View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by INRIA a CCSD electronic archive server

Simple simpl

Pierre Boutillier

To cite this version:

Pierre Boutillier. Simple simpl. 2013. hal-00816918

HAL Id: hal-00816918
https://hal.archives-ouvertes.fr/hal-00816918

Preprint submitted on 23 Apr 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Simple `simpl`

Pierre Boutillier

Univ Paris Diderot, Sorbonne Paris Cité, PPS, UMR 7126 CNRS, PiR2, INRIA Paris-Rocquencourt, F-75205 Paris, France

Abstract. We report on a new implementation of a reduction strategy in Coq to simplify terms during interactive proofs.
By "simplify", we mean to reduce terms as much as possible while avoiding to make them grow in size. Reaching this goal amounts to a discussion about how not to unfold uselessly global constants. Coq's `simpl` is such a reduction strategy and the current paper describes an alternative more efficient abstract-machine-based implementation to it

Introduction

This article discusses the issue of reduction in an extension of the λ-calculus that contains algebraic data structures, fixpoints and global constants. This is representative of the core of the programming language of a proof assistant such as Coq [9]. Its grammar is presented Fig.1.

$$\begin{array}{rl} \mathrm{t}, \mathrm{u} := \mathrm{x} \;|\; \mathrm{t}\ \mathrm{u} \;|\; \texttt{fun}\ \mathrm{x} \to \mathrm{t} \;|\; \mathrm{c} \;| \\ C_i \;|\; \texttt{case}\ \mathrm{t}\ \texttt{of}\ u_1 \ldots u_p\ \texttt{end} \;|\; \texttt{fix}\ \mathrm{f} := \mathrm{t} \\ \gamma := \varepsilon \;|\; \gamma; c\ :=\ t \end{array}$$

Fig. 1. Language syntax

Algebraic datatypes are introduced by constructors. The information stored is just "This is the ith constructor of its datatype". Algebraic datatypes are eliminated by what we call a *destructor*. It is either a case analysis alone or case analyses inside a fixpoint definition. Branches of a Case analysis do not bind arguments of constructors directly. The branch of a constructor that has k arguments must start by k functional abstractions.

In a proof assistant, this language would be strongly typed and have type annotations. To remain focused on our goal, we assume everything to be well-typed.

Our language is endowed with a small-step semantics characterized by four non-structural rules:

β-reduction functional elimination $\qquad \gamma \vdash (\texttt{fun}\ \mathrm{x} \to \mathrm{t})\ \mathrm{u} \mapsto \mathrm{t}[\mathrm{u}/\mathrm{x}]$
when a function is applied, the linked variable is substituted by the argument.

δ-reduction constant unfolding $\gamma;\, \mathrm{c} := \mathrm{t};\, \gamma' \vdash \mathrm{c} \mapsto \mathrm{t}$

A global constant is substituted by its definition.

ι-reduction Algebraic datatypes elimination

Pattern matching over the ith constructor is substituted by its ith branch applied to the constructor arguments.

$$\gamma \vdash \texttt{case}\ C_i\ u_1 \ldots u_n\ \texttt{of}\ |\ t_1 \ldots |\ t_p\ \texttt{end} \mapsto t_i\ u_1 \ldots u_n$$

A fixpoint applied to a constructor is substituted by its body.

$$\gamma \vdash (\texttt{fix}\ \mathrm{f} := \mathrm{t})\ (C_i\ u_1 \ldots u_)) \mapsto \mathrm{t}[\texttt{fix}\ \mathrm{f} := \mathrm{t}/\mathrm{f}]\ (C_i\ u_1 \ldots u_))$$

Big-step semantic can be obtained by building an abstract machine and following untyped normalization by evaluation technique [5, 8]. This technique answers by default a normal form whose length may be significantly bigger than the original term. Especially, when global constants and free variables are involved, unfolded form may be more difficult to recognize at first glance.

Reducing while remaining concise may seem contradictory but a simple example can illustrate what enhancement can be achieved over straightforward normalization. Take the unary integers defined as either zero (O) or the successor of an integer n ($S\ n$) and plus the constant defined as

plus := **fun** m →
  fix pl := **fun** n →
    case n **of** | m | **fun** n' → S (pl n') **end**.

The normal form w.r.t. $\beta\iota\delta$-reduction of plus x (S (S y)) is not S (S (plus x y)) but S (S (**fix** pl := **fun** n → **case** n **of** | x | **fun** n' → S (pl n') **end** y)).

We could have defined fixpoints in a *generative* way: a fixpoint is a toplevel object that defines a global constant and recursive calls are calls to this constant. However, the design considered here deals with anonymous fixpoints. Names are local to a fixpoint expression. Only the skeleton has a meaning and two definitions equivalent modulo renaming of binders are equivalents. We will not discuss further the difference between the two approaches. Further thoughts can be found in [10, 1]. Let us stop at the level of a user of a proof assistant: a goal is easier to read and more intuitive if it uses constants than expanded local definitions.

The Coq proof assistant features a popular reduction strategy called *simpl* performs reductions but unfold a constant only if

1. it leads to an algebraic data-structure destructor being in head position
2. this destructor can be eliminated by ι-reduction

Moreover, recursive calls are substituted by the constant that has been unfolded instead of the fixpoint definition.

While it is a corner-stone of many Coq proofs, its complexity is exponential in the number of constant that will not be unfolded. The reduction of plus m (plus n p) exhibits the undesirable behaviour.

1. It tries to unfold the first plus,
2. sees that the ι-reduction depends of the the result of plus n p
3. tries recursively to reduce plus n p

4. sees that it does not reduce to a constructor.
5. does not unfold the first plus but calls itself on the subterms.
6. reduces plus n p a second time.

The constant plus would have tried to be unfolded 2^n time if it appears n time.

The implementation of `simpl` is also unpredictable in its behavior when dealing with *cascade* of constants (constants that unfolds to a fixpoint via a chain of δ-reductions). Actually, unfolding depends of the context in which the cascade occurs.

In this paper, we propose a new implementation directly based on a variant of Krivine Abstract Machine with algebraic datatypes proposed by Bruno Barras in [2, Chapter 2] but behaves just like simpl with respect to constant unfolding. The key idea is to maintain a list of constants convertible to the term being reduced.

Outline This paper is organized as follows. In section 1, we propose an implementation of a call by name reduction strategy for our calculus. In section 2, we describe how we improve on the previous section to keep track of the constants that are unfolded. In section 3, we discuss how the use of explicit substitutions makes it possible to improve both on the efficiency of the reduction of our machine and on its ability to refolded terms. In section 4, we analyze the extent to which our reduction strategy may be fine tuned by the users. Section 5 proposes a discussion of some of our implementation choices.

1 Call By Name abstract machine

We use OCaml as the language to describe our abstract machine and assume that the standard library over lists is known.

Term reduction is performed by translating terms to a particular language, compute in this language and translating back to terms. It follows the general picture of untyped normalization by evaluation. First, a term is evaluated to an abstract machine *state* composed of a term, called the *head* and a stack [7]. Stacks store the destructors of types. All of them are defined by

```
type stack_element = Zapp of term | Zcase of term list
  | Zfix of var * term
type stack = stack_element list
type state = term * stack
```

Evaluation starts from the source term in front of an empty stack. It pushes destructors of types on the stack and computation occurs when a constructor of type appears in head position.

Computational steps are a rewriting in OCaml of the reduction rules describe page 1. (the evaluation function is given as a continuation for tail-recursivity).

```
(** val compute_arrow : var → term → stack →
      (state → state) → state (* beta *) *)
```

```
let compute_arrow x t stack k = match stack with
  | Zapp u :: q → k (subst t u x, q)
  | _ → (mkLam x t, stack)

(** val strip_zapp : stack → term list * stack *)
let strip_zapp s =
  let rec aux acc = function
    | Zapp x :: q → aux (x :: acc) q
    | s → List.rev acc, s
  in aux [] s

(** val compute_algebraic : int → stack →
    (state → state) → state (* iota *) *)
let compute_algebraic i stack k =
  let args, stack' = strip_zapp stack in
  match stack' with
  | Zcase l :: q →
    k (List.nth i l, (List.map Zapp args) @ q)
  | Zfix (f, t) :: q →
    k (subst t (mkFix f t) f,
       Zapp (mkApp (mkConstruct i) args) :: q)
  | _ → (mkConstruct i, stack)
```

The code for the evaluation function is:

```
(** val eval : env → term → state *)
let eval env t =
  let rec cbn = function
  | App (t, u), s → cbn (t, Zapp u :: s)
  | Case (t, l), s → cbn (t, Zcase l :: s)
  | Fix (f, t), Zapp u :: s → cbn (u, Zfix f t :: s)
  | Lam (x, t), s → compute_arrow x t s cbn
  | Construct i, s → compute_algebraic i s cbn
  | Const c, s → cbn (const_value env c, s)
  | state → state
  in cbn (t,[])
```

Then the state is quoted back to a term by putting back the surrounding type destructors.

```
(* val quote : state → term *)
let rec quote = function
  | t, Zapp u :: q → quote (mkApp t u, q)
  | t, Zcase l :: q → quote (mkCase t l, q)
  | u, Zfix (f, t) :: q →
    quote (mkFix f t, Zapp u :: q)
  | t, [] → t
```

This process returns a weak head normal form. It has to be mapped over the subterms to reach a strong normal form. It is also naive about efficiency because it performs the substitutions one by one. We will address this inefficiency issue in section 3.

2 Trace of unfolded constants

Refolding We add to the framework of section 1 a list of constants convertible to the head term in which we will pick the "refolded" form of the term. This list gives us the log of the chained δ-reduction done by the machine.

Formally, it is not strictly composed of constants but of triples we call *unfolding*.

```
type cst_stk = (cst * term list * term list) list
```

Elements of the first list are called the *parameters* and those of the second the *arguments*.

A list p of unfoldings is a *refolding* of a term t (written $t \Vdash p$) if for any of its elements, the constant applied to the parameters is convertible to `t` applied to the arguments.

We define primitives over lists of unfolding such that

- if $t\ u \Vdash p$ then $t \Vdash$ `add_arg u p`
- if `fun` $x \to t \Vdash p$ then $t[u/x] \Vdash$ `add_param u p`
- if $c \Vdash p$ and c := t then $t \Vdash$ `add_cst c p`

by

```
(** val add_arg : term → cst_stk → cst_stk *)
let add_arg arg =
  List.map (fun (a,b,c) → (a,b,c @ [arg]))
(** val add_param : term → cst_stk → cst_stk *)
let add_param param p = List.map (fun (a,b,c) →
  match c with |[] → (a,b @ [param],c) |_::q → (a,b,q)) p
(** val add_cst : cst → cst_stk → cst_stk *)
let add_cst cst p = (c,[],[]) :: p
```

It is chosen such that if the stack is put back onto the head. in the manner of the quote function, up to and including the considered node, the list of unfolding would be a refolding of the head.

```
type stack_element = Zapp of term
  | Zcase of term list * cst_stk
  | Zfix of var * term * cst_stk
```

We now modify cbn to maintain a refolding.

```
let compute_arrow x t stack p k = match stack with
  | Zapp u :: q → k (add_params u p) (subst t u x, q)
```

```
  | _ → (mkLam x t, stack)

let eval env t =
  let rec cbn p = function
  | App (t, u), s → cbn (add_arg u p) (t, Zapp u :: s)
  | Case (t, l), s → cbn [] (t, Zcase (l,p) :: s)
  | Fix (f, t), Zapp u :: s → cbn [] (u,Zfix(f,t,p) :: s)
  | Lam (x, t), s → compute_arrow x t s p cbn
  | Construct i, s → compute_algebraic i s (cbn [])
  | Const c, s → cbn (add_cst c p) (const_value env c,s)
  | state → state
  in cbn [] (t,[])
```

Nothing is refolded for now but everything is set in place to use the fact that for example plus applied to one argument is convertible to `fix` pl := .... The missing part to provide the expected behavior is to take advantage of this information during ι-reduction and quotation.

Best unfolding We define functions refold_in_term t p and refold_in_state t p that will look in terms and state respectively. If it finds t applied to the arguments of the first unfolding of p, it is substituted by the constant applied to the parameters. Otherwise, it tries with the next triple.

The constant unfolded first is the deepest in the refolding, *i.e.* the top element of the cascade of constants. It is the one that we want to use in priority. It is also the one with the longest number of arguments. So, it is tried first but its arguments are not found in any situation and it cannot always be used (see Section 5 for illustrative examples). That is why the other unfolding must be kept and tried.

Concise term reconstruction Finally, we try to use constant instead of algebraic data-structure destructor during computation for fixpoint substitution

```
let compute_algebraic i stack k =
  let args, stack' = strip_zapp stack in
  match stack' with
  | Zcase (l, _) :: q →
    k (List.nth i l, (List.map Zapp args) @ q)
  | Zfix (f, t, p) :: q → k
    (subst (refold_in_term (mkVar f) p) (mkFix f t) f,
     Zapp (mkApp (mkConstruct i) args) :: q)
  | _ → (mkConstruct i, stack)
```

and at quotation

```
let rec quote = function
  | t, Zapp u :: q → quote (mkApp t u, q)
  | t, Zcase (l, p) :: q →
    quote (refold_in_state p (mkCase t l, q))
```

```
| u, Zfix (f, t, p) :: q →
  quote (refold_in_state p (mkFix f t, Zapp u :: q))
| t, [] → t
```

3 Refolding Algebraic Krivine Abstract Machine

Our machine works but it is not optimally efficient. There is a computational inefficiency as seen at the end of section 1. Worse, its ability to refold is too limited. Improving substitution is the key to correct that, let us see how and why.

Defining

succ := plus (S O)

we have that $\underline{\text{succ } (S \text{ n})}$ reduces to $\underline{S \text{ (succ n)}}$ but compute_algebraic does not answer that in Coq.

In Coq's standard library [9], the default definition of plus is a bit different. The `fix` f := ... construction takes as an annotation its recursive argument. It is the argument that generates the fixpoint unfolding (and not necessary the first one). The `fun` m → ... of our definition is put inside the fixpoint body. We end with plus := `fix` pl := `fun` m n → ... pl m n' .... This means that during reduction, first the fixpoint is unfolded, then m is substituted.

In this situation, when the fixpoint pl is reduced, the cst_stk tells that pl applied to the argument (S O) is succ and that pl is plus. In the body of pl, pl m appears. Consequently, plus is chosen and not succ. The substitution of m by S O that would have allow a substitution to succ is only done later.

Substitution is done by maximally refolding constants. It brings to a complexity problem when the term will be reduced further later during evaluation. The unfolded version should be used directly under these circumstances. For example, during the evaluation of $\underline{\text{plus m } (S \; (S \text{ n}))}$, plus is δ-reduced, pl is ι-reduced by using plus m for the recursive call, the case is ι-reduced, we have got $\underline{S \text{ (plus m } (S \text{ n}))}$. Then, plus is δ-reduced a second time and so on...

Later substitutions offer the opportunity to both use unfolded form during evaluation and the best constant during quotation. It allows also to get the most precise possible environment while choosing the constant to use.

Explicit substitutions are a lazy way to substitute. Our abstract machine takes advantage of them. It becomes very close to the Krivine abstract machine: we use closures instead of terms in the state of the machine. The difference is that our substitutions are composed of pairs: a term (to proceed evaluation) and a cst_stk (to quote gently).

4 Configurability by user

You cannot provide to users a tool to reduce their goals without allowing them to customize how they want the reduction to proceed. Enrico Tassi [9, 4] proposed a command `Arguments` in Coq to specify the behavior of the user-level reduction regarding to a particular constant.

A constant can be

– never refolded
– unfolded if only it is applied to at least k arguments (with k bigger than its number of arguments, you have an opaque constant)
– unfolded iff a given list of its arguments starts by a constructor
– unfolded only if no `case` will remain in the normal form.

Our framework is suitable to handle the 3 first situations.

– A constant not to refold is a constant that you do not put in the cst_stk.
– By counting the number of node Zapp at the beginning of the stack, you know the number of arguments a term is applied to.
– Whether argument i starts by a constructor is exactly the question we handle to trigger a fixpoint ι-reduction.
  A extra node Zconst is added. It takes as arguments a constant, the list of the $(i-1)$th first arguments and a list of argument numbers (the one, we want that starts by a constructor). When cbn crosses such a constant, it puts the first argument to check in head of the state and pop a Zconst. Then, compute_algebraic puts in head the next argument to check or unfold the constant if the list is empty.
– An heuristic answering "will all the `case` that unfolding this constant introduces be reduced ?" seems very specific to a system that unfolds constant one by one and backtracks. We do not take it into account.

5 Discussion

Despite its configurabity, some choices are hard coded in our framework. They are made explicit and discussed now.

Constant cascade Functional programmers often work by defining new constants using general combinators on data-structure (fold, map) instead of direct recursive definition. During proofs, it seems preferable to get goals talking about the toplevel constant instead of the combinator. Therefore, the machine deals with a list of constants with arguments and parameters and not only the last unfolded constant. In that respect, it goes further than just simulating a generative system.

Reduce or refold The combinator map does not change the "iterated" function. A constant defined from it can always be refolded. With fold_left, the accumulator changes. Consequently, going back to a toplevel constant that uses fold_left can be impossible. Answering a reduced result expressed using fold_left has been preferred. The former implementation does not reduce the term in order to leave it written using the toplevel constant. Here is the backward incompatibility between the implementations.

Guessing refolding The presented way of dealing with unfolding is too sensitive to the syntax and implies an expertise of the user to define constants that will be refolded. A single swap of arguments and the system is lost. If plus would have been defined by recurrence on its first argument and succ m by plus m (S O), the information stored in the cst_stk during the evaluation of succ (S n) would have been that pl (S n) (S O) is equivalent to succ (S n). This is true but useless since the recursive call talks about pl n (S O).

q

Split tree The reason why one can want not to unfold if several arguments does not reduce to a constructor can be seen with the example of substraction.

minus := `fix` mn := `fun` a b →
  `case` b `of` | a | `fun` b' →
   `case` a `of` | O | `fun` a' → mn a' b' `end end`

Only when both arguments start with a constructor will the second case be eliminated. However, the following implementation of division by 2shows that we need a more complex strategy.

div2 := `fix` d2 := `fun` m →
  `case` m `of` | O | `fun` n →
   `case` n `of` | O | `fun` p → S (d2 p) `end end`

In the current implementation, there is no way to specify constaints such as unfold only if it is 0, 1 or the successor of the successor of a number ...

Control number of reduction Directives such as "Do unfold" or "do not unfold" may not be precise enough. We have imagined a system where a δ-reduction consumes a token as long as some are available. The user can choose, for each constant, how many tokens it is allowed to use. It is useful, for instance, if you want to reduce function of $n + 2$ as a function of $n + 1$ but not of n.

Conclusion

We have described a Krivine abstract machine with algebraic datatypes that stores lists of equivalent form for the term it handles. We believe it is a good way to reconcile in a proof assistant a notion of local fixpoint and a generative intuition of users.

This new reduction strategy improves upon the former by offering a uniform behaviour for cascade of constants. Furthermore its implementation of constant refolding does not affect the complexity of the underlying Krivine Machine. It is available in Coq's development branch as the *cbn* tactics.

Besides its use in proof development, our abstract machine is leveraged by Coq's implicit argument inference mechanism to produce shorter terms.

It is, however, sensitive to how constants are defined. Moreover, it can be difficult to avoid undesirable reductions. Therefore, it cannot pretend to fully replace functional induction [3] or more step-by-step rewriting techniques [11].

References

1. Asperti, A., Ricciotti, W., Sacerdoti Coen, C., Tassi, E.: The matita interactive theorem prover. In Bjørner, N., Sofronie-Stokkermans, V., eds.: Automated Deduction – CADE-23. Volume 6803 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2011) 64–69
2. Barras, B.: Auto-validation d'un système de preuves avec familles inductives. Thèse de doctorat, Université Paris 7 (November 1999)
3. Barthe, G., Forest, J., Pichardie, D., Rusu, V.: Defining and reasoning about recursive functions: a practical tool for the Coq proof assistant. In: Functional and Logic Programming (FLOPS'06), Fuji Susono, Japon (2006)
4. Gonthier, G., Mahboubi, A., Tassi, E.: A Small Scale Reflection Extension for the Coq system. Rapport de recherche RR-6455, INRIA (2008)
5. Grégoire, B.: Compilation des termes de preuves: un (nouveau) mariage entre Coq et Ocaml. Thése de doctorat, spécialité informatique, Université Paris 7, école Polytechnique, France (December 2003)
6. Krivine, J.L.: Un interpréteur du lambda-calcul. Unpublished (1986)
7. Krivine, J.L.: A call-by-name lambda-calculus machine. Higher-Order and Symbolic Computation (HOSC) (2005) Extended version of [6].
8. Löh, A., McBride, C., Swierstra, W.: A tutorial implementation of a dependently-typed lambda calculus. Fundamentae Informatica **102** (April 2010) 177–207
9. The Coq development team: The Coq proof assistant reference manual. INRIA - CNRS - LIX - LRI - PPS. (2012) Version 8.4.
10. Paulin-Mohring, C.: Définitions Inductives en Théorie des Types d'Ordre Supérieur. Habilitation à diriger les recherches, Université Claude Bernard Lyon I (December 1996)
11. Sozeau, M.: Compilation du filtrage dépendant. Working version in french (2008)