. Note that S'' is used anyway also in the rule that is applied when this rule fails. The same rules modified posing $S'' = S$ are also admissible as call-by-name variants; in this variant the read back functions are never used by the reduction loop.

call-by-name:

SItem = term
EItem = term
to_stack$(\xi, t) = \mathcal{T}(\xi, t, \emptyset)$
to_env $t = t$
from_env $t = (\emptyset, t, \emptyset)$
$\mathcal{T}_s(t) = t$
$\mathcal{T}_e(t) = t$

call-by-value:

SItem = closure
EItem = term
to_stack $(\xi, t) = (\xi, t)$
to_env$(\xi, t) = \mathcal{R}(\xi, t, \emptyset)$
from_env $t = (\emptyset, t, \emptyset)$
$\mathcal{T}_s(\xi, t) = \mathcal{T}(\xi, t, \emptyset)$
$\mathcal{T}_e(t) = t$

call-by-value, read-back by name:

SItem = closure
EItem = term * term
to_stack $(\xi, t) = (\xi, t)$
to_env$(\xi, t) = (\mathcal{R}(\xi, t, \emptyset), \mathcal{T}(\xi, t, \emptyset))$
from_env $(t_v, t_n) = (\emptyset, t_v, \emptyset)$
$\mathcal{T}_s(\xi, t) = \mathcal{T}(\xi, t, \emptyset)$
$\mathcal{T}_e(t_v, t_n) = t_n$

call-by-need:

SItem = closure
EItem = (bool $*$ configuration) ref
to_stack $(\xi, t) = (\xi, t)$
to_env$(\xi, t) = ref(false, (\xi, t, \emptyset))$
from_env $c =$
match $!c$ with
$true, c' \to c'$
$|false, c' \to c := true, \mathcal{R}(c');\ snd\ !c$
$\mathcal{T}_s(\xi, t) = \mathcal{T}(\xi, t, \emptyset)$
$\mathcal{T}_e(c) = \mathcal{R}(snd\ !c)$

Table 5
Instantiation to specific strategies

Thus we can implement head reduction as reduction of the initial machine configuration followed by read-back: $E[\Gamma] \vdash t \rhd_h t''$ iff $\mathcal{R}(\emptyset, t, \emptyset) = (\xi, t', S)$ and $\mathcal{T}(\xi, t', S) = t''$.

8 Reduction Strategies for the GKAM$_{\mathrm{CIC}}$

In Table 5 we show (in pseudo ML syntax) how to instantiate the GKAM$_{\mathrm{CIC}}$ to obtain both standard and non-conventional reduction strategies. Notice that we allow ourselves to call recursively $\mathcal{R}(\_, \_, \_)$ and $\mathcal{T}(\_, \_, \_)$ even if the calls are not tail recursive. Non tail recursive calls are usually not allowed in reduction machines for performance reasons. What we gain is the ability to switch the reduction strategy easily for direct comparison.

Call-by-name is implemented by immediately reading back a closure before pushing it on the stack. Thus the stack and the environment are made of terms; pushing a stack item to the environment does not change it and fetching a machine state from the environment builds a new state from a closed term. An alternative implementation consists in keeping closures both on the stack and the environment, yielding an instantiated reduction machine that is isomorphic to the KAM extended to CIC.

Call-by-value is implemented by keeping closures on the stack (as for the KAM), but reducing them (starting a new machine) before pushing them to the environment (that holds closed terms). Thus arguments of an application are evaluated at most once and only if the head of the function reduces to a λ-abstraction.

The third strategy performs in parallel both call-by-value and call-by-name, keeping closures on the stack and closed terms in the environment both in reduced and unreduced form. When environment items are fetched during computation, the call-by-value component is returned; when they are fetched in the read-back procedure, the call-by-name components is returned to simulate "undoing" reduction of redexes not in head position.

Call-by-need is implemented as usual keeping mutable machine states in the environment and updating them with the normal form when an argument is evaluated.

The proof that the strategies really implement what is expected can be found in the author's PhD. thesis [10] for an extended version of the calculus.

Performance evaluation for the proposed reduction strategies have been given in table 4. The benchmarks support our claim that, at least in the average case, avoiding reduction and hence comparison of usually larger reducts is better than optimising reduction. We can observe this in two different ways: 1) convertibility strategies seem more effective than reduction strategies; 2) call-by-name (also in the call-by-need variant) gives better results than call-by-value, and a strategy that is somehow intermediate between the two of them from the point of view of convertibility or reducts gives intermediate results. However, we can also observe the existence of a very heavy theorem that can be type-checked in reasonable time only in a call-by-value setting. Similar theorems, all based on two level reasonings, are also found outside the standard library, in the user contributions. Notice, however, that several other theorems proved with the same approach in the standard library are type-checked in a reasonable time with our best combination of strategies.

Finally, we can notice an anomaly in the benchmarks. With call-by-value the further improved strategy is slower than the improved strategy (even considering the additional theorem accepted that requires at most 11.8s). The computational cost for the additional complexity of the further improved strategy is unlikely to be responsible for the 20s to be explained. The only explanation left is that the following situation must be frequent: the two terms to be converted have reducts that are δ-redexes characterised by the same height, but further reduction is required to make the corresponding sub-terms convertible. Thus the reduction machine is interrupted too early, the cost of read-back is paid and useless conversion is performed before backtracking (that restarts reduction). Further investigation should be attempted to monitor how frequently phenomenon occurs with other strategies and to understand if it can be avoided.

9 Conclusions and Future Work

We have presented the basic conversion algorithm, a simple almost syntax directed judgement to test convertibility (and with minimal modifications also cumulativity) of terms of the Calculus of (co)Inductive Constructions (CIC). The judgement has been presented as a rewrite system with critical pairs parameterised over a reduction judgement. Both convertibility and reduction strategies can be imposed to obtain executable algorithms from the judgement. We have presented a few improvements on the simplest convertibility strategy and we have evaluated their performance by means of benchmarks on a real world library. One of the improvements could also be applied to the code of the Coq proof assistant, with expected performance increasing. We have also presented a generic reduction machine parameterised over a reduction strategy. The machine has been used to perform the benchmarks, varying over the reduction strategy.

The aim of the paper was mainly investigative, since, as far as we know, precise comparisons of several reduction strategies for CIC and their role in type-checking are not available in the literature.

Even with the best convertibility and reduction strategy the benchmarks show that, for a few theorems, the basic conversion algorithm is not competitive with the one now adopted in the Coq proof assistant. Profiling the code it becomes evident that the bottleneck is the read-back function $\mathcal{T}(\_,\_,\_)$ that is invoked after each reduction, to convert configurations back to terms. Thus the evident improvement consists in avoiding the read-back function, considering a new almost syntax directed and strategy driven judgement to check convertibility over machine configurations. The machine configurations can be interpreted as terms of a reformulation of the calculus seen as an extension of the $\bar{\lambda}\mu\tilde{\mu}$-calculus of Curien and Herbelin [2]. This solution, that is the one adopted for a particular strategy in the Coq proof assistant, is under implementation in Matita and it seems to give results compatible with the ones of Coq. The subject of the second part of this work will be the description of this alternative algorithm and a performance evaluation of the conversion and reduction strategies presented here when applied to that algorithm. We claim that for CIC the relative performance of the strategies is actually independent from the choice of the algorithm, i.e. from the term representation and the exact rewriting rules employed.

Acknowledgements: I am very grateful to the anonymous referees for their great suggestions on improving the presentation of the paper.

References

[1] H. P. Barendregt. “Lambda calculi with types”, Handbook of logic in computer science (vol. II), pp. 117–309, 1992.

[2] P. L. Curien, H. Herbelin. “The duality of computation”, Proceedings of the 5th ACM SIGPLAN international conference of Functional programming (ICFP’00), ACM Press, pp. 233–243, 2000.

[3] E. Gimenez. “Codifying guarded definitions with recursive schemes”. Proceedings of the 1994 Workshop on Types for Proofs and Programs, LNCS 996, pages 39-59.

[4] G. Gonthier. *A computer-checked proof of the Four Colour Theorem*, Technical Report Microsoft Research Cambridge, http://research.microsoft.com/~gonthier/4colproof.pdf

[5] J. Harrison. “Metatheory and reflection in theorem proving: A survey and critique”. Technical Report CRC-053, SRI Cambridge, 1995.

[6] H. Herbelin. *Séquents qu’on calcule: de l’interprétation du calcul des séquents comme calcul de lambda-termes et comme calcul de stratégies gagnantes*, PhD thesis, Université Paris VII, 1995.

[7] H. Herbelin. Invited talk at HOR’06. Slides available at http://yquem.inria.fr/~herbelin/talks/hor06.pdf.

[8] Z. Luo. *An Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1990.

[9] R. Harper and R. Pollack. “Type checking with universes”. Theoretical Computer Science, 89:107–136, 1991.

[10] C. Sacerdoti Coen. *Mathematical Knowledge Management and Interactive Theorem Proving.* PhD. thesis, University of Bologna, 2004.

[11] C. Sacerdoti Coen. “From Proof-Assistants to Distributed Libraries of Mathematics: Tips and Pitfalls”. In Proc. Mathematical Knowledge Management 2003, Lecture Notes in Computer Science, Vol. 2594, pp. 30–44, Springer-Verlag.

[12] The Stratego program transformation language http://www.program-transformation.org/Stratego/WebHome

[13] B. Werner. *Une Theorie des Constructions Inductives.* PhD. thesis, Université Paris VII, 1994.

A Lifting and substitution

Lifting (Table A.1) and substitution (Table A.2) for the Calculus of (Co)Inductive Constructions follow the standard definitions for a calculus with de Bruijn indexes.

Table A.1
Lifting

$$
\begin{array}{l}
\uparrow^m t = \uparrow^m_0 t \\
\\
\uparrow^m_b n = n \quad \text{if } n <= b \\
\uparrow^m_b n = n + m \quad \text{if } n > b \\
\uparrow^m_b t = t \quad \text{if } t \in \{c, i, k, Set, Prop, Type(j)\} \\
\uparrow^m_b (t_1\ t_2) = \uparrow^m_b t_1 \uparrow^m_b t_2 \\
\uparrow^m_b \lambda : T.t = \lambda : \uparrow^m_b T. \uparrow^m_{b+1} t \\
\uparrow^m_b \lambda := t_1.t_2 = \lambda := \uparrow^m_b t_1. \uparrow^m_{b+1} t_2 \\
\uparrow^m_b \Pi : T.t = \Pi : \uparrow^m_b T. \uparrow^m_{b+1} t \\
\uparrow^m_b \langle t \rangle_h t' \{\overrightarrow{t_\alpha}\} = \langle \uparrow^m_b t \rangle_h \uparrow^m_b t' \{\overrightarrow{\uparrow^m_b t_\alpha}\} \\
\uparrow^m_b \mu_l \{\overrightarrow{t_\alpha : T_\alpha / n_\alpha}\} = \mu_l \{\overrightarrow{\uparrow^m_{b+r} t_\alpha : \uparrow^m_b T_\alpha / n_\alpha}\} \\
\uparrow^m_b \nu_l \{\overrightarrow{t_\alpha : T_\alpha / n_\alpha}\} = \nu_l \{\overrightarrow{\uparrow^m_{b+r} t_\alpha : \uparrow^m_b T_\alpha / n_\alpha}\}
\end{array}
$$

Table A.2
Substitution

$$
\begin{array}{l}
m\{N/m\} = \uparrow^{m-1} N \\
n\{N/m\} = n \quad \text{if } n < m \\
n\{N/m\} = n - 1 \quad \text{if } n > m \\
t\{N/m\} = t \quad \text{if } t \in \{c, i, k, Set, Prop, Type(j)\} \\
(t_1\ t_2)\{N/m\} = t_1\{N/m\}\ t_2\{N/m\} \\
(\lambda : T.t)\{N/m\} = \lambda : T\{N/m\}.t\{N/m+1\} \\
(\lambda := t_1.t_2)\{N/m\} = \lambda := t_1\{N/m\}.t_2\{N/m+1\} \\
(\Pi : T.t)\{N/m\} = \Pi : T\{N/m\}.t\{N/m+1\} \\
(\langle t \rangle_h t' \{\overrightarrow{t_\alpha}\})\{N/m\} = \langle t\{N/m\} \rangle_h t'\{N/m\} \{\overrightarrow{t_\alpha\{N/m\}}\} \\
(\mu_l \{\overrightarrow{t_\alpha : T_\alpha / n_\alpha}\})\{N/m\} = \mu_l \{\overrightarrow{t_\alpha\{N/m+r\} : T_\alpha\{N/m\}/n_\alpha}\} \\
(\nu_l \{\overrightarrow{t_\alpha : T_\alpha / n_\alpha}\})\{N/m\} = \nu_l \{\overrightarrow{t_\alpha\{N/m+r\} : T_\alpha\{N/m\}/n_\alpha}\}
\end{array}
$$