Experiments with Finite Tree Automata in Coq*

Xavier Rival[1,2] Jean Goubault-Larrecq[1,3]

[1] GIE Dyade, INRIA Rocquencourt
Domaine de Voluceau B.P. 105
78153 Le Chesnay Cedex
France

[2] Ecole Normale Supérieure
45, rue d'Ulm
75005 Paris
France

[3] LSV, ENS Cachan
61, av. du président-Wilson
94235 Cachan Cedex
France

Abstract. Tree automata are a fundamental tool in computer science. We report on experiments to integrate tree automata in Coq using shallow and deep reflection techniques. While shallow reflection seems more natural in this context, it turns out to give disappointing results. Deep reflection is more difficult to apply, but is more promising.

1 Introduction

Finite-state automata, and more generally tree automata [7], are of fundamental importance in computer science [12], and in particular in hardware or software verification. One particular domain of interest to the authors is cryptographic protocol verification, where tree automata [13] and slight extensions of them [9] have been used with success. In this domain in particular—but this is already true to some extent in verification in general—it is important to *document* the process by which verification is deemed to be complete, as well as to be able to *convince* any third party that the result of verification is correct. Both goals can be achieved by producing a formal proof in some trusted proof assistant. Our aim in this paper is to report on experience we gained in producing formal proofs of correctness of computations on tree automata in Coq [1]. As we shall see, there are many possible approaches to this apparently simple problem, and several unexpected pitfalls to each.

We survey related work in Section 2, recall the features of Coq that we shall use in Section 3, and give a short introduction to finite tree automata in Section 4. We study the most promising technique to check tree automata computations done by an external tool in Coq, namely *shallow reflection*, in Section 5. Rather surprisingly, this yields disastrous performance. We discuss reasons for this failure at the end of this section. This prompts us to investigate *deep reflection* in Section 6: in deep reflection, the algorithms themselves are coded, proved, and run in the target proof assistant. While this is a complex task to engage in, and seems to provide little return on investments, it is worthwhile to investigate, considering the results of Section 5. We shall see in Section 6 that this actually yields encouraging results. We conclude in Section 7.

* This work was done as part of Dyade, a common venture between Bull S.A. and INRIA.

2 Related Work

There does not seem to have been any work yet on integrating automata-theoretic techniques with proof assistants. In general, however, augmenting the capabilites of proof assistants by external tools is now standard. This is *shallow reflection*, where the external tool constructs a trace of its verification, which can then be checked by the machinery of the given proof assistant. For instance, this is how Harrison [10, 11] integrates computations done by an external binary decision diagram package, resp. an implementation of Stålmarck's algorithm into HOL. Similarly, the model-checker of Yu and Luo [17] outputs a proof that can be checked by Lego.

Shallow reflection has many advantages. First, it is a safe way to extend the capabilities of proof assistants in practice, as the core of the proof assistant is not modified or enriched in any way: the proof assistant still only trusts arguments it (re)checks. Second, the external tool can be swapped for another, provided the new one is instrumented to output traces that the proof assistant can still check. This is important for maintenance, where tools can be optimized or upgraded easily enough, with only an instrumentation effort. Third, it is much easier to check a series of computations done by an external algorithm A that to prove that A itself is correct.

However, some applications seem to require the latter, where algorithm A itself is coded, proved and run inside the proof assistant. This approach is called *deep reflection*, and was pioneered in [16, 3]. In Coq, one first application is Boutin's `Ring` tactic [2] deciding equalities in the theory of rings, mixing shallow and deep reflection. One of the largest deep reflection endeavours is certainly the work by Verma *et al.* [15], where binary decision diagrams are integrated in Coq through total reflection. In the last two cases, total reflection was called for, as traces produced by external tools would have grown far too much to be usable in practice: this is the case for binary decision diagrams, as Harrison shows [10]. Total reflection can roughly be thought as "replacing proofs in the logic by computations", and will be described in more detail in Section 3.

3 A Short Tour of Coq

Coq is a proof assistant based on the Calculus of Inductive Constructions (CIC), a type theory that is powerful enough to formalize most of mathematics. It properly includes higher-order intuitionistic logic, augmented with definitional mechanisms for inductively defined types, sets, propositions and relations. CIC is also a typed λ-calculus, and can therefore also be used as a programming language. We describe here the main features of Coq that we shall need later.

Amongst the *sorts* of CIC are `Prop`, the sort of all propositions, and `Set`, the sort of all specifications, programs and data types. What it means for `Prop` to be the sort of propositions is that any object F : `Prop` (read: F of type `Prop`) denotes a proposition, i.e., a formula. In turn, any object $\pi : F$ is a *proof* π of F. A formula is considered proved in Coq whenever we have succeeded to find a proof of it. Proofs are written, at least internally, as λ-terms, but we shall be content to know that we can produce them with the help of *tactics*, allowing one to write proofs by reducing the goal to subgoals, and eventually to immediate, basic inferences.

Similarly, any object $t : \mathtt{Set}$ denotes a data type, like the data type $\mathbb{N}$ of natural numbers, or the type $\mathbb{N} \to \mathbb{N}$ of all functions from naturals to naturals. Data types can, and will usually be, defined inductively. For instance, the standard definition of $\mathbb{N}$ in Coq is by giving its two constructors $\mathtt{0} : \mathbb{N}$ and $\mathtt{S} : \mathbb{N} \to \mathbb{N}$; that is, as the smallest containing $\mathtt{0} : \mathbb{N}$ and such that $\mathtt{S}(n) : \mathbb{N}$ for any $n : \mathbb{N}$. In particular, every natural number must be of the form $\mathtt{S}(\ldots \mathtt{S}(\mathtt{0}) \ldots)$, which is expressed by Peano's induction principle.

If $t : \mathtt{Set}$, and $p : t$, we say that p is a *program* of type t. Programs in Coq are purely functional: the language of programs is based on a λ-calculus with variables, application pq of p to q, abstraction $[x : T]p$ (the function mapping each x of type T to p), case splits (e.g., $\mathtt{Cases}\ n\ \mathtt{of}\ \mathtt{0} \Rightarrow v \mid \mathtt{S}(m) \Rightarrow fm\ \mathtt{end}$ either returns v if $n = \mathtt{0}$, or fm if n is of the form $\mathtt{S}(m)$), and functions defined by structural recursion on their last argument. For soundness reasons, Coq refuses to accept any non-terminating function. In fact, Coq refuses to acknowledge any definition of a recursive function that is not primitive recursive, even though its termination might be obvious.

In general, propositions and programs can be mixed. We shall exploit this in a very limited form: when π and π' are programs (a.k.a., descriptions of data), then $\pi = \pi'$ is a proposition, expressing that π and π' have the same value. Deep reflection can then be implemented as follows: given a property $P : T \to \mathtt{Prop}$, write a program $\pi : T \to \mathtt{bool}$ deciding P (where $\mathtt{bool}$ is the inductive type of Booleans). Then prove the correctness lemma $\forall x : T \cdot \pi(x) = \mathtt{true} \to P(x)$. To show a theorem of the form $P(a)$, apply the correctness lemma: it remains to prove that $\pi(a) = \mathtt{true}$. If this is indeed so, one call to the `Reflexivity` tactic will verify that the left-hand side indeed reduces to `true`, by *computing* with program π. At this point, the proof of $P(a)$ will be complete.

Not only data structures, but also predicates can be defined by induction. This corresponds to the mathematical practice of defining the smallest set or relation such that a number of conditions are satisfied. For instance, we may define the binary relation $\leq$ on natural numbers as having constructors $\mathtt{le\_n} : \forall n : \mathbb{N} \cdot n \leq n$ and $\mathtt{le\_S} : \forall n, m : \mathbb{N} \cdot n \leq m \to n \leq \mathtt{S}(m)$. This defines it as the smallest binary relation such that $n \leq n$ for all $n \in \mathbb{N}$ (the `le_n` clause) and such that whenever $n \leq m$, then $n \leq m + 1$ (the `le_S` clause). Note that the constructors `le_n` and `le_S` play the role of clause names, whose types are Horn clauses (In Prolog notation, they would be $n \leq n$ and $n \leq \mathtt{S}(m) \leftarrow n \leq m$.). It is in fact useful to think of inductively defined predicates as sets of Horn clauses, a.k.a. pure Prolog programs.

4 Tree Automata

A *signature* Σ is a collection of so-called *function symbols* f, together with natural numbers n called *arities*. In this paper, we shall always assume that Σ is finite. The set $T(\Sigma, X)$ of first-order *terms*, a.k.a. *trees*, over the signature Σ with variables in X is defined inductively by: $x \in T(\Sigma, X)$ for every $x \in X$, and $f(t_1, \ldots, t_n)$ for every n-ary function symbol f and every $t_1, \ldots, t_n \in T(\Sigma, X)$. Elements of $T(\Sigma, \emptyset)$ are called *ground terms* over Σ. Note that there is a ground term over Σ if and only if Σ

contains at least one constant, i.e., one 0-ary function symbol. In the sequel, we shall assume Σ fixed.

Tree automata are finite data structures that represent possibly infinite sets of terms, the so-called *regular tree languages*. (As in [7], we only consider languages of *finite terms*. For more information on regular infinite term languages, see [14].) Tree automata generalize ordinary (word) finite-state automata, where the word $a_1 a_2 \ldots a_n$ is represented as the linear term $a_1(a_2(\ldots(a_n(\epsilon))\ldots))$, where a_1, a_2, …, a_n are unary (arity 1) function symbols and ϵ is a distinguished end-of-word constant.

There are many ways in which tree automata, not to mention regular tree languages, can be described [7]. While they are all equivalent in theory, the precise choice of data structure weighs heavily on algorithms working on them. We define:

Definition 1 (Tree Automata). *A* tree automaton $\mathcal{A}$ *is a tuple* (Q, F, Δ)*, where* Q *is a finite set of so-called* states q*,* $F \subseteq Q$ *is the subset of* final states*, and* Δ *is a finite set of* transitions $f(q_1, \ldots, q_n) \to q$*, where* f *is an* n*-ary function symbol, and* q_1*, …,* q_n*,* q *are states in* Q*.*

A ground term t *is* recognized *at* q *in* $\mathcal{A}$ *if and only if the judgment* $\mathrm{Rec}(q, t)$ *is derivable in the system whose rules are:*

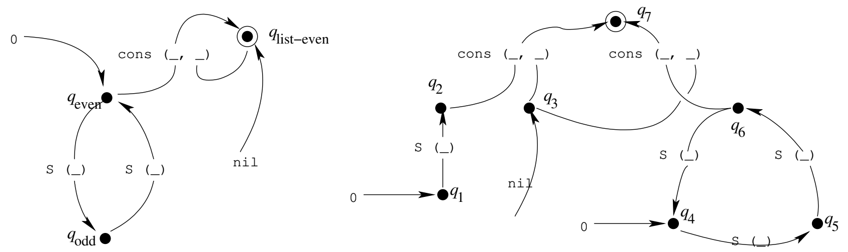
$$\frac{\mathrm{Rec}(q_1, t_1) \quad \ldots \quad \mathrm{Rec}(q_n, t_n)}{\mathrm{Rec}(q, f(t_1, \ldots, t_n))}\ (\mathrm{Rec})$$

where $f(q_1, \ldots, q_n) \to q$ *ranges over* Δ*. The term* t *is* recognized *by* $\mathcal{A}$ *if and only if it is recognized at some final state (in* F*).*

It might seem curious that tree automata do not have any *initial* states. Their roles are taken up by nullary transitions. Consider for example the automaton of Figure 1(a). Then the constant term $\mathtt{0}$ is recognized at q_{even} by following the nullary transition $\mathtt{0} \to q_{\text{even}}$. Note also that we have transitions of arity more than 1: consider the transition $\mathtt{cons}(q_{\text{even}}, q_{\text{list-even}}) \to q_{\text{list-even}}$ pictured around the $\mathtt{cons}(\_,\_)$ label in the upper right. The circled state $q_{\text{list-even}}$ is the only final state.

A classic alternate description of tree automata is through the use of rewrite systems [5]: tree automata are then rewrite systems consisting of rules of the form $f(q_1, \ldots, q_n) \to q$, where the states q_1, …, q_n, q are constants outside the signature Σ. Then a ground term t over Σ is recognized by a tree automaton if and only if t rewrites to some $q \in F$. This representation of tree automata recognizes terms bottom-up. For example, the term $\mathtt{cons}(\mathtt{S}(\mathtt{S}(\mathtt{0})), \mathtt{nil})$ is recognized by the automaton of Figure 1(a) by the rewrite sequence $\mathtt{cons}(\mathtt{S}(\mathtt{S}(\mathtt{0})), \mathtt{nil}) \to \mathtt{cons}(\mathtt{S}(\mathtt{S}(q_{\text{even}})), \mathtt{nil}) \to \mathtt{cons}(\mathtt{S}(q_{\text{odd}}), \mathtt{nil}) \to \mathtt{cons}(q_{\text{even}}, \mathtt{nil}) \to \mathtt{cons}(q_{\text{even}}, q_{\text{list-even}}) \to q_{\text{list-even}}$. In general, we may decide to recognize terms bottom-up or top-down, and while this makes no difference in theory, this does make a difference when considering *deterministic* tree automata [7]. In the sequel, we shall only consider non-deterministic tree automata, allowing us to avoid the bottom-up/top-down dilemma.

Another, perhaps less well-known data structure for representing tree automata is as sets of first-order Horn clauses [6, 4]. Each state q is translated to a unary predicate P_q—think of $P_q(t)$ meaning "t is recognized at state q"—while transitions $f(q_1, \ldots, q_n) \to q$ are translated to clauses of the form $P_q(f(x_1, \ldots, x_n)) \leftarrow P_{q_1}(x_1), \ldots, P_{q_n}(x_n)$,



(a) Lists of even natural numbers

(b) One element lists containing 1 or $3k + 2$, $k \in \mathbb{N}$

Fig. 1. Tree automata

and recognizability of t means derivability of the empty clause from the goal that is the disjunction of all $P_q(t)$, $q \in F$. It turns out that this representation is particularly attractive in Coq, where sets of first-order Horn clauses can be thought as particular inductive predicates (see Section 3).

In proof assistants that include a programming language, typically a form of λ-calculus as in Coq or HOL, an alternate representation is as a recognizer *function*—rather than as an axiomatized predicate. That is, we may define $\mathrm{Rec}(q, t)$ by primitive recursion on t by $\mathrm{Rec}(q, f(t_1, \ldots, t_n)) \hat{=} \bigvee_{(f(q_1,\ldots,q_n)\to q)\in\Delta} \bigwedge_{i=1}^{n} \mathrm{Rec}(q_i, t_i)$. This will actually be our first choice in Section 5. The example of Figure 1(a), for instance, becomes:

$$\mathrm{Rec}(q_{\text{list-even}}, t) \hat{=} \begin{cases} \mathtt{True} & \text{if } t = \mathtt{nil} \\ \mathrm{Rec}(q_{\text{even}}, t_1) \wedge \mathrm{Rec}(q_{\text{list-even}}, t_2) & \text{if } t = \mathtt{cons}(t_1, t_2) \\ \mathtt{False} & \text{otherwise} \end{cases} \tag{1}$$

$$\mathrm{Rec}(q_{\text{even}}, t) \hat{=} \begin{cases} \mathtt{True} & \text{if } t = \mathtt{0} \\ \mathrm{Rec}(q_{\text{odd}}, t_1) & \text{if } t = \mathtt{S}(t_1) \\ \mathtt{False} & \text{otherwise} \end{cases} \tag{2}$$

$$\mathrm{Rec}(q_{\text{odd}}, t) \hat{=} \begin{cases} \mathrm{Rec}(q_{\text{even}}, t_1) & \text{if } t = \mathtt{S}(t_1) \\ \mathtt{False} & \text{otherwise} \end{cases} \tag{3}$$

where $\mathtt{True}, \mathtt{False} : \mathtt{Prop}$ are verum and falsum respectively. Interpreting these equations as rewrite rules from left to right, we recognize $\mathtt{cons}(\mathtt{S}(\mathtt{S}(\mathtt{0})), \mathtt{nil})$ by the computation: $\mathrm{Rec}(q_{\text{list-even}}, \mathtt{cons}(\mathtt{S}(\mathtt{S}(\mathtt{0})), \mathtt{nil})) = \mathrm{Rec}(q_{\text{even}}, \mathtt{S}(\mathtt{S}(\mathtt{0}))) \wedge \mathrm{Rec}(q_{\text{list-even}}, \mathtt{nil}) = \mathrm{Rec}(q_{\text{even}}, \mathtt{S}(\mathtt{S}(\mathtt{0}))) \wedge \mathtt{True} = \mathrm{Rec}(q_{\text{odd}}, \mathtt{S}(\mathtt{0})) \wedge \mathtt{True} = \mathrm{Rec}(q_{\text{even}}, \mathtt{0}) \wedge \mathtt{True} = \mathtt{True} \wedge \mathtt{True}$, which is clearly provable. Note in passing that this recognizes terms top-down.

Many standard operations are computable on tree automata. Computing an automaton recognizing the intersection or the union of two languages given by automata can be done in polynomial (near-quadratic) time and space. Similarly, testing whether an automaton recognizes no term—the *emptiness* test—takes polynomial time. On the other hand, computing an automaton recognizing the complement of the language recognized by a given automaton $\mathcal{A}$ requires exponential time, as do testing whether $\mathcal{A}$ recognizes all terms—the *universality* test—, or whether all terms recognized by $\mathcal{A}$ are recognized by another automaton—the *inclusion* test. In implementations, the latter operations all require constructing a *deterministic* bottom-up automaton recognizing the same terms as $\mathcal{A}$, explicitly or implicitly, and this takes exponential time and space.

In this paper, we shall only be interested in polynomial time algorithms such as computing intersections and unions, and testing for emptiness. This allows us to only work with non-deterministic automata, therefore keeping our algorithms simple. More to the point, it turns out that this is essentially all we need in applications like [9], where non-deterministic top-down automata are used. Note that, then, we may always assume that there is exactly one final state in F, which we call *the* final state.

We shall also consider polynomial-time optimizations that aim at reducing the size of automata while keeping the languages they recognize intact. Such optimizations are important in practice, since the complexity of operations on automata are primarily functions of their sizes. Particularly useful optimizations are:

- Removal of empty states: say that $q \in Q$ is *empty* in $\mathcal{A}$ if and only if no ground term is recognized at q. Such states, together with transitions incident with q, can be removed from $\mathcal{A}$ without changing the language it recognizes. Empty states are frequently created in computing intersections of automata, notably.
- Removal of non-coaccessible states: say that $q \in Q$ is *coaccessible* in $\mathcal{A}$ if and only if we can reach q by starting from a final state and following transitions backwards. Removing non-coaccessible states and transitions between them leaves the language recognized by the automaton unchanged.

There are many other possible optimizations, e.g., merging states recognizing the same languages, erasing transitions $f(q_1, \ldots, q_n) \to q$ in the presence of $f(q'_1, \ldots, q'_n) \to q$ where a polynomial-time sufficient inclusion test ensures that all terms recognized at q_i are recognized at q'_i, $1 \leq i \leq n$, and so on.

5 Shallow Reflection

We start with a shallow reflection embedding of tree automata in Coq, where automata are described by mutually recursive functions $\mathrm{Rec}(q, \_)$. Such automata are not first-class citizens in Coq: the external tool that computes on tree automata has to output corresponding definitions of $\mathrm{Rec}(q, \_)$ functions for Coq, as well as expected properties about them, and proofs of these properties. (Automata *will be* first-class citizens in the deep reflection approach of Section 6.) This is in contrast with [10, 11, 17], where objects residing in the proof assistant are exported to the external tool, the latter computes on them, then produces a proof back for the proof assistant to check. Here the objects of interest, namely tree automata, are generated by and reside in the external tool, which also computes on them and feeds the proof assistant corresponding definitions, statements of correctness lemmata, and proofs of these lemmata.

While this is not as general as one may wish, this is exactly what we would want to check computations done in a tool such as CPV [9]: while CPV generates automata during the course of its verification process, an instrumented version of CPV will output their definitions to Coq, as well as correctness lemmata and proof scripts. The final Coq file will then validate the whole series of computations done by CPV.

5.1 Instrumenting Automaton Subroutines

Let us consider an example. Assume CPV calls its automaton intersection subroutine on the automata of Figure 1(a) and Figure 1(b). These automata have been produced by earlier computations. Our first change to the subroutines is to have them name the automata they produce, and output corresponding Coq functions. In particular, before the intersection subroutine is called, the input automata have received names, say A and B, and corresponding recursive functions $\texttt{rec\_}N\texttt{\_}q$ of type $\texttt{term} \to \texttt{Prop}$ are defined for every automaton name N and every state q of N: $\texttt{rec\_}N\texttt{\_}q(t)$ returns True if and only if t is recognized at state q in N. For example, the Coq function corresponding to the automaton B of Figure 1(b) is:

$$\begin{aligned}
\texttt{rec_B_}q_7(t) \;&\hat{=}\; \begin{cases} (\texttt{rec_B_}q_2(t_1) \wedge \texttt{rec_B_}q_3(t_2)) \vee & \\ (\texttt{rec_B_}q_6(t_1) \wedge \texttt{rec_B_}q_3(t_2)) & \text{if } t = \texttt{cons}(t_1, t_2) \\ \texttt{False} & \text{otherwise} \end{cases} \\
\texttt{rec_B_}q_2(t) \;&\hat{=}\; \begin{cases} \texttt{rec_B_}q_1(t_1) \text{ if } t = \texttt{S}(t_1) \\ \texttt{False} \qquad\quad \text{otherwise} \end{cases} \\
&\ldots
\end{aligned}$$

where the ellipsis abbreviates five other equalities. The definition of $\texttt{rec\_A\_}q$ follows easily from Equations (1)–(3). Note that this also requires the instrumented CPV tool to first output the definition of a type term of all terms, with constructors $\texttt{O} : \texttt{term}$, $\texttt{nil} : \texttt{term}$, $\texttt{S} : \texttt{term} \to \texttt{term}$, $\texttt{cons} : \texttt{term} \to \texttt{term} \to \texttt{term}$.

Intersection is usually computed by a product construction, where states in the intersection automaton are in one-to-one correspondence with pairs of states of A and B. (This is certainly so in CPV.) We now instrument the intersection subroutine so that it keeps track of this mapping. For example, assume that the intersection subroutine computes the automaton of Figure 2, claiming it to be the intersection of A and B. We have labeled each state in such a way that pairs of states are obvious, e.g., state $q_{\text{even},6}$ is the state that recognizes exactly those terms that are recognized at state q_{even} in A and at state q_6 in B. Note that all pairs are not represented: the CPV library does not compute non-coaccessible states, and eliminates empty states like $q_{\text{even},2}$.

Next, we instrument the intersection subroutine so that it generates the *correctness lemma*:

$$\forall t : \texttt{term} \cdot \texttt{rec\_C\_}q_{final}(t) \Longleftrightarrow \texttt{rec\_A\_}q_{final}(t) \wedge \texttt{rec\_B\_}q_{final}(t) \tag{4}$$

where q_{final} is the name of the final state in each automaton: $q_{\text{list-even},7}$ for C, $q_{\text{list-even}}$ for A, q_7 for B.

Finally, we let the intersection subroutine output a proof of the correctness lemma (4). It is interesting to examine how we might prove it in the case of the example $\texttt{C} = \texttt{A} \cap \texttt{B}$. This can only be proved by structural induction on the term t.

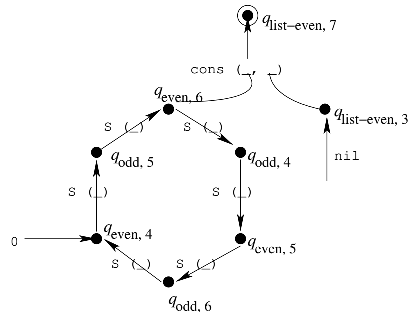


Fig. 2. Automaton recognizing one-element lists of $6k + 2$, $k \in \mathbb{N}$

However, we need to generalize the lemma first. Indeed, if $t = \texttt{cons}(t_1, t_2)$, then the body of (4) simplifies (by reduction in the λ-calculus) to $(\texttt{rec\_C\_}q_{\text{even},6}(t_1) \wedge \texttt{rec\_C\_}q_{\text{list-even},3}(t_2)) \iff (\texttt{rec\_A\_}q_{\text{even}}(t_1) \wedge \texttt{rec\_A\_}q_{\text{list-even}}(t_2)) \wedge ((\texttt{rec\_B\_}q_2(t_1) \wedge \texttt{rec\_B\_}q_3(t_2)) \vee (\texttt{rec\_B\_}q_6(t_1) \wedge \texttt{rec\_B\_}q_3(t_2)))$. This in turn obtains from the equivalences:

$$\begin{aligned}
\texttt{rec_C_}q_{\text{even},6}(t_1) &\iff \texttt{rec_A_}q_{\text{even}}(t_1) \wedge \texttt{rec_B_}q_6(t_1) \\
\texttt{rec_C_}q_{\text{list-even},3}(t_2) &\iff \texttt{rec_A_}q_{\text{list-even}}(t_2) \wedge \texttt{rec_B_}q_3(t_2) \\
\texttt{False} &\iff \texttt{rec_A_}q_{\text{even}}(t_1) \wedge \texttt{rec_B_}q_2(t_1)
\end{aligned}$$

where the latter expresses that there is no term recognized both at state q_{even} in A and at state q_2 in B—note that there is no state $q_{\text{even},2}$ in C: it was empty, so was deleted. However, proving (4) by induction does not generate these equivalences, and we have to prove the conjunction of (4) and the above equivalences (at least) simultaneously.

In general it is necessary to generalize (4) to the conjunction of mn statements, where m is the number of states in A and n is the number of states in B: for each state q_i in A and each state q_j in B, either there is a corresponding state q_{ij} in C and we generate C_{ij}, the conjunct $\texttt{rec\_C\_}q_{ij}(t) \iff \texttt{rec\_A\_}q_i(t) \wedge \texttt{rec\_B\_}q_j(t)$, or there is none and, if the intersection subroutine is correct, no term is recognized both at q_i and at q_j, so we generate C_{ij}, the conjunct $\texttt{False} \iff \texttt{rec\_A\_}q_i(t) \wedge \texttt{rec\_B\_}q_j(t)$. We let the instrumented intersection subroutine generate the generalized correctness lemma $\forall t : \texttt{term} \cdot \bigwedge_{i,j} C_{ij}$.

The proof script of the generalized correctness lemma is actually obvious, and is the same for all possible automata: do an induction on t; on all generated subgoals, compute, i.e., simplify as much as possible (use the `Simpl` tactic of Coq). All generated subgoals, if valid, are then provable using only rules of propositional logic (even intuitionistic): so use the `Tauto` tactic, which finds precisely such proofs. The proof of (4) is generated from the generalized correctness lemma using `Tauto` again.

Instrumenting the union subroutine is either similar (in tools that work on deterministic bottom-up automata, where union is computed by a similar product construction), or simpler. In the case of CPV, which works on non-deterministic top-down automata, this is definitely simpler: computing the union of two automata with disjoint sets of states amounts to generating a new final state q', and generating new transitions $f(q_1, \ldots, q_n) \to q'$ for each transition $f(q_1, \ldots, q_n) \to q$ in either input automaton such that q was final. It follows that the correctness lemma for unions only requires a case analysis, and no induction, and that it needs no prior generalization.

Instrumenting the emptiness test is also easy. Given an automaton `C`, either it is empty, and because CPV eliminates empty states, its corresponding function in Coq is the constant function `False`, so there is nothing to prove (in general, a polynomial-time emptiness test should be run in the automaton library to get this information); or it is not empty, and we instrument the code to output a witness t, i.e. a term t that is recognized by `C`. We then let the code output a non-emptiness lemma $\exists t : \texttt{term} \cdot \texttt{rec\_C\_}q_{final}(t)$, and its proof: exhibit t, simplify, call `Tauto`.

5.2 Experimental Evaluation

The approach of Section 5.1 has several nice features. It is easy to instrument most any automaton library to get similar results; we may choose to change or upgrade the automaton library, and this only requires modifying the instrumentation code; finally, the generated Coq proofs are short and do not depend on the automata we compute on.

However, as is apparent from the discussion on instrumenting intersection in Section 5.1, this approach generates huge generalized correctness lemmata. It is therefore necessary to evaluate the approach on a few practical examples. To test it, we used the signature `0`, `S`, `nil`, `cons` as above; we generated automata $\mathcal{A}_n$ recognizing the lists of natural numbers whose elements are all congruent to $n-1$ modulo n. $\mathcal{A}_n$ has $n+1$ states and $n+3$ transitions and contains only one cycle of length n. We then let Coq check proofs as generated by the method of Section 5.1, computing unions and intersections of automata $\mathcal{A}_n$. All tests were done on a Pentium Celeron 400Mhz notebook with 128 MO RAM running Linux 2.2.13.

As far as unions are concerned, we check unions of $\mathcal{A}_n$ and $\mathcal{A}_{n+1}$ for various values of n: this involves generating the automata, lemmata stating that all automata are not empty, the correctness lemma, and their proofs. The resulting automaton has $2n+4$ states; we sum up the times taken by Coq to do the whole verification of one instance (parsing, type-checking, checking non-emptiness and correctness proofs), the times taken to check the correctness lemma alone, and memory usage:

n	10	20	30	40	50	60	70	80
# states	24	44	64	84	104	124	144	164
total check (s.)	4	9	19	33	59	84	126	181
correctness (s.)	1	2	3	4	5	7	8	9
memory (Mb.)	7.5	8	9	10	12	14	17	18

Notice that checking that unions are computed correctly takes negligible time compared to parsing plus checking non-emptiness. This is to be expected, as checking unions only involves a case analysis and no induction.

We conducted exactly the same test with intersection. The results, as the reader may see, are extremely disappointing. When $n = 1$, checking takes 6s. and 8Mb; 74s. and 24Mb when $n = 2$; 513s. and 96Mb when $n = 3$; we ran out of memory when $n = 4$.

Both checking time and space increase quickly. This is clearly due to the large size of generalized correctness lemma. Although we might think that this is also aggravated by the relative inefficiency of `Tauto`, experience in replacing `Tauto` by more elementary tactics did not increase performance by any significant amount.

As mentioned in Section 4, tree automata can also be represented as sets of Horn clauses of a particular format, and sets of Horn clauses are naturally encoded in Coq as inductive predicates. It was suggested by C. Paulin-Mohring and B. Werner that this format usually has distinctive advantages over recursive functions as in Section 5.1: the main one is that induction and case analysis proofs only consider cases that actually happen. For example, if a state has two incoming transitions, the corresponding inductive predicate will have two clauses, even though terms might be built on more constructors (4 in our examples). In the encoding using recursive functions, any switch on the shape of terms actually always compiles to a n-way switch, where n is the number of constructors.

However, this alternate implementation of tree automata also turned out to be disappointing, as experience with hand-generated examples demonstrated. Compared to the technique of Section 5.1, proof scripts are harder to generate. In principle, it is enough to use the `Auto` resolution tactic instead of `Tauto`; however the use of conjunctions $\wedge$ and disjunctions $\vee$ foils this strategy: in Coq, conjunctions and disjunctions are not primitive and need to be decomposed by some other mechanism. The crucial point however is that checking proofs, in particular for intersection, in this new scheme is only roughly twice as fast as with the scheme of Section 5.1, and uses as much memory.

6 Deep Reflection

As said in Section 3, deep reflection means implementing tree automata and operations on them in Coq's λ-calculus, and proving the correctness of the latter in Coq. However, tree automata may be represented in several different ways, so one first contribution here is the choice of a workable data structure for tree automata in Coq (Section 6.1). Tree automata remain complex objects, and some algorithms will clearly not be optimal. This is to avoid excessive complications in proofs, which are already complex. We shall describe the computation of unions, intersections, and removal of empty and non-coaccessible states in Sections 6.2, 6.3 and 6.4 respectively. This will be rather cursory, and we shall only stress salient features.

6.1 Data Structures

We have chosen to represent tree automata as top-down non-deterministic tree automata. Previous experience with CPV [9] indicates that this is probably one of the simplest possible implementations. Such automata are described as a table mapping each state q to the set of possible transitions reaching q. This set is itself organized as a table mapping each function symbol f labeling such a transition to the set of lists $[q_1, \ldots, q_n]$ of states such that $f(q_1, \ldots, q_n) \to q$ is a transition. Note that all lists in a given set have the same length n.

To represent tables, we use the map library of [8]. This provides a data type $\mathtt{ad}$ of *addresses*, a.k.a. *keys*, and a type constructor $\mathtt{Map}$, such that $\mathtt{Map}(\tau)$ is the type of *maps* over τ, i.e., tables mapping keys to objects of type τ. The implementation of maps over τ is an efficient binary tree representation, where addresses are essentially finite lists of bits, or alternatively natural numbers in binary. The map library provides functions such as $\mathtt{MapGet}$ to fetch the object associated with a given key—or return the constant $\mathtt{NONE}$ if there is none—, and $\mathtt{MapPut}$ to produce a new map obtained from a given map m by adding a binding from a given key a to a given object x, erasing any possible preexisting binding to a. We shall simply write $f(a)$ to denote $\mathtt{MapGet}(f, a)$ when it is *defined*, i.e., does not return $\mathtt{NONE}$.

This is used to define *pre-automata* as objects of type $\mathtt{Map}(\mathtt{st})$ mapping states q to elements of $\mathtt{st}$; the latter map function symbols of arity n to sets of lists of states of length n, as indicated above. Since maps only take addresses, i.e., elements of $\mathtt{ad}$ as indices, this requires us to encode states as addresses, as well as function symbols.

In particular, we define *terms*, of inductive type $\mathtt{term}$ with one constructor $\mathtt{app}$: $\mathtt{ad} \to \mathtt{term\ list} \to \mathtt{term\ list}$, where $\mathtt{term\ list}$ is defined simultaneously as the type of finite lists of terms. For example, $f(t_1, \ldots, t_n)$ is coded as $\mathtt{app}(f, [t_1, \ldots, t_n])$. Each function symbol f : $\mathtt{ad}$ has an arity; this is summed up in a *signature* Σ, i.e., a map from function symbols f to their arities, of type $\mathtt{Map}(\mathbb{N})$.

Returning to pre-automata, the only remaining difficulty is to represent sets of lists of states of length n. The standard way to represent finite sets with maps is to encode them as elements of $\mathtt{Map}(*)$, where $*$ is a one-element type. However, this only allows us to encode sets of addresses, and while states are addresses, lists of states are not. Since addresses are natural numbers, it is in principle feasible to encode lists of addresses as natural numbers, solving this conundrum. However, this solution is complex, not particularly efficient in practice, and hard to reason about formally. Instead, we create a new data structure to represent sets of lists of states of length n as binary trees whose nodes are labeled with states: each branch in the tree is taken to denote the list of all states labeling nodes to the left of which the branch goes, provided it is of length n. Formally, the type $\mathtt{prec\_list}$ of sets of lists of states is the inductive type with two constructors $\mathtt{prec\_cons} : \mathtt{ad} \to \mathtt{prec\_list} \to \mathtt{prec\_list} \to \mathtt{prec\_list}$ and $\mathtt{prec\_empty} : \mathtt{prec\_list}$. The formal semantics $[\![pl]\!]_n$ of pl : $\mathtt{prec\_list}$ as a set of lists of length n is defined by: $[\![\mathtt{prec\_empty}]\!]_0 \hat{=} \{[]\}$; $[\![\mathtt{prec\_empty}]\!]_{n+1} \hat{=} \emptyset$; $[\![\mathtt{prec\_cons}\ q\ \ell\ r]\!]_0 \hat{=} [\![\emptyset]\!]$; $[\![\mathtt{prec\_cons}\ q\ \ell\ r]\!]_{n+1} \hat{=} \{q :: pl \mid pl \in [\![\ell]\!]_n\} \cup [\![r]\!]_{n+1}$, where $q;;l$ is the list obtained from l by adding q in front.

Recall that, then, $\mathtt{st} \hat{=} \mathtt{Map}(\mathtt{prec\_list})$ and the type of pre-automata is $\mathtt{predta} \hat{=} \mathtt{Map}(\mathtt{st})$. *Tree automata*, of type $\mathtt{dta}$, are pairs of a pre-automaton A and a state, taken to be the final state. They are *well-formed* w.r.t. a signature Σ provided the following conditions hold:

1. each state q is mapped to a map $A(q)$ from function symbols f to sets of lists *of length* n, where $n = \Sigma(f)$ is the arity of f;
2. for every state q, every function symbol f, every list $[q_1, \ldots, q_n] \in A(q)(f)$, q_i is in the domain of A for every i, $1 \le i \le n$ (no dangling pointers);
3. Similarly, the final state is in the domain of A.

All three conditions 1–3 are necessary to establish the correctness of operations in following sections. In particular, Condition 1 is crucial to be able to give a semantics to `prec_lists`.

The semantics of tree automata is defined by mutually inductive predicates `rec_predta : predta` $\to$ `ad` $\to$ `term` $\to$ `Prop` (recognizing a term at a given state), `rec_st : predta` $\to$ `st` $\to$ `term` $\to$ `Prop` (recognizing a term at a state given by its set of incoming transitions), `rec_prec_list : predta` $\to$ `prec_list` $\to$ `term list` $\to$ `Prop` (recognizing a list of terms at a list of states in some set of lists of states), where $\texttt{rec\_predta}(A, q, t)$ provided $A(q)$ is defined and $\texttt{rec\_st}(A, A(q), t)$; where $\texttt{rec\_st}(A, tr, t)$ provided $t = \texttt{app}(f, [t_1, \ldots, t_n])$, $tr(f)$ is defined and $\texttt{rec\_prec\_list}(A, tr(f), [t_1, \ldots, t_n])$; finally, $\texttt{rec\_prec\_list}(A, \texttt{prec\_empty}, [])$, $\texttt{rec\_prec\_list}(A, \texttt{prec\_cons}(q, \ell, r), t :: tl)$ provided either $\texttt{rec\_predta}(A, q, t)$ and $\texttt{rec\_prec\_list}(A, \ell, tl)$, or $\texttt{rec\_prec\_list}(A, r, t :: tl)$.

6.2 Implementing Union

Computing unions is done by the algorithm used in CPV, described in Section 5.1: generate a new final state q', and generate new transitions $f(q_1, \ldots, q_n) \to q'$ for each transition $f(q_1, \ldots, q_n) \to q$ in any of the input automata $\mathcal{A}$ or $\mathcal{B}$ where q was final. However, this is only correct provided $\mathcal{A}$ and $\mathcal{B}$ have disjoint sets of states. In implementations like CPV, states are memory locations pointing to maps of type `st`, and disjointness is not required for correctness, as any state common to both automata points to the same sub-automaton. The latter condition is hard to ensure (we might have modeled a full store as in [15], but this looked like overkill), while the disjointness condition is simpler to implement in the representation of Section 6.1: therefore, in Coq, we first copy each input automaton by adding a 0 bit in front of any address in $\mathcal{A}$ (i.e., changing q into $2q$) and a 1 bit in front of any address in $\mathcal{B}$ (i.e., changing q into $2q + 1$), then add a new final state and all relevant incoming transitions.

We prove that this algorithm is correct: the term t is recognized by the union of two well-formed automata if and only if t is recognized by one or the other, by structural induction on t. That the input automata are well-formed (Conditions 1–3) is needed in the proof. Condition 1 is needed to give a meaning $[\![pl]\!]_n$ to `prec_lists` $A(q)(f)$, where n is the arity of f. Conditions 2 and 3 are needed to give meaning to the set of all transitions $f(q_1, \ldots, q_n) \to q$ where q is given. We also prove that the union of two well-formed automata is again well-formed, one condition at a time. Notice that we never had to impose any well-formedness condition on terms, only on automata.

6.3 Implementing Intersection

Intersection of two automata $\mathcal{A}$ and $\mathcal{B}$ is trickier. The standard product construction consists in generating new states $\langle q, q' \rangle$ that are in one-to-one correspondence with pairs of states q of $\mathcal{A}$ and q' of $\mathcal{B}$. There is a transition $f(\langle q_1, q'_1 \rangle, \ldots, \langle q_n, q'_n \rangle) \to \langle q, q' \rangle$ in the computed intersection automaton $\mathcal{C}$ if and only if $f(q_1, \ldots, q_n) \to q$ is a transition in $\mathcal{A}$ and $f(q'_1, \ldots, q'_n) \to q'$ is a transition in $\mathcal{B}$, and the final state of $\mathcal{C}$ is $\langle q_{final}, q'_{final} \rangle$, where q_{final} is final in $\mathcal{A}$ and q'_{final} is final in $\mathcal{B}$. Our first problem here is that states $\langle q, q' \rangle$ should be of type `ad`, so we need a one-to-one mapping from pairs of addresses to addresses. There are several solutions to this. The one we choose is one of the simplest, and also of the most efficient to compute: looking at q and q' as sequences of bits, we interleave them to get $\langle q, q' \rangle$. For example, if q is 10010 in binary and q' is 1101, then $\langle q, q' \rangle$ is 1001011001. Formally, $\langle q, q' \rangle$ is $\langle \lfloor q'/2 \rfloor, q \rangle$ followed by 0 if q' is even, by 1 if q' is odd, and $\langle 0, 0 \rangle = 0$.

Linking in such an explicit way the states $\langle q, q' \rangle$ to q and q', i.e., making sure that we can get back q and q' easily from $\langle q, q' \rangle$ without the help of any outside machinery is a great help in proofs of correctness. The construction of the intersection automaton $\mathcal{C}$ is then direct: generate all states $\langle q, q' \rangle$, and add transitions as specified above. This involves quite many nested structural inductions, though: we have to induct on the trees that encode the pre-automata $\mathcal{A}$ and $\mathcal{B}$, then on the trees that encode transitions with symbol function f, for each f in each of the two input automata, then induct on two `prec_lists`. It then turns out that the intersection construction applied to two automata indeed computes the intersection, provided Condition 1 is satisfied; this is proved by structural induction on the terms fed to each automaton. Curiously, we don't need Conditions 2 or 3 here: intersection is still correct without them. Finally, we prove that the intersection construction preserves well-formedness, as expected.

Nonetheless, this intersection construction is naive: it generates many empty or non-coaccessible states. To correct this, it is common practice to only generate states $\langle q, q' \rangle$ *by need*. Intuitively, generate $\langle q_{final}, q'_{final} \rangle$; then, for each f, for each pair of transitions $f(q_1, \ldots, q_n) \to q_{final}$ in $\mathcal{A}$ and $f(q'_1, \ldots, q'_n) \to q'_{final}$ in $\mathcal{B}$, generate the states $\langle q_1, q'_1 \rangle, \ldots, \langle q_n, q'_n \rangle$ (observe that they may fail to be new), and recurse. Note that this is not well-founded induction, and a loop-checking mechanism has to be implemented: when some state $\langle q, q' \rangle$ is required that has already been generated, stop recursing. Coding such an algorithm in Coq, and above all proving it, is daunting—recall that the naive algorithm already requires 6 nested inductions. Thus, we have refrained from doing so.

6.4 Removing Empty and Non-Coaccessible States

Instead, we have implemented algorithms to delete empty and non-coaccessible states as separate functions. This is already rather involved, as this involves some form of loop-checking. First, we observe that, from the logical point of view, recursing with loop checks is just an implementation of a least fixpoint computation. For example, define the set $\mathcal{NV}$ of *non-void states* of $\mathcal{A}$ as the smallest such that for every transition $f(q_1, \ldots, q_n) \to q$ such that $q_1, \ldots, q_n \in \mathcal{NV}$, then $q \in \mathcal{NV}$. While this is naturally coded in Coq as an inductive predicate, such a predicate does not lend itself to computation. To compute $\mathcal{NV}$, we must instead compute the least fixpoint of the function NV mapping any set S of states to $NV(S) \hat{=} S \cup \{q \mid f(q_1, \ldots, q_n) \to q \text{ transition in } \mathcal{A}, q_1 \in S, \ldots, q_n \in S\}$. We then prove that both definitions are equivalent, by well-founded induction on the definition of the non-vacuity predicate in one direction, and by Tarskian fixpoint induction in the other direction.

This however requires that we develop a Tarskian theory of fixpoints of monotonic functions over complete lattices—at least over complete Boolean lattices of finite height. Indeed, subsets of a given set of states Q of cardinality n form a Boolean lattice of height n. In Coq, we represent subsets of Q as objects of type `Map(bool)`, constrained to have Q as domain. We show in Coq that this can be given the structure of a complete Boolean lattice of cardinality 2^n and height n, taking as ordering $m \leq^* m'$ if and only if $m(q) \leq m'(q)$ for every $q \in Q$, where $\leq$ is the ordering `false` $\leq$ `true` in `bool`. Independently, we show in Coq that, for every complete lattice L of height n, with bottom element $\perp$, for every monotonic $f : L \to L$, $f^n(\perp)$ is the least fixpoint of f. This is especially convenient for computation, as $f^n(\perp)$ can be defined by structural recursion on n in Coq, while general fixpoints cannot be defined as recursive functions.

Summing up, we obtain a function that computes the set of non-void states of a given automaton, together with a proof of its correctness. Observing now that a state q is non-void if and only if it is non-empty, where a state q of $\mathcal{A}$ is *non-empty* if and only if there is a term t recognized at q in $\mathcal{A}$, we deduce that the automaton $\mathcal{B}$ obtained from $\mathcal{A}$ by only keeping non-void (a.k.a, non-empty) states has the same semantics as $\mathcal{A}$. Therefore, composing empty state removal with the naive intersection algorithm yields another correct intersection algorithm. (The empty state removal algorithm is actually slightly trickier, in that it also removes transitions $f(q_1, \ldots, q_n) \to q$ where at least one q_i, $1 \leq i \leq n$, is empty, even when q is non-empty; and also in that it does not remove the final state, even if it is empty. These considerations complicate the algorithm and the proofs, but are not essential to the present discussion.)

Removal of non-coaccessible states is coded, proved and applied to intersection in a similar way. In both the non-emptiness and non-coaccessibility cases, the precise correctness theorems proved rest on input automata being well-formed; we also show that these removal algorithms preserve well-formedness.

6.5 Experimental Evaluation

It is interesting to evaluate how deep reflection works in practice, and whether it yields any significant advantage over shallow reflection (Section 5.2). We report on times and memory consumed for operations of union, intersection, and removal of useless states. Noticing that useless state removal is slow, we also include measures of efficiency for removal by shallow reflection (see below). We also compare these results, obtained by running our algorithms inside Coq's λ-calculus, with results obtained by running the OCaml program that one gets by Coq's extraction feature. While efficiency depends on many parameters (number of states, number of transitions, maximal arity of function symbols), we have decided to focus on evaluating the algorithms on the same examples as in Section 5.2 for the sake of comparison (this also tests how the algorithms fare in the presence of cycles), and examples featuring large transition tables. Tests were run on the same Pentium Celeron notebook with 128 Mb RAM.

Results on the cyclic examples $\mathcal{A}_n$ of Section 5.2 are as follows:

n	10	15	20	30	50	500	$n_{\max}$
$\mathcal{A}_n \cup \mathcal{A}_{n+1}$	$0.57(26)$	$0.75(26)$	$1.19(26)$	$1.71(27)$	$2.89(28)$	$51.8(48)$	1500
$\mathcal{A}_n \cap \mathcal{A}_{n+1}$	$2.09(27)$	$4.55(32)$	$10.6(36)$	$32.1(45)$	–	–	70
$\emptyset$-removal	$104.7(48)$	$438.9(95)$	–	–	–	–	25
$\emptyset$-check	$3.16/0.25$	$6.65/0.56$	$21.8/1.04(38)$	$44.0/2.34(49)$	–	–	50

Results are in the form $t(m)$ where times t are in seconds, and memory sizes m are in megabytes. The last column reports the largest n for which computation ran in the available 128 Mb plus 200 Mb swap space, without a time limit. (But times are only reported for cases where swap usage is negligible.) The $\emptyset$-removal row measures the time to remove empty states from automata computed in the second row. The $\emptyset$-check row measures the efficiency of the following procedure: instead of computing the fixpoint $\mathcal{NV}$ in Coq, let an external tool written in OCaml do it and feed it to Coq; then Coq checks that $\mathcal{NV}$ indeed is a fixpoint, and removes all empty states. Reported measures are $t/t_{\mathcal{NV}}(m)$, where t is time to remove empty states given $\mathcal{NV}$, $t_{\mathcal{NV}}$ is the time to check that $\mathcal{NV}$ is a fixpoint, and m is the amount of memory used. Although the last two rows report on computations done on $\mathcal{A}_n \cap \mathcal{A}_{n+1}$, times do not include intersection computation times. Note that input automata and $\mathcal{A}_n \cup \mathcal{A}_{n+1}$ have $O(n)$ states and transitions, while all others have $O(n^2)$ states and transitions.

Clearly, intersections are costly, but empty state removal is even costlier. Here shallow reflection helps (last row). Also, results are much better than with a wholly shallow reflection approach (Section 5.2), but not quite usable: CPV [9] typically needs to handle automata of 200–300 states.

Comparatively, the extracted OCaml code computes unions up to $n = 150,000$, intersections up to $n = 400$. It is roughly at least 300 times faster and consumes about 100 times less memory, although it is hard to draw a comparison: there are too few values of n for which both Coq computations succeed and OCaml times are measurable at all. Note that the extracted OCaml code would in general be able to deal with automata of the size that CPV generates. We believe that this vindicates using compiled OCaml code as a reduction machine inside Coq, instead of the current λ-calculus interpreter.

In a second series of tests, we worked with automata $\mathcal{B}_n$ on a signature with one constant α and one binary function symbol β, so that terms are just binary trees; $\mathcal{B}_n$ has $n+1$ states s_k^n, $0 \leq k \leq n$, that recognizes trees with k occurrences of β, and the final state is s_n^n. $\mathcal{B}_n$ has $1 + n(n+1)/2$ transitions, and is therefore dense. Tests (see table below) are similar to the ones above, except we test intersections by computing $\mathcal{B}_n \cap \mathcal{B}_n$ on two disjoint copies of $\mathcal{B}_n$—testing $\mathcal{B}_n \cap \mathcal{B}_{n+1}$ would result in an empty automaton. Note that unions have $O(n)$ states and $O(n^2)$ transitions, while intersections have $O(n^2)$ states and $O(n^4)$ transitions.

n	5	10	15	25	50	100
$\mathcal{B}_n \cup \mathcal{B}_{n+1}$	$0.45(26)$	$1.47(30)$	$2.54(32)$	$5.76(37)$	$23.9(50)$	$156(100)$
$\mathcal{B}_n \cap \mathcal{B}_n$	$2.79(31)$	$29.9(50)$	$196(87)$	–	–	–
$\emptyset$-check	$0.89/0.36(32)$	$18.4/3.14(46)$	$62.7/12.5(91)$	–	–	–

Again, OCaml is 400–600 times faster, and uses negligible memory compared to Coq.

7 Conclusion

We have presented two ways to formally verify tree automata computations in a proof assistant like Coq. The surprising conclusion is that the most promising method, checking computations by shallow reflection, gives disastrous results. On the contrary, deep reflection works much better—with a zest of shallow reflection for useless state removal—, up to the point that an extracted OCaml version—hence a certified algorithm—tackles computation on tree automata of sizes comparable to those dealt in a realistic cryptographic protocol verification application. These sizes are modest, still, and more work remains to be done. In particular, computing states of intersection automata by need instead as eagerly, as we did, is required. However, this is definitely a lot of work—our naive, deep reflection implementation and proofs already take about $14,000$ lines of Coq—and doing so involves implementing unbounded recursion with loop checks, i.e., computing least fixpoints, which we have seen was inefficient in Coq. Replacing the λ-calculus interpreter of Coq by a machine that would compile λ-terms to OCaml and run the compiled code also appears as a good way to gain efficiency.

References

1. B. Barras, S. Boutin, C. Cornes, J. Courant, J.-C. Filliâtre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saibi, and B. Werner. The Coq proof assistant reference manual: Version 6.3.1. Technical report, INRIA, France, 1999.
2. S. Boutin. Using reflection to build efficient and certified decision procedures. In M. Abadi and T. Ito, editors, *TACS'97*. Springer-Verlag LNCS 1281, 1997.
3. R. Boyer and J. S. Moore. Metafunctions: Proving them correct and using them efficiently as new proof procedures. In *The Correctness Problem in Computer Science*. Acad. Press, 1981.
4. W. Charatonik and A. Podelski. Set-based analysis of reactive infinite-state systems. In B. Steffen, editor, *TACAS'98*, pages 358–375. Springer Verlag LNCS 1384, 1998.
5. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, T. Sophie, and M. Tommasi. Tree automata techniques and applications. Available at `http://www.grappa.univ-lille3.fr/tata`, 1997.
6. L. Fribourg and M. Veloso Peixoto. Automates concurrents à contraintes. *Technique et Science Informatique*, 13(6):837–866, 1994.
7. F. Gécseg and M. Steinby. Tree languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, pages 1–68. Springer Verlag, 1997.
8. J. Goubault-Larrecq. Satisfiability of inequality constraints and detection of cycles with negative weight in graphs. Part of the Coq contribs, available at `http://pauillac.inria.fr/coq/contribs/graphs.html`, 1998.
9. J. Goubault-Larrecq. A method for automatic cryptographic protocol verification. In *FMPPTA'2000*, pages 977–984. Springer Verlag LNCS 1800, 2000.
10. J. Harrison. Binary decision diagrams as a HOL derived rule. *The Computer Journal*, 38:162–170, 1995.
11. J. Harrison. Stålmarck's algorithm as a HOL derived rule. In J. von Wright, J. Grundy, and J. Harrison, editors, *TPHOL'96*, pages 221–234. Springer Verlag LNCS 1125, 1996.
12. J.-P. Jouannaud. Rewrite proofs and computations. In H. Schwichtenberg, editor, *Proof and Computation*, volume 139 of *NATO series F: Computer and Systems Sciences*, pages 173–218. Springer Verlag, 1995.
13. D. Monniaux. Abstracting cryptographic protocols with tree automata. In *SAS'99*, pages 149–163. Springer-Verlag LNCS 1694, 1999.
14. W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 4, pages 133–191. Elsevier Science Publishers B. V., 1990.
15. K. N. Verma, J. Goubault-Larrecq, S. Prasad, and S. Arun-Kumar. Reflecting BDDs in Coq. In *ASIAN'2000*, pages 162–181. Springer Verlag LNCS 1961, 2000.
16. R. W. Weyhrauch. Prolegomena to a theory of mechanized formal reasoning. *Artifical Intelligence*, 13(1, 2):133–170, 1980.
17. S. Yu and Z. Luo. Implementing a model checker for LEGO. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *FME'97*, pages 442–458. Springer-Verlag LNCS 1313, 1997.