A Compiled Implementation of Strong Reduction

Benjamin Grégoire
INRIA Rocquencourt
Domaine de Voluceau, B.P. 105
78153 Le Chesnay, France
Benjamin.Gregoire@inria.fr

Xavier Leroy
INRIA Rocquencourt
Domaine de Voluceau, B.P. 105
78153 Le Chesnay, France
Xavier.Leroy@inria.fr

Abstract

Motivated by applications to proof assistants based on dependent types, we develop and prove correct a strong reducer and β-equivalence checker for the λ-calculus with products, sums, and guarded fixpoints. Our approach is based on compilation to the bytecode of an abstract machine performing weak reductions on non-closed terms, derived with minimal modifications from the ZAM machine used in the Objective Caml bytecode interpreter, and complemented by a recursive "read back" procedure. An implementation in the Coq proof assistant demonstrates important speedups compared with the original interpreter-based implementation of strong reduction in Coq.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Language Classifications—*applicative (functional) languages*; D.3.4 [**Programming Languages**]: Processors—*compilers, interpreters*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*operational semantics, partial evaluation*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*lambda calculus and related systems, proof theory*; I.1.3 [**Symbolic and Algebraic Manipulation**]: Languages and Systems—*evaluation strategies*; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving

General Terms

Languages, Theory, Experimentation, Verification

Keywords

Strong reduction, beta-equivalence, normalization by evaluation, abstract machine, virtual machine, Calculus of Constructions, Coq

1 Introduction

It is folklore that β-reduction in the λ-calculus is the computation model underlying functional programming languages. Actually, functional languages implement only *weak* β-reduction, whereby reductions are not performed on function bodies until functions are applied to actual arguments. Efficient compiled implementations of weak reduction are widely known and deployed, based either on environments and closures or on graph reduction, and implemented either as abstract machines or by direct machine code generation.

In contrast, this paper focuses on *strong* reduction, where β-reductions are also performed on function bodies. The practical need for strong reduction appears in two areas. The first area is partial evaluation, also known as program specialization: specializing a function applied to some known arguments (the other arguments remaining unknown) amounts to strongly normalizing a partial application of the function. The other area where strong reduction is required, which prompted the work presented here, is type checking / proof checking in type systems / logics based on dependent types, such as LF or the Calculus of Constructions [14, 7, 21], which are at the basis of proof assistants such as Alf, Coq, Elf, Lego and NuPRL. In these systems, dependent types may contain arbitrary terms, and types are compared up to β-equivalence of the terms they contain, as captured by the following conversion rule:

$$\frac{E \vdash a : \tau \qquad \tau \overset{\beta}{\approx} \tau'}{E \vdash a : \tau'} \text{ (conv)}$$

Thus, type checking in these systems, or equivalently proof checking, involve performing strong β-reductions inside the types to be compared, until the reducts are syntactically equal.

Most, if not all, proof assistants of the Coq/HOL family implement these strong reductions in a purely interpretative way, by walking over a tree-based representation of terms. Despite various implementation tricks involving explicit substitutions, this interpretative approach can become a performance bottleneck when developing and checking proofs with large computational content, such as proofs based on *reflection*. Proof by reflection is characterized by the use of efficient decision procedures, proved correct once and for all, to replace long proof derivations. A typical example is the use of computations on binary decision diagrams to prove results on boolean formulas [23]. Another example is Appel and Haken's famous proof of the 4-color theorem [2], which involves checking colorability of a finite, but large, number of well-chosen planar graphs: rather than developing a proof of 4-colorability for every such graph, a proved decision procedure is invoked on all of them.

The present paper reports on the design and implementation of a strong evaluator and equivalence tester for the Coq proof assistant that eliminates most of the interpretive overhead via compilation to the byte-code of a virtual machine. In sections 2 and 3, we first show how strong reduction can be implemented by a recursive combination of *weak symbolic reduction* (weak reduction of terms containing free variables) and *readback* (reconstruction of a term in normal form from the head normal form returned by the weak symbolic evaluator). While developed independently, this combination is similar to online type-directed partial evaluation [12, 22]; however, we give a new presentation of this approach that paves the way to an implementation of weak symbolic reduction that avoids explicit run-time tests "is this term symbolic or not?".

In section 4, we develop one such efficient implementation of weak symbolic reduction, as an abstract machine and its associated compilation scheme. The abstract machine is a minor extension of the ZAM abstract machine [16] that is at the heart of the Objective Caml bytecode interpreter [17], and reuses all the work that has been expended in making the latter efficient.

Both the abstract machine and its compilation scheme have been proved correct with respect to the weak reduction semantics, and the proof was mechanically checked by Coq; section 5 reports on this proof development. Performance figures obtained on a modified version of the Coq proof assistant are reported in section 6. We end this paper by a discussion of related work in section 7, and some concluding remarks in section 8.

2 Strong Reduction for the Pure λ-calculus

We first present our strong reduction technique on the simplest functional language of all: the pure λ-calculus.

2.1 The Pure λ-calculus

The syntax of the calculus is given by

Terms: $a ::= x \mid \lambda x.a \mid a_1\ a_2$

Terms are identified up to renaming of λ-bound variables (α-conversion). The strong reduction relation $\Rightarrow$ consists of the familiar β-reduction rule, plus a context rule allowing reduction in arbitrary subterms, including below a lambda:

$$\begin{array}{rcll} (\lambda x.a)\ a' & \Rightarrow & a\{x \leftarrow a'\} & (\beta) \\ \Gamma(a) & \Rightarrow & \Gamma(a') \quad \text{if } a \Rightarrow a' & (\text{context}) \end{array}$$

with $\Gamma ::= \lambda x.[\,] \mid [\,]\ a \mid a\ [\,]$.

We write $\stackrel{*}{\Rightarrow}$ for the reflexive and transitive closure of $\Rightarrow$. In the following, we assume that all λ-terms a considered are strongly normalizing: there are no infinite reduction sequences starting from a. In our setting, this property is guaranteed by the type system of the Calculus of Constructions.

We are interested in two computational problems. The first is to compute the normal form $\mathcal{N}(a)$ of a closed, strongly normalizing term a. This is the unique term such that $a \stackrel{*}{\Rightarrow} \mathcal{N}(a)$ and $\mathcal{N}(a)$ does not reduce. The second problem is to decide whether two closed, strongly normalizing terms a_1 and a_2 are β-equivalent, written $a_1 \approx a_2$. This equivalence is defined by $a_1 \approx a_2$ if and only if there exists a term a such that $a_1 \stackrel{*}{\Rightarrow} a$ and $a_2 \stackrel{*}{\Rightarrow} a$.

2.2 Strong Reduction by Iterated Symbolic Weak Reduction and Readback

To compute the normal form of a closed term a, our approach is first to compute the head normal form (also called value) of a using an off-the-shelf weak evaluator. Such an evaluator performs β-reductions anywhere in the term except under a λ-abstraction. Thus, the value of a is of the form $\lambda x.a'$, where a' is, in general, not in normal form. To obtain the normal form of a, all that remains to do is recursively compute the normal form $\mathcal{N}(a')$ of a'; then, we will have $\mathcal{N}(a) = \mathcal{N}(\lambda x.a') = \lambda x.\ \mathcal{N}(a')$.

A slight difficulty arises here: a' is not necessarily closed, since the formal parameter x may occur free in a'. Thus, we cannot just run our off-the-shelf weak evaluator on a', since such evaluators work only on closed terms. To circumvent this problem, we enrich the term algebra with the ability to represent and manipulate free variables during weak reduction. More formally, we inject the pure λ-terms a into the following algebra of extended terms b:

Extended terms: $b ::= x \mid \lambda x.b \mid b_1\ b_2 \mid [\tilde{x}\ v_1\ \ldots\ v_n]$

Values: $v ::= \lambda x.b \mid [\tilde{x}\ v_1\ \ldots\ v_n]$

Here, $\tilde{x}$ is a constant, uniquely associated with the identifier x, but not subject to alpha-conversion. The extended term $[\tilde{x}]$ is a run-time representation of the free variable x. Since free variables can be applied during weak reduction, we also need run-time representations $[\tilde{x}\ v_1\ \ldots\ v_n]$ for applications of a free variable x to arguments $v_1, \ldots, v_n$.

We now formalize the symbolic weak reduction of extended, closed terms b. ("Symbolic" here means that this weak reduction knows how to handle free variables as represented by $[\ldots]$ terms.) To be more specific, and to match exactly the implementation of symbolic weak reduction by an abstract machine presented in section 4, we impose a call-by-value, right-to-left evaluation strategy. The symbolic weak reduction relation $\to$ is defined by the following three rules:

$$\begin{array}{rcll} (\lambda x.b)\ v & \to & b\{x \leftarrow v\} & (\beta_v) \\ [x\ v_1\ \ldots\ v_n]\ v & \to & [x\ v_1\ \ldots\ v_n\ v] & (\beta_s) \\ \Gamma_v(a) & \to & \Gamma_v(a') \quad \text{if } a \to a' & (\text{context}_v) \end{array}$$

with $\Gamma_v ::= [\,]\ v \mid b\ [\,]$.

Rule (β_v) is the familiar call-by-value function application rule. It handles the case where the function part of an application evaluates to a known function. Rule (β_s) ("symbolic" beta reduction) handles the case where the function part evaluates to the representation of a free variable $[\tilde{x}]$ or of an application of a free variable $[x\ v_1\ \ldots\ v_n]$. Finally, our choice of contexts Γ_v precludes reductions under function abstractions, and forces the argument part of an application to be evaluated to a value before starting evaluation of the function part (right-to-left strategy).

As usual, we write $\stackrel{*}{\to}$ for the transitive and reflexive closure of $\to$. We define the value $\mathcal{V}(b)$ of an extended closed term b as the normal form of b for the relation $\stackrel{*}{\to}$. Notice that such a normal form is necessarily a value: reduction cannot get stuck.

Now that we know how to reduce weakly terms containing free variables, we can express precisely the strong normalization procedure outlined earlier: first, normalize weakly; second, *read back* the resulting value as a normalized term, recursing over the bodies of functions if needed.

$$\mathcal{N}(b) = \mathcal{R}(\mathcal{V}(b)) \quad (1)$$
$$\mathcal{R}(\lambda x.b) = \lambda y.\ \mathcal{N}((\lambda x.b)\ [\tilde{y}]) \quad (y \text{ fresh}) \quad (2)$$
$$\mathcal{R}([\tilde{x}\ v_1\ \ldots\ v_n]) = x\ \mathcal{R}(v_1)\ \ldots\ \mathcal{R}(v_n) \quad (3)$$

The readback function $\mathcal{R}$ transforms values v into normalized source terms a. For function values $\lambda x.b$ (equation 2), reading back consists in applying the value to the run-time representation of a fresh free variable $[\tilde{y}]$. ("Fresh", here, means that y does not occur at all in b; the freshness condition can be made fully precise by using negative de Bruijn indices.) We then compute the value of this term, or equivalently of the term $b\{x \leftarrow [\tilde{y}]\}$, and read it back as a normalized term a. The normal form of $\lambda x.b$ is then $\lambda y.a$.

Reading back the value $[\tilde{x}\ v_1\ \ldots\ v_n]$ (equation 3) simply consists in extracting the variable x from $\tilde{x}$, reading back the values to which it is applied, and reconstructing the application $x\ \mathcal{R}(v_1)\ \ldots\ \mathcal{R}(v_n)$.

Example: Consider the following source term

$$a = (\lambda x.x)(\lambda y.\ (\lambda z.z)\ y\ (\lambda t.t))$$

Weak symbolic evaluation reduces it to $v = \lambda y.\ (\lambda z.z)\ y\ (\lambda t.t)$. This is a functional value, hence the readback procedure $\mathcal{R}$ restarts weak symbolic evaluation on $b = (\lambda y.\ (\lambda z.z)\ y\ (\lambda t.t))\ [\tilde{u}]$. The value returned by the second round of weak symbolic evaluation is $v' = [\tilde{u}\ (\lambda t.t)]$. Readback of $\lambda t.t$ triggers the evaluation of $(\lambda t.t)\ [\tilde{w}]$, which returns $[\tilde{w}]$. Thus, $\mathcal{R}(\lambda t.t) = \lambda w.w$. Then, $\mathcal{R}(v') = u\ (\lambda w.w)$. Finally,

$$\mathcal{N}(a) = \mathcal{R}(v) = \lambda u.\ u\ (\lambda w.w)$$

Remark: The readback of functional values (equation 2) could alternatively be written as

$$\mathcal{R}(\lambda x.b) = \lambda y.\ \mathcal{N}(b\{x \leftarrow [\tilde{y}]\})$$

The two forms are equivalent, since the first step of weak reduction of $(\lambda x.b)\ [\tilde{y}]$ transforms this term into $b\{x \leftarrow [\tilde{y}]\}$. However, by writing (2) as a function application rather than a direct substitution, we make it clear that the readback procedure does not need to "look inside" function values: it remains applicable even if function values are represented as opaque closures of compiled code, like our abstract machine-based weak evaluator of section 4 does. The only requirement that the readback function $\mathcal{R}$ puts on the representation of values used by the weak evaluator $\mathcal{V}$ is that the representations of functions $\lambda x.b$ and of "accumulators" $[\tilde{x}\ v_1\ \ldots\ v_n]$ can be distinguished at run-time, and that the components $x, v_1, \ldots, v_n$ of the latter can be recovered from the latter.

2.3 Correctness of the Normalization Procedure

We now proceed to show the correctness of the function $\mathcal{N}$ defined above. We start with partial correctness: assuming the computation of $\mathcal{N}(a)$ terminates, we show that it is indeed the normal form of the pure term a.

REMARK 1. *For all extended terms b, $\mathcal{N}(b)$ and $\mathcal{R}(b)$, if defined, are terms that belong to the following grammar:*

$$n ::= \lambda x.n \mid x\ n_1\ \ldots\ n_k$$

Hence, they are pure λ-terms in normal form.

Now, define the following translation $\overline{b}$ from an extended term b to a pure λ-term, obtained by erasing the distinction between run-time representations of free variables and the free variables themselves:

$$\overline{\tilde{x}} = x$$
$$\overline{\lambda x.b} = \lambda x.\overline{b}$$
$$\overline{b_1\ b_2} = \overline{b_1}\ \overline{b_2}$$
$$\overline{[\tilde{x}\ v_1\ \ldots\ v_n]} = x\ \overline{v_1}\ \ldots\ \overline{v_n}$$

LEMMA 2. *If $b \to b'$, then $\overline{b} \stackrel{*}{\Rightarrow} \overline{b'}$.*

PROOF. By cases on the reduction rule used. If $b \to b'$ by rule (β_v), then $\overline{b} \Rightarrow \overline{b'}$ by rule (β). If $b \to b'$ by rule (β_s), then $\overline{b} = \overline{b'}$. Finally, for context reductions, notice that translations of call-by-value contexts $\overline{\Gamma_v}$ are a subset of Γ contexts, and conclude by induction on the number of context rules used. □

LEMMA 3. *If $\mathcal{N}(b)$ is defined, then $\overline{b} \stackrel{*}{\Rightarrow} \mathcal{N}(b)$. If $\mathcal{R}(v)$ is defined, then $\overline{v} \stackrel{*}{\Rightarrow} \mathcal{R}(v)$.*

PROOF. Both statements are proved simultaneously by "course of value" induction (induction on the number of recursive calls to $\mathcal{N}$ and $\mathcal{R}$).

Since $\mathcal{N}(b) = \mathcal{R}(\mathcal{V}(b))$, we have $\overline{b} \stackrel{*}{\Rightarrow} \overline{\mathcal{V}(b)}$ by lemma 2, and $\overline{\mathcal{V}(b)} \stackrel{*}{\Rightarrow} \mathcal{R}(\mathcal{V}(b))$ by induction hypothesis. It follows that $\overline{b} \stackrel{*}{\Rightarrow} \mathcal{N}(b)$.

Consider now the case $\mathcal{R}(\lambda x.b)$. Write $b' = (\lambda x.b)\ [\tilde{y}]$. By induction hypothesis, $\overline{b'} \stackrel{*}{\Rightarrow} \mathcal{N}(b')$. Since $\mathcal{N}(b')$ is in normal form by remark 1, and $\overline{b'} \Rightarrow \overline{b}\{x \leftarrow y\}$, confluence of the λ-calculus ensures that $\overline{b}\{x \leftarrow y\} \stackrel{*}{\Rightarrow} \mathcal{N}(b')$. Hence, $\lambda y.\ \overline{b}\{x \leftarrow y\} \stackrel{*}{\Rightarrow} \lambda y.\mathcal{N}(b')$. Since $\lambda y.\ \overline{b}\{x \leftarrow y\} = \overline{\lambda x.b}$ up to alpha-conversion, the expected result $\overline{\lambda x.b} \stackrel{*}{\Rightarrow} \mathcal{N}(b')$ follows.

The last case to consider is $\mathcal{R}([\tilde{x}\ v_1\ \ldots\ v_n])$. Applying the induction hypothesis to the v_i, we obtain that $\overline{v_i} \stackrel{*}{\Rightarrow} \mathcal{R}(v_i)$ for $i = 1, \ldots, n$. Thus, $\overline{[\tilde{x}\ v_1\ \ldots\ v_n]} = x\ \overline{v_1}\ \ldots\ \overline{v_n} \stackrel{*}{\Rightarrow} x\ \mathcal{R}(v_1)\ \ldots\ \mathcal{R}(v_n) = \mathcal{R}([\tilde{x}\ v_1\ \ldots\ v_n])$. □

LEMMA 4. *For all pure λ-terms a, if $\mathcal{N}(a)$ is defined, then it is the normal form of a.*

PROOF. Since a is a pure λ-term (containing no $[\ldots]$), we have $\overline{a} = a$. By lemma 3, $a \stackrel{*}{\Rightarrow} \mathcal{N}(a)$. Moreover, $\mathcal{N}(a)$ is in normal form by remark 1. □

We now turn to proving that $\mathcal{N}$ always terminates when given a strongly-normalizing argument.

LEMMA 5. *If $\overline{b}$ is strongly normalizing, then $\mathcal{V}(b)$ is defined.*

PROOF. By way of contradiction, consider an infinite weak reduction sequence starting at b and performing infinitely many β_v and β_s reductions (possibly under context). There can only be a finite number of β_s reductions in a row, without an intervening β_v step, since the number of application nodes in the term decreases by one during a β_s reduction. Thus, the infinite weak reduction sequence contains infinitely many β_v reductions. As shown in lemma 2, each such β_v reduction corresponds to a β reduction on translated terms. Thus, we can construct an infinite sequence of β reductions starting at $\overline{b}$. This contradicts the hypothesis that $\overline{b}$ is strongly normalizing. □

LEMMA 6. *If $\overline{v}$ is strongly normalizing, then $\mathcal{R}(v)$ is defined.*

PROOF. Consider the ordering $\succ$ on pure terms defined as the transitive closure of the following two cases:

- $a \succ a'$ if $a \Rightarrow a'$;
- $a \succ a'$ if a' is a strict subterm of a up to a renaming of free variables.

It is easy to see that $\succ$ is well-founded on strongly normalizing terms, in the sense that there exists no infinitely decreasing chains $a \succ a_1 \succ \ldots$ starting with a strongly normalizing term a. (From such a chain, we could trivially construct an infinite reduction sequence starting with a.)

The result then follows by well-founded induction using the $\succ$ ordering. For the function case, we have $\mathcal{R}(\lambda x.b) = \lambda y.\mathcal{R}(\mathcal{V}(b'))$ with $b' = b\{x \leftarrow [\tilde{y}]\})$. Since $\overline{b'} = \overline{b}\{x \leftarrow y\}$, $\overline{b'}$ is a strict subterm of $\overline{\lambda x.b}$ up to renaming of x by y. Hence, $\overline{b'}$ is strongly normalizing, and lemma 5 establishes the termination of $\mathcal{V}(b')$. Moreover, $\overline{\mathcal{V}(b')}$ is a reduct of $\overline{b'}$ by lemma 2. Hence, $\overline{\lambda x.b} \succ \overline{b'} \succeq \overline{\mathcal{V}(b')}$. We can therefore apply the induction hypothesis to $\mathcal{R}(\mathcal{V}(b'))$, and obtain the expected result.

For the case $\mathcal{R}([\tilde{x}\ v_1\ \ldots\ v_n]) = x\ \mathcal{R}(v_1)\ \ldots\ \mathcal{R}(v_n)$, it is obvious that $\overline{v_1}, \ldots, \overline{v_n}$ are strict subterms of $\overline{[\tilde{x}\ v_1\ \ldots\ v_n]} = x\ \overline{v_1}\ \ldots\ \overline{v_n}$, and the result follows by induction hypothesis. □

As a corollary of lemmas 4, 5 and 6, we obtain the (total) correctness of the strong normalization procedure:

THEOREM 1. *If a is a closed, strongly normalizing pure λ-term, then $\mathcal{N}(a)$ is defined and is the normal form of a.*

2.4 Deciding β-equivalence

The previous results give a naive procedure to decide the β-equivalence of two strongly normalizing terms a_1 and a_2: compute $\mathcal{N}(a_1)$ and $\mathcal{N}(a_2)$ and compare the normal forms for syntactic equality. However, it is often not necessary to go all the way to normal forms to find a common reduct. The following more efficient procedure $\mathcal{E}(a_1, a_2)$ reduces weakly the terms a_1 and a_2, compares their values, and recurses only if syntactically different functions are obtained.

- $\mathcal{E}(b_1, b_2)$ if and only if $\mathcal{E}_v(\mathcal{V}(b_1), \mathcal{V}(b_2))$.
- $\mathcal{E}_v(v_1, v_2)$ if $v_1 \equiv v_2$.
- $\mathcal{E}_v(\lambda x_1.b_1, \lambda x_2.b_2)$ if $\mathcal{E}_v((\lambda x_1.b_1)\ [\tilde{y}], (\lambda x_2.b_2)\ [\tilde{y}])$ where y is a fresh variable.
- $\mathcal{E}_v([\tilde{x}\ v_1\ \ldots\ v_n], [\tilde{x}\ w_1\ \ldots\ w_n])$ if $\mathcal{E}_v(v_i, w_i)$ for all $i = 1, \ldots, n$.
- $\mathcal{E}_v(v_1, v_2)$ is false otherwise.

In this definition, the $\equiv$ relation stands for any relation that is finer than β-convertibility and can be computed cheaply, so that the "early return" test (second case above) can be profitably applied at each step. Syntactic equality up to α-conversion is an obvious candidate for $\equiv$ in an interpreted setting. In a compiled implementation such as that of section 4, syntactic equality might not be easily decidable, in which case $\equiv$ can be machine-representation "equality" (compare code pointers embedded in function closures by address, and other values by structure).

The correctness of the $\mathcal{E}$ predicate defined above can easily be proved along the lines of section 2.3, and we omit the proofs.

3 Extension to the Calculus of Constructions

We now progressively extend the approach presented in section 2 with the additional features of the (type-erased) term language of the Calculus of Constructions: inductive types (a generalization of products and sums) and fixpoints.

3.1 Booleans and Conditional

Before considering inductive types in their full generality, it is helpful to present a familiar special case: booleans and the `if...then...else...` construct. The syntax of the source calculus becomes:

$$\text{Terms:}\quad a ::= x \mid \lambda x.a \mid a_1\ a_2 \mid \texttt{true} \mid \texttt{false} \mid \texttt{if}\ a_1\ \texttt{then}\ a_2\ \texttt{else}\ a_3$$

The associated reduction rules are:

$$\begin{array}{rcl}
(\lambda x.a)\ a' & \Rightarrow & a\{x \leftarrow a'\} \\
\texttt{if true then}\ a\ \texttt{else}\ a' & \Rightarrow & a \\
\texttt{if false then}\ a\ \texttt{else}\ a' & \Rightarrow & a' \\
\Gamma(a) & \Rightarrow & \Gamma(a') \quad \text{if } a \Rightarrow a'
\end{array}$$

with $\Gamma ::= \lambda x.[\,] \mid [\,]\ \ a \mid a\ \ [\,] \mid \texttt{if}\ [\,]\ \texttt{then}\ a_2\ \texttt{else}\ a_3 \mid \texttt{if}\ a_1\ \texttt{then}\ [\,]\ \texttt{else}\ a_3 \mid \texttt{if}\ a_1\ \texttt{then}\ a_2\ \texttt{else}\ [\,]$.

We must also add booleans and the conditional construct to the algebra of extended terms. However, a new case arises: the run-time representation of a free variable $[\tilde{x}]$ can now appear in the `if` part of a conditional, in addition to in function position of an application. Just like rule (β_s) simply "remembers" the arguments to an application of a free variable, we need to "remember" the `if...then...else...` construct that could not be reduced because the condition was a free variable. To capture this phenomenon, we introduce a new syntactic category k of *accumulators*, or run-time representations of non-closed, non-reducible terms.

$$\begin{array}{ll}
\text{Extended terms:} & b ::= x \mid \lambda x.b \mid b_1\ b_2 \\
& \quad \mid \texttt{true} \mid \texttt{false} \\
& \quad \mid \texttt{if}\ b_1\ \texttt{then}\ b_2\ \texttt{else}\ b_3 \\
& \quad \mid [k] \\
\text{Accumulators:} & k ::= \tilde{x} \mid k\ v \mid \texttt{if}\ k\ \texttt{then}\ b\ \texttt{else}\ b' \\
\text{Values:} & v ::= \lambda x.b \mid \texttt{true} \mid \texttt{false} \mid [k]
\end{array}$$

Symbolic weak reduction is then defined by the following rules:

$$\begin{array}{rcl}
(\lambda x.b)\ v & \to & b\{x \leftarrow v\} \\
[k]\ v & \to & [k\ v] \\
\texttt{if true then}\ b\ \texttt{else}\ b' & \to & b \\
\texttt{if false then}\ b\ \texttt{else}\ b' & \to & b' \\
\texttt{if}\ [k]\ \texttt{then}\ b\ \texttt{else}\ b' & \to & [\texttt{if}\ k\ \texttt{then}\ b\ \texttt{else}\ b'] \\
\Gamma_v(a) & \to & \Gamma_v(a') \quad \text{if } a \to a'
\end{array}$$

with $\Gamma_v ::= [\,]\ v \mid b\ [\,] \mid \texttt{if}\ [\,]\ \texttt{then}\ b\ \texttt{else}\ b'$. It only remains to extend the readback procedure to booleans and the new syntactic class of accumulators. The only point worth noticing is that reading back a suspended conditional $\texttt{if}\ k\ \texttt{then}\ b\ \texttt{else}\ b'$ involves normalizing both arms b and b' of the conditional, and reconstructing an `if...then...else` with the readback of k and the normal

$$
\begin{array}{rcl}
(\lambda x.b)\ v & \to & b\{x \leftarrow v\} \\
[k]\ v & \to & [k\ v] \\
\texttt{case}\ C(\vec{v})\ \texttt{of}\ (C_i(\vec{x}_i) \to b_i)_{i\in I} & \to & b_j\{\vec{x}_j \leftarrow \vec{v}\} \qquad \text{if } C = C_j \text{ and } |\vec{v}| = |\vec{x}_j| \\
\texttt{case}\ [k]\ \texttt{of}\ (C_i(\vec{x}_i) \to b_i)_{i\in I} & \to & [\texttt{case}\ k\ \texttt{of}\ (C_i(\vec{x}_i) \to b_i)_{i\in I}] \\
\Gamma_v(a) & \to & \Gamma_v(a') \qquad \text{if } a \to a' \\
\\
\mathcal{N}(b) & = & \mathcal{R}(\mathcal{V}(b)) \\
\mathcal{R}(\lambda x.b) & = & \lambda y.\ \mathcal{N}((\lambda x.b)\ [\tilde{y}]) \quad (y \text{ fresh}) \\
\mathcal{R}(C(v_1,\dots,v_n)) & = & C(\mathcal{R}(v_1),\dots,\mathcal{R}(v_n)) \\
\mathcal{R}([k]) & = & \mathcal{R}'(k) \\
\mathcal{R}'(\tilde{x}) & = & x \\
\mathcal{R}'(k\ v) & = & \mathcal{R}'(k)\ \mathcal{R}(v) \\
\mathcal{R}'(\texttt{case}\ k\ \texttt{of}\ (C_i(\vec{x}_i) \to b_i)_{i\in I}) & = & \texttt{case}\ \mathcal{R}'(k)\ \texttt{of}\ (C_i(\vec{y}_i) \to \mathcal{N}(b\ (C_i(\overrightarrow{[\tilde{y}_i]}))))_{i\in I} \\
& & \text{where } b = \lambda x.\ \texttt{case}\ x\ \texttt{of}\ (C_i(\vec{x}_i) \to b_i)_{i\in I} \\
& & \text{and the } \vec{y}_i \text{ are sequences of fresh variables with } |\vec{y}_i| = |\vec{x}_i|
\end{array}
$$

Figure 1. Symbolic weak reduction and readback for inductive types

forms of b and b'.

$$
\begin{array}{rcl}
\mathcal{N}(b) & = & \mathcal{R}(\mathcal{V}(b)) \\
\mathcal{R}(\lambda x.b) & = & \lambda y.\ \mathcal{N}((\lambda x.b)\ [\tilde{y}]) \quad (y \text{ fresh}) \\
\mathcal{R}(\texttt{true}) & = & \texttt{true} \\
\mathcal{R}(\texttt{false}) & = & \texttt{false} \\
\mathcal{R}([k]) & = & \mathcal{R}'(k) \\
\mathcal{R}'(\tilde{x}) & = & x \\
\mathcal{R}'(k\ v) & = & \mathcal{R}'(k)\ \mathcal{R}(v) \\
\mathcal{R}'(\texttt{if}\ k\ \texttt{then}\ b\ \texttt{else}\ b') & & \\
& = & \texttt{if}\ \mathcal{R}'(k)\ \texttt{then}\ \mathcal{N}(b)\ \texttt{else}\ \mathcal{N}(b')
\end{array}
$$

3.2 Inductive Types

Inductive types in the Calculus of Constructions [21] are similar to datatypes in ML and Haskell: they consist in one or several alternatives, identified by unique constructors C, each constructor carrying zero, one or several terms as arguments. A built-in `case` construct allows shallow pattern-matching on terms of inductive types. Once enriched with inductive types, the source calculus becomes:

Terms: $a ::= x \mid \lambda x.a \mid a_1\ a_2$
$\quad \mid C(\vec{a}) \mid \texttt{case}\ a\ \texttt{of}\ (C_i(\vec{x}_i) \to a_i)_{i\in I}$

We write $\vec{z}$ for a sequence of elements of the syntactic class z, and $|\vec{z}|$ for the number of elements in such a sequence.

The reduction rules for inductive types are:

$$
\begin{array}{rcl}
(\lambda x.a)\ a' & \Rightarrow & a\{x \leftarrow a'\} \\
\texttt{case}\ C(\vec{a})\ \texttt{of}\ (C_i(\vec{x}_i) \to a_i)_{i\in I} & \Rightarrow & a_j\{\vec{x}_j \leftarrow \vec{a}\} \\
& & \text{if } C = C_j \text{ and } |\vec{a}| = |\vec{x}_j| \\
\Gamma(a) & \Rightarrow & \Gamma(a') \quad \text{if } a \Rightarrow a'
\end{array}
$$

Extended terms and weak symbolic reduction are modified accordingly, taking into account the fact that a `case` construct can be applied to an accumulator, and needs to be remembered unevaluated as a whole in this case.

Extended terms:
$b ::= x \mid \lambda x.b \mid b_1\ b_2 \mid [k]$
$\quad \mid C(\vec{b}) \mid \texttt{case}\ b\ \texttt{of}\ (C_i(\vec{x}_i) \to b_i)_{i\in I}$

Accumulators:
$k ::= \tilde{x} \mid k\ v \mid \texttt{case}\ k\ \texttt{of}\ (C_i(\vec{x}_i) \to b_i)_{i\in I}$

Values:
$v ::= \lambda x.b \mid C(\vec{v}) \mid [k]$

Reduction contexts:
$\Gamma_v ::= [\,]\ v \mid b\ [\,] \mid C(\vec{b},[\,],\vec{v})$
$\quad \mid \texttt{case}\ [\,]\ \texttt{of}\ (C_i(\vec{x}_i) \to b_i)_{i\in I}$

Figure 1 defines the weak reduction rules and the normalization and readback procedures in the presence of inductive types. As in the case of the conditional construct in section 3.1, there are two weak reduction rules for `case`, depending on whether the argument is a constructed value $C(\vec{v})$ or an accumulator $[k]$. In the former case, reduction proceeds with the appropriate arm of the `case` construct. In the latter case, an accumulator is built that "remembers" the whole `case` statement.

The readback functions are as in section 3.1, with the treatment of `case` generalizing that of `if...then...else`: we recursively normalize each arm of the `case`, after replacing the pattern-bound variables x_i by run-time representations of fresh free variables; then, a `case` statement is reconstructed from the normal forms of the arms and the readback of the accumulator being matched upon. To emphasize the fact that the `case` construct can be compiled, and thus its arms are not necessarily recoverable from the code, we present the recursive normalization of each arm $C_i(\vec{x}_i) \to b_i$ as an application of the matching function $\lambda x.\texttt{case}\ x\ \texttt{of}\ \dots$ to the argument $C_i(\overrightarrow{[\tilde{y}_i]})$, composed of the constructor of the i^{th} arm applied to the correct number of fresh free variables. The compiled code for the matching function then selects the i^{th} arm and perform the substitution $\vec{x}_i \leftarrow \overrightarrow{[\tilde{y}_i]}$ all by itself.

3.3 Fixpoints

The Calculus of Constructions supports the definition of recursive functions via fixpoints. Fixpoints are introduced by a family of operators $\texttt{fix}_n$, where the positive integer n indicates the position of the argument that is used to guard the recursion and prevent infinite

Writing $b = \lambda f.\lambda x_1 \ldots \lambda x_n.b'$,

$$\begin{array}{rcl}
\mathtt{fix}_n(b)\ v_1\ \ldots\ v_{n-1}\ (C(\vec{v})) & \to & b'\{f \leftarrow b, x_1 \leftarrow v_1, \ldots, x_{n-1} \leftarrow v_{n-1}, x_n \leftarrow C(\vec{v})\} \\
\mathtt{fix}_n(b)\ v_1\ \ldots\ v_{n-1}\ [k] & \to & [\mathtt{fix}_n(b)\ v_1\ \ldots\ v_{n-1}\ k]
\end{array}$$

$$\begin{array}{rcl}
\mathcal{R}(\mathtt{fix}_n(b)\ v_1\ \ldots\ v_i) & = & \mathcal{N}(\mathtt{fix}_n(b))\ \mathcal{R}(v_1)\ \ldots\ \mathcal{R}(v_i) \qquad \text{if } i < n \\
\mathcal{R}'(\mathtt{fix}_n(b)\ v_1\ \ldots\ v_{n-1}\ k) & = & \mathcal{N}(\mathtt{fix}_n(b))\ \mathcal{R}(v_1)\ \ldots\ \mathcal{R}(v_{n-1})\ \mathcal{R}'(k) \\
\mathcal{N}(\mathtt{fix}_n(b)) & = & \mathtt{fix}_n(\lambda g.\lambda y_1 \ldots \lambda y_n.\mathcal{N}(b\ [\tilde{g}]\ [\tilde{y_1}]\ \ldots\ [\tilde{y_n}]) \qquad \text{where } g, y_1, \ldots, y_n \text{ are fresh}
\end{array}$$

Figure 2. Symbolic weak reduction and readback for fixpoints

unrolling:

Terms: $a ::= \ldots \mid \mathtt{fix}_n(\lambda f.\lambda x_1 \ldots \lambda x_n.a)$

The reduction rule for the $\mathtt{fix}_n$ operators is not the standard unrolling rule

$$\mathtt{fix}_n(\lambda f.a) \not\Rightarrow a\{f \leftarrow \mathtt{fix}_n(\lambda f.a)\}$$

since such a rule would allow infinite unrolling, and thus render type-checking undecidable. Instead, the calculus provides a guarded unrolling rule, allowing unrolling of the recursive definition only if the n^{th} argument is a constructed term: writing $a = \mathtt{fix}_n(\lambda f.\lambda x_1 \ldots \lambda x_n.a')$, we have

$$\begin{array}{l}
a\ a_1\ \ldots\ a_{n-1}\ (C(\vec{a}_n)) \\
\quad \Rightarrow \quad a'\{f \leftarrow a, x_1 \leftarrow a_1, \ldots, x_{n-1} \leftarrow a_{n-1}, x_n \leftarrow C(\vec{a}_n)\}
\end{array}$$

Further statically-checked restrictions on the body a' of the recursive definition ensures that it recursively applies f only to strict subterms of the n^{th} argument $C(\vec{a}_n)$. This ensures that all recursive definitions proceed by structural induction, and are thus guaranteed to normalize strongly. At first sight, this restriction might appear to restrict severely the expressiveness of the logic, but this is not so: the guard argument over which the function is structurally recursive can also be an inductive proof term such as a proof of accessibility in a well-founded ordering, guaranteeing the termination of the function. Thus, general recursive definitions can also be handled, as long as they are provably total in the logic.

As a consequence of the guarded unrolling rule, applications of $\mathtt{fix}_n(b)$ to fewer than n values are values (weak normal forms), while applications of $\mathtt{fix}_n(b)$ to $n-1$ values and an accumulator are themselves accumulators:

Extended terms:
$b ::= \ldots \mid \mathtt{fix}_n(\lambda f.\lambda x_1 \ldots \lambda x_n.b)$

Values:
$v ::= \ldots \mid \mathtt{fix}_n(\lambda f.\lambda x_1 \ldots \lambda x_n.b)\ v_1\ \ldots\ v_i \quad \text{if } 0 \le i < n$

Accumulators:
$k ::= \ldots \mid \mathtt{fix}_n(\lambda f.\lambda x_1 \ldots \lambda x_n.b)\ v_1\ \ldots\ v_{n-1}\ k$

The additional reduction rules and readback rules for fixpoints are shown in figure 2. The main point to notice is that in the source calculus, the normal form of a non-applied fixpoint $\mathtt{fix}_n(\lambda f.\lambda x_1 \ldots \lambda x_n.a)$ is simply $\mathtt{fix}_n(\lambda f.\lambda x_1 \ldots \lambda x_n.a')$ where a' is the normal form of a. In other terms, we normalize a without assuming anything known on the function name f nor on the parameters x_i, and in particular without assuming that f unrolls to $\mathtt{fix}_n(\ldots)$: the guarded unrolling rule does not apply here. For the same reason, an application of $\mathtt{fix}_n(a)$ to fewer than n arguments in normal form, or to $n-1$ normal forms and one normal form that is not a constructor application, is itself in normal form.

4 An Abstract Machine for Weak Symbolic Reduction

We now turn to implementing weak symbolic reduction by compilation to a suitable abstract machine. This abstract machine is a slight extension of the ZAM, which underlies the bytecode interpreter of Objective Caml [16, 17], In the terminology of [20], the ZAM is an environment- and closure-based abstract machine following the "push-enter" model, and implementing a call-by-value evaluation strategy.

The purpose of this section is to demonstrate that minor modifications of an existing abstract machine for weak reduction suffice to turn it into an abstract machine for weak symbolic reduction, without impacting significantly the evaluation speed for closed terms. Consequently, we present a minimal subset of the ZAM, omitting several aspects of the real ZAM that are irrelevant for our purposes, such as the optimization of tail-calls, the use of a register to cache the top of the stack, and the actual representation of environments (as a stack-allocated part and a heap-allocated part) and closures (single block, minimal environments). These omitted aspects are discussed in [16] and in the first author's forthcoming dissertation.

4.1 Machine States and Machine Values

The machine state has four components:

- A code pointer c representing the code being executed as a sequence of instructions.
- An environment e: a sequence of machine values $\hat{v}_1 \ldots \hat{v}_n$ associating the value $\hat{v}_i$ to the variable having de Bruijn index i.
- A stack s (a sequence of machine values and return contexts) holding function arguments, intermediate results, and function return contexts.
- An integer n counting the number of function arguments available on the stack.

The machine-level values $\hat{v}$ manipulated by the abstract machine are pointers to heap blocks, written $[T : \hat{v}_1 \ldots \hat{v}_n]$, where T is a tag attached to the block (a small integer), and $\hat{v}_1 \ldots \hat{v}_n$ are the values contained in the block. We use tag 0 for representations of accumulator terms $[k]$, tags $1 \ldots n$ to encode the constructor C of constructed terms $C(\vec{v})$, and a distinct tag T_λ for function closures. The values of the calculus are represented by heap blocks as follows:

- A function value is represented by a closure $[T_\lambda : c, e]$ of the compiled code c for the function body, and an environment e associating values to the variables free in $\lambda x.b$.
- For inductive types, we assume that the constructors of an inductive type are numbered consecutively starting at 1 when

the type is declared. We write $\#C$ for the tag number associated with constructor C. The value $C(\vec{v})$ is then represented by the heap block $[\#C : \vec{\hat{v}}]$, where $\vec{\hat{v}}$ are the representations of $\vec{v}$.

- Finally, an accumulator $[k]$ is represented by the heap block $[0 : \texttt{ACCU}, \hat{k}]$. This is a pseudo-closure with the single machine instruction ACCU as code part, and an encoding $\hat{k}$ of k as environment part. This encoding is as follows:
  - $[0 : \tilde{x}]$ represents the free variable $\tilde{x}$.
  - $[1 : \hat{k}, \hat{v}_1, \ldots, \hat{v}_n]$ represents the suspended application $k\ v_1\ \ldots\ v_n$.
  - $[2 : \hat{k}, c, e]$ represents the suspended case statement $\texttt{case}\ k\ \texttt{of}\ (C_i(\vec{x}_i) \rightarrow b_i)$, where c and e are the code and the environment for the function $\lambda x.\texttt{case}\ x\ \texttt{of}\ (C_i(\vec{x}_i) \rightarrow b_i)$.
  - $[3 : c, e, \hat{k}]$ represents the suspended fixpoint application $\texttt{fix}_n(\lambda f.\lambda x_1 \ldots \lambda x_n.b)\ v_1\ \ldots\ v_{n-1}\ k$, where c and e are the code and the environment for the partially applied fixpoint $\texttt{fix}_n(\lambda f.\lambda x_1 \ldots \lambda x_n.b)\ v_1\ \ldots\ v_{n-1}$.

The transitions of the abstract machine are shown in figure 3, and the halting configurations in figure 4.

4.2 Compilation and Execution of Functions

The compilation scheme for the ZAM is presented as a function $[\![b]\!]\ c$, where b is an expression and c an instruction sequence representing the continuation of b. It returns an instruction sequence that evaluates b, leaves its value at the top of the stack, and continues in sequence by executing the code c. For the pure λ-calculus, the compilation scheme is defined as follows:

$$
\begin{array}{rcl}
[\![x]\!]\ c & = & \texttt{ACCESS}(i); c \\
 & & \text{where } i \text{ is the de Bruijn index of } x \\
[\![\lambda x_1 \ldots x_m.b]\!]\ c & = & \texttt{CLOSURE}(\underbrace{\texttt{GRAB}; \ldots; \texttt{GRAB};}_{m \text{ times}} [\![b]\!]\ \texttt{RETURN}); c \\
[\![b\ b_1\ \ldots\ b_m]\!]\ c & = & \texttt{PUSHRETADDR}(c); \\
 & & [\![b_n]\!]\ \ldots\ [\![b_1]\!]\ [\![b]\!]\ \texttt{APPLY}(m)
\end{array}
$$

A reference to a variable x is compiled to an ACCESS instruction carrying the de Bruijn index of the variable. The execution of $\texttt{ACCESS}(i)$ looks up the i^{th} entry in the machine environment and pushes it on the stack.

A curried function $\lambda x_1 \ldots x_m.b$ compiles to a CLOSURE instruction, which at run-time builds a closure of its argument (a piece of code) with the current environment, and pushes the closure on the stack. The argument of CLOSURE is the code for the body b of the function, preceded by m GRAB instructions and followed by a RETURN instruction.

As its name suggests, GRAB attempts to pop one function argument from the stack and add it in front of the current environment. (This corresponds exactly to a β_v reduction step.) This is possible only if the count n of available arguments is non-zero, and causes n to be decremented. If $n = 0$, all arguments have been consumed already by previous GRAB instructions, hence the curried function is partially applied. The virtual machine then returns a closure of the current code (including the aborted GRAB) and the current environment to the caller of the function.

The RETURN instruction is the dual of GRAB: if all arguments have been consumed ($n = 0$), the value of the function body is returned to its caller; if some arguments remain ($n > 0$), the curried function was "over-applied", and its result, which must be a function, is to be applied to the remaining argument. The latter is achieved by tail-calling the closure found on the top of the stack with the remaining arguments.

The code for a multiple application $b\ b_1\ \ldots\ b_m$ first pushes a return frame $\langle c, e, n \rangle$ containing the code c to be executed when the applied function returns, as well as the current environment and argument count (instruction PUSHRETADDR). Then, the arguments $b_m \ldots b_1$ and the function b are evaluated right-to-left, and their values pushed on the stack. Finally, the $\texttt{APPLY}(m)$ instruction branches to the code of the closure obtained by evaluating b, setting the argument count to m.

The "push-enter" nature of the ZAM is apparent in the fact that we compile curried functions and multiple applications as a whole, and evaluate arguments right-to-left. For an application $b\ b_1\ \ldots\ b_m$, this avoids the construction of intermediate closures representing the applications $b\ b_1$, $(b\ b_1)\ b_2$, ..., like an "eval-apply" machine such as the SECD would do. The compilation scheme given above remains correct if applied to non-maximal curried functions and applications (e.g. if we treat $b\ b_1\ b_2$ as two applications, $(b\ b_1)\ b_2$), but becomes less efficient at run-time.

At this point, the reader is probably wondering how we treat the β_s reduction rule, corresponding to the application of an accumulator $[k]$. Recall that such an accumulator is represented by a pseudo-closure $[0 : \texttt{ACCU}, \hat{k}]$. Thus, the APPLY instruction will cause the ACCU instruction to be executed with $\hat{k}$ being moved to the environment register of the machine. The ACCU instruction simply pops all provided arguments $\hat{v}_1, \ldots, \hat{v}_n$ off the stack, constructs an accumulator $[0 : \texttt{ACCU}, [1 : \hat{k}, \hat{v}_1, \ldots, \hat{v}_n]]$ representing the application of k to these arguments, and returns this accumulator to the caller. Thus, the effect of the β_s reduction rule is achieved without testing at run-time whether the function part of an application is a real closure or an accumulator. In other terms, we implement symbolic weak reduction of function applications without any run-time overhead on applications of regular functions.

4.3 Inductive Types

Inductive types lead to the addition of the following two compilation rules:

$$
\begin{array}{l}
[\![C(b_1, \ldots, b_n)]\!]\ c = [\![b_n]\!]\ \ldots\ [\![b_1]\!]\ \texttt{MAKEBLOCK}(n, \#C); c \\
[\![\texttt{case}\ b\ \texttt{of}\ (C_1(\vec{x}_1) \rightarrow b_1\ \ldots\ C_n(\vec{x}_n) \rightarrow b_n)]\!]\ c = \\
\quad \texttt{PUSHRETADDR}(c); \texttt{SWITCH}([\![b_1]\!]\ \texttt{RETURN}, \ldots, [\![b_n]\!]\ \texttt{RETURN})
\end{array}
$$

In the latter rule, we assume that the arms of the case are ordered so that $\#C_i = i$ for $i = 1, \ldots, n$. No generality is lost since the type system of the Constructions guarantees the exhaustiveness of case statements.

The compilation and execution of constructor applications is straightforward: the arguments of the constructor are evaluated, and the MAKEBLOCK instruction then creates the representation of the constructed term, tagged with the constructor number.

For the case statement, a return frame to the continuation c is pushed first. The SWITCH instruction then discriminates on the tag of the matched value, and branches to the code of the corresponding case arm, after adding the fields of the matched value to the envi-

Code	Environment	Stack	Num. arg.	
$\texttt{ACCESS}(i);c$	e	s	n	
c	e	$e(i).s$	n	
$\texttt{CLOSURE}(c');c$	e	s	n	
c	e	$[T_\lambda : c', e].s$	n	
$\texttt{PUSHRETADDR}(c');c$	e	s	n	
c	e	$\langle c', e, n\rangle.s$	n	
$\texttt{APPLY}(i)$	e	$[T : c', e'] : s$	n	
c'	e'	s	i	
$\texttt{GRAB};c$	e	$v : s$	$n+1$	
c	$v.e$	s	n	
$c_0 = \texttt{GRAB};c$	e	$\langle c', e', n'\rangle.s$	0	
c'	e'	$[T_\lambda : c_0, e] : s$	n'	
$\texttt{RETURN}$	e	$v.\langle c', e', n'\rangle.s$	0	
c'	e'	$v.s$	n'	
$\texttt{RETURN}$	e	$[T : c', e'].s$	n	if $n > 0$
c'	e'	s	n	
$\texttt{ACCU}$	k	$v_1 \dots v_n.\langle c', e', n'\rangle : s$	n	
c'	e'	$[0 : \texttt{ACCU}, [1 : k, v_1, \dots, v_n]].s$	n'	
$\texttt{MAKEBLOCK}(T, m);c$	e	$v_1 \dots v_m.s$	n	
c	e	$[T : v_1, \dots, v_m].s$	n	
$\texttt{SWITCH}(c_1, \dots, c_m)$	e	$[T : v_1, \dots, v_p].s$	n	if $1 \leq T \leq m$
c_T	$v_p \dots v_1.e$	s	0	
$c_0 = \texttt{SWITCH}(c_1, \dots, c_m)$	e	$[0 : \texttt{ACCU}, k].s$	n	
$\texttt{RETURN}$	e	$[0 : \texttt{ACCU}, [2 : k, c_0, e]].s$	0	
$\texttt{CLOSUREREC}(c');c$	e	s	n	
c	e	$v.s$	n	where $v = [T_\lambda : c', v.e]$
$\texttt{GRABREC};c$	e	$[T : \vec{v}].s$	$n+1$	if $T > 0$
c	$[T : \vec{v}].e$	s	n	
$\texttt{GRABREC};c$	e	$[0 : \texttt{ACCU}, k].s$	$n+1$	
$\texttt{RETURN}$	e	$[0 : \texttt{ACCU}, [3 : c, e, k]].s$	n	
$c_0 = \texttt{GRABREC};c$	e	$\langle c', e', n'\rangle.s$	0	
c'	e'	$[T_\lambda : c_0, e].s$	n'	

Figure 3. Transitions of the abstract machine. Each two-line entry represents a transition; the first line is the machine state before the transition, and the second line is the state after.

Code	Environment	Stack	Num. arg.	Result value
$\texttt{RETURN}$	e	$v.\varepsilon$	0	v
$c_0 = \texttt{GRAB};c$	e	ε	0	$[T_\lambda : c_0, e]$
$c_0 = \texttt{GRABREC};c$	e	ε	0	$[T_\lambda : c_0, e]$
$\texttt{GRABREC};c$	e	$[0 : \texttt{ACCU}, k].\varepsilon$	1	$[0 : \texttt{ACCU}, [3 : c, e, k]]$

Figure 4. Final configurations for the abstract machine. The rightmost column is the result value (weak head normal form) for the execution

ronment, thus binding the pattern variables $\vec{x}_i$. The RETURN at the end of the code for each arm then restores the original environment and branches back to the continuation c. (The SWITCH instruction is careful to set the count of extra arguments to 0, ensuring that the RETURN will never perform over-application.)

If the matched value has tag 0, denoting a `case` on an accumulator k, the SWITCH instruction builds the accumulator $[0 : \mathtt{ACCU}, [2 : k, c_0, e]]$ where c_0 is the code consisting of the SWITCH instruction and the code for the arms of the `case`. Later, the readback procedure can restart the abstract machine with code c_0, environment e, stack $v.\varepsilon$ and number of arguments 0, where v is the machine representation of $C_i(\tilde{\vec{y}}_i)$. The code for the i^{th} arm will evaluate, then stop on its final RETURN instruction, since a final configuration is reached (first case in figure 4). Thus, this achieves the effect of computing the value of $b_i\{\vec{x}_i \leftarrow \tilde{\vec{y}}_i\}$.

This "replay" facility is the reason why a PUSHRETADDR-RETURN pair is used to implement the control-flow merging of the arms of the `case`, rather than a more standard GOTO instruction. Apart from the slightly higher cost of the PUSHRETADDR-RETURN combination compared with a GOTO, this compilation scheme support symbolic execution and readback for `case` statements without impacting the evaluation speed on non-symbolic, closed terms. (The actual implementation uses a jump table for the SWITCH instruction, covering both the case tag $= 0$ and tag ≥ 1, so that an additional test for tag $= 0$ is not required.)

4.4 Fixpoints

The compilation of fixpoint definitions is as follows:

$$[\![\mathtt{fix}_n(\lambda f.\lambda x_1 \ldots \lambda x_n.b)]\!]\ c = \\ \mathtt{CLOSUREREC}(\underbrace{\mathtt{GRAB}; \ldots; \mathtt{GRAB}}_{n-1 \text{ times}}; \mathtt{GRABREC}; [\![b]\!]\ \mathtt{RETURN}); c$$

The CLOSUREREC instruction constructs a closure for a recursive function. In the simplified presentation given in this paper, this is a cyclic closure $v = [T_\lambda : c, v.e]$ where the first slot of the environment, corresponding to the recursive variable f in the source term, points back to the closure itself [8]. (The actual implementation uses the scheme described in [1] instead of cyclic closures.)

The code part of the recursive closure consists of the compilation of the body b of the definition, preceded by $n-1$ GRAB instructions and one GRABREC instruction. The $n-1$ GRAB instructions absorb the first $n-1$ parameters, which are not subject to a guard condition. The last parameter, however, is guarded: evaluation should not proceed if it is bound to an accumulator. Hence, GRABREC checks whether the argument is an accumulator, and if so, constructs and returns an accumulator representing the suspended application of the fixpoint.

5 A Mechanically-checked Proof of Correctness

The first author has developed a fully formal specification and proof of correctness of the approach described in this paper, using the Coq proof assistant. Such a mechanically-checked correctness proof is required to ensure that we do not compromise the logical consistency of the Coq system by plugging a buggy evaluator in it. (The correctness of the kernel of the Coq proof- and type-checker has previously been mechanically proved in Coq by Barras [5, 3].) We now give a high-level overview of this 5000-line development; a detailed presentation is beyond the scope of this paper and will be published separately.

The bulk of the development consists in proving that the modified ZAM machine and its compilation scheme (including the optimizations that we left out in section 4) faithfully implement the weak symbolic reduction semantics. The remainder of the proof shows the correctness of the readback, strong normalization and equivalence testing procedures; this part of the proof is much shorter and easier, and follows the lines of section 2.3, using de Bruijn indices instead of variable names.

Following [13], the correctness of the modified ZAM and its compilation scheme is established by proving a simulation result between the reductions of the source term and the transitions of the abstract machine executing the compiled code for the term. The first step consists in setting up a decompilation relation $S \Uparrow b$ between machine states $S = (c, e, s, n)$ and source terms b, and also between machine values and source values $\hat{v} \Uparrow v$. This decompilation relation can be viewed as a left inverse of the compilation function: if we compile a term and build an initial machine state with this code, the decompilation of this initial state gives us back the original term. However, decompilation is also defined for intermediate machine states, reached after one or several transitions, and where the code part of the state is not the image of a source term by the compilation function. Several examples of decompilation relations for different, simpler abstract machines are shown in [13]. Ours is broadly similar, with the exception that [13] uses explicit substitutions in the source language, while we perform classical substitution as part of the decompilation predicate in order to keep the source language unchanged.

Once the decompilation relation is set up, we can show the main simulation result: one transition of the abstract machine corresponds to zero, one or several reductions on the source term.

LEMMA 7 (SIMULATION). *If $S \Uparrow b$ and the machine performs a transition from state S to state S', then there exists a source term b' such that $S' \Uparrow b'$ and $b \xrightarrow{*} b'$.*

We also show that initial states of the machine decompile to the original term. There are two kinds of initial states of interest: $([\![b]\!]\ \mathtt{RETURN}, \varepsilon, \varepsilon, 0)$, corresponding to executing the code of a closed term b, and $(\mathtt{APPLY}(n), \varepsilon, \hat{f}.\hat{v}_1 \ldots \hat{v}_n.\varepsilon, 0)$, corresponding to applying the closure $\hat{f}$ to the arguments $\hat{v}_1, \ldots, \hat{v}_n$ during the readback procedure.

LEMMA 8 (INITIAL STATES).

1. *$([\![b]\!]\ \mathtt{RETURN}, \varepsilon, \varepsilon, 0) \Uparrow b$ if b is closed.*
2. *$(\mathtt{APPLY}(n), \varepsilon, \hat{f}.\hat{v}_1 \ldots \hat{v}_n.\varepsilon, 0) \Uparrow f\ v_1\ \ldots\ v_n$ if $\hat{f} \Uparrow f$ and $\hat{v}_i \Uparrow v_i$ for $i = 1, \ldots, n$.*

Symmetrically, final machine configurations decompile to values.

LEMMA 9 (FINAL STATES). *If S is a final machine configuration and $\hat{v}$ the associated return value, as defined in figure 4, then there exists a source value v such that $S \Uparrow v$ and $\hat{v} \Uparrow v$.*

To show the correctness of the abstract machine and its compilation scheme, it remains to show two properties. First, the machine does not get stuck when executing the compiled code of a term that is not stuck.

Test	Our system	Coq CBV	Coq Lazy	OCaml bytecode
1. Normalization of `factorial`(9) (Peano integers)	14.2s	61.6s	466s	0.347s
2. Normalization of `is_even`$($`factorial`$(9))$ (Peano integers)	0.447s	46.9s	4.82s	0.357s
3. Normalization of 256×64 (Church integers)	0.106s	0.116s	1.99s	n/a
4. Normalization of $\lambda x.\lambda y.\ (128+x) \times (128+y)$ (Church integers)	0.082s	0.107s	1.81s	n/a
5. Equivalence of `factorial`(8) and `factorial`$'(8)$ (Peano integers)	0.096s	n/a	9.02s	??
6. Equivalence of 256×64 and 64×256 (Church integers)	0.094s	n/a	2.00s	n/a

Figure 5. Benchmark results for the synthetic tests (on a Pentium III 700Mhz, 512 Mb)

Proof	Our system	Coq	OCaml bytecode	OCaml native
Coq's standard theories	135s	131s	n/a	n/a
4-color theorem, perimeter 11	1.68s	56.7s	1.18s	0.30s
4-color theorem, perimeter 12	6.50s	259s	6.18s	1.92s
4-color theorem, perimeter 13	14.8s	680s	15.5s	4.11s
4-color theorem, perimeter 14	69.6s	out of memory	73.1s	19.8s

Figure 6. Benchmark results for the checking of actual proofs (on a Pentium III 700Mhz, 512 Mb)

LEMMA 10 (PROGRESS). *If $S \Uparrow b$, and S is not a final state, and b is not stuck (i.e. b reduces or b is a value), then the machine can perform a transition from S.*

Second, the abstract machine always terminates when given the code for a term that evaluates to a value. Given the simulation lemma 7, the only way the machine could fail to terminate is by performing an infinite number of consecutive "silent" transitions, that is, transitions between two states that decompile to the same term. To show that this cannot happen, we define a non-negative integer measure $|S|$ on machine states S, and show that it strictly decreases at each silent transition. Again, [13] gives examples of such measures for other machines.

LEMMA 11 (NO STUTTERING). *If the machine performs a transition from S to S', and $S \Uparrow b$ and $S' \Uparrow b$, then $|S'| < |S|$.*

As a corollary of the previous lemmas, we obtain the total correctness of the abstract machine and its compilation scheme:

THEOREM 2. *Let S be a machine state that decompiles to b. If $b \xrightarrow{*} \mathcal{V}(b)$, then the abstract machine started in state S terminates with a return value $\hat{v}$ that decompiles to $\mathcal{V}(b)$. If b diverges (reduces infinitely), the abstract machine started in state S performs an infinite number of transitions.*

6 Experimental Results

The first author has implemented the approach presented here in the context of the Coq proof assistant, version 7. The implementation consists of a bytecode compiler and a readback procedure and equivalence tester written in OCaml, and an abstract machine interpreter written in C. The Coq proof checker was modified to use our equivalence tester.

Synthetic tests. Figure 5 gives the time it takes to strongly normalize or to decide the equality of various (artificial) test terms. For comparison purposes, we also give timings for the current interpreter-based normalization and equivalence testing procedures of the Coq system. For normalization, Coq supports both call-by-value and lazy call-by-name strategies, while only the latter is supported for equivalence testing. When the example does not require normalization under λ, we also give the times for the OCaml bytecode interpreter. The timings for our system and for OCaml include compilation times in addition to evaluation times, in order to make the comparison with Coq's interpreter fairer.

Overall, our compiled implementation is 10 to 100 times faster than Coq's lazy call-by-name interpreter, and 1.1 to 100 times faster than Coq's call-by-value interpreter.

Example 1 computes factorial 9 using Peano integers. This involves no normalization under λ. The result of the weak evaluation is 362880 applications of the constructor `Succ` to the constant `Zero`, which is then transformed into an isomorphic source term during readback. Example 2 computes the parity of factorial 9. Compared with example 1, example 2 involves more work during weak reduction, but much less work during readback, since its result is an atomic constant. It therefore normalizes much faster than example 1 in our system, approaching the speed of the OCaml bytecode interpreter.

Example 3 computes 256×64 using Church integers. Normalization under λ is required, and the normal form is quite large, requiring significant readback effort. Example 4 computes $\lambda x.\lambda y.\ (128+x) \times (128+y)$ using Peano arithmetic; it requires not only normalization under λ, but also large amounts of normalization of `case` and `fix` constructs applied to the free variables x, y. In both cases, we observe significant speed-ups compared with Coq's lazy call-by-name interpreter, but only a 10% to 30% improvement compared with Coq's call-by-value interpreter.

Examples 5 and 6 perform equivalence testing instead of strong normalization. In both cases, we compare two terms that evaluate to the same (large) normal form, but are syntactically different enough to force full reduction to determine their equivalence.

Proof checking. Figure 6 gives total proof checking times obtained with the original Coq implementation and with our modified implementation. The efficiency of our implementation depends greatly on whether the proofs being checked involve significant amount of computation. For proofs that involve few computations, such as the standard library of theories distributed with the Coq system, the times are nearly identical. This was to be expected, since almost all beta-equality tests performed are trivial: they compare small terms that are syntactically identical. This is the least favorable case for our implementation, since we pay the price of compiling the terms down to bytecode, yet gain essentially nothing in return. This compilation overhead is quite low, however, resulting in a slowdown of only 3% compared with the original Coq implementation.

At the other end of the spectrum, proofs that involve significant amounts of computation exhibit speedups by one order of magnitude. As a representative example, we used Gonthier and Werner's Coq proof of the 4-color theorem, and more precisely the part of the proof that checks 4-colorability of a large number of elementary planar graphs. This part of the proof consists mostly in evaluating a function deciding 4-colorability on the elementary graphs. Here, our implementation is 33 to 45 times faster than the original Coq system. Owing to its more efficient use of memory, our implementation is able to complete the checking of the whole proof, while the original Coq runs out of memory on the largest configurations (of perimeter 14).

To compare the performances of our reducer with those of the OCaml system, we used Coq's extraction facility to extract from the Coq development the Caml code that defines the decision procedure and applies it to the elementary graphs. As figure 6 shows, our reducer runs nearly at the same speed as OCaml's bytecode interpreter; the speed ratio varies between 1.4 and 0.95. Compiling the extracted Caml code with the OCaml native-code compiler results in speed ratios between 3.5 and 5.6, which is typical of the speedups obtained by going from bytecode interpretation to native-code generation.

7 Related Work

Implementations of strong reduction. The work most closely related to us is Crégut's abstract machine for strong reduction [9, 10]. This machine is derived from Krivine's machine and implements a lazy evaluation strategy. The code is executed by expansion to Motorola 68000 assembly code. Like ours, Crégut's machine can handle terms containing free variables. However, reductions under lambdas are not performed by a separate readback phase as in our approach, but directly by the abstract machine: when encountering a term $\lambda x.b$ in head position, Crégut's machine immediately proceeds to reducing b (with a free x variable). This is potentially faster than our readback scheme when the goal is to compute the normal form of a term, but prevents early stopping when the goal is to compare two terms for β-equality. Also, strong reduction of `case` statements and recursive definitions is mentioned in [10] but not formalized.

Some of the mechanisms we use can be found in optimized interpreters performing strong reduction. Barras' reducer [4] uses environments and closures, and a reduction strategy based on Crégut's machine. Nadathur and Wilson [19] represent terms in such as way that reduction can be easily stopped at weak head normal form, while keeping enough information to restart strong evaluation later. Still, these approaches remain essentially based on interpretation of a tree-based data structure, without compilation to bytecode.

Online partial evaluation. The connections between strong normalization and partial evaluation (program specialization) [15] are well known: specializing a function $\lambda x.\lambda y.a$ for $x = b$ amounts to strongly normalizing the partial application $(\lambda x.\lambda y.a)\ b$. Classical partial evaluation comes in two flavors: offline partial evaluation, where a preliminary binding-time analysis classifies sub-terms into dynamic and static terms; and online partial evaluation, where the tests "static or dynamic?" are performed on the fly during specialization. Our approach does not rely on binding-time analysis, and therefore is a form of online partial evaluation.

Our decomposition of strong normalization into (compiled) weak symbolic reduction and (interpreted) readback is non-standard in traditional online partial evaluation. However, a similar decomposition is used in the online type-directed partial evaluation of Danvy [12] and Sumii and Kobayashi [22]. (The "type-directed" qualifier is a bit of a misnomer, since this approach does not exploit static typing information, unlike the original type-directed partial evaluation discussed below.) The main difference between online TDPE and our work is that online TDPE, like traditional online partial evaluation, operates on a standard two-level language, and instruments elementary operations of the language (function applications, `case` statements, etc) with run-time tests "static or dynamic?"; in contrast, our presentation in terms of accumulators allows implementing function application and `case` statements without explicit run-time tests "proper value or accumulator?", thus entailing essentially no additional cost compared with a standard, non-symbolic weak evaluator. However, our approach requires a specially adapted virtual machine, while online TDPE can be implemented on top of any functional language using only source-level transformations.

TDPE and normalization by evaluation. Type-directed partial evaluation (TDPE), also known as normalization by evaluation, computes normal forms by combining a standard weak evaluator with a type-directed reification procedure that reconstructs normalized terms from values [11, 6]. A function value is reified by applying it to a generic argument derived from the type of the function domain (an η-long form of a free variable, where all dynamic application nodes are replaced by static application nodes), and recursively normalizing this application.

This alternation between evaluation and reification looks superficially similar to our approach, but differs in several key aspects. First, TDPE uses η-expansions and produces an η-long $\beta\eta$ normal form, while our approach does not perform η-expansions and produces pure β normal forms. Since η conversion is not valid in the Calculus of Constructions, TDPE is not applicable in our context.

The second difference between TDPE and our approach is that the reification procedure of TDPE is based on the types of the terms to be normalized, while our readback procedure proceeds by examination of the shape of their values, without needing type information. While the recursion on the type structure that underlies TDPE is extremely elegant, it has been worked out only for simply-typed lambda calculus and for system F [24]. Extending TDPE to a type system as rich as that of the Constructions is an open issue.

By abandoning the guidance of the type system, our approach loses one of the advantages of TDPE: the ability to reuse an existing weak evaluator unchanged. Indeed, we had to modify the abstract machine so that it deals correctly with free variables (accumulators). However, our modifications are minimal, and this advantage of TDPE holds only when the language contains only functions and products: to handle sums, TDPE requires a weak evaluator that features control operators (prompts or `call/cc`), which few do; in contrast, our approach requires a few more modifications for sums, but the total of these modifications still requires less implementation work than adding control operators to a weak evaluator that does not have them initially.

8 Conclusions and Future Work

We have shown how minimal modifications to an existing abstract machine-based weak evaluator, combined with a simple, type-oblivious readback procedure, lead to an efficient implementation of strong reduction and equivalence testing for a purely functional language featuring products, sums, and guarded recursion.

We have presented this approach in the context of call-by-value weak evaluation and an environment-based abstract machine. It would be interesting to investigate their applicability to other weak evaluators, such as lazy evaluation and graph reduction.

The application of this work to the Coq proof assistant raises interesting type-theoretic issues. First, we test equivalence between type-erased terms, while the Calculus of Construction performs conversion between type-annotated terms. The logical consistency of the former approach follows from Miquel's model-theoretic results [18] in the case of the core Calculus of Constructions, but remains to be extended to inductive types.

A related issue is the treatment of proof terms during equivalence checking. The types being compared for equivalence can contain arbitrary proof terms. Proof terms are often large and costly to normalize. However, the general principle of proof irrelevance suggests that it might not be necessary to normalize them, since a proof of a given proposition is (morally) just as good as any other proof of this proposition. This suggests replacing all proof terms contained in types by a single constant `P` (a nullary constructor) before testing the equivalence of two types. Again, more work is needed to prove that the corresponding relaxed conversion rule is logically consistent.

Acknowledgments

Henri Laulhère conducted preliminary experiments on modifying the Objective Caml compiler to implement strong normalization circa 1998; the results of these experiments were communicated and explained to us by Alexandre Miquel, and partly inspired this work. We thank Bruno Barras and Benjamin Werner for their expertise on Coq and the Calculus of Constructions, and the anonymous reviewers for pointing us to the work on online TDPE.

References

[1] A. W. Appel. *Compiling with continuations*. Cambridge University Press, 1992.

[2] K. Appel and W. Haken. Every planar map is four colorable. *Illinois J. Math*, 21:429–567, 1977.

[3] B. Barras. *Auto-validation d'un système de preuves avec familles inductives*. PhD thesis, University Paris 7, 1999.

[4] B. Barras. Programming and computing in HOL. In *Theorem Proving in Higher Order Logics 2000*, volume 1869 of *LNCS*, pages 17–37. Springer-Verlag, 2000.

[5] B. Barras and B. Werner. Coq in Coq. Submitted for publication, 2000.

[6] U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In *Logic in Computer Science '91*, pages 203–211. IEEE Computer Society Press, 1991.

[7] T. Coquand and G. Huet. The calculus of Constructions. *Inf. and Comp.*, 76(2/3):95–120, 1988.

[8] G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. *Science of Computer Programming*, 8(2):173–202, 1987.

[9] P. Crégut. An abstract machine for lambda-terms normalization. In *Lisp and Functional Programming 1990*, pages 333–340. ACM Press, 1990.

[10] P. Crégut. *Machines à environnement pour la réduction symbolique et l'évaluation partielle*. PhD thesis, University Paris 7, 1991.

[11] O. Danvy. Type-directed partial evaluation. In *23rd symp. Principles of Progr. Lang*, pages 242–257. ACM Press, 1996.

[12] O. Danvy. Online type-directed partial evaluation. Technical Report RS-97-53, BRICS, 1997.

[13] T. Hardin, L. Maranget, and B. Pagano. Functional runtimes within the lambda-sigma calculus. *Journal of Functional Programming*, 8(2):131–176, 1998.

[14] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, 1993.

[15] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[16] X. Leroy. The ZINC experiment: an economical implementation of the ML language. Technical report 117, INRIA, 1990.

[17] X. Leroy, J. Vouillon, and D. Doligez. The Objective Caml system. Software and documentation available on the Web, `http://caml.inria.fr/`, 1996–2002.

[18] A. Miquel. *Le Calcul des Constructions implicite: syntaxe et sémantique*. PhD thesis, Université Paris 7, 2001.

[19] G. Nadathur and D. S. Wilson. A notation for lambda terms: A generalization of environments. *Theoretical Comput. Sci.*, 198:49–98, 1998.

[20] S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: the spineless tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, 1992.

[21] F. Pfenning and C. Paulin-Mohring. Inductively defined types in the Calculus of Constructions. In *Mathematical Foundations of Programming Semantics*, volume 442 of *LNCS*, pages 209–228. Springer-Verlag, 1990.

[22] E. Sumii and N. Kobayashi. Online type-directed partial evaluation for dynamically-typed languages. *Computer Software, Iwanami Shoten*, 17(3):38–62, 2000.

[23] K. N. Verma, J. Goubault-Larrecq, S. Prasad, and S. Arun-Kumar. Reflecting BDDs in Coq. In *6th Asian Computing Science Conference (ASIAN'2000)*, number 1961 in LNCS, pages 162–181. Springer-Verlag, 2000.

[24] R. Vestergaard. The polymorphic type theory of normalisation by evaluation. Draft, 2001.