Stålmarck's Algorithm in Coq: A Three-Level Approach

Laurent Théry

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Stålmarck's Algorithm in Coq: A Three-Level Approach

Laurent Théry

N° 4353

January 2002

THÈME 2

ISSN 0249-6399 ISRN INRIA/RR--4353--FR+ENG

Stålmarck's Algorithm in Coq: A Three-Level Approach

Laurent Théry

Thème 2 — Génie logiciel
et calcul symbolique
Projet Lemme

Rapport de recherche n° 4353 — January 2002 — 20 pages

Abstract: One of the most efficient method to check if a boolean proposition is a tautology is Stålmarck's algorithm. This paper reports on some experiments in integrating this algorithm inside the Coq theorem prover in a safe way.

Key-words: Stålmarck's algorithm, boolean formulae, satisfiability, Coq

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

L'algorithme de Stålmarck dans Coq: une approche à trois niveaux

Résumé : Une des méthodes les plus efficaces pour vérifier si une formule booléenne est une tautologie est l'algorithme de Stålmarck. Ce rapport propose et évalue différentes méthodes d'intégration de cet algorithme à l'intérieur du système de preuve Coq.

Mots-clés : algorithme de Stålmarck, formule booléenne, satisfiabilité, Coq

1 Introduction

Proving properties formally on machine is still a time-consuming activity. To increase the productivity, more automation should be added. There are different ways to perform this addition. In this paper we present some of our experiments in integrating a proof procedure based on Stålmarck's algorithm [13] inside the Coq prover [8]. Stålmarck's algorithm is a tautology checker. Given a boolean proposition that contains variables, it checks if this proposition is true for all possible values of its variables. Integrating such an algorithm in a prover makes it possible to automate proofs of some universally quantified formulae.

The most direct way to perform this integration is to simply add *externally* some implementation of the algorithm to the kernel of the prover. Such an extension can be troublesome. If the implementation is incorrect, we may introduce theorems that are not valid. In turn, these theorems can be used to prove any proposition since $\forall A, B.\, A \wedge \neg A \Rightarrow B$ is always a valid theorem. The standard answer to this problem is to use *reflection* [6] and prove the correctness of the implementation inside the prover prior its integration. Proving the correctness of a reasonable implementation of Stålmarck's algorithm is possible and has already been done in a previous work [9].

In systems like Coq, an alternative way is to express the computation directly inside the prover. In that case the problem of finding a proof is translated into an equivalent syntactic problem. It is then possible to use computation to solve the syntactic problem. This is the so-called *two-level approach* [4, 3]. Following this line, we get double benefit. First, the extension is done *internally* so it is safe by construction. Second, if the system records a proof object, what would be recorded in that case is a single function call. This means that the size of the proof object is independent of the actual amount of computation that is needed to establish the proposition.

Finally a last possibility is to perform a mix between external and internal integration. The idea is to separate the search for a proof, that is usually cpu-intensive, from checking the proof. The search is done externally producing a trace. This trace is then checked internally by the prover.

In this paper we present these three distinct approaches and show how practical they are for the integration of Stålmarck's algorithm. The paper is structured as follows. We first give an overview of the algorithm. Then we present a certified implementation that can be used as a base for the three approaches. Finally we compare the different approaches on standard examples and draw some conclusions.

2 The algorithm

In this section we give a quick overview of Stålmarck's algorithm. A more complete introduction can be found in [13]. We consider boolean formulae. Atomic boolean formulae are composed of the constants $\top$ (true) and $\bot$ (false) and a set of variables $(v_i)_{i\in\mathbb{N}}$. We also have the unary constructor $\neg$ (negation) and the binary constructors & (conjunction), #

(disjunction), $\mapsto$ (implication), $=$ (equivalence). A boolean formula is a well-formed expression composed of constants, variables and constructors. For example, the following formula is a boolean expression containing four variables v_1, v_2, v_3 and v_4:

$$((v_1 \mapsto v_2) \,\&\, (v_3 \mapsto v_4)) \mapsto ((v_1 \,\&\, v_3) \mapsto (v_2 \,\&\, v_4))$$

Stålmarck's algorithm is a tautology checker. This means that it checks if a formula is valid (true) for any assignment of its variables. The previous example is a tautology. An example of a formula that is not a tautology is $v_1 \mapsto v_2$, since $v_1 = \top$ and $v_2 = \bot$ invalidates the formula.

The algorithm does not work directly on the formula but on a derived form: a list of *triplets*. To construct this data-structure we first use *signed* variable to capture negation, so $\neg v_i$ is represented as $-v_i$. Then we assign new variables to all the sub-expressions of the formula (sub-expressions starting with negation are handled by signed variables). For our example we have:

$$\underbrace{\underbrace{\underbrace{(\underbrace{(v_1 \mapsto v_2)}_{v_5} \,\&\, \underbrace{(v_3 \mapsto v_4)}_{v_6})}_{v_9} \mapsto \underbrace{(\underbrace{(v_1 \,\&\, v_3)}_{v_7} \mapsto \underbrace{(v_2 \,\&\, v_4)}_{v_8})}_{v_{10}}}_{v_{11}}$$

This gives us the following list of 7 triplets:

$$\begin{array}{lcll} v_{11} & := & v_9 \mapsto v_{10} & (1) \\ v_{10} & := & v_7 \mapsto v_8 & (2) \\ v_9 & := & v_5 \,\&\, v_6 & (3) \\ v_8 & := & v_2 \,\&\, v_4 & (4) \\ v_7 & := & v_1 \,\&\, v_3 & (5) \\ v_6 & := & v_3 \mapsto v_4 & (6) \\ v_5 & := & v_1 \mapsto v_2 & (7) \end{array}$$

The value of the formula is the value of v_{11}.

The algorithm accumulates information about the respective values of the different variables. For each kind of triplets, there is a set of rules that describes how new information can be gained. For example, given the triplet $v_i := v_j \mapsto v_k$, a first rule says that if $v_i = \bot$ holds, we obtain two new equations $v_j = -v_k = \top$. Another rule for the same triplet says that if $v_j = \top$ holds, one new equation $v_i = v_k$ is gained. These rules can be seen as a way of propagating information. Appendix A gives the complete set of rules for the triplet $v_i := v_j \,\&\, v_k$.

The algorithm works by refutation. It assumes that the formula is false and tries to reach a contradiction, i.e an equation that is impossible to satisfy $v_i = -v_i$ or $\top = \bot$. On our example, the propagation alone is sufficient to establish that the formula is a tautology. We

start with the state where $v_{11} = \bot$ and apply the following propagation:

$$
\begin{array}{rrrrrrrrrr}
 & & & v_{11} = \bot & on & (1), & we\ get & v_9 = \top & and & v_{10} = \bot \\
 & & & v_{10} = \bot & on & (2), & we\ get & v_7 = \top & and & v_8 = \bot \\
 & & & v_7 = \top & on & (5), & we\ get & v_1 = \top & and & v_3 = \top \\
 & & & v_9 = \top & on & (3), & we\ get & v_5 = \top & and & v_6 = \top \\
v_5 = \top & and & v_1 = \top & on & (7), & we\ get & v_2 = \top & & & \\
v_6 = \top & and & v_3 = \top & on & (6), & we\ get & v_4 = \top & & & \\
v_2 = \top & and & v_4 = \top & on & (4), & we\ get & v_8 = \top & & &
\end{array}
$$

The last equation is a contradiction since we know that $v_8 = \bot$ from the second line. An alternative way to see this propagation is to keep the representation of the formula and to mark each variable with its value and at which step of the propagation it has been assigned. For example $v_7^{\top^2}$ means that the variables v_7 has been assigned the value true at the second step of the propagation:

$$
\underbrace{\underbrace{\underbrace{(v_1^{\top^3} \mapsto v_2^{\top^5})}_{v_5^{\top^4}} \,\&\, \underbrace{(v_3^{\top^3} \mapsto v_4^{\top^6})}_{v_6^{\top^4}}}_{v_9^{\top^1}} \mapsto \underbrace{\underbrace{(v_1^{\top^3} \,\&\, v_3^{\top^3})}_{v_7^{\top^2}} \mapsto \underbrace{(v_2^{\top^5} \,\&\, v_4^{\top^6})}_{v_8^{\bot^2}}}_{v_{10}^{\bot^1}}}_{v_{11}^{\bot^0}}
$$

Most of the time the propagation alone is not sufficient to get a contradiction and conclude. A simple way to complete propagation is to allow case splitting. Whenever propagation has not reached a contradiction, there are two possibilities. If all the variables are assigned, the formula is not a tautology since we have found an assignment that makes it false. Otherwise we pick one variable v_i not yet assigned and create two branches. On the first branch we add the new equation $v_i = \top$ and on the second branch the equation $v_i = \bot$ and we iterate the same process (propagation+case-splitting) on both branches. This method gives us an execution that looks like a binary tree as depicted in Figure 1. Since there is a finite number of variables, iterating propagation and case splitting terminates. If all branches of the exploration ends with a contradiction, our initial formula is a tautology.

The original idea in Stålmarck's algorithm is to make use of intersection in order to avoid case explosion. This is done in the *dilemma rule* that is depicted in Figure 2. When applying the rule we still create two branches and perform the propagation but instead of branching again we merge the two branches by considering a state that contains all the equations that are valid in both branches ($S' = S_3 \cap S_4$). If both branches give a contradiction, the result of the intersection is a contradiction. If only one branch produces a contradiction, the result of the intersection is the result of other branch. In any case, if the new state is not contradictory, it contains at least the initial state ($S \subseteq S'$). If this inclusion is strict ($S \subset S'$) we have gained information.

The dilemma rule is iterated on all variables till a contradiction is reached or no more information is gained. This gives us an execution that looks like Figure 3. This process can

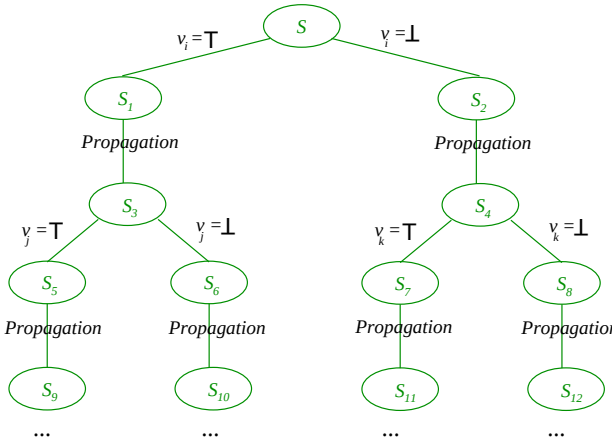

Fig. 1. Case splitting

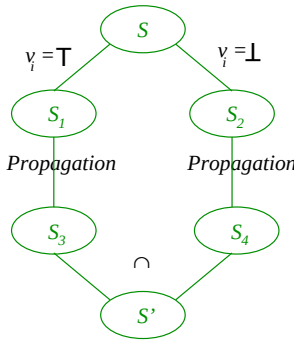

Fig. 2. The dilemma rule

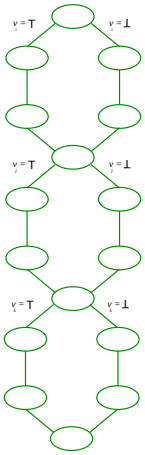

Fig. 3. Iterating the dilemma rule

be nested. At Level n, if $n = 0$ we call the propagation after the branching as in Figure 2, otherwise we recursively call the process at Level $n-1$ after the branching. The nice property of Stålmarck's algorithm is that Level 2 is sufficient to detect most tautologies.

3 Deriving an implementation

The implementation that we are going to use for our integration is a slightly different version of the implementation described in [9]. This implementation was only executable outside Coq using the extraction mechanism [11]. Here we want to be able to compare the different approaches of integrating the algorithm. So the implementation should be executable both inside *and* outside Coq. To get this, we face two main problems. First of all, the language of Coq is purely functional. So no side-effect is allowed. Second, functions must conform to a very strict and limited criterion of termination. In the following we explain how these aspects have been handled.

3.1 Functional implementation

As it is usually the case, signed variables are implemented using integers: v_i is represented by i and $-v_i$ by $-i$. The integer 1 has a special status and is used to represent $\top$ (and -1 represents $\bot$). In Coq we use the binary representation of numbers. This corresponds to the type *positive* whose definition is the following:

Inductive *positive* : *Set* :=
  `xO` : *positive* → *positive*
| `xI` : *positive* → *positive*
| `xH` : *positive*.

`xH` can be interpreted as 1, `(xO xH)` as 2, `(xI xH)` as 3, `(xO (xO xH))` as 4 and so on. To get our signed variables, we just need to add a sign to our *positive* objects:

Inductive *rZ* : *Set* :=
  `rZPlus` : *positive* → *rZ*
| `rZMinus` : *positive* → *rZ*

A triplet is represented as a set of three signed variables and a binary operation:

Inductive *triplet*: *Set* :=
  `Triplet` : *rBoolOp* → *rZ* → *rZ* → *rZ* → *triplet*

where *rBoolOp* is an enumerate type containing all the logical connectors.

A central implementation issue is how to represent the state of the algorithm, i.e the set of equations that are valid at a given moment. On this set of equations, we want to be able to decide quickly if two signed variables are equal. For this we use the same data-structure as in the union-find algorithm [12] but without aliasing. We associate to each variable the smallest variable to which it is equal. Let us take an example with 5 variables and the following set of equations $\{2 = -3, 3 = -1, 4 = 5\}$. We have the following list of associations: $\{1 \mapsto 1, 2 \mapsto 1, 3 \mapsto -1, 4 \mapsto 4, 5 \mapsto 4\}$. Two variables are equal if they point to the same elements modulo an elementary rule of signs. The association list alone does not contain enough information to be able to add new equations. We also need to keep some kind of back pointer information: which variables are pointing to a given variable. Since we do not allow aliasing, variables are either pointed to or point to. This means that the back pointer information can be kept in the corresponding field of the variables that are pointed to. In our example we have $\{1 \mapsto [2, -3], 2 \mapsto 1, 3 \mapsto -1, 4 \mapsto [5], 5 \mapsto 4\}$. Now if we want to add for example the equation $v_4 = -v_2$. Thanks to the association list we get that v_4 is minimal and that $-v_2$ points to $-v_1$. So adding the equation $v_4 = -v_2$ is equivalent to adding $v_4 = -v_1$. To do this we need to change the association list so that v_4 points to $-v_1$ and all the variables that were pointing to v_4 now point to v_1. This gives us the new association list: $\{1 \mapsto [2, -3, -4, -5], 2 \mapsto 1, 3 \mapsto -1, 4 \mapsto -1, 5 \mapsto -1\}$.

To implement association lists in Coq, we use functional arrays. As arrays may not be initialised, we first need to define a type to handle uninitialised values:

Inductive *Option*[*A* : *Set*]: *Set* :=
  `Some` : *A* → (*Option A*)
| `None` : (*Option A*)

With this definition, `None` represents an uninitialised value and (`Some` *A a*) represents the initialised value a of type *A*. The next step is to define binary trees:

```
Inductive Tree[A : Set]: Set :=
  rEmpty: (Tree A)
| rSplit: (Option A) → (Tree A) → (Tree A) → (Tree A)
```

We use the binary representation of our variables to implement our functional arrays. To find the value of an index we use its binary representation as a path in the tree:

```
Fixpoint rTreeGet [A : Set, t : (rTree A), r : positive] : (Option A) :=
  Cases r t of
          _       rEmpty       ⟹ (None A)
    |    xH     (rSplit a _ _) ⟹ a
    |  (xO r')  (rSplit _ t' _) ⟹ (rTreeGet A t' r')
    |  (xI r')  (rSplit _ _ t'') ⟹ (rTreeGet A t'' r')
  end.
```

Similarly we define the update of an array as follows:

```
Fixpoint rTreeSet [A : Set, t : (rTree A), r : positive] : A → (rTree A) :=λa: A.
  Cases r t of
       xH       rEmpty      ⟹ (rSplit A (Some A a) (rEmpty A) (rEmpty A))
    |  xH   (rSplit _ t'1 t'2) ⟹ (rSplit A (Some A a) t'1 t'2))
    | (xO r')    rEmpty      ⟹
                    (rSplit A (None A) (rTreeSet A (rEmpty A) r' a) (rEmpty A))
    | (xO r') (rSplit b t'1 t'2) ⟹
                    (rSplit A b (rTreeSet A t'1 r' a) t'2)
    | (xI r')    rEmpty      ⟹
                    (rSplit A (None A) (rEmpty A) (rTreeSet A (rEmpty A) r' a))
    | (xI r') (rSplit b t'1 t'2) ⟹
                    (rSplit A b t'1 (rTreeSet A t'2 r' a))
  end.
```

The use of functional arrays has a cost in term of efficiency. The access of an element in the array is not constant but linear to the length of the index in its binary representation. This is the prize to pay for being purely functional.

3.2 Termination

All functions that can be defined in Coq must terminate. In order to be able to evaluate these functions inside Coq, we must use the Fix constructor only. This constructor has a very restricted criterion to ensure termination: one specific argument of the function must strictly decrease in size in all the recursive calls. Defining functions so they conform to this criterion can be problematic sometime. For example in our development we need to be able to append two ordered lists. We have an arbitrary type A, on which a function *ltC* of type $A \to A \to bool$ is available to compare two elements of A, i.e. $(ltC\ a\ b) = \texttt{true}$ means $a \leq b$. One would naturally write the *append* function as:

```
Fixpoint append [l1 : (list A), l2 : (list A)] : (list A) :=
```

```
Cases l1 l2 of
        nil          _       ⇒ l2
  |      _          nil      ⇒ l1
  |  (cons a t1)  (cons b t2) ⇒
       Cases (ltC a b) of
         true ⇒ (cons a (append t1 l2))
       | false ⇒ (cons b (append l1 t2))
       end
end.
```

Unfortunately this definition is not accepted by Coq since in one recursive call it is the first argument that structurally decreases while in the other it is only the second one. So there is not a single argument of the function that decreases at each recursive call. To overcome this problem, we need to define locally a recursive function that handles one part of the recursion. This is done using the **Fix** command:

```
Definition append:
  Fix aux1{aux1/1 : (list A) → (list A) → (list A) := λl1, l2: (list A).
    Cases l1 l2 of
            nil          _       ⇒ l2
      |      _          nil      ⇒ l1
      |  (cons a t1)  (cons b t2) ⇒
           Cases (ltC a b) of
              true ⇒ (cons a (aux1 t1 l2))
            | false ⇒
                let f =
                  Fix aux2{aux2/1 : (list A) → (list A) := λl3: (list A).
                      Cases l3 of
                              nil    ⇒ l1
                        | (cons c t3) ⇒
                            Cases (ltC a c) of
                               true ⇒ (cons a (aux1 t1 l3))
                             | false ⇒ (cons c (aux2 t3))
                      end
                  }
                in (f l2)
    end
  }.
```

In this last definition both functions aux_1 and aux_2 have their first argument decreasing structurally in the recursive call.

This function *append* is actually the only definition where the structural argument of termination was problematic. For the overall algorithm, finding a simple argument of termination is quite direct. If we look at the different steps described in Section 2 all the iterations are done till no new information is gained. But if n is the number of variables, there will never be more than n iterations. So all these functions are defined recursively on an integer that is initially set to n.

3.3 Running the algorithm

The algorithm works directly on boolean expressions. These expressions are defined as follows:

Inductive *Expr*: *Set* :=
  `V` : *positive* → *Expr*
  `N` : *Expr* → *Expr*
  `Node` : *boolOp* → *Expr* → *Expr*

where *boolOp* is the type of the boolean connectors:

Inductive *boolOp*: *Set* :=
  `And` : *boolOp*
  `Or` : *boolOp*
  `Imp` : *boolOp*
  `Eq` : *boolOp*

On these expressions, we have defined a predicate *Tautology* that indicates if a boolean expression is a tautology. The algorithm in Coq is defined by the function *run*. Its type is: $nat \rightarrow Expr \rightarrow (rArray\ vM) * bool * (list\ rZ)$. The first integer indicates what is the level of nesting, i.e. if it is set to 0 we get only propagation, set to 1 the dilemma rule on one variable. The second argument is the boolean expression. The result is composed of 3 elements. The first one is the final association list. The second one is a boolean that indicates if we have reached a contradiction. The third one is the list of variables which values have been modified by the execution. The correctness of the algorithm is stated by the following theorem:

Theorem *runCorrect*:
  ∀*n*: *nat*. ∀*e*: *Expr*.
    Cases (*run n e*) **of**
      (_, *true* , _) ⟹ (*Tautology e*)
    | (_, *false*, _) ⟹ *True*
    end.

If the boolean that is returned is true then we can conclude that our expression is a tautology. If not, we cannot say anything. We can evaluate this function directly inside Coq. So for example if we want to check if the expression $v_2 \vee \neg v_2$ is a tautology, we enter the following command in Coq:

```
Eval Compute in  Cases (run 0 (Node Or (V (xO xH)) (N (V (xO xH))))) of
  (triple _ b _) => b end.
= true : bool
```

Table 1 shows how the algorithm runs internally and externally on standard examples. The machine we have used is a 750Mhz PC-linux machine. For each example we give the nested level of the formula, the number of its connectors, the number of its variables, the time and space required to execute the extracted algorithm, the time and space required to execute the algorithm inside Coq.

These benchmarks show first that our implementation is not very efficient. The extracted version is in a factor of 100 slower than the commercial product developed by Prover Technology. Our heuristics to select dilemma variables are very naive and could be largely improved.

name	level	nodes	variables	extracted: time	extracted: size	internal: time	internal: size
puz002_1	0	21	11	0m0.011s	1M824	0m02.120s	39M224
syn058_1	0	55	16	0m0.010s	1M824	0m02.570s	42M580
syn063_1	0	157	5	0m0.010s	1M824	0m05.510s	66M012
spoor4_neg	0	412	181	0m0.040s	1M828	0m08.490s	83M848
spoor2_neg	0	437	187	0m0.020s	1M828	0m05.540s	45M248
spoor3_neg	0	455	190	0m0.020s	1M828	0m05.600s	42M696
puz012_1	0	1427	144	0m0.290s	2M640	0m30.060s	100M824
syn029_1	1	9	3	0m0.010s	1M804	0m01.830s	39M220
transp_be	1	21	8	0m0.010s	1M824	0m02.700s	41M792
ztwaalf1_be	1	111	15	0m0.240s	1M824	2m24.010s	39M220
puz030_1	1	221	25	0m0.020s	1M828	0m11.830s	81M464
aim50_1_6_no3	1	239	50	0m0.310s	1M828	2m14.020s	82M220
hostint1_be	1	257	10	0m0.100s	1M828	0m56.160s	82M204
dk17_be	1	327	65	0m1.290s	2M088	9m53.850s	86M072
rip04_be	2	97	19	0m00.730s	1M824	5m33.760s	255M
u100	2	199	100	0m12.170s	1M836	$<$ 20m	$<$ 760M
aim50_1_6_no1	2	239	50	0m01.170s	1M818	$<$ 20m	$<$ 760M
dubois20	2	479	60	0m19.580s	2M092	$<$20m	$<$ 760M
u200	2	399	200	0m59.830s	1M848	$<$ 20m	$<$ 760M
aim100_1_6_no2	2	479	100	0m04.300s	2M092	$<$ 20m	$<$ 760M
dubois25	2	599	75	0m41.700s	2M352	$<$ 20m	$<$ 760M

Table 1. Run benchmarks

Second, the evaluation inside Coq is even slower and requires a fair amount of memory. We are only able to run the smallest Level 2 example. The other ones are outside our reach.

3.4 Getting a tactic

From the algorithm inside Coq, it is possible to derive a tactic. For this, we need to fill the gap between boolean formulae and propositions. This is done using the standard two-level approach. We first define a translation function from our boolean expressions to the propositions. This translation takes as a parameter a function that interprets boolean variables as propositions.

Fixpoint $ExprToProp\,[f : positive \to Prop,\ e : Expr] :\quad Prop :=$
  Cases e **of**
    $(\texttt{V}\ n) \Longrightarrow (f\ n)$
  | $(\texttt{N}\ e') \Longrightarrow \neg(ExprToProp\ f\ e')$
  | $(\texttt{Node And}\ e'_1\ e'_2) \Longrightarrow (ExprToProp\ f\ e'_1) \wedge (ExprToProp\ f\ e'_2)$
  | $(\texttt{Node Or}\ e'_1\ e'_2) \Longrightarrow (ExprToProp\ f\ e'_1) \vee (ExprToProp\ f\ e'_2)$

| (Node Imp e'_1 e'_2) $\Longrightarrow$ (*ExprToProp* f e'_1) $\Rightarrow$ (*ExprToProp* f e'_2)
| (Node Eq e'_1 e'_2) $\Longrightarrow$ (*ExprToProp* f e'_1) $\Longleftrightarrow$ (*ExprToProp* f e'_2)

end.

Now, using the excluded-middle as an axiom, it is possible to show that if an expression is a tautology, its translation as a proposition is true. This is given by the following theorem:

Theorem *ExprToPropTautology*:
$\forall f$: *positive* $\rightarrow$ *Prop*. (f xH) $\Rightarrow$ $\forall e$: *Expr*. (*Tautology* e) $\Rightarrow$ (*ExprToProp* f e)

The extra condition (f xH) comes from our encoding, we have to ensure that v_1 is interpreted as *True*.

A tactic StalRun can now be derived, It works as follows. It first takes the current goal and transforms it into a boolean equivalent. In this transformation it keeps a data-structure that maps the boolean variables to their actual value as a proposition. It then builds a proof term using this boolean expression and the theorems *runCorrect* and *ExprToPropTautology*. Let us see how it works in practice. Consider the following goal:

```
Require StalRun.

Theorem test: (a:Prop) a \/~a.
```

Applying the StalRun tactic ends the proof:

```
 test < StalRun.
Subtree proved!

 test < Qed.
```

We can display the proof term:

```
Print test.
test =
[a:Prop]
 (ExprToPropTautology
   (rArrayGetP (rArraySetP (rArrayInitP [_:positive]True) (xO xH) a)) I
   (Node Or (V (xO xH)) (N (V (xO xH))))
   (runCorrect (S (S O)) (Node Or (V (xO xH)) (N (V (xO xH))))))
     : (a:Prop)a\/~a
```

It is composed of an application of the theorem *ExprToPropTautology*. The first argument is the mapping function from boolean variables to propositions. It is given by a look-up function inside a particular array (*rArrayGetP*). This array is built as follows. First, an empty array is created (*rArrayInitP*) that gives the interpretation *True* to all the variables. Second, the value a is associated to the variable (xO xH) (*rArraySetP*). The second argument of *ExprToPropTautology* is a proof that (f xH) holds. Since (f xH) evaluates to *True*, I, the proof of *True*, is given. The third argument is the boolean expression. Finally the last argument is a proof that the boolean expression is a tautology. This is given by an appropriate application of the theorem *runCorrect* with a nested level of 2 and the boolean expression.

4 The 3-Level Approach

Table 1 shows that using directly the algorithm inside Coq cannot be a reasonable solution for integrating the algorithm. The performance of the programming language inside Coq being limited, we need to reduce as much as possible the computation done inside Coq. The basic idea of this section is to define a notion of execution trace that contains information that can be used as hints to replay more efficiently the same computation. The proof process is split in two. The algorithm is run externally and returns an execution trace. Then, the trace is checked internally inside the prover. This last step should be algorithmically simpler than running the initial algorithm.

4.1 Traces

In order to figure out what a good trace could be for the algorithm, we need to find places in the algorithm where useless computation may be avoided.

A first idea is to record only the successful applications of the dilemma rule, i.e applications where after the intersection new information has been gained. The other applications do not contribute to the final result. This gives us a first possible datatype for the traces:

```
Inductive Trace: Set :=
    emptyTrace : Trace
|     seqTrace : Trace → Trace → Trace
| dilemmaTrace : rZ → Trace → Trace → Trace.
```

In the following we refer to this version as the version `v`.

Another information that can be recorded is which triplets in the propagation effectively gain some information. The fact that we do not need to record the exact rule that has been applied has been proved in our formalisation: two rules that are applicable on the same triplets always generate the same information. Recording the triplet gives us a second possible data-structure (version `v+t`):

```
Inductive Trace: Set :=
    emptyTrace : Trace
| tripletTrace : triplet → Trace
|     seqTrace : Trace → Trace → Trace
| dilemmaTrace : rZ → Trace → Trace → Trace.
```

Finally if we look at the basic operations, the most expensive one is the computation of the intersection at the end of the dilemma rule. The information of which new equations have been gained can be stored in the trace. So, when checking the trace, it is sufficient to verify that these new equations hold in the resulting states of the two branches so we can add them to the initial state of the dilemma rule. Adding this information to the two previous traces gives a third possibility (version `v+i`):

```
Inductive Trace: Set :=
     emptyTrace : Trace
|      seqTrace : Trace → Trace → Trace
|  dilemmaTrace : rZ → (list rZ * rZ) → Trace → Trace → Trace.
```

and a forth one (version `v+t+i`):

```
Inductive Trace: Set :=
     emptyTrace : Trace
|  tripletTrace : triplet → Trace
|      seqTrace : Trace → Trace → Trace
|  dilemmaTrace : rZ → (list rZ * rZ) → Trace → Trace → Trace.
```

With any of these traces, it is then possible to define inside Coq a function *checkTrace* that verifies a trace. Its type is $Expr \rightarrow Trace \rightarrow (rArray\ vM) * bool$. The correctness theorem associated with the function that checks traces is the following:

```
Theorem checkTraceCorrect:
  ∀e: Expr. ∀t: Trace.
    Cases (checkTrace e t) of
      (_, true ) ⟹ (Tautology e)
    | (_, false) ⟹ True
    end.
```

This technique of generating a trace is not new and has already been used in different context for example recently in [5]. An external program sometimes called *oracle* is used to generate the trace. The originality of our approach is that we are going to verify that our external program generates correct traces. This is done by first modifying our previous algorithm so it produces traces. The type of the new *run* function is then $nat \rightarrow Expr \rightarrow (rArray\ vM) * bool * (list\ rZ) * Trace$. The theorem of correctness remains unchanged:

```
Theorem runCorrect:
  ∀n: nat. ∀e: Expr.
    Cases (run n e) of
      (_, true , _, _) ⟹ (Tautology e)
    | (_, false, _, _) ⟹ True
    end.
```

But we also prove formally that the trace that is produced is correct:

```
Theorem runTraceCorrect:
  ∀n: nat. ∀e: Expr.
    Cases (run n e) of
      (_, true , _, T) ⟹   Cases (checkTrace e T) of
                             (_, b) ⟹ b = true
                           end
    | (_, false, _, _) ⟹ True
    end.
```

It is then possible to use the extracted version of this algorithm *run* as the oracle for gen-

erating the trace. The main benefit of this 3-Level approach is that we ensure that only correct traces are generating. Also interfacing the extracted program to play the role of the oracle is straightforward since the trace produced by the program is exactly the one needed by the function that checks the trace. For this we just wrote few lines of Ocaml code. This capability of calling any extracted code from the Coq system could also be automatically generated during the process of extraction.

4.2 Getting a tactic

In a similar way than in Section 3.4, we can derive a tactic `StalTac` that uses the 3-Level approach. It proceeds initially as the tactic `StalRun`. First it finds the boolean equivalent of the formula. Then it calls externally the extracted program *run* that produces a trace. With this trace it builds a proof term using the *checkTrace* program. If we consider the same goal as in Section 3.4:

```
Require StalTac.

Theorem test: (a:Prop) a \/~a.
 test < StalTac.
Subtree proved!

 test < Qed.
```

Now if we display the proof term, it makes explicit call to the *checkTraceCorrect* theorem and contains a trace composed of a single triplet application:

```
Print test.
test =
[a:Prop]
 (ExprToPropTautology
   (rArrayGetP (rArraySetP (rArrayInitP [_:rNat]True) (xO xH) a)) I
   (Node Or (V (xO xH)) (N (V (xO xH))))
   (checkTraceCorrect (Node Or (V (xO xH)) (N (V (xO xH))))
     (tripletTrace
       (Triplet Or (rZPlus (xI xH)) (rZPlus (xO xH))
         (rZMinus (xO xH))))))
```

Tables 2 and 3 give the performance in term of time and space on the different versions. The benefit of having a trace becomes already clear at Level 1. At Level 2, it is now possible to prove all the examples. This was impossible with the initial algorithm. Unfortunately the benefit in term of space is not so big. For the relative performances of the different kind of traces, we can see that they vary from one example to another. If we look more specifically to the examples of Level 2, the more informative the trace is the better it performs. This confirms the fact that computation should be minimised inside Coq.

name	v	v+t	v+i	v+t+i
puz002_1	0m02.010s	0m01.770s	0m02.300s	0m01.670s
syn058_1	0m02.530s	0m02.230s	0m03.040s	0m02.000s
syn063_1	0m07.200s	0m04.200s	0m08.420s	0m03.900s
spoor3_neg	0m06.080s	0m06.290s	0m07.180s	0m05.770s
spoor2_neg	0m05.970s	0m06.060s	0m06.830s	0m05.850s
spoor4_neg	0m10.660s	0m08.880s	0m12.060s	0m08.210s
puz012_1	0m37.070s	0m27.890s	0m42.140s	0m26.960s
syn029_1	0m01.730s	0m01.750s	0m01.710s	0m01.600s
transp_be	0m02.730s	0m02.320s	0m02.710s	0m01.970s
ztwaalf1_be	0m42.660s	0m25.990s	0m41.890s	0m21.550s
puz030_1	0m14.900s	0m09.960s	0m12.730s	0m07.260s
aim50_1_6_no3	0m12.990s	0m10.790s	0m12.760s	0m08.010s
hostint1_be	1m20.460s	0m50.680s	0m59.770s	0m24.210s
dk17_be	4m32.830s	2m48.380s	3m27.930s	1m23.240s
rip04_be	0m34.210s	0m20.350s	0m31.800s	0m13.690s
u100	0m37.240s	0m29.420s	0m37.340s	0m33.900s
aim50_1_6_no1	0m20.240s	0m16.980s	0m17.190s	0m13.540s
dubois20	2m27.640s	1m50.280s	1m36.010s	1m19.730s
u200	1m48.820s	1m37.590s	1m50.550s	1m41.780s
aim100_1_6_no2	1m38.160s	1m13.870s	1m16.480s	1m01.190s
dubois25	3m39.910s	2m52.150s	2m38.390s	2m11.630s

Table 2. Trace benchmarks (Time)

5 Conclusions

The first conclusion we can draw from these experiments is that the only realistic integration is the one that would use reflection. Figures given in Tables 1 and 2 show that any other solution is largely out-performed by the extracted code. This does not come as a surprise. We are comparing the naive interpretor of Coq with the highly-optimised compiler of Ocaml. What is more surprising is the actual gap. From our experiments, it appears that one second of running time (for the extracted code) is the maximum computation we can get inside Coq. This is bad news.

It is not clear at the moment how reflection could be effectively used in Coq. A promising approach is the one presented in [1, 2] where the kernel of the Coq system has been formalized and proved correct in Coq itself. This could provide a way of bootstrapping the system: the code of the prover would be obtained by extracting algorithms formalized inside Coq. In that case, adding safely new code to the system would be straightforward.

The second conclusion is that the mix integration using traces is for the time being the best we can get. Results given in Tables 2 and 3 show that following this approach it is possible to prove tautologies of significant size. We have proposed a new approach to do this integration by developing directly inside the prover the program that is generating the

name	v	v+t	v+i	v+t+i
puz002_1	36M860	38M468	36M752	34M112
syn058_1	41M476	39M244	36M760	34M888
syn063_1	61M084	46M168	62M776	44M372
spoor3_neg	67M632	70M008	63M104	65M396
spoor2_neg	63M792	64M888	62M060	64M880
spoor4_neg	78M636	71M284	76M908	68M208
puz012_1	122M716	115M916	121M356	109M184
syn029_1	36M344	24M308	27M272	27M984
transp_be	40M696	38M208	40M076	32M756
ztwaalf1_be	75M924	84M816	72M992	74M444
puz030_1	75M196	68M472	73M324	61M812
aim50_1_6_no3	76M232	73M076	67M668	65M136
hostint1_be	86M176	93M256	76M860	78M660
dk17_be	105M464	152M200	81M768	121M696
rip04_be	75M648	79M944	73M500	71M364
u100	85M080	87M824	78M336	77M068
aim50_1_6_no1	78M028	78M964	73M064	68M464
dubois20	90M372	99M240	87M196	85M412
u200	102M228	107M752	92M148	93M412
aim100_1_6_no2	90M628	104M616	86M688	89M764
dubois25	97M144	109M248	94M156	90M044

Table 3. Trace benchmarks (Size)

trace. Doing so, we can ensure that the traces that are generating are always valid, i.e. every time the program recognises a tautology it will be accepted by the checker. As we said the integration with the extracted code is straightforward since the data-structures that are manipulated are similar to the ones used by the checker. It would even be possible to extend the extraction mechanism so to automatically generate the code to call the extracted code directly inside Coq. With this extension implementing such a mixed tactic could be done without any knowledge of the internal structure of the Coq system.

What we have done may seem like overkill: we have used a program that is proved correct to generate a trace that is then checked. Note that strictly speaking only the theorem *checkTraceCorrect* is needed. So we could have spared the effort of proving the other theorems Also the constrain for the program *run* to be runnable inside Coq is unnecessary. We did it only in order to be able to do the comparison with the other approaches.

A drawback of our approach is that we need to develop our own implementation of the algorithm to generate traces. Modifying an existing algorithm and adding some trace information is a seducing alternative. Most probably such an implementation would be highly optimised. However, this is feasible only if the existing implementation already generates some kind of traces, which is rare, or if we have access to its code. Furthermore, adding traces could not be so easy and have a cost in term of development and efficiency. Also, it

is necessary to establish a connection between the trace generator and the prover. In our approach all these problems do not exist.

Finally with respect to the efficiency of the trace generator, the results given in Table 2 shows that the bottle neck of the mix integration is the checking part. Having a more sophisticate trace generator would not improve much our performance.

Another aspect that could be worth investigating is the quality of the traces we produce. Our traces may still contain information that are not necessary for showing that the formula is a tautology. Our criterion was to record operations when some new information is gained but nothing tells us that this new information is needed to reach the contradiction. Solutions developed in systems like GRASP [10] are clearly relevant here. They track the source of conflict in the area of boolean satisfiability. Also, it could be possible to find by exhaustive search the minimal traces (in term of numbers of applications of dilemma rules and\or of propagation rules). This would give us an idea of which performance could ultimately be obtained using traces.

Finally our comparisons still need to be completed. We are missing one possible way of integrating safely the algorithm. This is by means of tactics. Tactics provide a complete programming language to build proof terms. The idea is to translate each step of computation by some proof steps. This approach has already been followed in [7]. The proof terms are usually very large but easy to check.

References

1. Bruno Barras. Verification of the Interface of a Small Proof System in Coq. In E. Gimenez and C. Paulin-Mohring, editors, *Workshop on Types for Proofs and Programs*, pages 28–45, Aussois, France, December 1996. Springer-Verlag LNCS 1512.
2. Bruno Barras. *Auto-validation d'un système de preuves avec familles inductives*. Phd, University Denis Diderot, November 1999.
3. Gilles Barthe, Mark Ruys, and Henk Barendregt. A Two-Level Approach Towards Lean Proof-Checking. In *TYPES'95*, volume 1158 of *LNCS*, pages 16–35, 1995.
4. Samuel Boutin. Using Reflection to Build Efficient and Certified Decision Procedures. In *TACS'97*, volume 1281 of *LNCS*, pages 515–529, Sendai, Japan, 1997.
5. Olga Caprotti and Martijn Oostdijk. Formal and efficient primality proofs by use of Computer Algebra oracles. *Journal of Symbolic Computation*, 32(1/2):55–70, July 2001.
6. John Harrion. Metatheory and Reflection in Theorem Proving: A Survey and Critique. Technical Report CRC-053, SRI International Cambridge Computer Science Research Centre, 1995.
7. John Harrison. Stålmarck's algorithm as a HOL derived rule. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, number 1125 in LNCS, pages 221–234, Turku, Finland, August 1996. Springer-Verlag.
8. Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. The Coq Proof Assistant: A Tutorial: Version 6.1. Technical Report 204, INRIA, 1997.
9. Pierre Letouzey and Laurent Théry. Stålmarck's Algorithm in Coq. In *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs'00*, number 1869 in LNCS, pages 387–404, Portland, Oregon, September 2000. Springer-Verlag.

10. Joao P. Marques-Silva and Karem A. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 220–227, November 1996.
11. Christine Paulin-Mohring and Benjamin Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15(5-6):607–640, 1993.
12. Robert Sedgewick. *Algorithms in C*, pages 441–449. Addison-Wesley, 1990.
13. Mary Sheeran and Gunnar Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. In *FMCAD '98*, volume 1522 of *LNCS*. Springer-Verlag, November 1998.

A Propagation rules for the triplet $v_i := v_j \,\&\, v_k$

(1) **if** $v_i = -v_j$, **propagate** $v_j = \top$ **and** $v_k = \bot$
(2) **if** $v_i = -v_k$, **propagate** $v_j = \bot$ **and** $v_k = \top$
(3) **if** $v_j = v_k$, **propagate** $v_i = v_k$
(4) **if** $v_j = -v_k$, **propagate** $v_i = \bot$
(5) **if** $v_i = \top$, **propagate** $v_j = \top$ **and** $v_k = \top$
(6) **if** $v_j = \top$, **propagate** $v_i = v_k$
(7) **if** $v_j = \bot$, **propagate** $v_i = \bot$
(8) **if** $v_k = \top$, **propagate** $v_i = v_j$
(9) **if** $v_k = \bot$, **propagate** $v_i = \bot$

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399