Full reduction at full throttle

Mathieu Boespflug[1], Maxime Dénès[2], and Benjamin Grégoire[2]

[1] McGill University, mboes@cs.mcgill.ca
[2] INRIA Sophia Antipolis - Méditerranée, {Maxime.Denes,Benjamin.Gregoire}@inria.fr

Abstract. Emerging trends in proof styles and new applications of interactive proof assistants exploit the computational facilities of the provided proof language, reaping enormous benefits in proof size and convenience to the user. However, the resulting proof objects really put the proof assistant to the test in terms of computational time required to check them. We present a novel translation of the terms of the full Calculus of (Co)Inductive Constructions to OCaml programs. Building on this translation, we further present a new fully featured version of Coq that offloads much of the computation required during proof checking to a vanilla, state of the art and fine tuned compiler. This modular scheme yields substantial performance improvements over existing systems at a reduced implementation cost.
The work presented here builds on previous work described in [11], but we place particular emphasis in this paper on the fact that this scheme is in fact an instance of untyped normalization by evaluation [8, 14, 1, 4].

Introduction

Many proof assistants share with many programming language a common basis in the typed λ-calculus. Systems in the lineage of Church's original higher order logic reuse the typed λ-calculus as both the language for the objects of the discourse (terms) and the language of formulae and propositions about these objects. Following Automath dependently typed systems push the envelope even further to unify objects of discourse, propositions, and proofs, all as terms of the typed λ-calculus. Either way, seeing as the λ-calculus is also a programming language, both strands of proof assistants thus intrinsically support discoursing about programs. In dependently typed programming languages, this is made plain by the following inference rule,

$$\frac{\Gamma \vdash M : A \qquad A =_\beta B}{\Gamma \vdash M : B}\ \text{(conv)}$$

which allows replacing part of a proposition for another if an oracle agrees that the two propositions are related to each other through computation (i.e. they are *convertible*).

As a matter of fact, in dependently typed theories, support for reasoning about programs is so good that new proof methodologies have emerged in recent years [5, 16, 9, 12, 10] to establish results in pure and applied mathematics by reducing them to the computation of a program proven to be correct. The point of this approach is to turn many of the deduction steps into computation steps instead, which do not appear explicitely in the proof term, so as to yield smaller proofs whose size is independent of the number of computation steps required. However, efficiency of proof checking hinges on the complexity of the algorithm realized by the computation steps and on the performance of the evaluator used to carry them out.

Proof by reflection [5, 13] is one such methodology. Consider subclasses $\mathscr{C}$ of the class of all propositions. The idea is that the propositions of somesuch subclasses $\mathscr{C}$ are better established by appeal to some metalogical property shared by these propositions rather than to write independent proofs for each of the properties in $\mathscr{C}$. For instance, the standard library of Coq gives $(<)$ as an inductively defined predicate. But notice that ground instances of $n < m$ are provable only if the computation $f(n,m)$ yields the boolean value true as an answer, where

$$f(x,y) \mathrel{\hat{=}} \begin{cases} \mathsf{true} & \text{if } \max(x+1-y,0)=0 \\ \mathsf{false} & \text{otherwise} \end{cases}$$

One can prove $f(x,y) = \mathsf{true}$ by reflexivity in Coq because this proposition is equivalent to $\mathsf{true} = \mathsf{true}$ by the inference rule (conv) above. Now if we have at hand a proof p showing that $\forall x y.\, f(x,y) = \mathsf{true} \Rightarrow x < y$, then the term $p\ x\ y\ \mathsf{refl}$ is a proof of $x < y$. This is a single, generic proof that works for all ground inequalities and that is independent of the sizes of x and y. This style of reflection scales up in the large to complicated semi-decision procedures, as in the proof of the four colour theorem [9], and is promoted in the small by the SSReflect methodology [10]. Proofs by reflection also extend to non-ground propositions via *quoting* (lifting to an object representation) and syntactic transformations (of which a generic proof of correctness is established) [5, 12].

Existing functional programming language runtime environments would be most appropriate to decide the convertibility condition which lies at the heart of most every proof by reflection. However, this strategy faces two problems:

- In a conventional functional programming language, one can only analyze and compare values at base type (lists, integers,...). Inhabitants of function types are effectively black boxes that cannot be compared.
- Programs of a functional language are always closed terms, whereas in our context, we may have to compare (and so to evaluate) open terms (with free variables referencing assumptions in the typing context).

These constraints allow runtimes to only deal with substitutions of closed terms into closed terms, which allows for efficient implementation strategies and elides any issues with name capture. Normal forms cannot always be reached purely by closed substitutions. Only weak head normal forms are computed, meaning that they are only *weakly reduced*. Our objective is to implement *full reduction* to normal form of potentially open terms of dependently typed terms at higher order type, with much of the same performance profile and optimizations as might be available in a mature, optimizing compiler for run-of-the-mill programming languages. We conspicuously avoid modifying any existing compiler to do so, let alone writing our own, by reusing as-is the OCaml compiler.

This design choice follows a long history of previous approaches to normalization using off-the-shelf components. Normalization by Evaluation (NbE) is one such appealingly elegant family of approaches. The point there is to obtain the sought normal forms not by the usual iteration of a one-step reduction relation, but rather by constructing a *residualizing* model D of the set Λ of terms, given by a denotation $[\![\_]\!] : \Lambda \to D$, supporting an inverse $\downarrow : D \to \Lambda$ functional [3] (called *reification* or *readback*) such that

1. if $t \longrightarrow t'$ then $[\![t]\!] = [\![t']\!]$ (soundness) ;
2. if t is a term in normal form, then $\downarrow [\![t]\!] = t$ (reproduction).

Then it is easy to see that if $t \longrightarrow^* t'$ where t' is normal, $\downarrow [\![t]\!] = \downarrow [\![t']\!] = t'$, so composing the interpretation with reification gives a normalization function for terms whose normal form exists. In typed NbE and type directed partial evaluation (TDPE) [7], reification is actually done in a type directed way. But such approaches need to be adapted as the type system changes and scaling them up to powerful type systems such as the Calculus of Inductive Constructions (CIC) [6, 17] with a hierarchy of universes as implemented in Coq is non-trivial. Untyped variants of NbE have been proposed [8, 14, 1, 4], but the generality of the untyped approaches have so far come at the cost of adding tags to the interpretation of object syntax to deeply embed it into the host language. While the performance penalty of this tagging can be mitigated in many circumstances [4], some of the interpretive overhead introduced by the tagging invariably remains. Memory allocation and locality of code is negatively impacted and some simple common compiler optimizations (such as uncurrying) need to be redone at the level of the object syntax interpretation.

The implementation of full reduction that we describe here matches the generality of untyped NbE, since it works for all (open) terms of the λ-calculus. It also matches the performance of typed NbE, since the interpretation of terms introduces zero interpretive overhead. One no longer needs to choose between full generality and full performance. The approach used here is inspired by [11], but unlike this earlier work which requires modified versions of the stock OCaml compiler and virtual machine, we achieve full genericity in the underlying evaluator. We do not need to maintain a custom version of this underlying evaluator — meaning a better separation of concern between writing proof assistants[3] and writing compilers.

The structure of this paper is as follows. In Section 1, we offer as a first contribution a unifying view of untyped normalization by evaluation and of the normalization algorithm of [11], showing that the latter can be seen as an instance of the former. We then show how to implement this algorithm by translation of the source language to a functional language, without modifying the underlying compiler or virtual machine, and generalize the algorithm to full reduction of terms of the CIC (Section 2). We proceed to adding coinductive datatypes (Section 3). These encodings are, to the best of our knowledge, novel features in an NbE algorithm. In Section 4, we show through a number of high-level and real world use cases that our solution fares very favourably relative to existing implementations of the conversion rule in Coq, typically attaining a fivefold speedup.

[3] Our implementation is available in a development branch of Coq at http://github.com/maximedenes/native-coq, and in a future release of the proof assistant.

1 Implementations for the λ-calculus

1.1 Calculus of symbolic reduction

Finding the normal form of a term t by evaluation hinges upon distinguishing the head of the value of t, to continue reduction in the body of the abstraction if the value of t is of the form $\lambda x.\ t'$. However, t' is not, in general, a closed term (x can appear free), which evaluators for functional programming languages cater for exclusively.

The symbolic calculus of [11] introduces a new kind of value to represent free variables and terms whose evaluation is "stuck" because of a free variable in head position, as well as a new reduction rule. Hence, the weak reduction of open terms can be simulated by weak reduction of closed symbolic terms. The syntax of the symbolic calculus is as follows:

$$\begin{aligned} \textbf{Term} \ni t &::= x \mid t_1\, t_2 \mid v \\ \textbf{Val} \ni v &::= \lambda x.t \mid [\tilde{x}\, v_1 \ldots v_n] \end{aligned}$$

Where $[\tilde{x}\, v_1 \ldots v_n]$ is a value, called an accumulator, representing the free variable x applied to the arguments $v_1 \ldots v_n$. The reduction rules of the calculus are:

$$\begin{aligned} (\lambda x.\, t)\, v &\to t\{x \leftarrow v\} && (\beta_v) \\ [\tilde{x}\, v_1 \ldots v_n]\, v &\to [\tilde{x}\, v_1 \ldots v_n\, v] && (\beta_s) \\ \Gamma(t) &\to \Gamma(t') \quad \text{if } t \to t' \quad (\text{with } \Gamma ::= t\,[\,] \mid [\,]\, v) && \text{context} \end{aligned}$$

The β_v rule is the standard β-reduction rule in call by value[4], the context rule allows reduction in any subterm which is not an abstraction (so called weak reduction). Finally, the β_s rule expresses that free variables behave like boxes accumulating their arguments when applied.

We write $\overset{*}{\to}$ for the reflexive and transitive closure of $\to$. We define the value $\mathscr{V}(t)$ of a closed symbolic term t as the normal form of t for the relation $\overset{*}{\to}$. Since infinite reduction sequences are possible, this normal form does not necessarily exist. However, if the normal form exists then it must be a value because the reduction cannot get stuck on closed symbolic terms [11].

Given a translation from terms t to symbolic terms $[\![t]\!]$, we can now express precisely how to get the normal form of t with respect to the β rule, by iteration of weak symbolic reduction and readback: first, compute $\mathscr{V}([\![t]\!])$ by weak symbolic reduction (equation 1); second, inspect the resulting value and recursively normalize the subterms (*readback*).

$$\begin{aligned} \mathscr{N}(t) &= \mathscr{R}(\mathscr{V}(t)) && (1) \\ \mathscr{R}(\lambda x.t) &= \lambda y.\mathscr{N}((\lambda x.t)\,[\tilde{y}]) \text{ where } y \text{ is fresh} && (2) \\ \mathscr{R}[\tilde{x}\, v_1 \ldots v_n] &= x\, \mathscr{R}(v_1) \ldots \mathscr{R}(v_n) && (3) \end{aligned}$$

[4] We commit to call-by-value in the rest of this paper, but we could equally well have chosen any other standard evaluation strategy, such as call-by-need. Anyway, since CIC is strongly normalizing and confluent, the result is independent of the evaluation strategy.

The normalization algorithm takes a closed symbolic term and returns a λ-term in normal form. If the value is an accumulator the readback function simply injects the symbolic variable $\tilde{x}$ into the source variable x and applies it to the readback of the accumulated arguments (equation 3).

If the value is a function $\lambda x.\ t$, the readback function simply normalizes the application of the function to a fresh[5] symbolic variable $[\tilde{y}]$ (equation 2). Note that the application reduces in one step to $t\{x \leftarrow [\tilde{y}]\}$, so $\mathcal{N}((\lambda x.t)[\tilde{y}])$ will return the normal form of t modulo renaming of x into y. This trick is key to using stock evaluators for symbolic reduction: it means that we do not need to be able to peek inside a function to examine its body. Functions are black boxes whose only observable behaviour is through applying it to some value, as usual.

In [11], the authors use a modification of the OCaml abstract machine to get an efficient reducer for the symbolic calculus. This come at a price: while the implementation effort is reduced, the new abstract machine and associated compiler must be maintained separately, *ad infinitum*. Furthermore, the efficiency is limited to what abstract machines can afford us, which is often much less than compilation to native code. This paper could have been about a new ahead-of-time or just-in-time native code compiler for this abstract machine, but such a specialist task should hardly be the concern of developers of proof assistants and would likely achieve little over reusing as is an existing compiler to native code for an existing language. In the next section, we present a modular interface that we instantiate in two ways, giving two implementations of the above symbolic calculus, both times as vanilla OCaml programs. The first is inspired by higher order abstract syntax (HOAS) and the second uses the reflective capabilities of OCaml.

1.2 Abstract setting

To perform the normalization of a source term (here a λ-term), we first translate it into a program of our target language (here OCaml) that computes a value. Then, by inspection of the head of the obtained value, we will readback the value into a source term in normal form. To do so we assume that we have a module for values with the following interface:

```
module type Values = sig
  type t
  val app : t -> t -> t
  type atom =
    | Var of var
  type head =
    | Lam of t -> t
    | Accu of atom * t list
  val head : t -> head
  val mkLam : (t -> t) -> t
  val mkAccu : atom -> t
end
```

[5] The freshness condition here can be made precise by using de Bruijn *levels* for symbolic variables.

The first component of the signature is the type representing values. We assume that given two values we are able to compute the value corresponding to the application of one to the other (the app function). Secondly we assume that we are able to discriminate on the head of any value (the head function). In the case of the λ-calculus, a value is either a λ-abstraction (constructor `Lam`) or an accumulator which is an atom applied to its arguments[6] (constructor `Accu`). Finally, we assume an injection function for atoms. We assume that term abstractions are represented as OCaml functions, with `mkLam` injecting functions to terms. The following laws should hold: `head (mkAccu a) = Accu(a,[])` and `head (mkLam f) = Lam f`.

The compilation of a λ-term to an OCaml program is easily given, as follows:

$$\begin{aligned}
[\![x]\!]^B &= \begin{cases} x & \text{if } x \in B \\ \mathsf{mkAccu}(\mathsf{Var}\ x) & \text{otherwise} \end{cases} \\
[\![\lambda x.t]\!]^B &= \mathsf{mkLam}\,(\mathsf{fun}\ x \to [\![t]\!]^{B\cup\{x\}}) \\
[\![t_1\ t_2]\!]^B &= \mathsf{app}\ [\![t_1]\!]^B\ [\![t_2]\!]^B
\end{aligned}$$

The compiler takes as input a λ-term t and a set of bound variables B, returning an OCaml program computing the weak head normal form of t, viewed as a symbolic value. Bound variables in t are compiled to OCaml variable. Otherwise, the code builds the accumulator $[\tilde{x}]$ corresponding to the symbolic variable $\tilde{x}$. The compilation of an abstraction builds an OCaml function (the set B of bound variables is extended with bound variable x). For the application we use the app function to perform application of the functional part to the argument.

The normalization algorithm is thus a straightforward translation of the normalization algorithm presented in the previous section.

$$\begin{aligned}
\mathcal{N}_\Lambda\ t &= \mathcal{R}_V\ [\![t]\!]^\emptyset \\
\mathcal{R}_V\ v &= \mathcal{R}\ (\mathsf{head}\ v) \\
\mathcal{R}\ (\mathsf{Lam}\ f) &= \lambda y.\mathcal{R}_V\ (f\ (\mathsf{mkAccu}\ (\mathsf{Var}\ y))\ \text{where } y \text{ is fresh} \\
\mathcal{R}\ (\mathsf{Accu}(a, [v_n; \ldots v_1])) &= (\mathcal{R}_A\ a)\ (\mathcal{R}_V\ v_1) \ldots (\mathcal{R}_V\ v_n) \\
\mathcal{R}_A\ (\mathsf{Var}\ x) &= x
\end{aligned}$$

1.3 Tagged normalization

A natural implementation for the type `t` of the `Values` module, suggested in [4, 1], consists in using the type head directly:

```
type t = head
let head v = v
let app t v = match t with
  | Lam f -> f v
  | Accu(a, args) -> Accu(a,v::args)
let mkLam f = Lam f
let mkAccu a = Accu(a,[])
```

[6] Arguments are stored in reverse order to allow efficiently extending the list of arguments when applying an accumulator.

In this case, much of the implementation follows immediately from the laws above. If the first argument models an abstraction, the `app` function must unbox the first argument to get the representing function and perform the substitution (modelled as application of OCaml values). Otherwise, the first argument is an accumulator. The fact that an accumulator was applied to `v` is recorded by extending the list of arguments of the accumulator with this new argument.

The representation we described features a succinct implementation. However, explicit tagging of all values to inform about the shape of their heads entails a sizeable performance penalty. Several optimizations are suggested in [4] to mitigate the impact of the costly representation of application, such as uncurrying or specialization of constructors. Although the improvement is significant, we remove the cost of tagging entirely by doing away with tags. We show in the next section how to encode accumulators as special infinite arity functions. Applications then no longer need to dispatch on heads, given this uniform calling convention for both ordinary functions and accumulators.

1.4 Tagless normalization

Already Grégoire and Leroy [11] remark that an accumulator can be viewed as a primitive function keeping a trace of all arguments fed to it. We show how to write such a function within OCaml, without extending the language with a new primitive.

The OCaml runtime manipulates a variety of different kinds of values: integers, floats, or pointers to arrays, constructed values of a user defined datatype, closures, etc. At a low-level, integers are distinguished from pointers to heap allocated blocks by the content of their least significant bit, which is always set to 1 in the case of integers[7]. A memory block, written $[T : v_0; ...; v_n]$, is composed by its tag T (a small integer) and its fields $v_0 \dots v_n$. Closures are encoded by a block $[T_\lambda : C; v_1; \dots; v_n]$ where the first field C is a code pointer and $v_1, \dots; v_n$ are the values associated to the free variables of the function (i.e. the environment of the closure).

In [11], accumulators are represented using the same layout as closures: $[0 : \texttt{ACCU}; k]$ where k is the memory representation of the accumulator and `ACCU` is a code pointer to a single instruction. When applied to an argument this instruction builds a fresh accumulator block containing the representation of k applied to the new argument. The major advantage of this technique is that the compilation scheme of the application is unchanged (accumulators can be seen as closures), so there is no penalty to the evaluation of an application. In particular, there is no penalty to the evaluation of closed terms. A second advantage is that the tag used for accumulator blocks is 0 which allows to distinguish the kind of block we obtain by a simple inspection of the tag (the tag T_λ used for closure is not 0).

Our idea is to use the same trick but directly in OCaml. Remember that an accumulator is a function expecting one argument, accumulating this argument and recursively returning an accumulator. Such a function, can be defined as follows[8]:

[7] This information is used by the garbage collector, and it is the reason why Ocaml integers are limited to 31 (resp. 63) bits on a 32 (resp. 64) bits machine.

[8] The `-rectypes` option is needed.

```
type t = t -> t
let rec accu atom args = fun v -> accu atom (v::args)
let mkAccu atom = accu atom []
```

Given an atom `a` the value of `mkAccu a` is a function expecting one argument `v`. This argument is stored in the list `args`, and the result is itself an accumulator. This is what we expect, but the tag of the object is T_λ and not 0. Fortunately, the tag of objects can be changed using the `Obj` module of Ocaml. This leads to the following code for `accu`:

```
let rec accu atom args =
  let res = fun v -> accu atom (v::args) in
  Obj.set_tag (Obj.repr res) 0; (res : t)
```

The result is an object of type `t` and its tag is now 0. Finally we need to write the `head` function. For this we inspect the tag of the value: if the tag is not 0 then it is a closure so we can return the value itself; if the tag is 0 we need to get the atom and the accumulating arguments. The atom is stored a position 3 in the closure and the list of arguments at position 4. The `Obj.field` function allows to get them. This leads to the following code:

```
type t = t -> t
let app f v = f v
let mkLam f = f
let getAtom o = (Obj.magic (Obj.field o 3)) : atom
let getArgs o = (Obj.magic (Obj.field o 4)) : t list
let rec head (v:t) =
  let o = Obj.repr v in
  if Obj.tag o = 0 then Accu(getAtom o, getArgs o) else Lam(v)
```

Note that the `app` function simply performs the application (without matching the functional part) and that the `mkLam` function is the identity. In practice, these operators are inlined by the compiler and hence effectively disappear in the output code. The tags in the previous implementation played two roles: they allowed `App f t` to do the right thing depending on whether `f` was a function or an accumulator, and guide the readback. With our tagless implementation, we have made the calling convention for functions and accumulators uniform, and rely on the runtime of the target language to inform readback. We do not need tags during readback because the runtime can already distinguish between different kinds of values. Finally, the presence of unsafe operations, like `Obj.repr` and `Obj.magic` (which are untyped identity functions), is not a matter of concern in the sense that our *source* language is typed so we have no particular safety requirement in our *target* language.

2 Extension to the Calculus of Inductive Constructions

In this section, we extend our approach to the Calculus of Inductive Constructions. Decidability of type checking only holds for *domain-full* terms, where variables are all explicitly annotated with their type at their binding site. However, one can still safely erase annotations on terms of the CIC when testing convertibility [2]. Furthermore, given a term with erased annotations it is possible to recover the type annotations from its type. We hence only consider a domain-free variant of the CIC.

2.1 The symbolic CIC

The syntax of the symbolic calculus is extended with sorts, dependent products, inductive types, constructors, pattern matching and fixpoints:

$$\begin{aligned}
\textbf{Term} \ni t, P &::= x \mid t_1\, t_2 \mid v \mid C_i(\boldsymbol{t}) \mid \textbf{case}_{\langle P\rangle}\,\textbf{of}\,(C_i(\boldsymbol{x_i}) \to t_i)_{i\in I} \mid \textbf{fix}_m\,(f : T := t)\\
\textbf{Val} \ni v &::= \lambda x.t \mid [k\, \boldsymbol{v}] \mid C_i(\boldsymbol{v})\\
\textbf{Atom} \ni k &::= \tilde{x} \mid s \mid \Pi x : t.t \mid \textbf{case}_{\langle P\rangle}\, k\,\textbf{of}\,(C_i(\boldsymbol{x_i}) \to t_i)_{i\in I} \mid \textbf{fix}_m\,(f : T := t)
\end{aligned}$$

It is worth noting that we only represent here fully applied constructors. Indeed, since the η rule is admissible for the Calculus of Inductive Construction (and available in recent implementations of Coq), we may perform η-expansions where needed to preserve this invariant.

$$\textbf{case}_{\langle P\rangle}\, C_i(\boldsymbol{v})\,\textbf{of}\,(C_i(\boldsymbol{x_i}) \to t_i)_{i\in I} \to_\iota t_i\{\boldsymbol{x_i} \leftarrow \boldsymbol{v}\} \qquad (\iota_v^1)$$

$$\textbf{case}_{\langle P\rangle}\,[k]\,\textbf{of}\,(C_i(\boldsymbol{x_i}) \to t_i)_{i\in I} \to_\iota [\textbf{case}_{\langle P\rangle}\, k\,\textbf{of}\,(C_i(\boldsymbol{x_i}) \to t_i)_{i\in I}] \qquad (\iota_s^1)$$

$$\textbf{fix}_m\,(f : T := t)\, v_1 \dots v_{m-1}\, C_i(\boldsymbol{v}) \to_\iota t\{f \leftarrow \textbf{fix}_m\,(f : T := t)\}\, v_1 \dots v_{m-1}\, C_i(\boldsymbol{v}) \qquad (\iota_v^2)$$

$$\textbf{fix}_m\,(f : T := t)\, v_1 \dots v_{m-1}\, [k] \to_\iota [\textbf{fix}_m\,(f : T := t)\, v_1 \dots v_{m-1}\, k] \qquad (\iota_s^2)$$

The rules ι_v^1 and ι_v^2 are the usual reduction rules for case analysis and fixpoints in the CIC. Fixpoints reduce only if their recursive argument (with index denoted by m) is a constructor. This prevents infinite unrolling of fixpoints during normalization. ι_s^1 and ι_s^2 are the symbolic counterparts, and handle the case when the argument is an accumulator. A new accumulator is created to represent the application of a fixpoint or a case analysis that cannot be reduced any further.

$$\begin{aligned}
\mathscr{R}_A(\textbf{case}_{\langle P\rangle}\, k\,\textbf{of}\,(C_i(\boldsymbol{x_i}) \to t_i)_{i\in I}) &= \textbf{case}_{\langle P\rangle}\, \mathscr{R}(k)\,\textbf{of}\,(C_i(\boldsymbol{x_i}) \to \mathscr{N}(f\,(C_i(\overrightarrow{[\tilde{x}_i]}))))_{i\in I}\\
&\quad \text{where } f = \lambda x.\textbf{case}_{\langle P\rangle}\, x\,\textbf{of}\,(C_i(\boldsymbol{x_i}) \to t_i)_{i\in I}\\
\mathscr{R}_A(\textbf{fix}_m\,(f : T := t)) &= \textbf{fix}_m\,(f : T := \mathscr{N}((\lambda f.t)\,[\tilde{f}]))
\end{aligned}$$

Fig. 1. Readback algorithm for pattern matching and fixpoints

Fig. 1 describes the readback algorithm. Readback of an accumulator representing a case analysis requires to normalize branches. As is the case for abstractions, bodies cannot be accessed directly, hence the need to apply the expression to trigger reduction. More precisely, if $\textbf{case}_{\langle P\rangle}\, t\,\textbf{of}\,(C_i(\boldsymbol{x_i}) \to t_i)_{i\in I}$ is in weak head normal form (t evaluates to an accumulator), we apply $\lambda x.\textbf{case}_{\langle P\rangle}\, x\,\textbf{of}\,(C_i(\boldsymbol{x_i}) \to t_i)_{i\in I}$ successively to the constructors $C_i(\boldsymbol{x_i})$ where the $\boldsymbol{x_i}$ are accumulators representing free variables, recursively applying readback on the result.

```
                $[\![s]\!]^B$ = mkAccu (Sort $s$)
        $[\![\Pi x : T.U]\!]^B$ = mkAccu (Prod($[\![T]\!]^B$, $[\![\lambda x.U]\!]^B$))
             $[\![C_i(\boldsymbol{t})]\!]^B$ = mkConstruct i [| $[\![\boldsymbol{t}]\!]^B$ |]
$[\![\mathbf{case}_{\langle P\rangle}\ t\ \mathbf{of}(C_i(\boldsymbol{x_i}) \to t_i)_{i\in I}]\!]^B$ =
                      let rec case c =
                        match c with
                        | $[\![C_1(\boldsymbol{x_1})]\!]^{B\cup\{x_1\}}$ -> $[\![t_1]\!]^{B\cup\{x_1\}}$
                        | ...
                        | $[\![C_n(\boldsymbol{x_n})]\!]^{B\cup\{x_n\}}$ -> $[\![t_n]\!]^{B\cup\{x_n\}}$
                        | _ -> mkAccu Match($\tilde{I}$,c,$[\![P]\!]^B$,case)
                      in case $[\![t]\!]^B$
$[\![\mathbf{fix}_m(f : T := t)]\!]^B$ =
                      let fnorm f = $[\![t]\!]^{B\cup\{f\}}$ in
                      let rec f =
                        mkLam (fun $x_1$ -> ... -> mkLam (fun $x_m$ ->
                        if is_accu $x_m$ then
                          mkAccu (Fix (fnorm,$[\![T]\!]^B$,$m$)) $x_1$ ... $x_m$
                        else fnorm f $x_1$ ... $x_m$) ... )
                      in f
```

Fig. 2. Compilation scheme of the extended calculus of symbolic reduction

2.2 The translation

The signature of our module `Values` needs to be extended accordingly to represent all possible heads and shapes of accumulators:

```
module type Values = sig
 type head =
   | ...
   | Construct of int * t array
 type atom =
   | Var of var
   | Sort of sort
   | Prod of t * t
   | Match of annot * t * t * (t -> t)
   | Fix of (t -> t) * t * int
  ...
  val mkConstruct : int -> t array -> t
end
```

Here again, we keep track only of the informations that are relevant for the conversion test. Constructors are identified by an index into the list of constructors of the inductive type it belongs to, and carry a vector of arguments. Case analyses are characterized by the term being matched, the predicate which expresses the (possibly dependent) return type, and the branches.

The compilation scheme is extended accordingly, as shown in Fig. 2, where `is_accu` is a simple auxiliary function defined as:

```
let is_accu v = match head v with
```

```
  | Accu _ -> true
  | _ -> false
```

Compilation of pattern matching builds a recursive closure which can reduce to a branch if applied to a constructor or otherwise to an accumulator storing information that is necessary for the reification. In particular, the recursive closure `case` is stored in the accumulator. This function plays the role of $\lambda x.\mathbf{case}_{\langle P\rangle}\, x\, \mathbf{of}(C_i(\boldsymbol{x_i}) \to t_i)_{i\in I}$ in the readback algorithm (cf Fig. 1). The use of such a recursive closure prevents exponential duplication of code.

The same kind of trick is used for fixpoints: the `fnorm` function parametrized by `f` encapsulates the body and the fixpoint itself is represented by a recursive closure expecting m arguments. If the m-th argument is an accumulator, then rule ι_s^2 applies, otherwise the fixpoint is reduced, as in ι_v^2.

Following our first approach (with explicit tagging), a concrete implementation of constructors could be:

```
mkConstruct i args = Construct(i,args)
```

Instead, we map inductive types and constructors of the CIC to datatypes and constructors of the host language, thus avoiding any allocation overhead, superfluous indirections and benefiting from the host language's efficient implementation of pattern matching. Thus, the following inductive type

```
Inductive I := C1 : T1 | ... | Cn : Tn
```

is translated to:

```
type I = Accu_I of t | C1 of t * ··· * t | ... | Cn of t * ··· * t
```

where the signatures match the arity of each constructor. This allows us to interpret a constructor in the source term by a constructor of the host language.

```
mkConstruct i v = Obj.magic Ci(v)
```

OCaml represents non-constant constructors by a memory block and distinguishes them according to their tag. Since `Accu_I` is the first non-constant constructor of the generated type, it will be attributed tag 0. This is compatible with our function `head`, which relies on the fact that we reserved tag 0 for accumulators.

2.3 Optimizations

To make the compilation scheme presented above more explicit, let us consider the addition over naturals defined as:

```
Fixpoint add (m n:nat):= match m with O => n | S p => S(add p n) end.
```

Strict application of our compilation scheme would yield:

```
let norm_add f m n =
  let rec case_add m = match m with
    | Accu_nat _ ->
      mk_sw_accu [...] (cast_accu m) pred_add (case_add f n)
    | Construct_nat_0 -> n
    | Construct_nat_1 p -> Construct_nat_1 (f p n)
in
let rec add m n = if is_accu m then
    mk_fix_accu [...] fixtype_add normtbl_add m n
  else norm_add add m n
in add
```

In the code above, some type information and annotations necessary to the reification have been elided, while some others are referred to by `pred_add`, `fixtype_add` and `normtbl_add`.

However, in our real implementation, several optimizations are performed. First, in order to avoid building at each recursive call a closure representing the pattern matching, we inline one level of pattern matching. Then, if a fixpoint starts a case analysis on the recursive argument, as it is often the case, we can avoid the call to `head` since the pattern matching will capture the case when the argument is an accumulator. On our function `add`, the final code looks like:

```
let rec case_add f n m = match m with
  | Accu_nat _ ->
    mk_sw_accu [...] (cast_accu m) pred_add (case_add f n)
  | Construct_nat_0 -> n
  | Construct_nat_1 p -> Construct_nat_1 (f p n)

let norm_add f m n = case_add f n m

let rec add m n = match m with
  | Accu_nat _ -> mk_fix_accu [...] fixtype_add normtbl_add m n
  | Construct_nat_0 -> n
  | Construct_nat_1 p -> Construct_nat_1 (f p n)
```

3 Coinductive types

The Calculus of (Co)Inductive Constructions supports the definition of co-recursive data, which can be built using constructors of coinductive datatypes or using cofixpoints. Co-recursive data can be infinite objects like the elements of type `stream`:

```
CoInductive stream := Cons : nat -> stream -> stream.
CoFixpoint sone := Cons 1 sone.
CoFixpoint snat x := Cons x (snat (1+x)).
```

`sone` represents the infinite list of 1 and `snat x` the infinite list of naturals starting from `x`. To prevent infinite unrolling, the evaluation of cofixpoint is lazy. This means that `sone` is in normal form, and so is `snat 0`. Only pattern matching can force the evaluation of a cofixpoint, the reduction rule is the following[9]:

$$\begin{array}{lcl}\textbf{case}\ c\ \vec{a}\ \textbf{with}\ \ldots & \longrightarrow & \textbf{case}\ (t\{f \leftarrow c\})\ \vec{a}\ \textbf{with}\ \ldots\\ \text{where } c = \textbf{cofix}\ f\ := t & & \end{array}$$

[9] The guard condition ensures that the reduction of a cofixpoint always produces a constructor.

Straightforward implementation of the reduction rule would lead to an inefficient evaluation strategy, since there is no sharing between multiple evaluations of $c\ \boldsymbol{a}$. To get an efficient strategy for cofixpoints, we use the same idea as OCaml, which roughly consists in representing a term of type `'a Lazy.t` by a reference either to a value of type `'a` (when the term has been forced) or to a function of type `unit -> 'a`. When forcing a lazy value, two cases appear: the reference points to the result of a previous evaluation, which can be directly returned, or to a function, in which case it is evaluated and the result is stored in the reference.

However, reduction rules of cofixpoints require that we keep track of the original term which has been forced. This leads to the following implementation:

```
type atom =
  | ...
  | Acofix_e of t * (t -> t) * t
  | Acofix of t * (t -> t) * (unit -> t)

let update_atom v a =
   Obj.set_field (Obj.magic v) 3 (Obj.magic a)

let force v =
  if is_accu v then match get_atom v with
     | Acofix_e(_,_,v') -> v'
     | Acofix (t,norm, f)  ->
        let v' = app_list f (args_accu v) () in
        update_atom v (Acofix_e(t,norm v')); v'
     | _ -> v
  else v
```

To force a value, we first check if it is an accumulator. If not, it means that it is a constructor of a coinductive type which is returned unchanged. Otherwise, if the atom is an already evaluated cofixpoint, we return the stored result. If it is a cofixpoint which has not been evaluated, the function is applied to its accumulated arguments (through the `app_list` routine) and the accumulator is updated with a new atom. In the last case, the accumulator is a neutral term.

Coinductive types have strictly the same compilation scheme as inductive types (c.f. Section 2.2). For cofixpoints, we use the following scheme:

$[\![\textbf{cofix}\, f\ :\ T\ :=\ t]\!]^B =$

```
    let fnorm f = [[t]]^{B∪{f}} in
    let f = mk_accu dummy_atom in
    update_atom f (Acofix([[T]]^B, fnorm, fun _ -> fnorm f));
    f
```

This is directly adapted from the compilation of fixpoints (c.f. Section 2.2). It is worth mentioning that the accumulator `f` is created first with a dummy atom and then updated with the real one whose definition depends on `f` (under a lambda abstraction). We use this construction to circumvent a limitation on the right hand side of **let rec** in OCaml. Finally, the compilation of pattern matching adds a `force` if the matched term has a coinductive type.

	Standard reduction	**Bytecode interpreter**	**Native compilation**
BDD	4min53s (100%)	21,98s (7,5%)	11,36s (3,9%)
Four colour	not tested	3h7min (100%)	34min47s (18,6%)
Lucas-Lehmer	10min10s (100%)	29,80s (4,9%)	8,47s (1,4%)
Mini-Rubik	Out of memory	15,62s (100%)	4,48s (28,7%)
Cooper	not tested	48,20s (100%)	9,38s (19,5%)
RecNoAlloc	2min27s (100%)	14,32s (9,7%)	1,05s (0,7%)

	Standard reduction	**Bytecode interpreter**	**Native compilation**
BDD	6min1s (100%)	18,08s (5,0%)	11,28s (3,1%)
Four colour	not tested	2h24min (100%)	46min34s (32,3%)
Lucas-Lehmer	15min56s (100%)	21,59s (2,3%)	10,04s (1,1%)
Mini-Rubik	Out of memory	14,06s (100%)	3,99s (28,4%)
Cooper	not tested	37,88 (100%)s	10,18s (26,9%)
RecNoAlloc	4min3s (100%)	11,21s (4,6%)	1,47s (0,6%)

Table 1. Benchmarks run on a (a) 64 bits architecture and (b) 32 bits architecture, both with 4GB memory.

4 Benchmarks

In order to assess the performance of our low-level approach, we compared our implementation (**Native compilation**) with two preexisting implementations of term conversion in the Coq proof assistant. The first one (**Standard reduction**) is based on an abstract machine that manipulates syntactic representations of terms using a lazy evaluation strategy. The second one (**Bytecode interpreter**), is the bytecode based virtual machine using a call-by-value evaluation strategy described in [11].

To ensure a meaningful comparison, we extracted most of our benchmarks from real-world use cases in a variety of settings:
BDD is an implementation of binary decisions diagrams [16], which checks if a given proposition is a tautology. In our example, we ran it on an expression of the pigeonhole principle: if n pigeons are put in $n-1$ holes, there cannot be only one pigeon in each hole.
Four colour is the reducibility check of configurations in the formal proof of the four colour theorem by Gonthier and Werner [9], which represents most of the computation time of the whole proof.
Lucas-Lehmer is an implementation of Lucas-Lehmer primality test which decides if a given Mersenne number is prime or not.
Mini-Rubik checks that any position of the 2x2x2 Rubik's is solvable in at most 11 moves, using machine integers and arrays which we were able to port to our approach without extra cost, because the whole OCaml language is accessible to our compiler. The original formalization was described in [15].
Cooper implements Cooper's quantifier elimination on a formula with 5 variables.
RecNoAlloc triggers 2^{27} trivial recursive calls (i.e. without memory allocation to store the result). This aims at measuring pure performance impact, when garbage collection is not significantly involved.

We ran the benchmarks on two different architectures (32 and 64 bits), because some optimizations of Coq's bytecode interpreter like the use of threaded code are available only on 32-bits environments. This accounts for the different ratios between **Bytecode interpreter** and **Native compilation** since the latter is not impacted by such limitations.

Most of the results show a speed-up factor ranging from 2 to 5, which is typical of the expected speed-ups when going from bytecode interpretation to native-code compilation. It is worth noting that the performance improvement is particularly significant on examples involving less garbage collection. This is highlighted by **RecNoAlloc** where the speed-up factor lies between 7 and 14, depending on the architecture.

We also used **Cooper** and **RecNoAlloc** to assess the gap between our reference implementation of tagless normalization with a preliminary tagged version, achieving respectively 1.5 and 2.5 speed-up factors in favour of the former. Also, the program obtained through extraction on **RecNoAlloc** produces an OCaml program running at 76% of the time spent by its **Native compilation** equivalent. The performance penalty over the extracted version can be attributed to the overhead of having to check the guard condition at every recursive call, showing that our compilation scheme achieves close to the best performance that could possibly be achieved on a call-by-value evaluator.

Conclusion

The move towards greater automation and reduced proof sizes shifts away some of the burden of formalizing mathematics from the user and the tactics. The flip side is a greater pressure on the implementation of the proof checker, with the checking of the proof of the four colour theorem quite simply unfeasible without introducing some form of compilation of proof terms. Even checking based on a bytecode compilation scheme takes upward of 3 hours on a desktop machine. We have presented in this paper a more than fivefold improvement on this total checking time, enabling the widespread use of proof by reflection techniques in ever larger and greater developments in mathematics and software verification.

The approach we propose is cheap enough that it can readily be implemented in most any interactive proof environment where computing with proofs is pervasive, without expert knowledge on compilation technology. The correctness of our approach is in part contingent upon the correctness of the compiler, whose entire code enters the trusted base. However, the chosen compiler is already in the trusted base of the proof assistant if the target language of the translation described here and the implementation language of the proof assistant coincide. A certified compiler for the target language would certainly be of interest here to reduce the trusted base.

Since the whole target language is available to our translation routine, we have successfully implemented new features such as machine-native integers and persistent arrays nearly for free, whereas approaches based on ad-hoc compilers or runtimes require extra work each time an extension is needed.

Our tagged implementation is portable across any functional programming language, while our tagless implementation makes use of OCaml specific extensions that are already partially implemented in other typed languages. In particular, the `unpackClosure#` primitive of the GHC compiler for Haskell might well be sufficient for our purposes. Further investigation into the feasibility and trade-offs of this optimization for other languages is left as future work.

References

1. Aehlig, K., Haftmann, F., Nipkow, T.: A compiled implementation of normalization by evaluation. In: Mohamed, O., Muñoz, C., Tahar, S. (eds.) TPHOL, LNCS, vol. 5170, pp. 39–54. Springer Berlin / Heidelberg (2008)
2. Barras, B., Grégoire, B.: On the role of type decorations in the calculus of inductive constructions. In: Computer Science Logic, 19th International Workshop, CSL 2005, 14th Annual Conference of the EACSL, Oxford, UK, August 22-25, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3634, pp. 151–166. Springer (2005)
3. Berger, U., Schwichtenberg, H.: An inverse of the evaluation functional for typed λ-calculus. In: LICS'91. pp. 203–211 (1991)
4. Boespflug, M.: Conversion by evaluation. In: Proceedings of the Twelfth Internation Symposium on Practical Aspects of Declarative Languages. Madrid, Spain (2010)
5. Boutin, S.: Using reflection to build efficient and certified decision procedures. In: Abadi, M., Ito, T. (eds.) Theoretical Aspects of Computer Software, Lecture Notes in Computer Science, vol. 1281, pp. 515–529. Springer Berlin / Heidelberg (1997)
6. Coquand, T., Paulin, C.: Inductively defined types. In: Proceedings of the international conference on Computer logic. pp. 50–66. Springer-Verlag (1990)
7. Danvy, O.: Type-directed partial evaluation. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 242–257. POPL '96, ACM, St. Petersburg Beach, Florida, United States (1996)
8. Filinski, A., Rohde, H.K.: A denotational account of untyped normalization by evaluation. In: Proceedings of the 7th International Conference in Foundations of Software Science and Computation Structures. Springer Berlin / Heidelberg (2004)
9. Gonthier, G.: The four colour theorem: Engineering of a formal proof. In: Kapur, D. (ed.) ASCM. Lecture Notes in Computer Science, vol. 5081, p. 333. Springer Berlin / Heidelberg, Singapore (2007)
10. Gonthier, G., Mahboubi, A.: A Small Scale Reflection Extension for the Coq system. Research Report RR-6455, INRIA (2008)
11. Grégoire, B., Leroy, X.: A compiled implementation of strong reduction. In: Proceedings of the seventh ACM SIGPLAN international conference on functional programming. pp. 235–246. ACM (2002)
12. Grégoire, B., Mahboubi, A.: Proving equalities in a commutative ring done right in coq. In: Hurd, J., Melham, T. (eds.) Theorem Proving in Higher Order Logics, Lecture Notes in Computer Science, vol. 3603, pp. 98–113. Springer Berlin / Heidelberg (2005)
13. Harrison, J.: Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK (1995)
14. Lindley, S.: Normalisation by evaluation in the compilation of typed functional programming languages. Ph.D. thesis, University of Edinburgh. College of Science and Engineering. School of Informatics. (2005)
15. Thery, L.: Proof pearl: Revisiting the mini-rubik in coq. In: Theorem proving in higher order logics: 21st international conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008: proceedings. vol. 5170, p. 310. Springer Verlag (2008)
16. Verma, K.N., Goubault-Larrecq, J., Prasad, S., Arun-Kumar, S.: Reflecting bdds in coq. In: He, J., Sato, M. (eds.) ASIAN. LNCS, vol. 1961, pp. 162–181. Springer (2000)
17. Werner, B.: Une Théorie des Constructions Inductives. Ph.D. thesis, Université Paris-Diderot - Paris VII (05 1994)