A native compiler for Coq: current status, perspectives and discussion[1]

Maxime Dénès, *jww M. Boespflug and B. Grégoire*

Marelle Team, INRIA Sophia-Antipolis

February 19, 2013

[1] This work has been partially funded by the FORMATH project, nr. 243847, of the FET program within the 7th Framework program of the European Commission.

Context

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash A \equiv_{\beta\delta\iota\zeta} B}{\Gamma \vdash t : B}\ \text{conv}$$

Most conversion tests involve very little computation.

Context

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash A \equiv_{\beta\delta\iota\zeta} B}{\Gamma \vdash t : B}\ \text{conv}$$

Most conversion tests involve very little computation.

But some need a bit more:

```
Lemma red000to106 : reducible_in_range 0 106 the_configs.
Proof. CheckReducible. Qed.
```

Current implementations of the conversion rule

Bruno Barras' reduction machine

- Based on Krivine's Abstract Machine, CBN with sharing
- Well-suited for most tests, when smart comparison matters
- Limited performance for (very) heavy computations

Current implementations of the conversion rule

Bruno Barras' reduction machine

- Based on Krivine's Abstract Machine, CBN with sharing
- Well-suited for most tests, when smart comparison matters
- Limited performance for (very) heavy computations

Benjamin Grégoire's bytecode compiler

- Ad-hoc bytecode compiler and runtime, based on OCaml
- Call by value
- Compiled approach: slower for small tests, faster for heavy ones

Current implementations of the conversion rule

Bruno Barras' reduction machine

- Based on Krivine's Abstract Machine, CBN with sharing
- Well-suited for most tests, when smart comparison matters
- Limited performance for (very) heavy computations

Benjamin Grégoire's bytecode compiler

- Ad-hoc bytecode compiler and runtime, based on OCaml
- Call by value
- Compiled approach: slower for small tests, faster for heavy ones

Why a modified runtime?

Why a modified runtime?

Weak reduction

```
# let f x = print_endline "Reducing"; (fun y -> y) x;;
val f : 'a -> 'a = <fun>
# f;;
- : 'a -> 'a = <fun>
# f 42;;
Reducing
- : int = 42
```

Why a modified runtime?

Weak reduction

```
# let f x = print_endline "Reducing"; (fun y -> y) x;;
val f : 'a -> 'a = <fun>
# f;;
- : 'a -> 'a = <fun>
# f 42;;
Reducing
- : int = 42
```

Strong reduction

```
Coq < Eval compute in (fun (x:nat) => (fun y => y) x).
      = fun x : nat => x
      : nat -> nat
```

Compiling strong reduction

B. Grégoire and X. Leroy remark that strong reduction can be reduced to iterated weak evaluation on open terms.

In this talk, we will:

1. Show that their work is an instance of Normalization by Evaluation
2. Define another instance that can be encoded in regular OCaml
3. Improve over this first attempt to achieve better performance
4. Extend to full CIC
5. Discuss the current state of the implementation

Compiling strong reduction

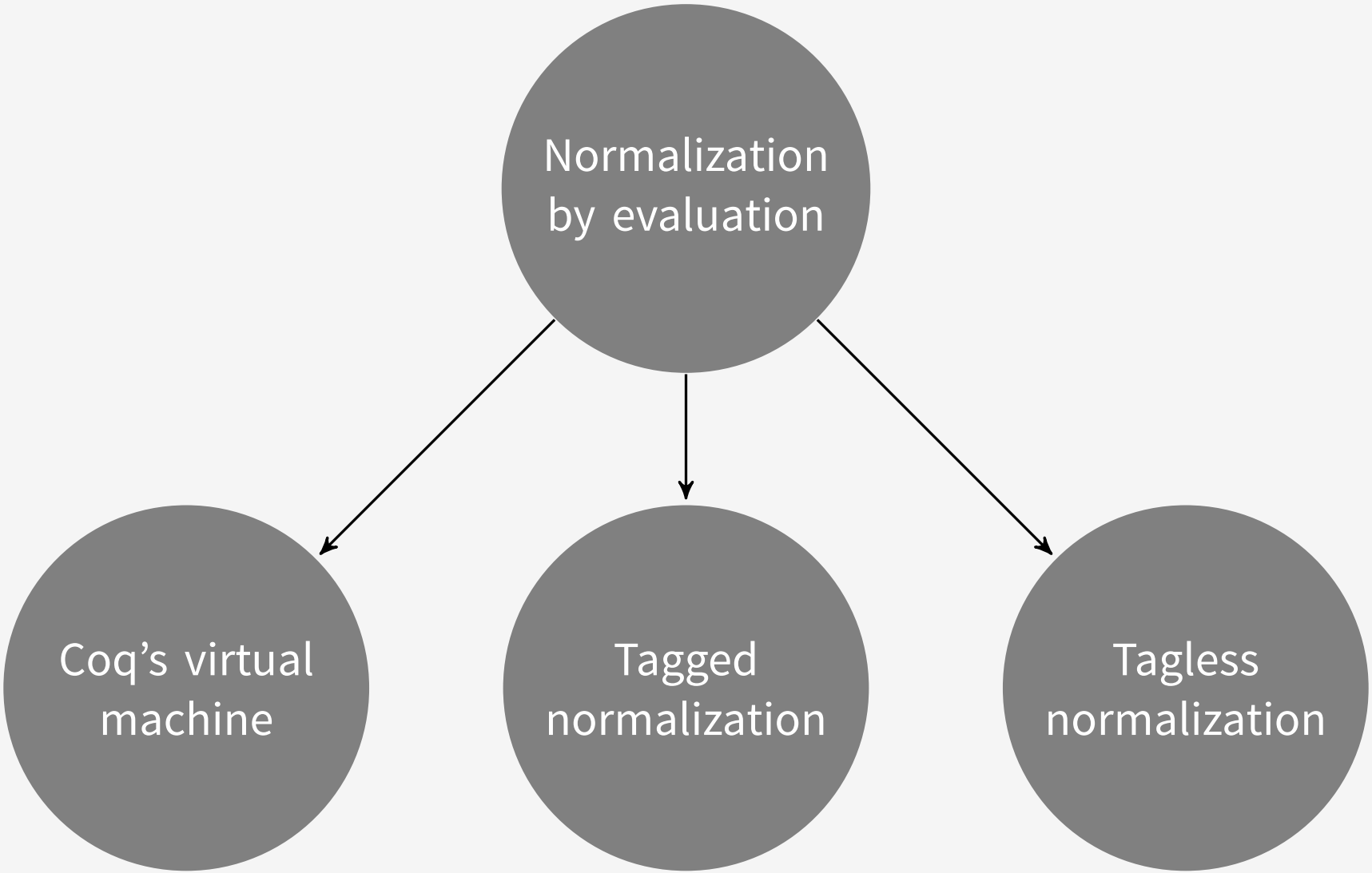
B. Grégoire and X. Leroy remark that strong reduction can be reduced to iterated weak evaluation on open terms.

In this talk, we will:

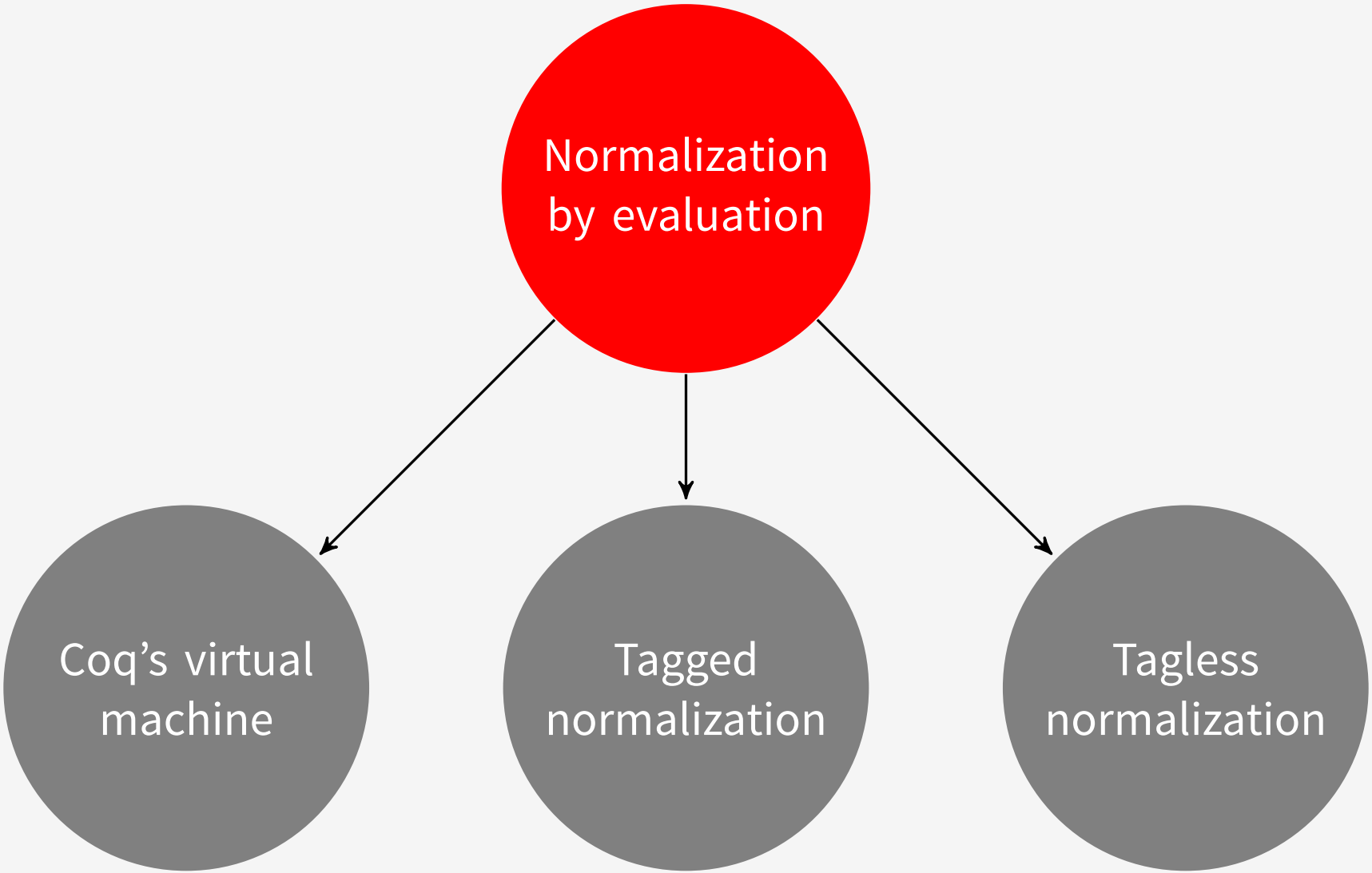
1. Show that their work is an instance of Normalization by Evaluation
2. Define another instance that can be encoded in regular OCaml
3. Improve over this first attempt to achieve better performance
4. Extend to full CIC
5. Discuss the current state of the implementation

Note: we will focus on normalization, but conversion is easier.

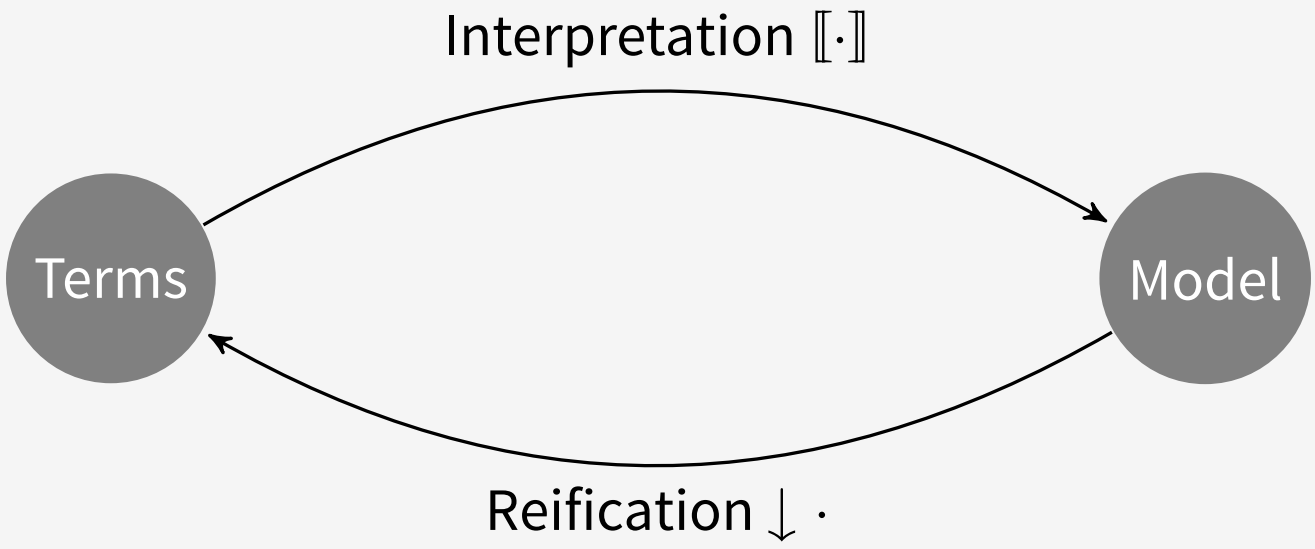
Outline


Normalization by evaluation
Coq's virtual machine
Tagged normalization
Tagless normalization

Outline


Normalization by evaluation
Coq's virtual machine
Tagged normalization
Tagless normalization

Normalization by evaluation



With:

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

Model: the calculus of symbolic reduction

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

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

Interpretation and reification

$$[\![\cdot]\!] = \mathcal{V}(\ulcorner\cdot\urcorner)$$

$\ulcorner\cdot\urcorner$: λ-terms $\longrightarrow$ closed symbolic terms

$\mathcal{V}(\cdot)$: weak symbolic reduction

$$\mathcal{N}(t) = \mathcal{R}(\mathcal{V}(\ulcorner t \urcorner))$$

$$\mathcal{R}(\lambda x.t) = \lambda y.\, \mathcal{R}(\mathcal{V}((\lambda x.t)\ [\tilde{y}])) \text{ where } y \text{ is fresh}$$

$$\mathcal{R}[\tilde{x}\, v_1 \dots v_n] = x\, \mathcal{R}(v_1) \dots \mathcal{R}(v_n)$$

Interpretation and reification

$$[\![\cdot]\!] = \mathcal{V}(\ulcorner\cdot\urcorner)$$

$\ulcorner\cdot\urcorner$: λ-terms $\longrightarrow$ closed symbolic terms

$\mathcal{V}(\cdot)$: weak symbolic reduction

$$\mathcal{N}(t) = \mathcal{R}(\mathcal{V}(\ulcorner t \urcorner))$$

$$\mathcal{R}(\lambda x.t) = \lambda y.\, \mathcal{R}(\mathcal{V}((\lambda x.t)\ [\tilde{y}])) \text{ where } y \text{ is fresh}$$

$$\mathcal{R}[\tilde{x}\, v_1 \dots v_n] = x\, \mathcal{R}(v_1) \dots \mathcal{R}(v_n)$$

The interface must provide ways to:

Interpretation and reification

$$[\![\cdot]\!] = \mathcal{V}(\ulcorner\cdot\urcorner)$$

$\ulcorner\cdot\urcorner$: λ-terms $\longrightarrow$ closed symbolic terms

$\mathcal{V}(\cdot)$: weak symbolic reduction

$$\mathcal{N}(t) = \mathcal{R}(\mathcal{V}(\ulcorner t \urcorner))$$

$$\mathcal{R}(\lambda x.t) = \lambda y.\, \mathcal{R}(\mathcal{V}((\lambda x.t)\ [\tilde{y}])) \text{ where } y \text{ is fresh}$$

$$\mathcal{R}[\tilde{x}\, v_1 \dots v_n] = x\, \mathcal{R}(v_1) \dots \mathcal{R}(v_n)$$

The interface must provide ways to:

- Implement the β_v and β_s rules

Interpretation and reification

$$[\![\cdot]\!] = \mathcal{V}(\ulcorner\cdot\urcorner)$$

$\ulcorner\cdot\urcorner$: λ-terms $\longrightarrow$ closed symbolic terms

$\mathcal{V}(\cdot)$: weak symbolic reduction

$$\mathcal{N}(t) = \mathcal{R}(\mathcal{V}(\ulcorner t \urcorner))$$

$$\mathcal{R}(\lambda x.t) = \lambda y.\, \mathcal{R}(\mathcal{V}((\lambda x.t)\ [\tilde{y}])) \text{ where } y \text{ is fresh}$$

$$\mathcal{R}[\tilde{x}\, v_1 \ldots v_n] = x\, \mathcal{R}(v_1) \ldots \mathcal{R}(v_n)$$

The interface must provide ways to:

- Implement the β_v and β_s rules
- Distinguish kinds of values

Interpretation and reification

$$[\![\cdot]\!] = \mathcal{V}(\ulcorner\cdot\urcorner)$$

$\ulcorner\cdot\urcorner$: λ-terms $\longrightarrow$ closed symbolic terms

$\mathcal{V}(\cdot)$: weak symbolic reduction

$$\mathcal{N}(t) = \mathcal{R}(\mathcal{V}(\ulcorner t \urcorner))$$

$$\mathcal{R}(\lambda x.t) = \lambda y.\, \mathcal{R}(\mathcal{V}((\lambda x.t)\ [\tilde{y}])) \text{ where } y \text{ is fresh}$$

$$\mathcal{R}[\tilde{x}\, v_1 \dots v_n] = x\, \mathcal{R}(v_1) \dots \mathcal{R}(v_n)$$

The interface must provide ways to:

- Implement the β_v and β_s rules
- Distinguish kinds of values
- Apply abstractions

Interpretation and reification

$$[\![\cdot]\!] = \mathcal{V}(\ulcorner\cdot\urcorner)$$

$\ulcorner\cdot\urcorner$: λ-terms $\longrightarrow$ closed symbolic terms

$\mathcal{V}(\cdot)$: weak symbolic reduction

$$\mathcal{N}(t) = \mathcal{R}(\mathcal{V}(\ulcorner t \urcorner))$$

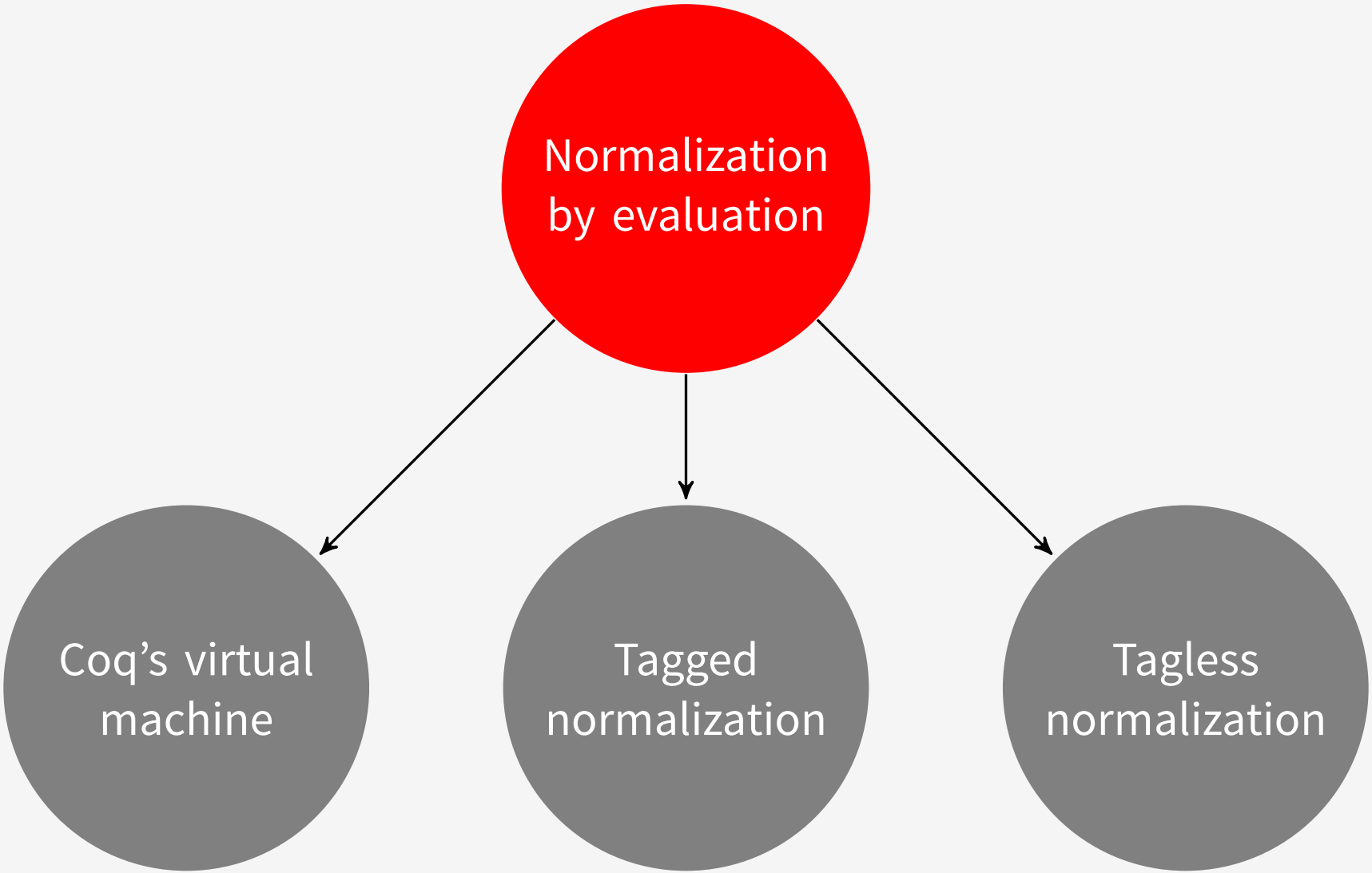
$$\mathcal{R}(\lambda x.t) = \lambda y.\, \mathcal{R}(\mathcal{V}((\lambda x.t)\ [\tilde{y}])) \text{ where } y \text{ is fresh}$$

$$\mathcal{R}[\tilde{x}\, v_1 \dots v_n] = x\, \mathcal{R}(v_1) \dots \mathcal{R}(v_n)$$

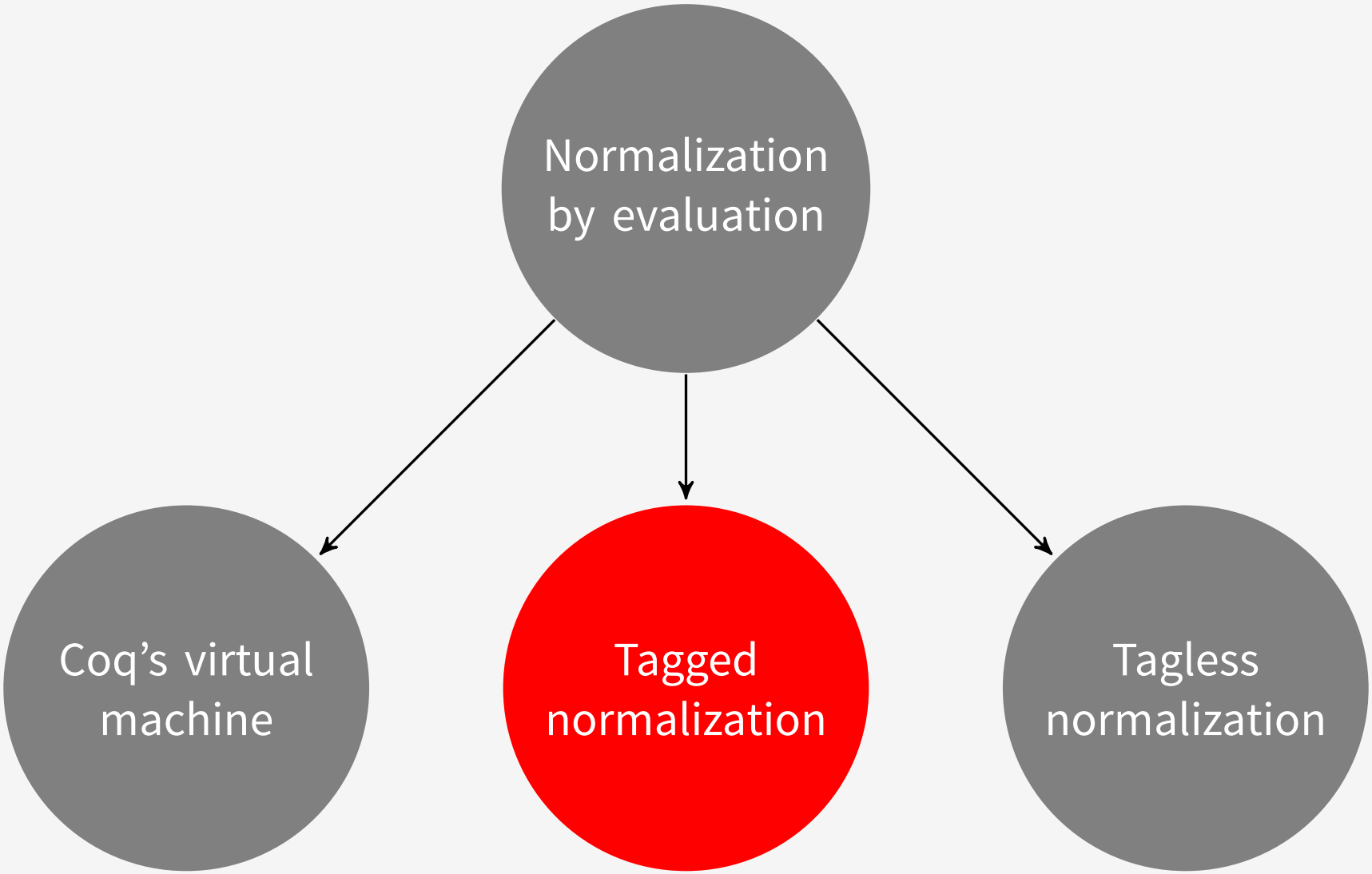
The interface must provide ways to:

- Implement the β_v and β_s rules
- Distinguish kinds of values
- Apply abstractions
- Retrieve names of variables from accumulators

Outline


Normalization by evaluation
Coq's virtual machine
Tagged normalization
Tagless normalization

Outline


Normalization by evaluation
Coq's virtual machine
Tagged normalization
Tagless normalization

Tagged normalization

```
type atom =
  | Var of var
type term =
  | Lam of term -> term
  | Accu of atom
```

Tagged normalization

```
type atom =
  | Var of var
type term =
  | Lam of term -> term
  | Accu of atom

let app t v = match t with
  | Lam f -> f v
  | Accu(a, args) -> Accu(a,v::args)
```

Tagged normalization

```
type atom =
  | Var of var
type term =
  | Lam of term -> term
  | Accu of atom

let app t v = match t with
  | Lam f -> f v
  | Accu(a, args) -> Accu(a,v::args)
```

$$\ulcorner x \urcorner^{B} = \begin{cases} x & \text{if } x \in B \\ \mathsf{Accu}(\mathsf{Var}\, x) & \text{otherwise} \end{cases}$$

$$\ulcorner \lambda x.t \urcorner^{B} = \mathsf{Lam}\,(\mathsf{fun}\, x \to \ulcorner t \urcorner^{B \cup \{x\}})$$

$$\ulcorner t_1\ t_2 \urcorner^{B} = \mathsf{app}\ \ulcorner t_1 \urcorner^{B}\ \ulcorner t_2 \urcorner^{B}$$

Unsatisfactory solution

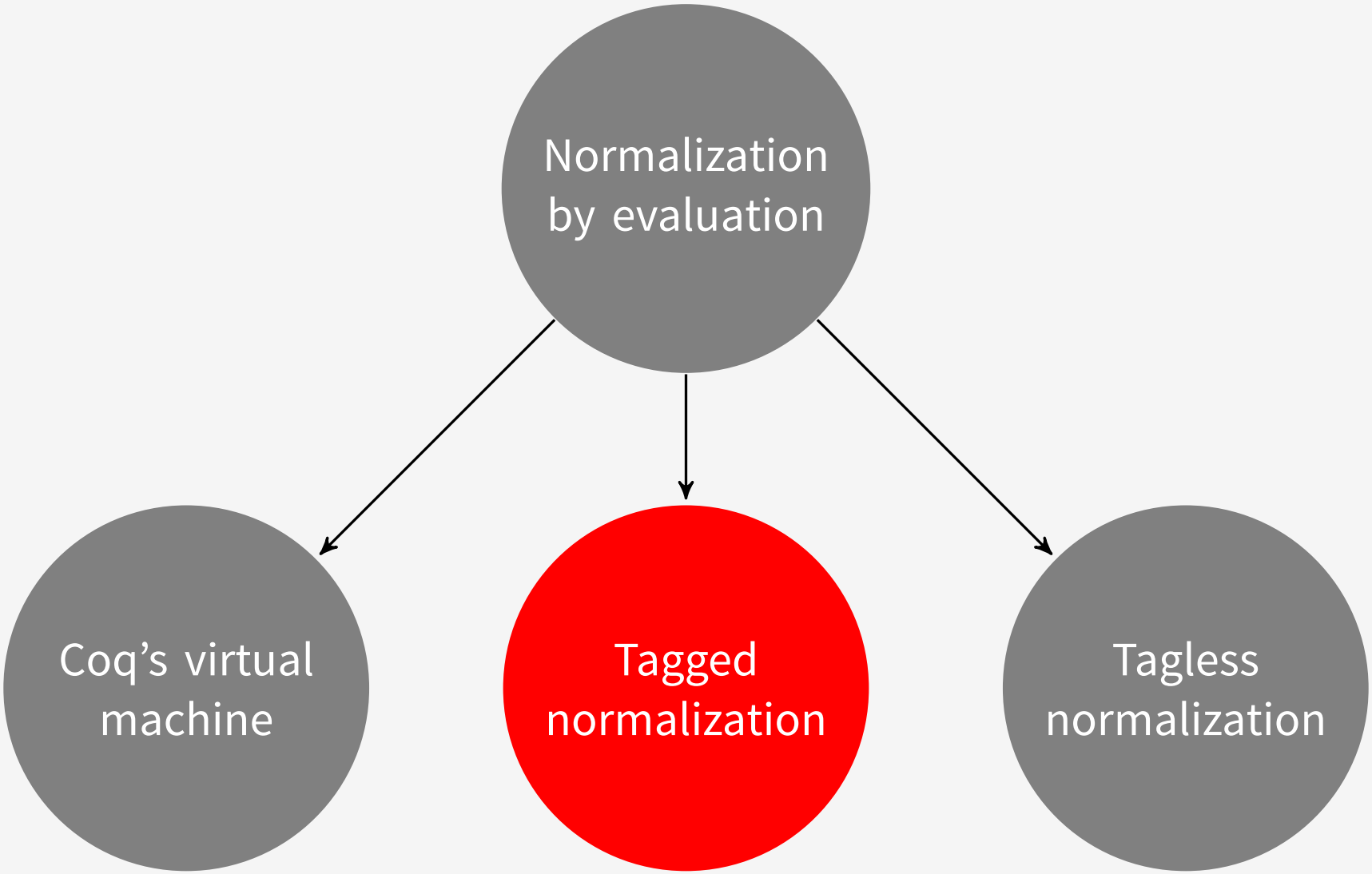
Pros:

- No need to maintain an ad-hoc compiler and interpreter
- Easily portable

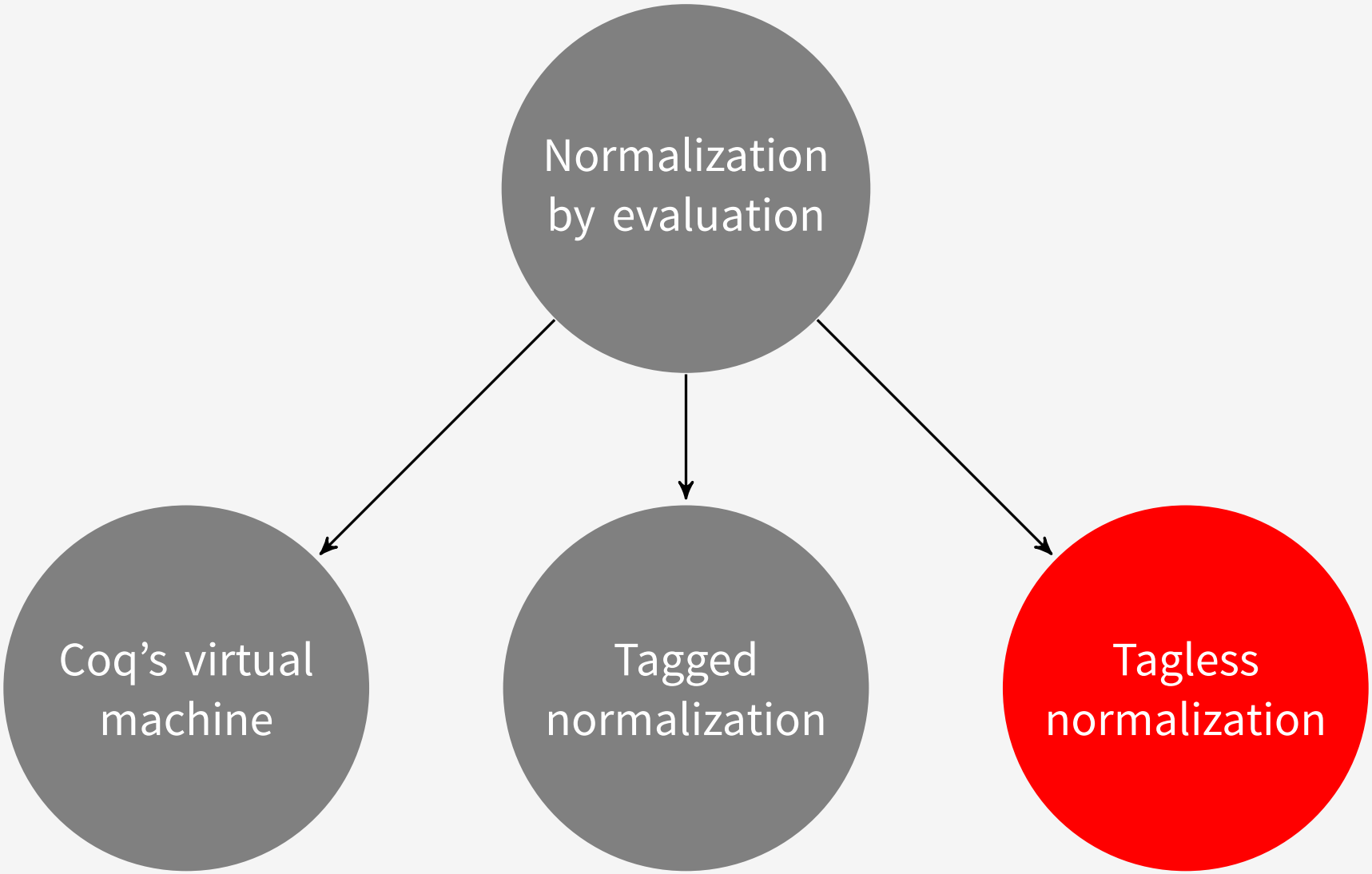
Cons:

- Breaking compiler optimizations has a negative impact on performance
- Efficient interpretation of pattern matching is unclear

Outline


Normalization by evaluation
Coq's virtual machine
Tagged normalization
Tagless normalization

Outline


Normalization by evaluation
Coq's virtual machine
Tagged normalization
Tagless normalization

Getting rid of tags

Closures memory layout in OCaml is:

We would like to use the same layout for accumulators, but how to:

Getting rid of tags

Closures memory layout in OCaml is:

We would like to use the same layout for accumulators, but how to:

- Implement the β_v and β_s rules ?
- Distinguish kinds of values ?
- Apply abstractions ?
- Retrieve information from accumulators ?

Getting rid of tags

Closures memory layout in OCaml is:

We would like to use the same layout for accumulators, but how to:

- Implement the β_v and β_s rules ?

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

- Distinguish kinds of values ?
- Apply abstractions ?
- Retrieve information from accumulators ?

Getting rid of tags

Closures memory layout in OCaml is:

We would like to use the same layout for accumulators, but how to:

- Implement the β_v and β_s rules ?

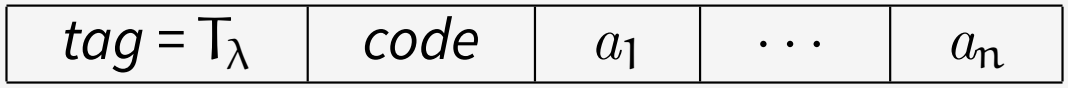
```
type term = term -> term
let rec accu atom args = fun v -> accu atom (v::args)
let mkAccu atom = accu atom []
```

- Distinguish kinds of values ?
  Assigning tag `0` to accumulators thanks to `Obj.set_tag`
- Apply abstractions ?

- Retrieve information from accumulators ?

Getting rid of tags

Closures memory layout in OCaml is:



We would like to use the same layout for accumulators, but how to:

- Implement the β_v and β_s rules ?
  ```
  type term = term -> term
  let rec accu atom args = fun v -> accu atom (v::args)
  let mkAccu atom = accu atom []
  ```
- Distinguish kinds of values ?
  Assigning tag 0 to accumulators thanks to `Obj.set_tag`
- Apply abstractions ?
  Using the native application of the language
- Retrieve information from accumulators ?

Getting rid of tags

Closures memory layout in OCaml is:

We would like to use the same layout for accumulators, but how to:

- Implement the β_v and β_s rules ?
  ```
  type term = term -> term
  let rec accu atom args = fun v -> accu atom (v::args)
  let mkAccu atom = accu atom []
  ```
- Distinguish kinds of values ?
  Assigning tag `0` to accumulators thanks to `Obj.set_tag`
- Apply abstractions ?
  Using the native application of the language
- Retrieve information from accumulators ?
  Using `Obj.get_field` (we know where the atom and arguments are)

Extension to CIC

Pattern matching

```
Definition is_zero n :=
  match n with
  | O => true
  | S _ => false
  end.
```

Pattern matching

```
let     const_is_zero  x_n_0                    =
  begin match            x_n_0  with


  | Construct_nat_0 ->
                Construct_bool_0
  | Construct_nat_1 _ ->
                Construct_bool_1
  end
```

Pattern matching

```
let rec const_is_zero  x_n_0                          =
  begin match             x_n_0  with
  | Accu_nat _ ->
    mk_sw_accu [...] (cast_accu x_n_0) (pred_is_zero_0)
    const_is_zero
  | Construct_nat_0 ->
                Construct_bool_0
  | Construct_nat_1 _ ->
                Construct_bool_1
  end
```

Pattern matching

```
let rec const_is_zero (x_n_0 : term) =
  begin match Obj.magic (x_n_0) with
  | Accu_nat _ ->
    mk_sw_accu [...] (cast_accu x_n_0) (pred_is_zero_0)
    const_is_zero
  | Construct_nat_0 ->
    (Obj.magic Construct_bool_0 : term)
  | Construct_nat_1 _ ->
    (Obj.magic Construct_bool_1 : term)
  end
```

Fixpoints

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

let norm_id (x_id_1 : term) (x_n_0 : term) =
  case_id x_id_1 x_n_0

let const_id =
  let rec x_id_1 (x_n_0 : term) =
    if is_accu x_n_0 then
      mk_fix_accu [...] norm_id x_n_0
    else
      norm_id x_id_1 x_n_0
  in
  x_id_1
```

Fixpoints (with optimizations)

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

let const_id =
  let rec x_id_1 (x_n_0 : term) =
    begin match Obj.magic (x_n_0) with
    | Accu_nat _ ->
      mk_fix_accu [...] norm_id x_n_0
    | Construct_nat_0 ->
      (Obj.magic (Construct_nat_0) : term)
    | Construct_nat_1 x_p_2  ->
      (Obj.magic (Construct_nat_1 (x_id_1 x_p_2)) : term)
    end
  in
  x_id_7
```

Other extensions

We have also implemented:

- Coinductive types and cofixpoints
- Machine integers (native-coq branch only)
- Persistent arrays (native-coq branch only)
- Various optimizations (e.g. completely constructed values)

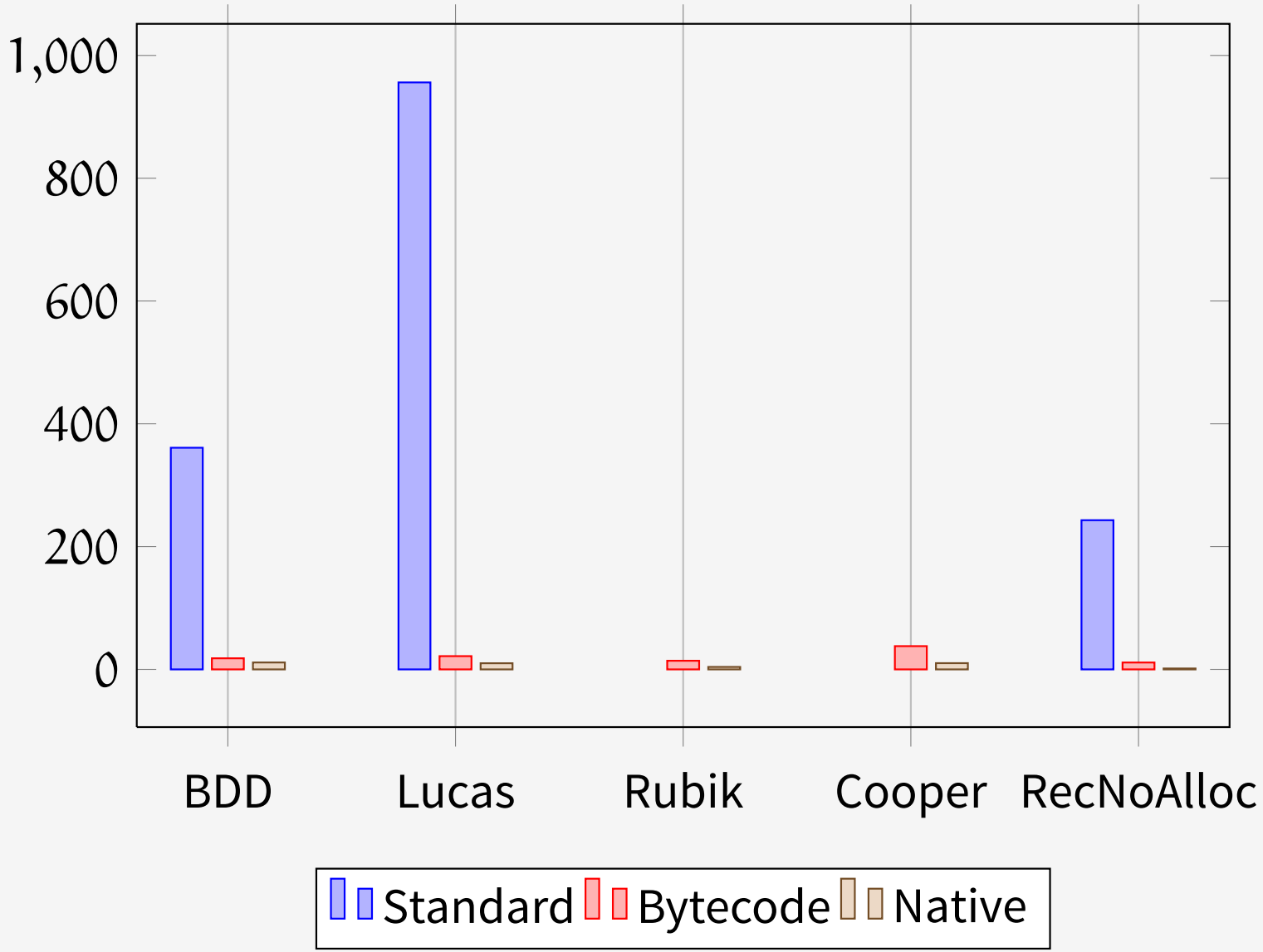
Other extensions

We have also implemented:

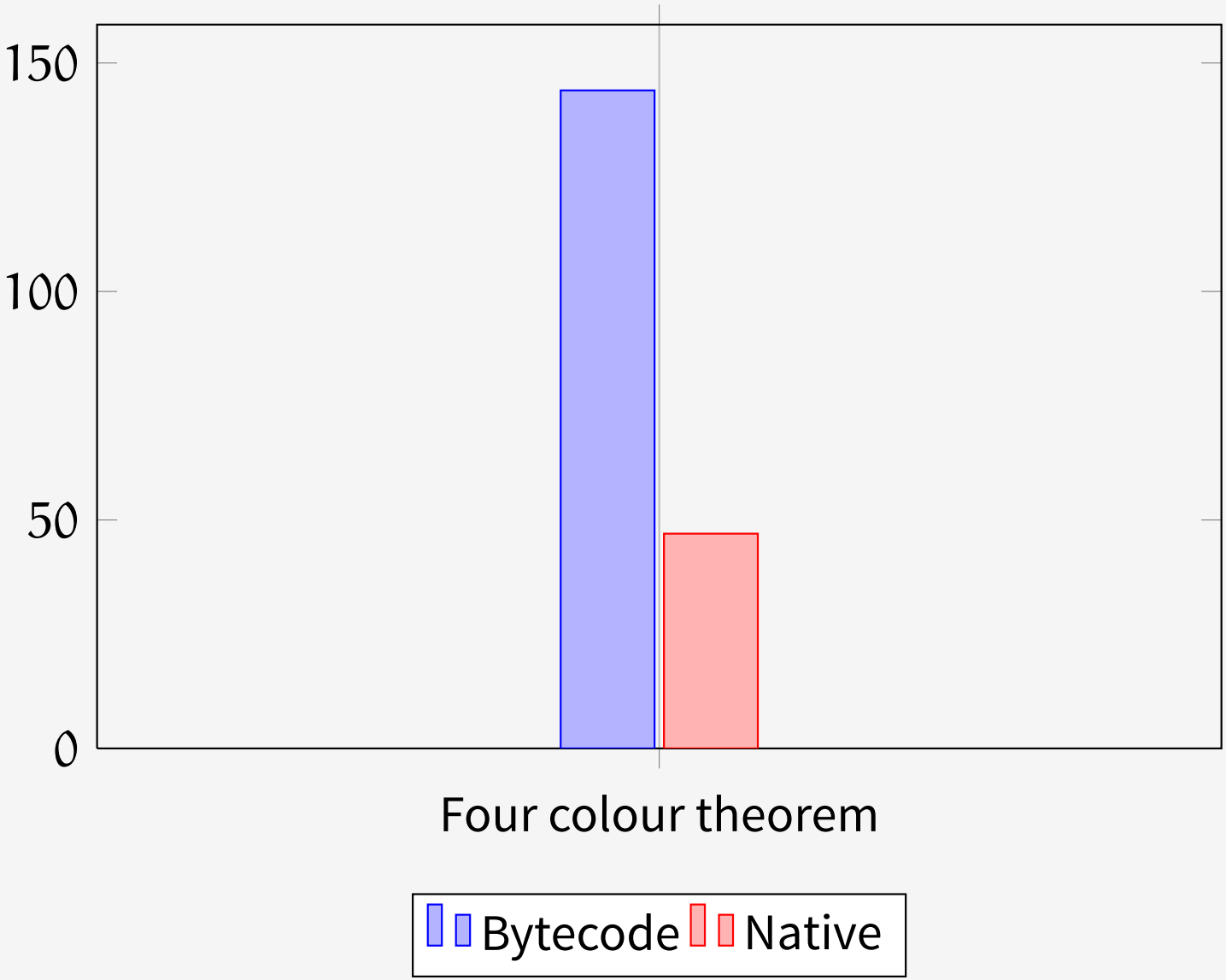
- Coinductive types and cofixpoints
- Machine integers (native-coq branch only)
- Persistent arrays (native-coq branch only)
- Various optimizations (e.g. completely constructed values)

No modification of the runtime system required!

Benchmarks (32 bits)



Benchmarks (32 bits)



Current state of the implementation

Available as a branch (native-coq) on github since 2011.

Porting it to Coq's trunk in two steps:

- Native compiler for Coq's usual theory
- Machine integers and arrays

Current state of the implementation

Available as a branch (native-coq) on github since 2011.

Porting it to Coq's trunk in two steps:

- Native compiler for Coq's usual theory
- Machine integers and arrays

On 2013-01-22 18:37:00, I committed the first part…

Current state of the implementation

Available as a branch (native-coq) on github since 2011.

Porting it to Coq's trunk in two steps:

- Native compiler for Coq's usual theory
- Machine integers and arrays

On 2013-01-22 18:37:00, I committed the first part…
…Two hours later coq.inria.fr was down

Current state of the implementation

Available as a branch (native-coq) on github since 2011.

Porting it to Coq's trunk in two steps:

- Native compiler for Coq's usual theory
- Machine integers and arrays

On 2013-01-22 18:37:00, I committed the first part…
…Two hours later coq.inria.fr was down
…But due to an apparently unrelated air conditioning problem!

Impact on user contributions

A few contribs have been broken because of the -rectypes flag (fixed).

Some others hit OCaml limitations in code size:

- Nijmegen/QArithSternBrocot (fixed)
- Orsay/FingerTree (fixed)
- Nijmegen/CoRN (fixed, but unification issue)
- Amsterdam/Coalgebras (to be fixed)

Apart from that, things went relatively smoothly.

Rough edges

A few issues still have to be addressed:

- `coq_makefile` should take into account generated files
- The checker should be patched to re-plug native casts to regular conversion.
- Filesystems are a bit polluted by generated files at the moment (move them to a subdirectory?)

Conclusion

- Bringing Coq even closer to a programming language

Conclusion

- Bringing Coq even closer to a programming language
- Performance not far from extracted code

Conclusion

- Bringing Coq even closer to a programming language
- Performance not far from extracted code
- Full target language available, easily extensible

Conclusion

- Bringing Coq even closer to a programming language
- Performance not far from extracted code
- Full target language available, easily extensible
- Another reason why OCaml is a nice language: low-level access to memory objects

Perspectives

Perspectives

- Machine integers and arrays should follow

Perspectives

- Machine integers and arrays should follow
- Replacing the VM requires smaller compilation cost
  - JIT techniques in OCaml?
  - Interfacing with OCaml bytecode compiler

Perspectives

- Machine integers and arrays should follow
- Replacing the VM requires smaller compilation cost
  - JIT techniques in OCaml?
  - Interfacing with OCaml bytecode compiler
- Formally verify the target compiler

Thank you!

Comments, suggestions, discussion welcome!

Boespflug, Mathieu, Maxime Dénès, and Benjamin Grégoire (2011). “Full Reduction at Full Throttle”. In: *Certified Programs and Proofs*. Vol. 7086. LNCS, pp. 362–377.

Grégoire, Benjamin and Xavier Leroy (2002). “A compiled implementation of strong reduction”. In: *International Conference on Functional Programming 2002*. ACM Press, pp. 235–246.