Formalizing Stålmarck's algorithm in Coq

Pierre Letouzey † and Laurent Théry ‡

† École Normale Supérieure, 45 rue d'Ulm, 75230 Paris France
`Pierre.Letouzey@ens.fr`
‡ INRIA, 2004 route des Lucioles, 06902 Sophia Antipolis France
`Laurent.Thery@sophia.inria.fr`

Abstract. We present the development of a machine-checked implementation of Stålmarck's algorithm. First, we prove the correctness and the completeness of an abstract representation of the algorithm. Then, we give an effective implementation of the algorithm that we prove correct.

1 Introduction

When formalizing an algorithm inside a prover, every single step has to be justified. The result is a presentation of the algorithm where no detail has been omitted. Mechanizing the proofs of correctness and completeness is the main goal of this formalization. Whenever such proofs are intricate and involve a large amount of case exploration, mechanized proofs may be an interesting complement to the ones on paper. Also, there is often a gap between an algorithm and its actual implementation. Bridging this gap formally and getting a reasonably efficient certified implementation is a valuable exercise.

In this paper we explain how this has been done for Stålmarck's algorithm [10] using the Coq prover [6]. This algorithm is a tautology checker. It is patented and has been successfully applied in industry. As it includes a number of heuristics, what we formalize is an abstract version of the algorithm. We prove different properties of the algorithm including correctness and completeness. We also cover two ways of ensuring that the result of an implementation is correct. We define execution traces and prove that these traces can be used to check that a formula is a tautology in a more elementary way. We also derive a certified implementation.

The paper is structured as follows. The algorithm is presented in Section 2. The formalization of the algorithm is described in Section 3. The notion of trace is introduced in Section 4. Finally the implementation is given in Section 5.

2 The algorithm

Stålmarck's algorithm is a tautology checker. It deals with boolean formulae, i.e. expressions formed with the two constants $\top$ (true), $\bot$ (false), the unary symbol $\neg$ (negation), the binary symbols & (conjunction), # (disjunction), $\mapsto$ (implication), $=$ (equivalence), and a set of variables $(v_i)_{i\in\mathbb{N}}$. For example, the

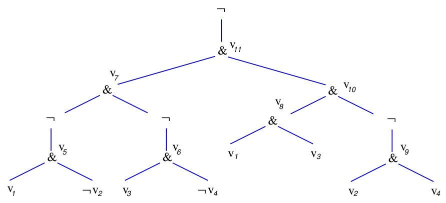

Fig. 1. Annotated tree-like representation of the formula

following formula is a boolean expression containing four variables v_1, v_2, v_3 and v_4:

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

It is also a tautology. This means that the formula is valid (true) for any value of its variables. The first step of the algorithm is to reduce the number of binary symbols using the following equalities:

$$\begin{aligned} A \# B &= \neg(\neg A\,\&\,\neg B) \\ A \mapsto B &= \neg(A\,\&\,\neg B) \\ \neg\neg A &= A \end{aligned}$$

With this transformation, we obtain an equivalent formula containing only conjunctions, equalities and negations. By applying this transformation on our example, we get:

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

The algorithm manipulates data structures called *triplets*. To handle negation, variables are *signed*: $\pm v_i$. A triplet is a group of three signed variables and a connector (either & or $=$), meaning that the first variable has the value of the result of applying the connector to the other two variables. The two kinds of triplets are written as $v_i := \pm v_j \,\&\, \pm v_k$ and $v_i := \pm v_j = \pm v_k$. Every reduced boolean expression has a corresponding list of triplets. If we consider the tree representation of the formula and annotate every binary tree with a fresh new variable as in Figure 1, taking every binary node of the annotated tree to form a triplet gives us the list of triplets representing the formula. In our example, we get the following list:

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

The value of the formula is the value of $-v_{11}$. The algorithm works by refutation. It assumes that the formula is false and tries to reach a contradiction by propagation. There is a set of rules for each kind of triplets that defines how to do this propagation. For the triplet $v_i := v_j \,\&\, v_k$ we have nine rules:

if	$v_i = -v_j$,	*propagate*	$v_j = \top$ *and* $v_k = \bot$	$\&_{i-j}$
if	$v_i = -v_k$,	*propagate*	$v_j = \bot$ *and* $v_k = \top$	$\&_{i-k}$
if	$v_j = v_k$,	*propagate*	$v_i = v_k$	$\&_{jk}$
if	$v_j = -v_k$,	*propagate*	$v_i = \bot$	$\&_{j-k}$
if	$v_i = \top$,	*propagate*	$v_j = \top$ *and* $v_k = \top$	$\&_{i\top}$
if	$v_j = \top$,	*propagate*	$v_i = v_k$	$\&_{j\top}$
if	$v_j = \bot$,	*propagate*	$v_i = \bot$	$\&_{j\bot}$
if	$v_k = \top$,	*propagate*	$v_i = v_j$	$\&_{k\top}$
if	$v_k = \bot$,	*propagate*	$v_i = \bot$	$\&_{k\bot}$

For the triplet $v_i := v_j = v_k$ we have twelve rules:

if	$v_i = v_j$,	*propagate*	$v_k = \top$	$=_{ij}$
if	$v_i = -v_j$,	*propagate*	$v_k = \bot$	$=_{i-j}$
if	$v_i = v_k$,	*propagate*	$v_j = \top$	$=_{ik}$
if	$v_i = -v_k$,	*propagate*	$v_j = \bot$	$=_{i-k}$
if	$v_j = v_k$,	*propagate*	$v_i = \top$	$=_{jk}$
if	$v_j = -v_k$,	*propagate*	$v_i = \bot$	$=_{j-k}$
if	$v_i = \top$,	*propagate*	$v_j = v_k$	$=_{i\top}$
if	$v_i = \bot$,	*propagate*	$v_j = -v_k$	$=_{i\bot}$
if	$v_j = \top$,	*propagate*	$v_i = v_k$	$=_{j\top}$
if	$v_j = \bot$,	*propagate*	$v_i = -v_k$	$=_{j\bot}$
if	$v_k = \top$,	*propagate*	$v_i = v_j$	$=_{k\top}$
if	$v_k = \bot$,	*propagate*	$v_i = -v_j$	$=_{k\bot}$

In our case, this simple mechanism of propagation is sufficient to establish that the formula is a tautology. We start with the state where $v_{11} = \top$ ($-v_{11} = \bot$) and apply the following rules:

$v_{11} = \top$,	*we get*	$v_7 = \top$ *and* $v_{10} = \top$	*by*	$\&_{i\top}$	*on*	(7)
$v_7 = \top$,	*we get*	$v_5 = \bot$ *and* $v_6 = \bot$	*by*	$\&_{i\top}$	*on*	(3)
$v_{10} = \top$,	*we get*	$v_8 = \top$ *and* $v_9 = \bot$	*by*	$\&_{i\top}$	*on*	(6)
$v_8 = \top$,	*we get*	$v_1 = \top$ *and* $v_3 = \top$	*by*	$\&_{i\top}$	*on*	(4)
$v_1 = \top$,	*we get*	$v_5 = -v_2$	*by*	$\&_{j\top}$	*on*	(1)
$v_2 = \top$,	*we get*	$v_9 = v_4$	*by*	$\&_{j\top}$	*on*	(5)
$v_3 = \top$,	*we get*	$v_6 = -v_4$	*by*	$\&_{j\top}$	*on*	(2)

The last equation is a contradiction since we know that $v_6 = \bot$ and $v_4 = v_9 = \bot$. Note that the order in which propagation rules are selected is arbitrary.

Most of the time the propagation alone is not sufficient to conclude. In that case the *dilemma rule* can be applied. This rule works as depicted in Figure 2. Given a state S, it takes two arbitrary variables v_i and v_j and creates two

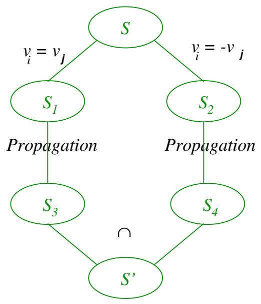

Fig. 2. The dilemma rule

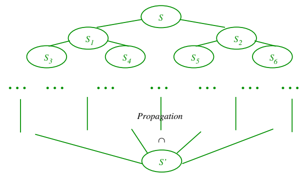

Fig. 3. The generalized dilemma rule

separates branches. In one branch, it adds the equation $v_i = v_j$ to get S_1. In the second branch it adds the equation $v_i = -v_j$ to get S_2. On each of these branches, the propagation is applied to obtain S_3 and S_4 respectively. Then the result of the dilemma rule is the intersection S' of S_3 and S_4 that contains all the equations that are valid independently of the relation between v_i and v_j. If one of the branches gives a contradiction, the result is the final state of the other branch. If we obtain a contradiction on both branches, a contradiction is reached. The dilemma rule is iterated on all pairs of variables taking the state resulting from the previous application as the initial state of the next one, till no new information is gained or a contradiction is reached. If no contradiction is reached, the same process is applied using four variables creating four branches $(v_i = v_j, v_k = v_l)$, $(v_i = v_j, v_k = -v_l)$, $(v_i = -v_j, v_k = v_l)$ and $(v_i = -v_j, v_k = -v_l)$). If the iteration on four variables is not sufficient to conclude, we can proceed using 6 then 8, $\ldots$, then $2 * n$ variables. This gives us a generalized schema of the dilemma rule as depicted in Figure 3. The nice property of this algorithm is that the dilemma rule with four variables v_i, v_j, v_k and v_l with the restriction that $v_j = v_l = \top$ is sufficient to find most of the tautologies occurring in formal verification.

3 Formalization of the algorithm

3.1 Triplets

To define triplets in Coq we first need a notion of signed variables. For this, we introduce the type rZ with two constructors on nat: `+` and `-`. We also take the convention that $\top$ is represented by `+0` and $\bot$ by `-0`. On rZ, we define the usual operations: complement $(-)$, absolute value $(||)$ and an order $<$ such that $i < j$ if and only if $|i| < |j|$.

A triplet is a set of three signed variables and a binary operation. We define the new type *triplet* with the only constructor `Triplet`:

Inductive $triplet : Set :=$
  $\texttt{Triplet} : rBoolOp \rightarrow rZ \rightarrow rZ \rightarrow rZ \rightarrow triplet$

where $rBoolOp$ is an enumerate type containing the two elements `rAnd` and `rEq`. In the following we use the usual pretty-printing convention for triplets: (`Triplet rAnd` i j k) and (`Triplet rEq` i j k) are written as $i := j \,\&\, k$ and $i := j = k$ respectively.

In order to define an evaluation on triplets, we first need to define an evaluation on rZ as:

Definition $rZEval : (nat \rightarrow bool) \rightarrow rZ \rightarrow bool :=$
  $\lambda f{:}\ nat \rightarrow bool.\ \lambda r{:}\ rZ.$
    Cases r **of**
      $(+\ n) \Longrightarrow (f\ n)$
      $|\ (-\ n) \Longrightarrow \neg(f\ n)$
    end.

For the triplet we simply check that the first variable is equal to the result of applying the boolean operation to the other two variables:

Definition $tZEval : (nat \rightarrow bool) \rightarrow triplet \rightarrow bool :=$
  $\lambda f{:}\ nat \rightarrow bool.\ \lambda t{:}\ triplet.$
    Cases t **of**
      (`Triplet` $r\ v_1\ v_2\ v_3) \Longrightarrow$
        $(rZEval\ v_1) = ((rBoolOpFun\ r)\ (rZEval\ v_2)\ (rZEval\ v_3))$
    end.

where the function $rBoolOpFun$ maps elements of $rBoolOp$ into their corresponding boolean operation.

As the algorithm manipulates a list of triplets, we introduce the notion of realizability: a valuation realizes a list of triplets if the evaluation of each triplet in the list gives true:

Definition $realizeTriplets : (nat \rightarrow bool) \rightarrow (list\ triplet) \rightarrow Prop :=$
  $\lambda f{:}\ (nat \rightarrow bool).\ \lambda L{:}\ (list\ triplet).\ \forall t{:}\ triplet.\ t\ in\ L \Rightarrow (tZEval\ f\ t) = \top.$

Another interesting notion is the one of valid equation with respect to a list of triplets. An equation $i = j$ is valid if for every valuation f that realizes a list of triplets L, f gives the same value for i and j:

Definition $validEquation : (list\ triplet) \rightarrow rZ \rightarrow rZ \rightarrow Prop :=$
  $\lambda L{:}\ (list\ triplet).\ \lambda p, q{:}\ rZ.\ \forall f{:}\ (nat \rightarrow bool).$
    $(realizeTriplets\ f\ L) \Rightarrow (f\ 0) = \top \Rightarrow (rZEval\ f\ p) = (rZEval\ f\ q).$

The condition $(f\ 0) = \top$ is here to keep the convention that $(+0)$ represents $\top$.

Not every list of triplets corresponds to a boolean expression. To express the notion of tautology on triplets we simply ask for the top variable of the generated list to be evaluated to true:

Definition $tTautology : Expr \rightarrow Prop :=$
  $\lambda e{:}\ Expr.$
    Cases $(makeTriplets\ e)$ **of**
      $(l, s) \Longrightarrow (validEquation\ l\ s\ \top)$
    end.

where $Expr$ is the type representing boolean expressions and $makeTriplets$ is the function that computes the list of triplets corresponding to a given expression and its top variable. With this definition, we have the following theorem:

Theorem $TautoEquivtTauto$:
  $\forall e{:}\ Expr.\ (tautology\ e) \iff (tTautology\ e).$

where the predicate $tautology$ defines the usual notion of tautology on boolean expressions.

3.2 States

All the operations of checking and adding equations are done with respect to a state. We have chosen to represent states as lists of pairs of signed variables.

Definition $State{:}\ Set := (list\ rZ * rZ).$

The inductive predicate $\sim$ defines when two variables are equal:

Inductive $\sim : State \rightarrow rZ \rightarrow rZ \rightarrow Prop :=$
  $\sim\!Ref : \forall a{:}\ rZ.\ \forall S{:}\ State.\ a \sim_S a$
  $|\ \sim\!In : \forall a, b{:}\ rZ.\ \forall S{:}\ State.\ (a, b)\ in\ S \Rightarrow a \sim_S b$
  $|\ \sim\!Sym : \forall a, b{:}\ rZ.\ \forall S{:}\ State.\ a \sim_S b \Rightarrow b \sim_S a$
  $|\ \sim\!Inv : \forall a, b{:}\ rZ.\ \forall S{:}\ State.\ a \sim_S b \Rightarrow -a \sim_S -b$
  $|\ \sim\!Trans : \forall a, b, c{:}\ rZ.\ \forall S{:}\ State.\ a \sim_S b \Rightarrow b \sim_S c \Rightarrow a \sim_S c$
  $|\ \sim\!Contr : \forall a, b, c{:}\ rZ.\ \forall S{:}\ State.\ a \sim_S -a \Rightarrow b \sim_S c.$

The logic of Coq is constructive. This means that the theorem $\forall P{:}\ Prop.\ P \vee \neg P$ is not valid. But instances of this theorem can be proved. In particular we have:

Theorem $stateDec{:} \forall S{:}\ State.\ \forall a, b{:}\ rZ.\ a \sim_S b \vee \neg(a \sim_S b).$

The property of being a contradictory state is defined as:

Definition $contradictory{:}\ State \rightarrow Prop := \lambda S{:}\ State.\ \exists a{:}\ rZ.\ a \sim_s -a.$

Note that from the definition of $\sim$, it follows that in a contradictory state all equalities are valid. Inclusion and equality on states are defined as:

Definition $\subset$: *State* $\rightarrow$ *State* $\rightarrow$ *Prop* :=
$\lambda S_1, S_2$: *State*. $\forall a, b$: *rZ*. $a \sim_{S_1} b \Rightarrow a \sim_{S_2} b$.

Definition $\equiv$: *State* $\rightarrow$ *State* $\rightarrow$ *Prop* := $\lambda S_1, S_2$: *State*. $S_1 \subset S_2 \wedge S_2 \subset S_1$.

A valuation realizes a state if all the equations of the state are valid:

Definition *realizeState* : (*nat* $\rightarrow$ *bool*) $\rightarrow$ *State* $\rightarrow$ *Prop* :=
λf: *nat* $\rightarrow$ *bool*. λS: *State*. $\forall a, b$: *rZ*. (a, b) *in* $S \Rightarrow$ (*rZEval* f a) = (*rZEval* f b).

We also need to define two basic functions on states: intersection and union. The union of two states is simply the state given by the concatenation of the two lists. In the following we use the notation $(a, b)+S$ to denote $[(a, b)] \cup S$.

The intersection of two states is not the intersection of their lists[1]. The function that computes the intersection of S_1 and S_2 first generates the list of all non-trivial equations of S_1, i.e. all pairs (a,b) such that $a \sim_{S_1} b$ and $(a \neq b)$. Then, it removes from this list the equations that are not valid in S_2. The resulting list represents $S_1 \cap S_2$.

3.3 One-step propagation

We formalize the one-step propagation as an inductive predicate $\rightarrow$ whose definition is given in Appendix A. Note that $S_1 \rightarrow_t S_2$ only means that there exists a rule that produces S_2 from S_1 using the triplet t. Because $\rightarrow$ is defined as a predicate, no particular strategy of rule application is assumed. The relation $\rightarrow$ is compatible with the equality as defined in Section 3.2:

Theorem $\rightarrow\equiv$*Ex*:
$\forall S_1, S_2, S_3$: *State*. $\forall t$: *triplet*.
$S_1 \rightarrow_t S_2 \Rightarrow S_1 \equiv S_3 \Rightarrow \exists S_4$: *State*. $S_3 \rightarrow_t S_4 \wedge S_4 \equiv S_2$.

Also a propagation only adds equations:

Theorem $\rightarrow\cup$*Ex*:
$\forall S_1, S_2$: *State*. $\forall t$: *triplet*. $S_1 \rightarrow_t S_2 \Rightarrow \exists S_3$: *State*. $S_2 \equiv S_3 \cup S_1$.

A corollary of this last theorem is that progatation always produces a bigger state:

Theorem $\rightarrow$*Incl*:
$\forall S_1, S_2$: *State*. $\forall t$: *triplet*. $S_1 \rightarrow_t S_2 \Rightarrow S_1 \subset S_2$.

In a similar way we can prove that the relation behaves as a congruence:

Theorem $\rightarrow$*CongruentEx*:
$\forall S_1, S_2, S_3$: *State*. $\forall t$: *triplet*.
$S_1 \rightarrow_t S_2 \Rightarrow \exists S_4$: *State*. $(S_3 \cup S_1) \rightarrow_t S_4 \wedge S_4 \equiv (S_3 \cup S_2)$.

This gives us as a corollary that the relation is monotone:

[1] For example, given the lists $[(1, 2); (2, 3)]$ and $[(1, 3)]$, their intersection as states is $[(1, 3)]$ while their intersection as lists is $[]$.

Theorem $\rightarrow$*MonotoneEx*:
$\forall S_1, S_2, S_3$: *State*. $\forall t$: *triplet*.
$S_1 \rightarrow_t S_3 \Rightarrow S_1 \subset S_2 \Rightarrow \exists S_4$: *State*. $S_2 \rightarrow_t S_4 \wedge S_3 \subset S_4$.

Another interesting property is that the relation is confluent:

Theorem $\rightarrow$*ConflEx*:
$\forall t_1, t_2$: *triplet*. $\forall S_1, S_2, S_3$: *State*. $S_1 \rightarrow_{t_1} S_2 \Rightarrow S_1 \rightarrow_{t_2} S_3 \Rightarrow$
$\exists S_4, S_5$: *State*. $S_2 \rightarrow_{t_2} S_4 \wedge S_3 \rightarrow_{t_1} S_5 \wedge S_4 \equiv S_5$.

Note that to establish these properties we do not use the particular equations that are checked or added. All relations with a shape similar to the one of $\rightarrow$ would have these properties.

The first semantic property that we have proved is that preserving the realizability for the one-step propagation is equivalent in some sense to evaluating the triplet to $\top$:

Theorem *realizeStateEvalEquiv*:
$\forall f$: *nat* $\rightarrow$ *bool*. $\forall S_1, S_2$: *State*. $\forall t$: *triplet*.
$(f\ 0) = \top \Rightarrow$ (*realizeState* f S_1) $\Rightarrow S_1 \rightarrow_t S_2$
$\Rightarrow$ ((*realizeState* f S_2) $\iff$ (*tEval* f t) = $\top$).

Another key semantic property is that no matter which rule is applied, the resulting state is the same:

Theorem $\rightarrow$*Eq*:
$\forall t$: *triplet*. $\forall S_1, S_2, S_3$: *State*. $S_1 \rightarrow_t S_2 \Rightarrow S_1 \rightarrow_t S_3 \Rightarrow S_2 \equiv S_3$.

Moreover, a triplet is essentially useful only once:

Theorem $\rightarrow$*Invol*:
$\forall t$: *triplet*. $\forall S_1, S_2, S_3, S_4$: *State*.
$S_1 \rightarrow_t S_2 \Rightarrow S_2 \subset S_3 \Rightarrow S_3 \rightarrow_t S_4 \Rightarrow S_3 \equiv S_4$.

The proofs of the above theorems are not very deep and mostly involve exploring the twenty-one possible rules of one-step propagation.

3.4 Propagation

The propagation consists in iterating the one-step propagation. We take the reflexive transitive closure of $\rightarrow_t$:

Inductive $\rightarrow^*$: *State* $\rightarrow$ (*list triplet*) $\rightarrow$ *State* $\rightarrow$ *Prop* :=
$\rightarrow^*$*Ref* : $\forall S_1, S_2$: *State*. $\forall L$: (*list triplet*). $S_1 \equiv S_2 \Rightarrow S_1 \rightarrow^*_L S_2$
| $\rightarrow^*$*Trans* : $\forall S_1, S_2, S_3$: *State*. $\forall L$: (*list triplet*). $\forall t$: *triplet*.
t *in* $L \Rightarrow S_1 \rightarrow_t S_2 \Rightarrow S_2 \rightarrow^*_L S_3 \Rightarrow S_1 \rightarrow^*_L S_3$.

All the properties of the one-step propagation can be lifted to the propagation. The exception is the theorem about realizability which has a simple implication since the propagation might use a strict subset of the list of triplets:

Theorem *realizeStateEval**:
$\forall f\colon nat \to bool. \forall S_1, S_2\colon State. \forall L\colon (list\ triplet).$
$(f\ 0) = \top \Rightarrow (realizeState\ f\ S_1) \Rightarrow S_1 \to^*_L S_2$
$\Rightarrow (realizeTriplets\ f\ L) \Rightarrow (realizeState\ f\ S_2).$

Finally the property that a triplet is useful only once is captured by:

Theorem $\to^*$ *TermEx*:
$\forall L\colon (list\ triplet). \forall S_1, S_2\colon State.\ S_1 \to^*_L S_2 \Rightarrow$
$(S_1 \equiv S_2) \vee (\exists t\colon triplet. \exists S_3\colon State.\ t\ in\ L \wedge S_1 \to_t S_3 \wedge S_3 \to^*_{L-[t]} S_2).$

where $L - [t]$ denotes the list obtained by removing t from L.

3.5 The dilemma rule

As we did for propagation, the dilemma rule is non-deterministic and modeled by a predicate. Also, we allow an arbitrary (but finite) number of splits:

Inductive $\to^d$: $State \to (list\ triplet) \to State \to Prop :=$
$\to^d Ref : \forall S_1, S_2\colon State. \forall L\colon (list\ triplet).\ S_1 \to^*_L S_2 \Rightarrow S_1 \to^d_L S_2$
$|\ \to^d Split : \forall a, b\colon rZ. \forall S_1, S_2, S_3, S_4\colon State. \forall L\colon (list\ triplet). \forall t\colon triplet.$
$(a, b){+}S_1 \to^d_L S_2 \Rightarrow (a, -b){+}S_1 \to^d_L S_3 \Rightarrow S_2 \cap S_3 \equiv S_4 \Rightarrow S_1 \to^d_L S_4.$

The relation $\to^d$ is compatible with the equality:

Theorem $\to^d \equiv$:
$\forall S_1, S_2, S_3, S_4\colon State. \forall L\colon (list\ triplet).$
$S_1 \to^d_L S_2 \Rightarrow S_3 \equiv S_1 \Rightarrow S_4 \equiv S_2 \Rightarrow S_3 \to^d_L S_4.$

The same theorems about inclusion also hold:

Theorem $\to^d \cup Ex$:
$\forall S_1, S_2\colon State. \forall L\colon (list\ triplet).\ S_1 \to^d_L S_2 \Rightarrow \exists S_3\colon State.\ S_2 \equiv S_3 \cup S_1.$

Theorem $\to^d Incl$:
$\forall S_1, S_2\colon State. \forall L\colon (list\ triplet).\ S_1 \to^d_L S_2 \Rightarrow S_1 \subset S_2.$

Unfortunately as we only have $(S_1 \cap S_2) \cup S_3 \subset (S_1 \cup S_3) \cap (S_2 \cup S_3)$ and not $(S_1 \cap S_2) \cup S_3 \equiv (S_1 \cup S_3) \cap (S_2 \cup S_3)$, the relation is not a congruence. A simple way to recapture this congruence would be to define $\to^d$ as:

Inductive $\to^d$: $State \to (list\ triplet) \to State \to Prop :=$
$\to^d Ref : \forall S_1, S_2\colon State. \forall L\colon (list\ triplet).\ S_1 \to^*_L S_2 \Rightarrow S_1 \to^d_L S_2$
$|\ \to^d Split : \forall a, b\colon rZ. \forall S_1, S_2, S_3, S_4\colon State. \forall L\colon (list\ triplet). \forall t\colon triplet.$
$(a, b){+}S_1 \to^d_L S_2 \Rightarrow (a, -b){+}S_1 \to^d_L S_3 \Rightarrow S_1 \subset S_4 \subset (S_2 \cap S_3) \Rightarrow S_1 \to^d_L S_4.$

but this would mean considering the merging of the two branches as a non-deterministic operation, so we prefer our initial definition. Even though the relation is not a congruence, it is monotone:

Theorem $\to^d$ *Monotone*:
$\forall L\colon (list\ triplet). \forall S_1, S_2, S_3\colon State.$
$S_1 \to^d_L S_3 \Rightarrow S_1 \subset S_2 \Rightarrow \exists S_4\colon State.\ S_2 \to^d_L S_4 \wedge S_3 \subset S_4.$

and it is also confluent:

Theorem $\to^d$ *Confluent*:
$\forall L\colon (list\ triplet). \forall S_1, S_2, S_3\colon State.$
$S_1 \to^d_L S_2 \Rightarrow S_1 \to^d_L S_3 \Rightarrow \exists S_4\colon State.\ S_2 \to^d_L S_4 \wedge S_3 \to^d_L S_4.$

The last property we have proved about the dilemma rule is that it preserves realizability:

Theorem *realizeStateEval*^{d}:
$\forall f\colon nat \to bool. \forall S_1, S_2\colon State. \forall L\colon (list\ triplet).$
$(f\ 0) = \top \Rightarrow (realizeState\ f\ S_1) \Rightarrow S_1 \to^d_L S_2$
$\Rightarrow (realizeTriplets\ f\ L) \Rightarrow (realizeState\ f\ S_2).$

3.6 Stålmarck's algorithm

Stålmarck's algorithm is the reflexive transitive closure of the dilemma rule:

Inductive $\to^s$: $State \to (list\ triplet) \to State \to Prop :=$
$\to^s Ref : \forall S_1, S_2\colon State. \forall L\colon (list\ triplet).\ S_1 \equiv S_2 \Rightarrow S_1 \to^s_L S_2$
$|\ \to^s Trans : \forall S_1, S_2, S_3\colon State. \forall L\colon (list\ triplet).$
$S_1 \to^d_L S_2 \Rightarrow S_2 \to^s_L S_3 \Rightarrow S_1 \to^s_L S_3.$

As for $\to^d$ we get the standard properties:

Theorem $\to^s \equiv$:
$\forall S_1, S_2, S_3, S_4\colon State. \forall L\colon (list\ triplet).$
$S_1 \to^s_L S_2 \Rightarrow S_3 \equiv S_1 \Rightarrow S_4 \equiv S_2 \Rightarrow S_3 \to^s_L S_4.$

Theorem $\to^s \cup Ex$:
$\forall S_1, S_2\colon State. \forall L\colon (list\ triplet).\ S_1 \to^s_L S_2 \Rightarrow \exists S_3\colon State.\ S_2 \equiv S_3 \cup S_1.$

Theorem $\to^s Incl$:
$\forall S_1, S_2\colon State. \forall L\colon (list\ triplet).\ S_1 \to^s_L S_2 \Rightarrow S_1 \subset S_2.$

Theorem $\to^s$ *Monotone*:
$\forall L\colon (list\ triplet). \forall S_1, S_2, S_3\colon State.$
$S_1 \to^s_L S_3 \Rightarrow S_1 \subset S_2 \Rightarrow \exists S_4\colon State.\ S_2 \to^s_L S_4 \wedge S_3 \subset S_4.$

Theorem $\to^s$ *Confluent*:
$\forall L\colon (list\ triplet). \forall S_1, S_2, S_3\colon State.$
$S_1 \to^s_L S_2 \Rightarrow S_1 \to^s_L S_3 \Rightarrow \exists S_4\colon State.\ S_2 \to^s_L S_4 \wedge S_3 \to^s_L S_4.$

Theorem *realizeStateEval*^{s}:
  $\forall f\colon nat \to bool.\forall S_1, S_2\colon State.\forall L\colon (list\ triplet).$
    $(f\ 0) = \top \Rightarrow (realizeState\ f\ S_1) \Rightarrow S_1 \to^s_L S_2$
      $\Rightarrow (realizeTriplets\ f\ L) \Rightarrow (realizeState\ f\ S_2).$

Only the last property is relevant for the correctness of the algorithm. From the theorem *realizeStateEval*^{s}, the following property is easily derived:

Theorem *stålmarckValidEquation*:
  $\forall L\colon (list\ triplet).\forall a, b\colon rZ.\forall S\colon State.$
    $[(a, -b)] \to^s_L S \Rightarrow (contradictory\ S) \Rightarrow (validEquation\ L\ a\ b).$

Once we have this theorem, we can glue together all the theorems about tautology to get the correctness:

Theorem *stålmarckCorrect*:
  $\forall e\colon Expr.\forall S\colon State.$
    Cases $(makeTriplets\ e)$ **of**
      $(l, s) \Longrightarrow [(s, \bot)] \to^s_l S \Rightarrow (contradictory\ S) \Rightarrow (Tautology\ e)$
    end.

Another property that has been formalized is the completeness of the algorithm:

Theorem *stålmarckComplete*:
  $\forall e\colon Expr.(Tautology\ e) \Rightarrow \exists S\colon State.$
    Cases $(makeTriplets\ e)$ **of**
      $(l, s) \Longrightarrow [(s, \bot)] \to^s_l S \wedge (contradictory\ S)$
    end.

This is proved by showing that if e is a tautology, we obtain a contradiction by applying the dilemma rule on all the variables in the list of triplets. The program extracted from the constructive proof of the theorem *stålmarckComplete* would then not be faster than the one that computes the truth table.

4 Trace

Our relation $\to^s$ contains all possible execution paths. The choice of the rules, the choice of the triplets and the choice of the variables for the dilemma rule are not explicitly given. A natural question arises on what an explicit implementation should produce to provide a certificate that it has reached a contradiction. Of course, a complete trace of the execution (states, rules, triplets, variables) is a valid certificate. In fact, we can do much better and keep only the triplets that have been used and the variables on which the dilemma rule has been applied. We define our notion of trace as:

Inductive *Trace*: *Set* :=
    `emptyTrace` : *Trace*
  | `tripletTrace` : $triplet \to Trace$
  | `seqTrace` : $Trace \to Trace \to Trace$
  | `dilemmaTrace` : $rZ \to rZ \to Trace \to Trace \to Trace.$

The semantics is given by the following predicate that evaluates a trace:

Inductive *evalTrace*: $State \to Trace \to State \to Prop$:=
    emptyEval : $\forall S_1, S_2\colon State.\ S_1 \equiv S_2 \Rightarrow (evalTrace\ S_1\ \texttt{emptyTrace}\ S_2)$
  | *tripletEval* : $\forall S_1, S_2\colon State.\forall t\colon triplet.$
      $S_1 \to_t S_2 \Rightarrow (evalTrace\ S_1\ (\texttt{tripletTrace}\ t)\ S_2)$
  | *seqEval* : $\forall S_1, S_2, S_3\colon State.\forall T_1, T_2\colon Trace.$
      $(evalTrace\ S_1\ T_1\ S_2) \Rightarrow (evalTrace\ S_2\ T_2\ S_3)$
        $\Rightarrow (evalTrace\ S_1\ (\texttt{seqTrace}\ T_1\ T_2)\ S_3)$
  | *dilemmaEval* : $\forall S_1, S_2, S_3, S_4\colon State.\forall a, b\colon rZ.\forall T_1, T_2\colon Trace.$
      $(evalTrace\ (a, b){+}S_1\ T_1\ S_2) \Rightarrow (evalTrace\ (a, -b){+}S_1\ T_2\ S_3)$
        $\Rightarrow S_2 \cap S_3 \equiv S_4 \Rightarrow (evalTrace\ S_1\ (\texttt{dilemmaTrace}\ a\ b\ T_1\ T_2)\ S_4).$

The fact that a trace determines a unique computation up to state equality is asserted by the following theorem:

Theorem *evalTraceEq*:
  $\forall S_1, S_2, S_3, S_4\colon State.\forall T\colon Trace.$
    $(evalTrace\ S_1\ T\ S_2) \Rightarrow (evalTrace\ S_3\ T\ S_4) \Rightarrow S_1 \equiv S_3 \Rightarrow S_2 \equiv S_4.$

Conversely it is possible to get a trace from any non-deterministic computation:

Theorem *stålmarckExTrace*:
  $\forall S_1, S_2\colon State.\forall L\colon (list\ triplet).$
    $S_1 \to^s_L S_2 \Rightarrow \exists T\colon Trace.(evalTrace\ S_1\ T\ S_2) \wedge T\ in\ L.$

The second condition requires all the triplets in the trace T to be in L.

5 Implementation

Because of space limitation, we are only going to sketch the different components of the implementation. In particular, we do not make explicit the rules of sign using the notation $\pm v$ to denote either $+v$ or $-v$.

5.1 Memory

We represent non-contradictory states using functional arrays. Appendix B lists the different axioms we are using in our development. The size of the array is *maxN*, the natural number that exceeds by at least one all the variables in the list of triplets. The type of the elements of the array is defined as follows:

Inductive *vM*: *Set* :=
    `ref` : $rZ \to vM$
  | `class` : $(list\ rZ) \to vM$

The value of the location i depends on the smallest element a such that $+i \sim a$. If $i \neq |a|$, the location i contains the value (`ref` a). Otherwise, it contains the value (`class` L), where L is the ordered list of the elements b such that $+i \sim b$ and $|b| \neq i$. All the constraints about the different values of the array are concentrated in the predicate *WellFormed*:

Definition $\mathit{Mem}\colon \mathit{Set} := \{r : (\mathit{Array}\ \mathit{maxN}\ \mathit{vM}) \mid (\mathit{WellFormed}\ \mathit{maxN}\ r)\}$.

Checking equality Given a memory m, it is easy to build a function min_m that returns for any element a of rZ the smallest b such that $a \sim_m b$. To check the equality between a and b in m, it is then sufficient to compare $(\mathit{min}_m\ a)$ and $(\mathit{min}_m\ b)$.

Adding an equation Given a memory m, it is also easy to build a function l_m that returns for any element a of rZ the ordered list of all the elements b such that $a \sim_m b$. The result of an addition to a memory is a triple $(\mathit{Mem}, \mathit{bool}, (\mathit{list}\ rZ))$. Since a memory can only represent a non-contradictory state, the boolean is set to true if the addition of the equation gives a contradiction, to false otherwise. The absolute values of the elements of the list are the locations of the arrays that have been modified by the update. To perform the addition of $a = b$ to m, we first compare $(\mathit{min}_m\ a)$ and $(\mathit{min}_m\ b)$. If $(\mathit{min}_m\ a) = (\mathit{min}_m\ b)$, the result is $(m, \bot, [])$. If $(\mathit{min}_m\ a) = -(\mathit{min}_m\ b)$, the result is $(m, \top, [])$. If $(\mathit{min}_m\ a) < (\mathit{min}_m\ b)$, the result is $(m', \bot, [(\mathit{min}_m\ a)] \cup (l_m\ b))$ where m' is obtained from m by setting the locations corresponding to the elements of $(l_m\ b)$ to $(\texttt{ref}\ \pm (\mathit{min}_m\ a))$ and the location $|(\mathit{min}_m\ a)|$ to $(\texttt{class}\ (\pm(l_m\ a) \cup \pm(l_m\ b)))$. The case where $(\mathit{min}_m\ b) < (\mathit{min}_m\ a)$ is symmetric to the previous one.

Intersection The function that computes the intersection takes three memories m_1, m_2, m_3 and two lists d_1, d_2 under the hypothesis that $m_1 \subset m_2$, $m_1 \subset m_3$, d_1 is the difference list between m_1 and m_2, and d_2 is the difference list between m_1 and m_3. It returns a 4-tuple (m'_1, m'_2, m'_3, d') such that $m'_1 = m_2 \cap m_3$, $m'_1 \equiv m'_2 \equiv m'_3$ and d' is the difference list between m_1 and m'_1. It proceeds by successive additions to m_1 of equations $a_i = b_i$ where the a_i are the elements of $d_1 \cap d_2$ and the b_i are the smallest element such that $a_i \sim_{m_2} b_i$ and $a_i \sim_{m_3} b_i$.

5.2 Propagation

The implementation of the one-step propagation is a composition of checking equalities and adding equations. It has the type $\mathit{Mem} \to \mathit{triplet} \to (\mathit{Mem}, \mathit{bool}, (\mathit{list}\ \ rZ))$. To do the propagation, we need to define a way to select triplets. As the difference lists give the variables whose values have been modified, triplets containing these variables are good candidate for applying one-step propagation. The type of the propagation function is $\mathit{Mem} \to (\mathit{list}\ rZ) \to (\mathit{Mem}, \mathit{bool}, (\mathit{list}\ rZ))$. The difference lists resulting from the application of the one-step propagation are then recursively used for selecting triplets. This way of propagating terminates since we cannot add infinitely many equations to a memory.

5.3 Dilemma

We have implemented only the instances of the dilemma rule that are of practical use: *dilemma1*, the dilemma with $(a,\top)$ for an arbitrary a, and *dilemma2*, the dilemma with $((a, \top), (b, \top))$ for arbitrary a and b. To perform the first one, we use three memories, one for each branch and one to compute the intersection. For the second one we use an extra memory. The first memory m_1 is used to compute each branch iteratively. The intermediate result is stored in the second memory m_2. At the end of each iteration we compute the intersection of m_1 and m_2 using the third memory m_3. We then switch m_2 and m_3 and use the last memory to reset m_1 and m_3 before proceeding to the next branch. Note that a dilemma with any number of variables could be implemented in the same way using four memories.

5.4 Stålmarck

At this stage, we have to decide the strategy to pick up variables for the application of the dilemma rules. Our heuristics are very simple and could be largely improved. We first add the initial equation and propagate. If no contradiction is reached, we iterate the application of the function *dilemma1* using minimal variables starting from +1 to $+\mathit{maxN}$. We perform this operation till a contradiction is reached or no more information is gained. In the second case, we do a double iteration with a, b such that $0 < a < b < \mathit{maxN}$ using the function *dilemma2*. Implementing this naïve strategy is straightforward and gives us the function *doStal* for which we have proved the following property:

Theorem *doStalCorrect*:
$\forall e\colon \mathit{Expr}.\, (\mathit{doStal}\ e) = \top \Rightarrow (\mathit{Tautology}\ e)$.

Note that it is the only property that we have proved for our implementation and clearly it is not sufficient. An algorithm that always returns $\bot$ would satisfy the above property. While our implementation is not complete since we use dilemma rules only up to four variables, we could prove some liveness property. This is feasible and would require to formalize the notion of n-hard formulae as described in [10].

5.5 Benchmark

Once the implementation has been proved correct in Coq, the extraction mechanism [9] enables us to get a functional Ocaml [8] program. The result is 1400 lines long. To be able to run such a program, we need to provide an implementation of the arrays that satisfies the axioms of Appendix B. A first possibility is to use balanced trees to implement functional arrays. A second possibility is to use tagged arrays, since we have taken a special care in the implementation in order to be able to use destructive arrays. The tag in the array prohibits illegal accesses. For example, the set function for such arrays looks like:

Problem	dilemma	variables	connectives	hand-coded	balanced trees	tagged arrays
puz030_1	1	25	221	0.04 s	0.07 s	0.05 s
syn072_1	1	30	518	0.04 s	0.17 s	0.14 s
dk17_be	1	63	327	1.58 s	2.76 s	1.89 s
ssa0432_003	1	435	2363	6.38 m	47 s	42 s
jnh211	1	100	3887	9.47 m	9.36 m	9.14 m
aim50.1.6no.1	2	50	238	2.38 m	31.98 s	21.55 s
counter_be	2	18	290	11.62 s	6.76 s	4.24 s
misg_be	2	108	279	1.07 m	1.18 m	52.74 s
dubois20	2	60	479	7.59 m	7.95 s	5.98 s
add2_be	2	144	407	4.00 m	1.44m	1.13 m

Fig. 4. Some benchmarks on a Pentium 450

```
let set tar m v = match tar with
  (ar,tag) -> if ((!tag) = true) then
                 (tag := false; Array.set ar m v;(ar,ref(true)))
              else raise Prohibited_access;;
```

If the program terminates without exception, the result is correct. Table 4 gives some execution times on standard examples taken from [5]. For each problem, we give which level of dilemma rules is needed, the number of variables, the number of connectives and compare three versions of the algorithm: the algorithm is directly hand-coded in Ocaml with slightly different heuristics, our certified version with balanced trees and with tagged arrays. The time includes parsing the formula and generating triplets. Even though the performance of an implementation largely depends on the heuristics, our certified version seems comparable with the hand-coded one and the one presented by John Harrison in [5]. However, we are aware that in order to get a realistic implementation, much more work on optimizations and heuristics has to be done. Prover 4.0, the propositional prover of Prover Technology, needs a maximum of two tenth of a second to conclude on any of the examples given in Table 4.

6 Conclusion

We hope that what we have presented shows that current theorem proving technology can be effectively used to reason about algorithms and their implementations. We have presented a formalization of Stålmarck's algorithm that includes formal proofs of the main properties of the algorithm. Then, we have proposed a format of execution traces and proved that it is adequate. Such certificates are important in practice. They represent a simple way of increasing the confidence in the correctness of specific results. Prover Technology commercializes the Prover Plug-In product for integration into CASE and EDA tools. Prover Plug-In is based on the Stalmarck's method, and supports a trace format [7] and an associated trace/proof checker. John Harrison [5] also presents a tactic based on Stålmarck's method for HOL [4] using traces: the search is handled by a program that generates traces, then, the prover uses these traces to build safe derivations of theorems. Finally, the effort for deriving a certified implementation is orthogonal to the one on traces since the correctness of results is ensured once and for all. The benchmarks given in Section 5.5 show that our implementation is relatively efficient and can be proposed as a reference implementation.

From the point of view of theorem proving, the most satisfying aspect of this work is the formalization of the algorithm. It is a relatively concise development of 3200 lines including 80 definitions and 200 theorems. The proof of correctness of the implementation is less satisfying. Proving the basic operations (addition and intersection) took 2/3 of the effort and represents more than 6000 lines of Coq. This does not reflect the effective difficulty of the task. The main reason why deriving these basic operations has been so tedious is that most of the proofs involves a fair amount of case-splitting. For example, proving properties of the addition often requires to take into account the signs and the relative value of the components of the equation. We have neither managed to abstract our theorems enough nor got enough automation so that we do not have to operate on the different cases manually. Moreover, the fact that we handle imperative features such as arrays in a functional way is a bit awkward. We plan in a near future to use improvements such those presented in [3] to reason directly on imperative programs inside Coq. Finally while the overall experience is quite positive, we strongly believe that for this kind of formalization to become common practice, an important effort has to be done in order to make proofs scripts readable by non-specialists. In that respect, recent efforts such as [2, 11–13] seem very promising.

References

1. Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin, and Laurent Théry, editors. *Theorem Proving in Higher Order Logics: 12th International Conference, TPHOLs'99*, number 1690 in LNCS, Nice, France, 1999. Springer-Verlag.
2. Yann Coscoy, Gilles Kahn, and Laurent Théry. Extracting text from proofs. In *Typed Lambda Calculus and its Applications*, volume 902 of *LNCS*, pages 109–123. Springer-Verlag, 1995.
3. J.-C. Filliâtre. Proof of Imperative Programs in Type Theory. In *International Workshop, TYPES '98, Kloster Irsee, Germany*, volume 1657 of *LNCS*. Springer-Verlag, March 1998.
4. Michael J. C. Gordon and Thomas F. Melham. *Introduction to HOL : a theorem proving environment for higher-order logic*. Cambridge University Press, 1993.
5. J. 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.
6. Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. The Coq Proof Assistant: A Tutorial: Version 6.1. Technical Report 204, INRIA, 1997.
7. Arndt Jonasson. Proof logging, definition of the log format. Draft, Logikkonsult NP, December 1997.

8. Xavier Leroy. Objective Caml. Available at `http://pauillac.inria.fr/ocaml/`, 1997.
9. Christine Paulin-Mohring and Benjamin Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15(5-6):607–640, 1993.
10. 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.
11. Don Syme. Three Tactic Theorem Proving. In Bertot et al. [1], pages 203–220.
12. Markus Wenzel. A Generic Interpretative Approach to Readable Formal Proof Documents. In Bertot et al. [1], pages 167–184.
13. Vincent Zammit. On the Implementation of an Extensible Declarative Proof Language. In Bertot et al. [1], pages 185–202.

A The predicate for one-step propagation

Inductive $\rightarrow = : \mathit{State} \rightarrow \mathit{triplet} \rightarrow \mathit{State} :=$

$$
\begin{array}{l}
\quad \rightarrow_{\&_{p-q}} : \forall p, q, r\colon rZ.\, \forall S\colon \mathit{State}.\ p \sim_S \text{-}q \Rightarrow S \rightarrow_{p:=q\,\&\,r} (q,\top)+(r,\bot)+S \\
\mid\ \rightarrow_{\&_{p-r}} : \forall p, q, r\colon rZ.\, \forall S\colon \mathit{State}.\ p \sim_S \text{-}r \Rightarrow S \rightarrow_{p:=q\,\&\,r} (q,\bot)+(r,\top)+S \\
\mid\ \rightarrow_{\&_{qr}} : \forall p, q, r\colon rZ.\, \forall S\colon \mathit{State}.\ q \sim_S r \Rightarrow S \rightarrow_{p:=q\,\&\,r} (p,r)+S \\
\mid\ \rightarrow_{\&_{q-r}} : \forall p, q, r\colon rZ.\, \forall S\colon \mathit{State}.\ q \sim_S \text{-}r \Rightarrow S \rightarrow_{p:=q\,\&\,r} (p,\bot)+S \\
\mid\ \rightarrow_{\&_{p\top}} : \forall p, q, r\colon rZ.\, \forall S\colon \mathit{State}.\ p \sim_S \top \Rightarrow S \rightarrow_{p:=q\,\&\,r} (q,\top)+(r,\top)+S \\
\mid\ \rightarrow_{\&_{q\top}} : \forall p, q, r\colon rZ.\, \forall S\colon \mathit{State}.\ q \sim_S \top \Rightarrow S \rightarrow_{p:=q\,\&\,r} (p,r)+S \\
\mid\ \rightarrow_{\&_{q\bot}} : \forall p, q, r\colon rZ.\, \forall S\colon \mathit{State}.\ q \sim_S \bot \Rightarrow S \rightarrow_{p:=q\,\&\,r} (p,\bot)+S \\
\mid\ \rightarrow_{\&_{r\top}} : \forall p, q, r\colon rZ.\, \forall S\colon \mathit{State}.\ r \sim_S \top \Rightarrow S \rightarrow_{p:=q\,\&\,r} (p,q)+S \\
\mid\ \rightarrow_{\&_{r\bot}} : \forall p, q, r\colon rZ.\, \forall S\colon \mathit{State}.\ r \sim_S \bot \Rightarrow S \rightarrow_{p:=q\,\&\,r} (p,\bot)+S \\
\mid\ \rightarrow_{=_{pq}} : \forall p, q, r\colon rZ.\, \forall S\colon \mathit{State}.\ p \sim_S q \Rightarrow S \rightarrow_{p:=q=r} (r,\top)+S \\
\mid\ \rightarrow_{=_{p-q}} : \forall p, q, r\colon rZ.\, \forall S\colon \mathit{State}.\ p \sim_S \text{-}q \Rightarrow S \rightarrow_{p:=q=r} (r,\bot)+S \\
\mid\ \rightarrow_{=_{pr}} : \forall p, q, r\colon rZ.\, \forall S\colon \mathit{State}.\ p \sim_S r \Rightarrow S \rightarrow_{p:=q=r} (q,\top)+S \\
\mid\ \rightarrow_{=_{p-r}} : \forall p, q, r\colon rZ.\, \forall S\colon \mathit{State}.\ p \sim_S \text{-}r \Rightarrow S \rightarrow_{p:=q=r} (q,\bot)+S \\
\mid\ \rightarrow_{=_{qr}} : \forall p, q, r\colon rZ.\, \forall S\colon \mathit{State}.\ q \sim_S r \Rightarrow S \rightarrow_{p:=q=r} (p,\top)+S \\
\mid\ \rightarrow_{=_{q-r}} : \forall p, q, r\colon rZ.\, \forall S\colon \mathit{State}.\ q \sim_S \text{-}r \Rightarrow S \rightarrow_{p:=q=r} (p,\bot)+S \\
\mid\ \rightarrow_{=_{p\top}} : \forall p, q, r\colon rZ.\, \forall S\colon \mathit{State}.\ p \sim_S \top \Rightarrow S \rightarrow_{p:=q=r} (q,r))+S \\
\mid\ \rightarrow_{=_{p\bot}} : \forall p, q, r\colon rZ.\, \forall S\colon \mathit{State}.\ p \sim_S \bot \Rightarrow S \rightarrow_{p:=q=r} (q,\text{-}r)+S \\
\mid\ \rightarrow_{=_{q\top}} : \forall p, q, r\colon rZ.\, \forall S\colon \mathit{State}.\ q \sim_S \top \Rightarrow S \rightarrow_{p:=q=r} (p,r)+S \\
\mid\ \rightarrow_{=_{q\bot}} : \forall p, q, r\colon rZ.\, \forall S\colon \mathit{State}.\ q \sim_S \bot \Rightarrow S \rightarrow_{p:=q=r} (p,\text{-}r)+S \\
\mid\ \rightarrow_{=_{r\top}} : \forall p, q, r\colon rZ.\, \forall S\colon \mathit{State}.\ r \sim_S \top \Rightarrow S \rightarrow_{p:=q=r} (p,q)+S \\
\mid\ \rightarrow_{=_{r\bot}} : \forall p, q, r\colon rZ.\, \forall S\colon \mathit{State}.\ r \sim_S \bot \Rightarrow S \rightarrow_{p:=q=r} (p,\text{-}q)+S.
\end{array}
$$

B Axioms for arrays

Parameter get: $\forall n\colon \mathit{nat}.\, \forall A\colon \mathit{Set}.\, \forall \mathit{Ar}\colon (\mathit{Array}\ n\ A).\, \forall m\colon \mathit{nat}.\, \forall H\colon m < n.\, A.$

Parameter set: $\forall n\colon \mathit{nat}.\, \forall A\colon \mathit{Set}.\, \forall \mathit{Ar}\colon (\mathit{Array}\ n\ A).\, \forall m\colon \mathit{nat}.\, \forall H\colon m < n.$
$\quad \forall v\colon A.\, (\mathit{Array}\ n\ A).$

Parameter gen: $\forall n\colon \mathit{nat}.\, \forall A\colon \mathit{Set}.\, \forall f\colon \mathit{nat} \rightarrow A.\, (\mathit{Array}\ n\ A).$

Axiom setDef_1: $\forall n\colon \mathit{nat}.\, \forall A\colon \mathit{Set}.\, \forall \mathit{Ar}\colon (\mathit{Array}\ n\ A).\, \forall m\colon \mathit{nat}.\, \forall H\colon m < n.$
$\quad \forall v\colon A.\, (\mathit{get}\ n\ A\ (\mathit{set}\ n\ A\ \mathit{Ar}\ m\ H\ v)\ m\ H) = v.$

Axiom setDef_2: $\forall n\colon \mathit{nat}.\, \forall A\colon \mathit{Set}.\, \forall \mathit{Ar}\colon (\mathit{Array}\ n\ A).\, \forall m_1, m_2\colon \mathit{nat}.$
$\quad \forall H_1\colon m_1 < n.\, \forall H_2\colon m_2 < n.\, \forall H\colon m < n.\, \forall v\colon A.\, m_1 \neq m_2 \Rightarrow$
$\qquad (\mathit{get}\ n\ A\ (\mathit{set}\ n\ A\ \mathit{Ar}\ m_1\ H_1\ v)\ m_2\ H_2) = (\mathit{get}\ n\ A\ \mathit{Ar}\ m_2\ H_2).$

Axiom genDef: $\forall n\colon \mathit{nat}.\, \forall A\colon \mathit{Set}.\, \forall m\colon \mathit{nat}.\, \forall f\colon \mathit{nat} \rightarrow A.\, \forall H\colon m < n.$
$\quad (\mathit{get}\ n\ A\ (\mathit{gen}\ n\ A\ f)\ m\ H) = (f\ m).$

Axiom getIrr: $\forall n\colon \mathit{nat}.\, \forall A\colon \mathit{Set}.\, \forall \mathit{Ar}\colon (\mathit{Array}\ n\ A).\, \forall m_1, m_2\colon \mathit{nat}.$
$\quad \forall H_1\colon m_1 < n.\, \forall H_2\colon m_2 < n.\, m_1 = m_2 \Rightarrow$
$\qquad (\mathit{get}\ n\ A\ \mathit{Ar}\ m_1\ H_1) = (\mathit{get}\ n\ A\ \mathit{Ar}\ m_2\ H_2).$

Axiom setIrr: $\forall n\colon \mathit{nat}.\, \forall A\colon \mathit{Set}.\, \forall \mathit{Ar}\colon (\mathit{Array}\ n\ A).\, \forall m_1, m_2\colon \mathit{nat}.$
$\quad \forall H_1\colon m_1 < n.\, \forall H_2\colon m_2 < n.\, \forall v\colon A.\, m_1 = m_2 \Rightarrow$
$\qquad (\mathit{set}\ n\ A\ \mathit{Ar}\ m_1\ H_1\ v) = (\mathit{set}\ n\ A\ \mathit{Ar}\ m_2\ H_2\ v).$