A Formally-Proven Algorithm for 2-Sat Problems

Laurent Théry
Laurent.Thery@sophia.inria.fr

Abstract

This notes explains how an algorithm that checks the satisfiability of a set of two clause has been formalised in the Coq prover using the SSReflect extension.

1 Introduction

Sat is a well-known problem in computer science. In this paper, we are interested by one of its instance: the 2-Sat problem. More precisely, we consider a set V of variables. Taking the variables $v_i \in V$ and their negation $\neg v_i$ forms the set L of literals. A 2-clause is composed of two literals $\{l_1, l_2\}$ and represents the logical formula $l_i \vee l_j$. A 2-clause is then satisfied if at least of its literal is true. A 2-Sat problem P is composed of a set of 2-clauses $\{C_1, \ldots, C_n\}$ and represents the logical formula, $C'_1 \wedge \cdots \wedge C'_n$ where C'_i is the logical formula associated with the 2-clause C_i. The problem is to decide whether P is satisfiable or not. In other words, the problem is to find an assignment e of the variables of V such that at least one of the literals of each 2-clause of P is satisfied by e. Let us take a concrete example and consider the set $S = \{\{x_1, x_2\}, \{x_1, \neg x_2\}, \{\neg x_1, x_2\}, \{\neg x_1, \neg x_2\}\}$. Four clauses and two variables compose this problem. So, one simply needs to check if any of the four possible assignments of these two variables satisfies the logical formula $(x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$. One can easily check that each of these assignments is refuted by exactly one of the 2-clauses. For example, $\{v_1 \mapsto \top, v_2 \mapsto \top\}$ does not satisfy $(\neg x_1 \vee \neg x_2)$. This formula is then unsatisfiable.

Satisfiability for 2-SAT problems can be solved in linear time. This is a classical result. It is usually shown by translating the initial problem P into another classic graph problem: computing strong-connected components. Standard graph algorithms in linear-time for this problem exist, Tarjan's one [4] or Kosaraju's one [3] to cite only two. Here, we are going to follow a more direct path. We formally prove the correctness of an algorithm that uses a variation of some standard SAT solving technique: the *unit propagation*. No detour into graph theory is then needed in order to get the linearity result. We state our algorithm in a functional setting using the SSREFLECT extension of the COQ theorem prover. The main inspiration for this algorithm comes from a paper by Alvaro del Val [2].

2 Unit Propagation

The semantic of logical formula is usually given in term of truth-tables. They consist in computing the truth value of a formula for each possible assignment. If there are n variables, 2^n assignments are to be considered. Let us try to concretize the problem and give a program that checks satisfiability. We first consider a set of clauses C and an assignment E and try to derive a naive algorithm that check satisfiability. Two auxiliary functions are needed. First, the function *unassigned* checks if an assignment has some unassigned variables if so it returns one of them. This is done in our functional setting by using an option type: `Some` v indicates that the variable v has no value yet and `None` indicates that all variables have a value. The second function *eval_clauses* evaluates a set of clause with respect to an assignment. It returns `SAT` if the assignment satisfies the set of clauses. It is the case when there exists at least one literal in each clauses that has a true value by the assignment. The function returns `UNSAT` otherwise. Now, the function *check_sat* uses recursion to explore all the possible assignments stopping at the first one that satisfies the set of clauses if it exists.

```
Function check_sat C E :=
  if unassigned E is Some v then
    if check_sat C (E + {v := ⊤}) is SAT then SAT
    else check_sat C (E + {v := ⊥})
  else eval_clauses C E
```

An immediate improvement is to extend the evaluation of clauses to partial assignments where some variables can have unknown values. The value returns by *eval_clauses* C E can be of three kinds. It is `SAT` if each clause of C has at least one literal that is true for E. It is `UNSAT` if one clause of C has all its literals that are false for E. Finally, it is `UNKNOWN` (C', v) when C' is the subset of C that contains all the clauses that are yet to be satisfied and v an unassigned variable.

```
Function check_sat C E :=
  let r := eval_clauses C E in
  if r is UNKNOWN (C', v) then
     if check_sat C' (E + {v := ⊤}) is SAT then SAT
     else check_sat C' (E + {v := ⊥})
  else r
```

Calling *eval_clauses* at every recursive call has two benefits. First, it detects early successes because as just one literal needs to be true, any completion of the partial assignment would satisfy the set of clauses. Second, it also detects dead-ends because if a clause has all its literals set to false there is no way the set of clauses can be satisfiable by refining the current partial assignment. These early detections insure that the partial assignment is in fact a pre-model of the formula we want to satisfy.

A further refinement is to carefully choose the variable returns by `UNKNOWN`. First, note that such a variable exists since *eval_clauses* has returned neither `SAT` nor `UNSAT`. Choosing the proper variable on which to do the splitting is a difficult problem. Nevertheless, there is one situation where this is easy. It is the case when a clause has all its variable set to false except one that is unknown. Choosing the variable in this literal is pertinent since it leads to a trivial splitting since the branch that sets this literal to false is immediately detected as unsatisfiable. A clause having such a property with respect to an assignment is called a unit clause. It is of course quite inefficient to generate such a trivial splitting, this is why these "forced" assignments are often collected in a single procedure *unit_propagation*. Note that this procedure is recursive since a new assignment of a variable can turn a clause that was not unit into a unit one. Calling *eval_clauses* C E can returned three possible result. It is `SAT` if each clause of C has at least one literal that is true for E modulo unit propagation. It is `UNSAT` if one clause of C

has all its literals that are false for E modulo unit propagation. Finally, it is `UNKNOWN` (E', C', v) when E' is E augmented of all the assignment found by unit propagation, C' is the subset of C of the clause that are not yet satisfied and v an unassigned variable. This leads to a new version of our checking algorithm.

```
Function check_sat C E :=
  let r := unit_propagation C E in
  if r is UNKNOWN (C', E', v) then
     if check_sat C' (E' + {v := ⊤}) is SAT then SAT
     else check_sat C' (E' + {v := ⊥})
  else r
```

What we have here is almost a naive version of the DPLL algorithm [1] that is the central part of most SAT solvers. The only missing feature is the pure literal simplification that is not relevant to the work presented here.

We have introduced unit propagation in the general setting. Now, we can explain what is specific to 2-SAT instances. First, unit propagation is very simple to implement. Suppose that we have a clause $\{l_i,\ l_j\}$ where both literals are not yet assigned. Assigning any of the two literals automatically either satisfies the clause or turns it into a unit clause. Second, when a propagation is successful and returns a subset C' of clauses, each clause of C' has both its literals unassigned. So, C' represents a sub-problem that can be solved independently. Only the immediate propagation can invalidate a branch of a split. In contrast, arbitrary deep backtracking can occur in the general setting. These observations lead to a tail-recursive version of the satisfiability checking for 2-SAT problems. We first define the recursive algorithm that finds an assignment for the variable v using the unit propagation.

```
Function rcheck_2sat C E v :=
  let r := unit_propagation C (E + {v := ⊤}) in
  if r is UNKNOWN (C', E', v') then rcheck_2sat C' E' v' else
     if  r is SAT then r else
      let r' := unit_propagation C (E + {v := ⊥}) in
      if r' is UNKNOWN (C'', E'', v'') then rcheck_2sat C'' E'' v'' else r'
```

Now, the top-level function just needs to call the previous function with an arbitrary unassigned variable.

```
Function check_2sat C E :=
  let r := unit_propagation C E in
  if r is UNKNOWN (C', E', v') then rcheck_2sat C' E' v' else r
```

Note that this version already shows that the time complexity for 2-SAT is not exponential anymore. Though, it is not sufficient to get linearity. It is possible to build a simple example that exhibits a quadratic behaviour. Let us consider the clauses $\{\neg x_i \vee x_{i+1} \mid i < n\} + \{\neg x_{n-1} \vee \neg x_n\}$. This problem is composed of Horn clauses (each clause has at most one positive literal) with no unit clause. So, the assignment that sets all the variables to false satisfies trivially the problem. Still, our naive program can have a hard time finding it. Suppose the heuristic for choosing the variable returned by the propagation is to select the unassigned variable with the smallest index. The program first tries to assign x_0 to $\top$, then it follows a chain of propagation to set all the x_i to true and discovers that there is a problem with x_{n-1}. The propagation having failed, so x_0 is assigned to $\bot$ with a trivial propagation that selects the variable x_1. The pattern of failed propagation is repeated for each variable $x_1, x_2, \ldots, x_{n-1}$. When trying to set the variable x_i to $\top$, a chain of propagation from x_i to x_n is necessary to invalidate it. This means that the clause $\neg x_0 \vee x_1$ is used once by propagation, $\neg x_1 \vee x_2$ twice and so on. Summing up, we get the quadratic behaviour.

The idea for getting a linear behaviour is to have a two-level propagation that is capable locally of erasing previous assignments. In order to do this, we extend the assignment of a variable to five values instead of the usual three: `U` for unknown, $\top^*$ for possibly true, $\bot^*$ possibly false, $\top$ for strongly true and $\bot$ for strongly false. Initially, the propagation assigns possible values ($\top^*$ or $\bot^*$). If there is no problem, the propagation terminates as usual. If there is a problem, this means that there is a variable x_i that, when supposed to have a possible value v^*, leads logically to the same variable having the value $\neg v^*$. This means that we are sure that x_i has strong value $\neg v$. So this new information can be propagated, possibly erasing possible values. Trying to erase a strong value ($\top$ or $\bot$) leads to report an unsatisfiable status. Let us show that the simple propagation of x_0 assigned to $\top^*$ on our example is almost sufficient to show that this formula is satisfiable. It works in two

steps. The first one is assigning possible values to all variables x_0, ..., x_n till a problem is found on x_{n-1}. Propagating the strong value $\bot$ to x_{n-1} erases all the possible values of x_0, ..., x_{n-1} and set them to $\bot$. Now, any value for v_n makes the formula satisfiable. Intuitively, a clause can be used at most two times during the checking problem: once to propagate possible value from one of its variable x_i to the other one x_j and another (possible) time to erase this information and to propagate in the opposite direction from x_j to x_i strong values.

The discussion so far has been very informal. In the rest of the paper, we propose a formal presentation of the algorithm and explains how it has been formally proved correct.

2.1 Formal algorithm

In this section, we explicit the algorithm. For this, we use the programming language provided by the Coq proof system with the SSReflect extension. We first need to define the data-structures. We first parametrise our development by a finite set V of variables and define literal as a pair of a boolean that indicates the polarity and a variable, clause as a pair of literals and clauses as a sequence of clauses. Then, we consider an arbitrary set of clauses *cls* that is used in the following to parametrise our development. Our goal is to decide if the set of clauses *cls* is satisfiable or not.

```
Variable V : finType.
Definition lit := bool × V.
Definition clause := lit × lit.
Definition clauses := seq clause.
Variable cls : clauses.
```

We first introduce a relation *imply* on literal that models logical implication. l_i *imply* l_j holds if for every assignment that satisfies *cls*, l_i is true then also l_j should be true. In term of clauses, this means that either $\{\neg l_i,\ l_j\}$ or $\{l_j,\ \neg l_i\}$ belongs to *cls*. Once, this relation is defined, we take advantage of the function `rgraph` of the finite graph library. Given a relation r and an element x, `rgraph` r x compute the sequence of all the elements y such that r x y. We use this function for the relation *imply*. Given a literal l_i that has been set to true, `rgraph` *imply* l_i returns the sequence of the literals that have also to be set to true by unit propagation.

```
Definition flip l := (¬ l.1, l.2).
Definition imply := [rel l1 l2 | (flip l1, l2) ∈ cls || (l2, flip l1) ∈ cls ].
Definition inext : lit → seq lit := rgraph imply.
```

The next ingredient is to define partial assignment. The five possible values are represented by an option type *val* on pairs of booleans. `U` is encoded as `None`, $\top^*$ as `Some (false, true)`, $\bot^*$ as `Some (false, false)`, $\top$ as `Some (true, true)` and $\bot$ as `Some (true, false)`. An assignment is then defined as a function from V to *val*. It is then straightforward to lift the evaluation and the update from variables to literals: *get* m l returns the value of the literal l in the assignment m, *set* m l r returns a new assignment that is an update of the assignment m where the literal l has value r.

```
Definition val := option (bool × bool).
Definition mem := {ffun var → val}.
Definition vflip : val → val := omap (fun v ↦ (v.1, ¬ v.2)).
Definition get m l : val := if l.1 then m l.2 else vflip (m l.2).
Definition set m l r : mem :=
  [ffun x ↦ if x = l.2 then (if l.1 then r else vflip r) else m x ].
```

The propagation works by mending all the places where the implication relation is violated. It takes two arguments, m the current assignment and s the stack of all the literals that have not yet been processed and should be set to true. At each step, the function tries to process the literal on top of the stack. The stack is not a simple sequence of literals but a sequence of pairs of a literal (that is at the origin of the propagation) and the sequence of its implied literals that have not yet been processed. Suppose the top of the stack s is $(l, l_1 :: ls_1) :: s_1$ and the assignment is m. The standard way of processing the literal l_1 is to create an assignment $m' = set\ m\ l_1$ (`Some` (b_1, *true*)) and a new stack $s' =$ `[::` (l_1, *inext* l_1), (l, ls_1) `&` s_1`]`. This is correct under two conditions. First, the propagation must still be valid, i.e. the literal l that is at the origin of the propagation for l_1 must still be true, *get* m $l =$ `Some(`b_1`, true)`. It means that no backtrack has occurred. Note that the mode b_1 of propagation (possible or strong) is stored in the value of the literal l. Second, the value of l' in m should be unknown. If it is already set to true, nothing has to be propagated. If it is possibly false, the mode

has to be changed and l' must be propagated as strongly true. If the value of l' is strongly false, the set of clauses is unsatisfiable.

```
Function propagate m s :=
  if s is (l, ls) :: s1 then
    if get m l is Some (b1, true) then
      if ls is l1 :: ls1 then
        if get m l1 is Some (b2, v) then
          if v then propagate m ((l, ls1) :: s1)
          else
            if b2 then (false, m)
            else
              let m1 := set m l1 (Some (true, true)) in
              propagate m1 [:: (l1, inext l1), (l, ls1) & s1 ]
        else
          let m1 := set m l1 (Some (b1, true)) in
          propagate m1 [:: (l1, inext l1), (l, ls1) & s1 ]
      else propagate m s1
    else propagate m s1
   else (true, m)
```

Note that this function has been implemented in such a way that the number of recursions gives a direct indication of the complexity of the propagation. We go even further in our formal proof and makes this number of recursions an explicit argument of the function. It is the argument that is ensuring termination. A call to *propagate* m s n performs n steps of propagation starting from the assignment m and the stack s. The returned value is either `None` if the number n of steps has been insufficient to complete the propagation or `Some` (b, m', n') where b indicates if the propagation has been successful, m' is the new assignment and n' the number of steps that have not been consumed ($n - n'$ steps have been sufficient to complete the propagation). This version of the propagation is given in Appendix A

In order to get our function that checks satisfiability, we choose a naive strategy and propagate to possibly true all the variables that are not assigned yet. We first define a function *check_satl* that performs the propagation for an arbitrary sequence *ls* of literals.

```
Function check_satl m n ls :=
  if ls is l :: ls₁ then
    if get m l is None then
      let m₁ := set m l (Some (false, true)) in
      if propagate n m₁ [:: (l, inext l)] is Some (true, n₂, m₂) then
        check_satl m₂ n₂ ls₁
      else None
    else check_satl m n ls₁
  else Some m.
```

Note that this function returns an option type on assignment. We get the final assignment if the set of propagation went fine otherwise `None` is returned either if one propagation finds the clauses unsatisfactory or the n parameter is not large enough to complete the computation.

Now, checking satisfiability is almost straightforward. The only difficulty is to give a large enough recursion level. This number is computed with respect to a current assignment m and a stack s. The code of *propagate* indicates the formula. Remember that we only propagate true values, possible ones or strong ones. We need the size of the sequence plus one (the empty list counts one) for each literal in the stack and the size of the sequence (*inext* l) plus one for the literal that are not assigned yet or assigned with a possibly false and could be rewritten into a strong true. This computation is done by the function *mweight*. The function *check_sat* calls the previous function *check_satl* with the empty assignment, a recursive weight given by *mweight* and the sequence of all variables (`enum` *var*) lifted to the positive literals.

```
Definition mweight m (s : seq (lit × seq lit)) :=

    ∑_{i ← s} (size i.2 + 1) + ∑_{l | l ∉ [seq i.1 | i ← s] && (get m l = None || get m l = Some (false, false))} (size (inext l) + 1).

Definition check_sat :=
  let m := [ffun x ↦ None] in
  let n := mweight m [::]  in
  check_satl m (n + 1) [seq (true, v) | v ← enum var ].
```

2.2 Formal proof

This algorithm is about propagating truth values. As we have seen, there are two flavours of truth: possibly true or strongly truth. We then define two predicates on literals with respect to an assignment: *is_true* m l tells that the literal l is true (possibly or strongly) for the assignment m and *is_true* m l that the literal l is strongly true for the assignment m.

```
Definition  is_true m l := oapp (fun x ↦ x.2) false (get m l).
Definition is_strue m l := oapp (fun x ↦ x.2 && x.1) false (get m l).
```

We use the application `oapp` for option type: `oapp` f d o applies f if o contains a value or returns d otherwise.

A propagation terminates when there is no more information that can be derived from the clauses. It can be captured by a predicate of well-formedness.

```
Definition wf m := ∀l1 l2, l2 ∈ inext l1 → is_true m l1 → is_true m l2.
```

During the propagation, it is the stack s that precisely indicates for which elements the well-formedness does not hold. This is in fact the key invariant of the propation. Suppose that l_1 is true but not l_2. There are two possibilities either the propagation $l_1 \mapsto l_2$ has not been performed yet and is in s or it has been performed but then l_2 has been proved to be strongly false and the proprogation $\neg l_2 \mapsto \neg l_1$ has not been performed yet.

```
∀l1 l2,
  (∀ls, (l1, ls) ∈ s → l2 ∉ ls) →
  (∀ls, is_strue m (flip l2) → (flip l2, ls) ∈ s → flip l1 ∉ ls) →
  l2 ∈ inext l1 → is_true m l1 → is_true m l2.
```

More has to be said about the stack if we want to prove this property as an invariant. The first fact is that pairs in the stack are well-formed: their second element is composed of literals that are implied by their first element.

```
∀l ls, (l, ls) ∈ s → ls ⊂ inext l
```

We define the notion of cycle. There is a cycle on l for the implication if $\neg l \mapsto^* l$. We use the `connect` relation defined in the finite graph and our *imply* relation.

```
Definition cycle l := connect imply (flip l) l.
```

We then require in our invariant that all strong values comes from a cycle

```
∀l, is_strue m l → cycle l
```

Now, we need to be more precise on the structure of the stack s. If we remember, this stack is a sequence composed of pairs. The pairs are composed of a literal and the sequence of its implied literals that have not been processed yet. The four remaining requirements is on the sequence composed of the first elements of the pairs, the originators, $s_1 = [\texttt{seq}\ i.1 \mid i \leftarrow s]$.

```
[∧ uniq s₁, istack s₁, ostack m s₁ &
    ∀l, l ∈ s₁ → is_true m l || is_strue m (flip l)].
```

The first two properties concern the structure of the literals. The sequence must be without duplication and composed of literals that are linked by implication. We use the `uniq` predicate of the sequence library to assert the lack of duplication. The linked nature of the sequence is stating by the fact that if we reverse the sequence, we get a path for the relation *imply*

```
Definition istack s :=
  if rev s is l :: ls then path imply l ls else true.
```

where `path` $r\ x\ s$ indicates that x `::` s is a path for the relation r. Note that if the order in which the propagation is performed is not relevant in

the general setting. Here, for 2-clauses, it is crucial that only one thread of implication is investigated at a time.

The third and forth properties correspond to the two modes of propagation. The predicate *ostack* indicates that the stack is in some sense ordered, the literals possibly true being after the ones strongly true.

```
Definition ostack m s :=
 let s_1 := [seq x ← s | is_true m x] in
  s_1 = [seq x ← s_1 | is_strue m x] ++ [seq x ← s_1 | ¬ is_strue m x ].
```

and the last property expresses that the value of a literal in s_1 is either true or has been swapped to strongly false.

Putting altogether all the properties we have described, we get the invariant of the propagation algorithms

```
Definition valid m s :=
 [∧
   let s_1 := [seq i.1 | i ← s ]) in
   [∧ uniq s_1, istack s_1, ostack m s_1 &
       ∀ l, l ∈ s_1 → is_true m l | is_strue m (flip l)],
   ∀ l ls, (l, ls) ∈ s → ls ⊂ inext l,
   ∀l_1 l_2,
       (∀ls, (l_1, ls) ∈ s → l_2 ∉ ls) →
       (∀ls, is_strue m (flip l_2) → (flip l_2, ls) ∈ s → flip l_1 ∉ ls) →
       l_2 ∈ inext l_1 → is_true m l_1 → is_true m l_2].
```

The first trivial property of this invariant is that if the stack is empty we get the well-formedness of our assignment.

```
Lemma wf_nil m : valid m [::] → wf m.
```

The next property states that if the propagation is positive from a valid stack, the assignment that is returned is well-formed.

```
Lemma valid_propage_true n m m1 s k :
  valid m s → propagate n m s = Some (true, k, m1) → valid m1 [::].
```

This is proved by induction following the code of the function propagate. Associated with this variant when the propagation succeeds, there is a variant that just says that the number of unknown in the assignment cannot decrease.

```
Lemma propage_reduce_none n k m m1 s l :
  propagate n m s = Some (true, k, m1) →
  get m1 l = None → get m l = None.
```

When the propagation fails, it can be proved quite directly that there is a cycle both for l and $\neg l$.

```
Lemma valid_propage_false n m m1 s k :
  valid m s → propagate n m s = Some (false, k, m1) →
  ∃ l, cycle l ∧ cycle (flip l).
```

Finally, when n is sufficiently large, the propagation must return an answer.

```
Lemma valid_propage_none n m s:
  valid m s → mweight m s < n → propagate n m s ≠ None.
```

More importantly, it is possible to show that successful propagation behaves well with respect to the weight of assignments.

```
Lemma valid_propage_red n k m m1 s :
  valid m s → propagate n m s = Some (true, k, m1) →
  mweight m s < n → mweight m1 [::] < k.
```

It is then possible to lift all the properties on the *propagate* function and prove the correctness of the *check_satl* function.

```
Lemma check_satl_correct m n ls :
  mweight m [::] < n → valid m [::] →
  if check_satl m n ls is Some m1 then
    [∧ valid m1 [::],
        ∀ l, l ∈ ls → get m1 l ≠ None &
        ∀ l, get m1 l = None → get m l = None]
  else ∃ l, cycle l ∧ cycle (flip l).
```

Now, in order to state the correctness of *check_sat*, we first have to define what it is for the set of clause *cls* to be satisfiable.

```
Definition valuation := {ffun var → bool}.
Definition eval (val : valuation) l := if l.1 then val l.2 else ¬ val l.2.
Definition satisfiable :=
  [∃val, ∀cl, cl ∈ cls → eval val cl.1 || eval val cl.2].
```

Now, thanks to the implicit conversion from option type to boolean that sends option value to true and none value to false, the correctness of the *check_sat* function can be simply stated as

```
Lemma check_sat_correct : satisfiable = check_sat.
```

Its proof directly follows from the correctness of *check_satl*.

In order to conclude the formalisation, we just need to prove the linearity of the algorithm. This can be done by providing a bound for the *mweight* function. It says that the maximum number of iterations of the propagate function is no more that twice the number of clauses and the number of variables.

```
Lemma check_sat_mweight :
  mweight [ffun x ↦ None] [::] ≤ 2 × (size cls + #|var|).
```

The complete specification and the proofs of this formalisation are available at http://www-sop.inria.fr/marelle/Laurent.Thery/TwoSat.html

3 Conclusion

Formalizing this very nice algorithm that solves 2-SAT problems in linear time has been an interesting exercise. Defining a tail recursive version of the algorithm that explicitly manipulates a stack of literals was a key decision in our formalisation. First, it made it possible to give a simple concrete statement on the complexity of the algorithm with the *mweight* function. Second, the stack parameter was used in a very effective manner to state the invariant of the propagation. We believe that the result is not only a very concise functional implementation of the algorithm but also a relatively simple formal proof of its correctness.

References

[1] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[2] Alvaro del Val. On 2-SAT and Renamable Horn. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence*, pages 279–284. AAAI Press / The MIT Press, 2000.

[3] Micha Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers and Mathematics with Applications*, 7:67–72, 1981.

[4] Robert Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1972.

A Propagation with step number

```
Function propagate n m s :=
  if n is n1. + 1 then
    if s is (l, ls) :: s1 then
      if get m l is Some (b1, true) then
        if ls is l1 :: ls1 then
          if get m l1 is Some (b2, v) then
            if v then propagate n1 m ((l, ls1) :: s1)
            else
              if b2 then Some (false, n, m)
              else
                let m1 := set m l1 (Some (true, true)) in
                propagate n1 m1 [:: (l1, inext l1), (l, ls1) & s1 ]
          else
            let m1 := set m l1 (Some (b1, true)) in
            propagate n1 m1 [:: (l1, inext l1), (l, ls1) & s1 ]
        else propagate n1 m s1
      else propagate n1 m s1
    else Some (true, n, m)
  else None
```