Extending Coq with Imperative Features and its Application to SAT Verification

Michaël Armand[1], Benjamin Grégoire[1], Arnaud Spiwack[2], and Laurent Théry[1]

[1] INRIA Sophia Antipolis - Méditerranée, France,
{Michael.Armand, Benjamin.Gregoire, Laurent.Thery}@inria.fr
[2] LIX, École Polytechnique, France,
Arnaud.Spiwack@lix.polytechnique.fr

Abstract. Coq has within its logic a programming language that can be used effectively to replace many deduction steps into a single computation, this is the so-called reflection. In this paper, we present a major improvement of the evaluation mechanism that preserves its correctness and makes it possible to deal with cpu-intensive tasks such as proof checking of SAT traces.

1 Introduction

In the Coq proof assistant [2], functions are active objects. For example, let us consider the sum of two natural numbers. The sum function is defined recursively on its arguments using, say, Peano's definition. Then `1+2` is an expression that can be *computed* to its expected value. In particular to prove `1+2=3`, we simply need to know that equality is reflexive, and the system takes care of checking that `1+2` and `3` compute to the same value. Note that this computation (also called normalisation of λ-calculus) is not restricted to ground terms, like in our example: it can act as a symbolic evaluation on any term. Furthermore, Coq being based on the Curry-Howard isomorphism, writing a proof or a program is essentially the same. These remarks are the bases of proofs by reflection, which consists in replacing many deduction steps by a single computation. This technique has become popular in the Coq community since a few years. Its most impressive application would be the formal proof of the four-colour theorem [6].

Using reflection can greatly improve the checking time of proofs. However, as one pushes the limits of it, efficiency can become a concern. In that respect, a major improvement has already been achieved through the introduction of a dedicated virtual machine [7] allowing Coq programs to compare with (bytecode compiled) OCaml [8] ones. Still, there are strong restriction left. First, there are no primitive data-structures. Every type is encoded using the constructs allowed by the system (primarily, inductive definitions). Also, there is no possibility to use destructive data-structures, which can be much more efficient than purely functional ones in some circumstances. To be able to go further on what can be efficiently programmed in Coq, we will add new data-structures such as native integers and destructive arrays. The challenge is to achieve this, changing as

little as possible, in order to preserve trust in the correctness of evaluation in Coq, and nevertheless to get an effective speed-up thanks to the new feature.

The paper is organised in the following way. In Section 2, we describe how it is possible in Coq to benefit from the arithmetic capabilities of the microprocessor. Section 3 is dedicated to arrays. We then propose two examples that illustrate the benefit of our extension. In Section 4, we present the toy example of the Mini-Rubik for which we use computation to prove that it is always solvable in less than 11 moves. Section 5 is dedicated to a more challenging example. In order to prove the unsatisfiabily of large boolean propositions, we replay in a reflexive way proof traces generated by SAT solvers.

2 Extending Coq with machine integers

Arithmetic is currently defined in Coq as a standard inductive type. Thus, computations with numbers do not differ from other data-structures: it is a plain symbolic evaluation. What we aim at, here, is to rely on the arithmetic of the processor to speed-up computations within Coq. In order to add machine integers, a first possibility is to extend the theory underlying the Coq logic with:

- one primitive type int;
- the constructors $0, 1, 2, \ldots, 2^n - 1$ of type int;
- the basic primitive functions over the type int such as $+, *, \ldots$;
- the corresponding reduction rules for each primitive function.

It is also necessary to give it an equational theory, for instance, Peano theory together with a lemma stating that $(2^n - 1) + 1 = 0$. However, this approach has some drawbacks:

- It adds a large amount of new constructions to the theory. This goes against de Bruijn's principle which states that keeping the theory and its implementation as small as possible highly contributes to the trust one has in a system. Furthermore, on a more practical side, it will have a deep impact in the implementation, since the terms will have to be extended with new syntactic categories (primitive types and primitive functions).
- It adds a lot of new reductions, not only for ground terms but also for theorems. For example, if we consider the theorem n_plus_zero that states that $\forall n\colon \mathsf{int}.\, n + 0 = n$, it could be convenient to have (n_plus_zero 7) reduces to (refl int 7) where refl represents the reflexivity of equality. It is not clear that way that one captures all the necessary reductions.

For these reasons, we have taken an alternative approach. Efficient evaluation in Coq, as provided by the virtual machine, uses a compilation step. Before evaluating a term, it transforms it into another representation that is more suitable for performing reduction. The idea is to introduce the native machine integers not as part of the theory of Coq but only in this compilation phase. So, the type int is defined using the standard commands as a type with a single constructor that contains n digits:

Definition digit := bool.
Inductive int : Type := In(d_{n-1} ... d_1 d_0 : digit) : int.

For the definition of primitive functions, we do not try to define them directly. We relate the machine numbers int with the relative numbers Z with the two functions $\overline{\bullet} : \mathsf{int} \to \mathsf{Z}$ and its inverse $[\bullet] : \mathsf{Z} \to \mathsf{int}$ and we prove that they satisfy the following two properties:

$$\forall\, i : \mathsf{int}.\ \overline{[i]} = i$$
$$\forall\, z : \mathsf{Z}.\ [z] = z \mod 2^n$$

Now, it is straightforward to define the primitive functions of int as the image of the corresponding function of Z. For example, addition for int is defined as follows:

Definition $i_1 +_{\mathsf{int}} i_2 := [\overline{i_1} +_{\mathsf{Z}} \overline{i_2}]$

It is also straightforward to derive the basic properties of these functions from the properties of the corresponding functions on Z. This set of definitions and properties will let the user manipulate the type int in Coq as any other type. So we preserve the property that everything is defined from base principle.

Now, the trick is to modify the compiler in such a way that it treats the type int as real machine integer. The main difficulty is that Coq requires strong reduction. This is not the case of traditional functional languages where only weak reduction is needed (no reduction under binders). Before explaining our modification to the compiler, we first give an overview of what symbolic weak and strong reductions are and then explain how the compiler works.

2.1 Strong reduction by symbolic weak reduction

In order to compute the strong normal form of a term t or to test the convertibility between two terms t_1 and t_2, the Coq system uses a compiled implementation of the symbolic calculus [7, **?**]. We briefly recall what symbolic computation is starting from the pure λ-calculus extended with inductive types.

Each inductive type is defined by a name I and a fixed number of constructors $|I|$. In this context the constructor In is represented by $C_{\mathsf{int},1}$. The syntax of the λ-calculus is extended with constructors and case analysis:

$$a ::= x \mid \lambda x.a \mid a_1\ a_2 \mid C_{I,i}(\boldsymbol{a}) \mid \mathsf{case}\ a\ \mathsf{of}\ (\boldsymbol{x}_i \Rightarrow a_i)_{1\le i\le |I|}$$

the reduction rules are

$$\begin{array}{rcll} (\lambda x.a_1)a_2 & \Rightarrow & a_1\{x \leftarrow a_2\} & (\beta) \\ \mathsf{case}\ C_{I,j}(\boldsymbol{a})\ \mathsf{of}\ (\boldsymbol{x}_i \Rightarrow a_i)_{1\le i\le |I|} & \Rightarrow & a_j\{\boldsymbol{x}_j \leftarrow \boldsymbol{a}\} & (\iota) \\ \Gamma(a) & \Rightarrow & \Gamma(a') \quad \text{if } a \Rightarrow a' & (\text{context}) \end{array}$$

Where the context Γ allows to reduce every where, in particular under binder like abstraction and inside the branches of a cases. We are interested in computing the strong $\beta\iota$-normal form strongly normalising λ-term a. This will be done by iteration of weak symbolic reduction and *readback*.

We first introduce the symbolic calculus:

$$
\begin{array}{lll}
\text{Extended terms} & b ::= x \mid \lambda x.b \mid b_1\ b_2 \mid C_{I,i}(\boldsymbol{b}) \mid \mathsf{case}\ b\ \mathsf{of}\ (\boldsymbol{x}_i \Rightarrow b_i)_{1\leq i\leq |I|} \mid [k] \\
\text{Accumulators} & k ::= h \mid k\ v \\
\text{Atoms} & h ::= \tilde{x} \mid \mathsf{case}\ k\ \mathsf{of}\ (\boldsymbol{x}_i \Rightarrow b_i)_{1\leq i\leq |I|} \\
\text{Values} & v ::= \lambda x.b \mid C_{I,i}(\boldsymbol{v}) \mid [k]
\end{array}
$$

The value $[h\ v_1 \dots v_n]$ is called an accumulator, it represents h apply to arguments $v_1 \dots v_n$. The atom $\tilde{x}$ is a symbolic variable it represents the free variable x, whereas $\mathsf{case}\ k\ \mathsf{of}\ (\boldsymbol{x}_i \Rightarrow b_i)_{1\leq i\leq |I|}$ represent a suspended cases which can not reduce since its argument does not reduce to a constructor.

The rules for weak reduction are defined as follows:

$$
\begin{array}{rcll}
(\lambda x.b)\ v & \to & b\{x \leftarrow v\} & (\beta_v) \\
[k]\ v & \to & [k\ v] & (\beta_s) \\
\mathsf{case}\ C_{I,j}(\boldsymbol{v})\ \mathsf{of}\ (\boldsymbol{x}_i \Rightarrow b_i)_{1\leq i\leq |I|} & \to & b_j\{\boldsymbol{x}_j \leftarrow \boldsymbol{v}\} & (\iota_v) \\
\mathsf{case}\ [k]\ \mathsf{of}\ (\boldsymbol{x}_i \Rightarrow b_i)_{1\leq i\leq |I|} & \to & [\mathsf{case}\ k\ \mathsf{of}\ (\boldsymbol{x}_i \Rightarrow b_i)_{1\leq i\leq |I|}] & (\iota_s) \\
\Gamma_v(b) & \to & \Gamma_v(b') \quad \text{if } b \to b' & (\text{context}_v)
\end{array}
$$

where $\Gamma_v(\bullet) ::= \bullet\ v \mid b\ \bullet \mid C_{I,i}(\boldsymbol{b}\ \bullet\ \boldsymbol{v}) \mid \mathsf{case}\ \bullet\ \mathsf{of}\ (\boldsymbol{x}_i \Rightarrow b_i)_{1\leq i\leq |I|}$

The rules (β_v) andι_v are the standard call-by-value function application rule and case reduction. The rule (β_s) ("symbolic" β-reduction) handles the case where the function part of an application is not a function: a free variable $[\tilde{x}]$ or an application of a free variable $[\tilde{x}\ v_1 \dots v_n]$ or a suspended case. In that case, the accumulator simply absorbs its argument. The rule (ι_s) explain what to do when an accumulator is argument of a case, we simply *remember* that the case construct can not reduce by producing a new accumulator.

The rule (context_v) enforces weak reduction (no reduction under binder) and a right to left evaluation order (the argument being evaluated before the functional part)[3].

In order to compute the normal form of a λ-term a, we first inject a into the symbolic calculus. This is done by replacing each free variable x of a by its corresponding symbolic value $[\tilde{x}]$. We obtain a closed symbolic term: variable $\tilde{x}$ are symbolic and not subject to substitution. In order to compute the normal form of a closed symbolic term b, we first compute its symbolic head normal form $\mathcal{V}(b)$ (also called value); then we *read back* the resulting value:

$$
\begin{array}{lll}
\mathcal{N}(b) & = \mathcal{R}(\mathcal{V}(b)) & (1) \\
\mathcal{R}(\lambda x.b) & = \lambda y.\mathcal{N}((\lambda x.b)\ [\tilde{y}]) \quad y \text{ fresh} & (2) \\
\mathcal{R}(C_{I,i}(\boldsymbol{v})) & = C_{I,i}(\mathcal{R}(\boldsymbol{v})) & (3) \\
\mathcal{R}([k]) & = \mathcal{R}'(k) & (4) \\
\mathcal{R}'(k\ v) & = \mathcal{R}'(k)\ \mathcal{R}(v) & (5) \\
\mathcal{R}'(\tilde{x}) & = x & (6) \\
\mathcal{R}'(\mathsf{case}\ k\ \mathsf{of}\ (\boldsymbol{x}_i \Rightarrow b_i)_{1\leq i\leq |I|}) & = \mathsf{case}\ \mathcal{R}'(k)\ \mathsf{of}\ (\boldsymbol{x}_i \Rightarrow \mathcal{N}(b\ C_i([\tilde{\boldsymbol{y}_i}])))_{1\leq i\leq |I|} & (7)
\end{array}
$$

where $b = \lambda x.\mathsf{case}\ x\ \mathsf{of}\ (\boldsymbol{x}_i \Rightarrow b_i)_{1\leq i\leq |I|}$
and $\boldsymbol{y}_i$ are sequences of fresh variables with $|\boldsymbol{y}_i| = |\boldsymbol{x}_i|$

[3] The evaluation order is important when using a virtual machine like the ZAM with n-ary applications to execute the symbolic calculus.

The readback function $\mathcal{R}$ is defined recursively. It transforms a value v into a normalised source term. Reading back an atom $\tilde{x}$ (equation 6) simply consists in extracting the variable x. Reading back an accumulator $k\ v$ (equations 5) consists in applying the readback of the functional part to the readback of the argument. The interesting case is for function $\lambda x.b$ (equation 2). It consists in applying the functional value to a value $[\tilde{y}]$ representing a fresh variable. Here, "fresh" means that y is not a free variable of b. Then, we compute the value of the application, which reduces in one step to $b\{x \leftarrow [\tilde{y}]\}$, and reads it back as a normalised term a. The normal form of $\lambda x.b$ is $\lambda y.a$, which is correct up to α-conversion. The same idea is used to obtain the normal form of the branches of a case, and so it normal form.

In [7], the authors prove the following the theorem in the case of the λ-calculus:

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

The normal form of a term can be obtain by recursively compute its symbolic weak normal form and reading back the resulting value. The efficiency of the process depend on the efficiency of the weak evaluation.

2.2 Compiling the symbolic calculus

Weak symbolic reduction can be implemented using a compiler and an abstract machine. The abstract machine we present here is a simplified version of the ZAM []. We write $\hat{v}$ the values manipulated by the abstract machine. They are pointers to heap allocated block $[T : \hat{v}_1, \ldots, \hat{v}_n]$, where T is a tag, and $\hat{v}_i$ are values belonging to the block.

A machine state $(e,\ c,\ s)$ has three components: an environment e that contains a sequence of machine values $\hat{v}_1, \ldots, \hat{v}_n$ (it associates to the variable of de Bruijn index i the value $\hat{v}_i$); a code pointer c that represents the term being executed; stack frame s that contains function arguments, intermediate results and return context $\langle c, e\rangle$. The semantics of instructions and the compilation rules are given Figure 1.

The compilation scheme $[\![b]\!]c$ takes a term b that has to be compiled and a code c that corresponds to the continuation of b. If b is normalising, the execution of $(e,\ [\![b]\!]c,\ s)$ leads to $(e,\ c,\ \hat{v} :: s)$ where $\hat{v}$ is the machine representation of the value v of b where the free variable have been substituted by the value in e.

The evaluation the code corresponding to a function $\lambda x.b$ builds a closure $[T_\lambda : c, e]$ where c is the code pointer corresponding to b and e the current environment. For application, first a return context is pushed on the stack then the argument and the function are evaluated, finally the APPLY instruction start the evaluation of the closure. For constructors, the instruction MAKEBLOCK(n, T) builds a block[4] $[T : \hat{v}_1 \ldots \hat{v}_n]$ which is the machine representation of constructors. The compilation of a case start by PUSHRA, which saves return context

[4] The compilation of a block erase the inductive name in the constructor, this is correct, see [].

$$
\begin{array}{rcl}
[\![x]\!]c & = & \mathsf{ACCESS}(i); c \quad \text{where } i \text{ is the de Bruijn index of } x \\
[\![\lambda x.b]\!]c & = & \mathsf{CLOSURE}(\mathsf{GRAB}; [\![b]\!]\mathsf{RETURN}); c \\
[\![b_1\ b_2]\!]c & = & \mathsf{PUSHRA}(c); [\![b_2]\!][\![b_1]\!]\mathsf{APPLY} \\
[\![C_{I,i}(b_1, \ldots, b_n)]\!]c & = & [\![b_n]\!] \ldots [\![b_1]\!]\mathsf{MAKEBLOCK}(n, i); c \\
[\![\mathsf{case}\ b\ \mathsf{of}\ (\boldsymbol{x}_i \Rightarrow b)_{1 \leq i \leq |I|}]\!]c & = & \\
 & & \mathsf{PUSHRA}(c); [\![b]\!]\mathsf{SWITCH}([\![b_1]\!]\mathsf{RETURN}, \ldots, [\![b_{|I|}]\!]\mathsf{RETURN})
\end{array}
$$

$$
\begin{array}{rcl}
(e,\ \mathsf{ACCESS}(i); c,\ s) & \rightsquigarrow & (e,\ c,\ e[i] :: s) \\
(e,\ \mathsf{CLOSURE}(c_f); c,\ s) & \rightsquigarrow & (e,\ c,\ [T_\lambda : c_f, e] :: s) \\
(e,\ \mathsf{GRAB}; c,\ \hat{v} :: s) & \rightsquigarrow & (\hat{v} :: e,\ c,\ s) \\
(e_f,\ \mathsf{RETURN},\ \hat{v} :: \langle c, e\rangle :: s) & \rightsquigarrow & (e,\ c,\ \hat{v} :: s) \\
(e,\ \mathsf{APPLY}, [T_\lambda : c_f, e] :: s) & \rightsquigarrow & (e_f,\ c_f,\ s) \\
(e,\ \mathsf{PUSHRA}(c_1); c_2,\ s) & \rightsquigarrow & (e,\ c_2,\ \langle c_1, e\rangle :: s) \\
(e,\ \mathsf{MAKEBLOCK}(n, T); c, \hat{v}_1 :: \ldots :: \hat{v}_n :: s) & \rightsquigarrow & (e,\ c,\ [T : \hat{v}_1 :: \ldots :: \hat{v}_n] :: s) \\
(e,\ \mathsf{SWITCH}(c_1, \ldots, c_m),\ [T : \hat{v}_1 :: \ldots :: \hat{v}_n] :: s) & \rightsquigarrow & (\hat{v}_n :: \ldots :: \hat{v}_1 :: e,\ c_T,\ s) \\
(e,\ \mathsf{SWITCH}(c_1, \ldots, c_m),\ [0 : \mathsf{ACCU}, \hat{k}] :: s & \rightsquigarrow & (e, \mathsf{RETURN},\ \hat{v} :: s) \\
\text{where } \hat{v} = [0 : \mathsf{ACCU}, [1 : \hat{k}, [T_\lambda : \mathsf{GRAB}; \mathsf{SWITCH}(c_1, \ldots, c_m)]]] & & \\
(e_f,\ \mathsf{ACCU}, \hat{v}_a :: \langle c,\ e\rangle :: s) & \rightsquigarrow & (e,\ c,\ [0 : \mathsf{ACCU}, e_f :: \hat{v}_a] :: s)
\end{array}
$$

Fig. 1. Compilation rules and semantics of the virtual machine

(used at the end of branches), then the argument is evaluated and the SWITCH instruction jump to the corresponding branches.

What happens for the symbolic calculus? When an APPLY instruction is executed, the top stack value is not necessarily a closure, it can be the machine representation of an accumulator. An accumulator $[k]$ is represented like a closure: $[0 : \mathsf{ACCU}, \hat{k}]$. Furthermore, k being of the form $h\ v_1 \ldots v_n$, $\hat{k}$ is represented as an environment, i.e by the sequence $\hat{h}, \hat{v}_1, \ldots, \hat{v}_n$. The ACCU instruction takes the top value of the stack, pushes it at the end of the environment, rebuilds an accumulate block and returns. In that way, the APPLY instruction does not need to perform an extra test.

For the same reason, when a SWITCH instruction is executed, the top value is not necessarily a constructor, it can be an accumulator. If the tag is 0, the matched value is an accumulator, the SWITCH instruction builds an accumulate block representing the suspended case. In practice, 0 branches are automatically added to cases by the compiler, thus the SWITCH instruction of the ZAM can be used without extra test. Note that for atoms, $\hat{x}$ is represented by the block $[0 : x]$ and $\mathsf{case}\ k\ \mathsf{of}\ (\boldsymbol{x}_i \Rightarrow b)_{1 \leq i \leq |I|}$ is represented by $[1 : \hat{k}, [T_\lambda : c, e]]$ where $\hat{k}$ is the machine representation of k, c and e are the code and the environment for the function $\lambda x.\mathsf{case}\ x\ \mathsf{of}\ (\boldsymbol{x}_i \Rightarrow b)_{1 \leq i \leq |I|}$.

In order to normalise a λ-term a using the virtual machine, we first compute $c = [\![a]\!]$ and start the evaluation of abstract machine in the state $(e,\ c,\ \emptyset)$, where e is an environment associating to each free variable x of a its value $[\tilde{x}]$ encoded by the heap block $[0 : \mathsf{ACCU}, [0 : \tilde{x}]]$. When the machine stops, we obtain a value v on the top of the stack. The readback function analyses which kind of value it

is. It can either be a closure, a constructor, or an accumulator. This can be done by a simple inspection of the tag. If the tag is T_λ we have to normalise $v\ [\tilde{y}]$, this is done by simply restarting the machine in the state $(e, \mathsf{APPLY}, v :: [\tilde{y}] :: \emptyset)$. The same technique is used to normalise the branches of a suspended case.

2.3 Adding machine integers

We are now ready to explain how we can take advantage of the compilation mechanism to boost the evaluation of a λ-calculus extended with inductive types using the machine-integer operations. Of course, the gain will only be effective for programs using the previously defined inductive type int.

We extend the λ-calculus with a global environment Δ associating global variables g to their definition (λ-term): $a ::= \ldots \mid g$ and $\Delta ::= \emptyset \mid \Delta :: (g, a)$. The reduction rules now depend on the global environment Δ and are extended with one rule for the reduction of global definitions: $g \quad \rightarrow \quad \Delta(g)$.

We assume that n has been chosen in such a way that the term $\mathsf{In}(d_{n-1}, \ldots, d_0)$ is isomorphic to the machine word $d_{n-1} \ldots d_0$ if the d_j are all constructors (true stands for the machine digit 1 and false for 0). In the following, the term $\mathsf{In}(d_{n-1}, \ldots, d_0)$ is written p if all the d_j are constructors. We write $\dot{p}$ for the machine representation of p. If m is a machine integer, we write $|m|$ its representation as a term of type int; we have $p = |\dot{p}|$ and $m = |\dot{m}|$.

In the following, we assume that we have a global definition $+$ performing the addition of two int. We denote by $+_a$ its associated definition and we write $+_M$ the processor addition. We assume that $+$ does what it is supposed to do, i.e.:

$$p_1 + p_2 \quad \Rightarrow^* \quad |\dot{p}_1 +_M \dot{p}_1|$$

This gives us a first way to boost the reduction of $+$ when the two arguments are of the form p_1 and p_2; instead of accessing to the global definition $+_a$ of $+$ and then reducing the application $+_a\ p_1\ p_2$, one can directly compute $|\dot{p}_1 +_M \dot{p}_2|$. This solution does not work so well when additions are nested. For example, during the reduction of $(p_1 + p_2) + p_3$, the machine word $\dot{p}_1 +_M \dot{p}_2$ will be injected into its constructor representation at the end of the evaluation of $p_1 + p_2$ and then immediately re-injected into a machine word to perform the second addition. We have chosen a different solution that overcomes this problem.

We extend the symbolic calculus with machine integers and their primitive operations. The idea is to try to maintain as long as possible the terms of type int in their machine representation. The syntax of the new symbolic calculus is extended with machine integers:

$$\begin{aligned} b &::= \ldots \mid m \\ v &::= \lambda x.b \mid [k] \mid C_{I,i}(v_1, \ldots, v_n) \mid m \end{aligned}$$

In the definition of value we reject the case $\mathsf{In}(v_1, \ldots, v_n)$ where the v_i are all true or false. New reduction rules are added to the calculus. First, a special case is added for the constructor In:

$$p \quad \rightarrow \quad \dot{p}$$

In other word, when a constructor of type int can be represented by a machine word, its value is the machine representation. Second, for each global definition representing a primitive operation over int, some special rules are added. Let us consider addition, we add two rules:

$$\begin{array}{rcl} m_1 + m_2 & \to & m_1 +_M m_2 \\ v_1 + v_2 & \to & \Delta_b(+)\ v_1\ v_2 \end{array}$$

The first rule applies when the two arguments of + are in machine representation. The result is given by the machine addition. The second one applies when one of the two arguments is not in machine representation. It can either be an accumulator or an In constructor with one of its arguments being an accumulator. In that case, the usual rule for global variable is used: the variable is replaced by is associating value in Δ_b.

Pattern matching on term of type int has also to be taken care of. The matched value can be a machine word whereas a constructor value or an accumulator is expected. For this reason, we add the rule:

$$\mathsf{case}\ m\ \mathsf{of}\ (\boldsymbol{x}_i \Rightarrow b)_{1\leq i\leq |I|} \quad \to \quad \mathsf{case}\ |m|\ \mathsf{of}\ (\boldsymbol{x}_i \Rightarrow b)_{1\leq i\leq |I|}$$

Finally the readback function needs only to be extended so to get rid of machine integers: $\mathcal{R}(m) = |m|$.

Theorem 1 can be extended to this new symbolic calculus:

Theorem 2. *If for all global definitions g with a special shortcut we have* $\Delta_b(g)\ \boldsymbol{m} \to^* g_M\ \boldsymbol{m}$. *For all closed term a, well typed and strongly normalising, then* $\mathcal{N}(a)$ *is defined and is then normal form of a.*

What remains to be modified is the virtual machine and the compilation scheme. Previously, the values of the virtual machine were only pointers to heap-allocated blocks. The values are now extended with machine integers. Two instructions TOINT and OFINT are added to the virtual machine. Their semantics is given by:

$$\begin{array}{rcll} (e, \mathsf{OFINT}; c, d_0 :: \ldots :: d_{n-1} :: s) & \rightsquigarrow & (e, c, m :: s)\ m = d_0 \ldots d_{n-1} & (1) \\ (e, \mathsf{OFINT}; c, v_0 :: \ldots :: v_{n-1} :: s) & \rightsquigarrow & (e, c, v :: s)\ \text{otherwise} & (2) \\ \text{where } v = [1 : v_0, \ldots, v_{n-1}] & & & \\ (e, \mathsf{TOINT}; c, m :: s) & \rightsquigarrow & (e, c, v :: s) & (3) \\ \text{where } v = [1 : d_0, \ldots, d_{n-1}] & & & \\ (e, \mathsf{TOINT}; c, v :: s) & \rightsquigarrow & (e, c, v :: s)\ \text{otherwise} & (4) \end{array}$$

We also add one instruction for each primitive operations. For example, the instruction ADD corresponds to the addition:

$$\begin{array}{rcll} (e, \mathsf{ADD}; c, m_1 :: m_2 :: s) & \rightsquigarrow & (e, c, m_1 +_M m_2 :: s) & (5) \\ (e, \mathsf{ADD}; c, v_1 :: v_2 :: s) & \rightsquigarrow & (e_+, c_+, v_1 :: v_2 :: \langle c, e\rangle :: s)\ \text{otherwise} & (6) \\ \text{where } \Delta_b(+) = [T_\lambda : c_+, e_+] & & & \end{array}$$

Finally, we modify the compiler with special cases for the compilation of the In constructor, for the primitive operations and for the pattern matching over elements of type int:

$$\begin{aligned}
[\![\mathsf{In}(b_0,\ldots,b_{n-1})]\!]c &= [\![b_{n-1}]\!]\ldots[\![b_0]\!]\mathsf{OFINT};c \\
[\![b_1+b_2]\!]c &= [\![b_2]\!][\![b_1]\!]\mathsf{ADD};c \\
[\![\mathsf{case}\ b\ \mathsf{of}\ (\boldsymbol{x}_i \Rightarrow b)_{1\le i\le |I|}]\!]c &= \\
\mathsf{PUSHRA}(c);[\![b]\!]\mathsf{TOINT};&\mathsf{SWITCH}([\![b_1]\!]\mathsf{RETURN},\ldots,[\![b_n]\!]\mathsf{RETURN}) \\
\text{if type of } b = \mathsf{int}&
\end{aligned}$$

The compilation of the In constructor generates an OFINT as last instruction and not a MAKEBLOCK as for the other constructors. The OFINT instruction checks if the n first arguments on the stack correspond to machine representation of digits (i.e a block representing the constructors true or false). If all the arguments are constructors, the instruction builds the corresponding machine word (this corresponds to the reduction rule $p \to \dot{p}$ at the symbolic level). If one of the argument is not a constructor, the instruction is equivalent to MAKEBLOCK$(n, 1)$.

The compilation of a + first evaluates its arguments, then the ADD checks if they are machine words. If it is the case, the instruction simply performs the addition. If not, the instruction gets the value $[T_\lambda : c_+, e_+]$ of the +, inserts a return context $\langle c, e\rangle$ and performs an APPLY.

The compilation of a pattern matching on an object of type int is also modified. A TOINT instruction is inserted just before the SWITCH. If the top value of the stack is a machine word, the TOINT instruction replaces it by its corresponding block representation. If not, the TOINT instruction does nothing. The semantics of the SWITCH does not need to be modified.

The readback function should be able to analyse the value it gets. Before adding machine integer, this was done by matching the tag of the heap block. Remember that a heap block is a pointer, i.e a machine integer. The problem is how to differentiate between pointers and integers. Note that the problem is the same for the implementation of the new instructions, which have to test if some values are blocks or integers. There is an easy solution. Since the implementation of the Coq virtual machine is based on the one of OCaml. The OCaml garbage collector makes the difference between a machine word representing a pointer and a machine word representing an integer using the following convention: a pointers is machine word with weak bit set to 0, integer is a machine word with weak bit set to 1. So, an integer p is encoded by the machine word $2p + 1$, this is why, for example, OCaml and now Coq have integers of only 31-bits on a 32-bits architecture.

2.4 Iterators

We have added to Coq the usual machine integer primitives. This leads to an efficient evaluation for programs based on those primitives. There still a problem to define functions recurring on an int. The Coqsystem ensures the termination of recursive functions, thus the following fold function, computing $F\ n_s\ (F\ n_{s+1}\ (\ldots(F\ n_e\ a)\ldots))$, is not accepted:

```
Definition foldi (A:Type) (F:A->A) (a:A) (n_s n_e:int) :=
  if n_start <= n_end then
     (fix aux (i:int) (ai:A) {
          if i = n_s then F i ai
          else aux (i-1) (F i ai)
      }) n_e a
  else a.
```

To do this we have added an inefficient version of `foldi` recurring on a `nat` (the Coq representation of Peano numbers). And a special case in the compiler for this function. The evaluation leads to the evaluation of the not accepted definition.

3 Extending Coq with persistent arrays

Arrays are among the most important data-structure in programming. Unfortunately, logics like the one of Coq are stateless. So, it is impossible to deal directly with destructive arrays as the ones we find in mainstream programming languages. The work-around is usually to use some flavour of purely functional arrays [9]. This works pretty well when arrays are rather small. For larger ones, not having an $O(1)$ access to elements of the array becomes quickly unmanageable.

In order to introduce destructive arrays, one way to go is to add states to the logic. Monads [13] are the standard way to do this. Unfortunately, monads are quite difficult to manage in a prover without developing some infrastructure (see [4, 11] for example). An alternative approach is to develop some kind of program analysis that is capable of discovering (automatically or semi-automatically) that it can safely use destructive arrays instead of functional ones (see [10] for example). If one wants this technique to be applicable to a large set of programs, such analysis is usually rather complex. Here, we are going to follow a third approach and use destructive arrays but with a functional interface. These arrays are called persistent arrays [1]. For Coq, persistent arrays are implemented in a very naive way. An array is simply composed of the list of elements of the array and a default value.

$$\mathsf{Inductive\ array}\ (A : \mathsf{Type}) : \mathsf{Type}\ := \mathsf{mkArray}(\mathsf{elems} : \mathsf{list}\ A)(\mathsf{default} : A).$$

As there is no exception in Coq, the default value is mainly used as a return value when accessing outside the range of the array. The two basic operations on arrays are then defined in a straightforward manner:

$$\begin{array}{l}\mathsf{Definition\ get}\ (A : \mathsf{Type})(t : \mathsf{array}\ A)(n : \mathsf{int}) := \mathsf{get\_elem}\ (\mathsf{default}\ t)\ (\mathsf{elems}\ t)\ n.\\ \mathsf{Definition\ set}\ (A : \mathsf{Type})(t : \mathsf{array}\ A)(n : \mathsf{int})(a : A) :=\\ \quad \mathsf{mkArray}\ (\mathsf{upd\_elem}\ (\mathsf{elems}\ t)\ n\ a)\ (\mathsf{default}\ t).\end{array}$$

where ($\mathsf{get\_elem}\ d\ l\ n$) returns the $n+1$-st element of the list l if n is less than the size of the list, d otherwise; and ($\mathsf{upd\_elem}\ l\ n\ a$) returns l where the

$n+1$-st element has been replaced by a if n is less than the size of the list, l otherwise. Both definitions are very inefficient. The access is linear in the number of elements and the update is also linear and furthermore reallocates a large part of the list.

Now, the virtual machine is going to conform to this functional behaviour but using destructive arrays. The idea is quite simple. Among all the versions of the array that may co-exist during execution, the last one (the newest one) is privileged and is represented by a destructive array. Look-ups and updates applied to this last version are then very efficient. Older versions of the array are not destructive arrays but point to the last version through a list of indirections. These indirections explain which modifications have to be applied in order to retrieve the values of the old array from the last version. Look-ups and updates of old versions applied to old versions are possible (this is a requirement of the functional interface) but rather slow (linear to the number of updates). For the implementation, we have directly adapted the OCaml code proposed by J.C. Filliâtre in his paper [5]. Persistent arrays are defined as follows:

```
type 'a parray_kind =
 |  Array of 'a array
 |  Updated of int * 'a * 'a parray
and 'a parray = ('a parray_kind) ref
```

A persistent array is a reference on a `parray_king` which is either a destructive array (`Array`) or an indirection `Updated`(i,v,t) indicating that the persistent array is t except that at position i the value is v. The OCaml implementation does not contain explicitly the default value. It is stored in the last position of the array.

For the get function, we look directly in the array or follow the indirections:

```
let rec get p n =
  match !p with
  | Array t ->
     let l =  Array.length t in
     if 0 <= n && n < l then Array.get t n else Array.get t (l-1)
  | Updated (k,e,p) -> if n = k then e else get p n
```

Note that in the path to the destructive array, there could be several occurrences of `n` (this location could have been updated several times) but we stop at the first one. For the `set` function, an indirection is added at the right position:

```
let set p n e =
  let kind = !p in
  match kind with
  | Array t ->
     if 0 <= n && n < Array.length t - 1 then
        let res = ref kind in
        p := Updated (n, Array.get t n, res);
        Array.set t n e; res
```

```
    else p
| Updated _ ->
    if 0<= n && n < length p then ref (Updated(n, e, p))
    else p
```

Note that if we are updating outside the array, everything is left unchanged.

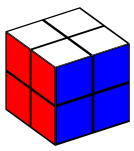
To link this implementation with the virtual machine of Coq we need two translation functions allowing to transform a Coq representation of array into its virtual machine representation and vise-versa. This two functions are trivial to implement[5].

The compilation of array primitives like get is a little bit different than the one of integers. This is due to the implicit polymorphism of the virtual machine implementation. For the virtual machine the `get` operation expects only two arguments, whereas the Coq version expects three arguments (the type A of the elements). So, the compilation scheme first evaluates the two last arguments, then the `get` checks if they are in machine representation. If not the argument A is evaluated and the three arguments are applied to the Coq implementation of get.

Finally we have added to primitive copy and reroot, the functional interface is just the identity but they can be very useful when writing program with persistent arrays. BE MORE EXPLICIT

4 First application: the Mini-Rubik

The Mini-Rubik is the pocket version of the famous Rubik cube. It is composed of 8 small cubes only:



This cube has 3,674,160 configurations. It is then quite easy to explore them completely with computers. Here, we explain how the property that the Mini-Rubik is always solvable in less than 11 moves has been proved formally.

First, we need to give a model. For this, we use indexation and associate to each configure of the Mini-Rubik a unique number from 1 to 3,674,160. This is done by taking one small cube as reference and considering that a configuration is composed of a permutation of the other 7 small cubes with for each of them a possible change of orientation. Ranking the permutation gives us a number

[5] OCaml arrays are limited in size, if the size of the Coq array is greater than the maximum OCaml size then the virtual machine use the inefficient Coq representation.

between 1 and 7!. A small cube has 3 coloured faces, so it has 3 possible orientations. Not all orientations are possible. An invariant of the Mini-Rubik is that the orientation of the last small cube can be deduced from the orientations of the other small cubes. This gives us 3^6 orientations. Our indexation is given by combining these two numbers ($3,674,160 = 7! \times 3^6$).

Second, we have to construct a reachability graph – a Cayley graph, using the terminology of group theory. As we are capable of indexing configuration, this is easy. In order to represent a set of configurations, we use an array of 3,674,160 booleans. We prove that it is solvable in 11 moves using an iterative process and two sets of configurations S_A and S_N. The first set S_A contains the configurations that have been reached so far. The second set S_N contains the new configurations that have been reached by the previous iteration. Initially these two set only contain the initial configuration. At each iteration, S_A and S_N are updated with all the configurations that can be reached in one move by one configuration in S_N. After 12 iterations, S_N should be empty.

This application is perfect for testing our extension. First, as $3,674,160 < 2^{31} - 1$, a configuration can be represented by a single native integer. Second, if it is not possible to allocate in Coq an array of 3,674,160 booleans (booleans are defined as an inductive type with two constructors, so one boolean takes one word in memory), it is easy to use binary encoding and use an array of machine integer of length $118522 = 3,674,160/31+1$. Furthermore, from a given configuration, there are 9 configurations reachable in one move. So this means that, in the iterations the look-ups in the array S_A will be 9 times more frequent than the updates. This is perfect since our look-ups cost much less than our updates. In [12], we have already presented a formal proof of the Mini-Rubik. In this version, machine integers were available but not efficient arrays. The arrays were implemented using an *ad hoc* functional data-structure. Checking the proof that the Mini-Rubik is solvable in 11 moves took 4 minutes. Modified with our new arrays, not only did it reduce to 10 seconds, but it also greatly simplified the implementation.

5 Second application: verifying SAT traces

The most efficient SAT solvers that are used to prove the unsatisfiability of booleans formulas are all based on the DPLL algorithm with learning (see [**?**] for a complete introduction). An interesting feature of this algorithm is that very little overhead is needed in order to generate a trace that explains why the formula is unsatisfiable. Format for traces may slightly vary from one solver to the other but they are all based on the simple resolution rule:

$$\frac{\neg x \vee C \qquad x \vee C'}{C \vee C'}$$

The variable x is called the resolution variable, C and C' are clauses, i.e disjunctions of literals. The reflexive method to prove the unsatisfiability of boolean formulas in Coq works as follows:

- The initial problem is turned by some CNF transformation into a list of clauses. These clauses are called the roots.
- The sat solver is called and returns the trace. This trace is composed of a list of resolution chains. Each chain corresponds to a clause that has been learned by the algorithm, so it is a logical consequence of the roots.
- A program written in Coq checks that the trace is correct: it build the clauses that correspond to the resolution chains and finally checks that the last clause is the empty clause, i.e. $\bot$ is a consequence of the roots.

From the implementation point of view, roots are represented by a list of lists of natural numbers. Each boolean variable x has a unique number n. The literal x is represented as $2n$, $\neg x$ as $2n+1$. The trace is also represented as a list of lists of natural numbers. Each number represents an index of a clause. The indexes are computed with the following convention: roots come first then the clauses built by resolutions. A resolution chain is a list of natural numbers $\{n_1, n_1, \ldots, n_k\}$. In order to build the resulting clause, it is traversed from left to right, C_{n_1} is resolved with C_{n_2}, the result is then resolved with C_{n_3} and so on.

For the implementation of the checker, we have directly translated the C code of zVerify, the checker of zChaff [**?**] into our functional setting. The checker represents ??? lines of code and its correctness proof is ??? line long. We use native integers to represent literals and indexes. Arrays are used for the set of clauses and for a temporary cache to compute the result of a resolution chain. In order to tackle large examples, we had to take a special care in memory usage. For this reason, one preprocessing is performed on traces for garbage collecting: we track when a clause is not used anymore, its index can then be reallocated. Traces processed by the checker are then list of tagged lists of natural numbers. The tag indicates if the new resulting clauses has to be appended or reallocated in that case it contains the index of substituted clause.

To end this section, Figure **??** presents some benchmarks performed on different versions of the checker. The machine used for these benchmarks is a ??? with ?? Gigabytes of memory ??.

– if x appears in C_1, then $[C_1, C_2] = [C_1, C_2']$
– if $\neg x$ appears in C_1, with $C_1 = \neg x \vee C_1'$, then $[C_1, C_2] = C_1' \vee C_2'$
– if neither x nor $\neg x$ appears in C_1, then $[C_1, C_2] = [x \vee C_1, C_2']$

Thus, we can encode C_2 as the list $x :: C_2'$ and build a recursive function on this list. Now we need an efficient way to represent C_1 so we can quickly check the occurrence of a given variable x. Once more, a good solution is to use an array, with size equal to the number of variables. The row x of the array will contain $\top$ if x occurs in C_1, *bot* if $\neg x$ occurs in C_1, and *undef* otherwise. With this representation, we can also add or remove a literal to C_1 in constant time. We can now efficiently compute the result of a resolution chain $C_{i_1}, \ldots, C_{i_k}$:

– take C_{i_1} as a list and store it as an array R
– iteratively update R to $[R, C_{i_l}]$ for $l = 1, \ldots, k$ using the recursive function C_{i_l}

– get the final result in R, convert it as a list and add it to the set of clauses ϕ'

Note that for each resolution chain, whatever the number of resolution is, we only need to remember the last derived clause R for each step. Thus, we can work in place with a single result array. These representation and algorithm were introduced by zChaff trace checker : zVerify [**?**].

6 Conclusion

In this paper, we have presented how Coq can be extended with some imperative features. We believe that what we have proposed here is a very good compromise between the impact the extension has on the architecture of the prover and the benefit in term of speed-up in proof checking. Our changes are localised to the abstract machine and its compiler. We didn't have to change any other part of the prover. We have also developed a systematic and simple methodology to add efficient data-structures with a functional interface to the abstract machine and its compiler. This contributes to the trust one can put in this extension. The methodology has been used to integrate machine integers and persistent arrays.

Some kind of destructive arrays are available in provers like ACL2 [3] Isabelle [4] or PVS [10], but some of these techniques are difficult to apply directly to a prover with a rich logic such as Coq and anyway all of them would require a major modification in the architecture of the prover. To our knowledge, the idea of using persistent arrays inside a prover is new. If it does not provide the full power of destructive arrays, for large applications, it gives a clear speed-up with respect to functional arrays. The loss in efficiency with respect to destructive arrays is largely compensated by the fact that we remain in the confortable setting of functional behaviour.

Our overall goal is not to have Coq evaluation to compete with mainstream programming languages. It is more to have a reasonable computing power within the prover. For example, being able to check the property of the Mini-Rubik in 4 minutes was sufficient enough. The sat example is more interesting. We manage to get within Coq what was done with extraction in [**?**]. Without our extension, it would have been impossible to handle large examples. The fact that we could very quickly be competitive with what was achieved by finely-tuned proof reconstruction [14] in HOL and Isabelle is clearly good news. It opens new perspectives for the use of reflexive methods inside Coq. Finally, if our initial motivation was efficiency, memory usage has revealed to be sometimes an even more crucial limiting factor. Our machine integers and persistent arrays are much more compact than their corresponding functional representations – or their traditional encoding. Unfortunately, and this is maybe the only drawback of having this light integration, these objects only exist within the abstract machine. In particular, they cannot be stored in proof objects, therefore have no impact on the size of compiled file. For this reason, we had to develop an *ad hoc* inductive type in order to store efficiently the proof traces generated by the SAT solver.

References

1. Henry G. Baker. Shallow binding makes functional arrays fast. *ACM SIGPLAN notices*, 26:1991–145, 1991.
2. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.
3. Robert S. Boyer and J Strother Moore. Single-threaded objects in ACL2. In *PADL'2001*, volume 2257, pages 9–27, 2001.
4. Lukas Bulwahn, Alexander Krauss, Florian Haftmann, Levent Erkök, and John Matthews. Imperative functional programming with isabelle/hol. In *TPHOLs'08*, volume 5170 of *LNCS*, pages 134–149, 2008.
5. Sylvain Conchon and Jean-Christophe Filliâtre. A persistent union-find data structure. In *ACM Workshop on ML*, pages 37–46, 2007.
6. Georges Gonthier. Formal proof – the four-color theorem. *Notices of the AMS, 55(11)*, 2008.
7. Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In *ICFP*, pages 235–246, 2002.
8. Xavier Leroy. Objective Caml. Available at `http://ocaml.inria.fr/`, 1997.
9. Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.
10. Natarajan Shankar. Static analysis for safe destructive updates in a functional language. In *LOPSTR*, volume 2372 of *LNCS*, pages 1–24. Springer, 2001.
11. Wouter Swierstra. A hoare logic for the state monad. In *TPHOLs'09*, volume 5674 of *LNCS*, pages 440–451. Springer, 2009.
12. Laurent Théry. Proof Pearl: Revisiting the Mini-Rubik in Coq. In *TPHOLS'08*, volume 5170 of *LNCS*, pages 310–319, 2008.
13. Philip Wadler. Monads for functional programming. In *Advanced Functional Programming*, volume 925 of *LNCS*, pages 24–52. Springer, 1995.
14. Tjark Weber and Hasan Amjad. Efficiently checking propositional refutations in hol theorem provers. *J. Applied Logic*, 7(1):26–40, 2009.