ANR-08-DEFI-005
DECERT

# Task 4: Preliminary Report on a Generic Proof Format for SMT Solvers

The Decert Consortium

January 5, 2010

# Chapter 1

# Introduction

Satisfiability Modulo Theory (SMT) consists in deciding the satisfiability of formulae belonging to a combination of theories *e.g.,* EUF[1] + LRA[2]. Over the past few years, the SMT competition (SMT-COMP) has witnessed a dramatic improvement of SMT-provers. Current SMT-provers are very optimised and engineered in order to decide formulae of industrial size.

For the sake of the SMT competition the output of SMT-solvers is just a string: `sat` or `unsat`. This information is rich enough to judge for the speed and for the relative soundness of the tools – especially when the status of formulae is known beforehand. As the tools are getting more complex the trust in the status (`sat` or `unsat`) is getting questionable.

The purpose of the current task is to tackle this problem and propose a generic *proof format* for SMT-provers. Compared to existing approaches our objective is to propose a format that is sufficiently generic such that:

1. any SMT-solver could generate a proof without too much effort;
2. the proof could be checked by an external verifier.

Our assessment is that current formats are either easy to generate but hard to check or hard to generate and easy to check. For instance, the SMT-solver `clsat` generates proofs that are already genuine proofs objects of the logical framework LF [10]. This approach is very challenging and intellectually attractive. Yet efficient generation and checking of LF proofs is still an open research area. Alike `cla`, in the Decert project we advocate for a clear distinction between the proof generated by the SMT-solver and the proof-object that would be generated by the checker. As a pay-off our proof format should be easier to produce and not require a substantial re-engineering of the SMT-solver. There are other proof-generating SMT-provers such as Z3 [4, 3] and CVC3 [1]. For those the proof format is not totally formalised and proof reconstruction in a skeptical proof assistant is not a trivial task [6]. One difficulty lies in the fact that certain proof steps of the SMT-provers are kept implicit and do not appear in the proof trace. We aim at providing a proof format that is *proof-assistant friendly*. Validating proofs generated by SMT-provers is the objective of Task 5 of WP2.

[1] EUF is the logic of equality and uninterpreted functions
[2] LRA stands for linear real arithmetic

When a formula is satisfiable, a checkable account of this fact is a detailed model of the formula, such that every term, atom, literal, and subformula has a precise value. Giving this detailed model may be a problem when handling formulas with quantifiers. If the formula is unsatisfiable, SMT-solvers derive an inconsistency from the original formula. A proof is thus a checkable derivation, from the original formula, to an object which is trivially recognised as being an inconsistency. The *context* of the proof is the set of all deduced facts. Initially, the context contains the original formula. Rules derive new formulas from formulas in the context, and add those new formulas to the context. At the end of the proof, the empty clause (noted `()`) belongs to the context.

Defining a proof format aims at providing a clear interface between SMT-solvers that generate proofs and checkers that verify the correctness of the generated proofs. This version of the document is a preliminary version of the proof format. Its origin is the current proof format of the SMT-prover veriT [2]. We volunteerly restrict ourself to a specific fragment of the logics current SMT-solvers can deal with. This will let us experiment rapidly and get feedback on how the format should evolve and be improved. We are considering only quantifier-free formulas with uninterpreted functions, equality and linear arithmetic. In particular, this version of the document does not take quantifiers into account, and thus does not mention skolemization and instantiation.

The work of the solver is usually composed of several phases. The proof format will have to address all of them. First, the formula may be rewritten: the formula is transformed to an hopefully simpler and equivalent one, by identifying subformulas and subterms that clearly can be rewritten to simpler equivalent forms. Provers implement some rewriting rules, and those rewriting rules will have their derivation counterparts.

SMT-solvers being based on the SAT-solvers technology, they also require conjunctive normal forms (CNF). The transformation phase, that translates the original formula to a set of clauses also requires a proof. Each clause of the CNF becomes a new fact added to the context. From those clauses new clauses can be derived using resolution. Resolution is a complete method for the satisfiability of propositional formulas.

Reasoning about theories interacts with the SAT-solver through conflict clauses. These conflict clauses are tautologies according to the theories in action. To each theory will correspond a set of rules that will add clauses in the context. Equality exchange between different parts of the reasoner can simply be seen as resolution.

# Chapter 2

# Proof Format Structure

## 2.1 Rationale

The basic idea is that a proof $\Pi$ is represented by a list of rules $\rho_i$. So we have $\Pi = [\rho_1, \rho_n, \ldots, \rho_n]$. The proof is then intrinsically sequential: each rule explains how a new fact can be derived from the previous ones. The checking process uses a context $\Gamma$ that is a (partial) mapping from identifiers to clauses. It proceeds as follows:

- Initially this context is empty:

$$\Gamma_{init} = \emptyset$$

- The invariant maintained throughout the checking of the proof is that all the clauses in the invariants are either tautologies of the logic or logic consequences of the initial formula.

- The deduction relation $\vDash$ is defined as follows:
  $\Gamma, \rho \vDash \varphi$ holds if given the context $\Gamma$ the rule $\rho$ allows to deduce the clause $\varphi$.

- The relation $\rightarrow$ between contexts is defined as follows:
  $\Gamma, \Pi \rightarrow \Gamma'$ holds if in the context $\Gamma$ the proof $\Pi$ allows to construct a context $\Gamma'$. It consists of the following two rules:

$$\frac{}{\Gamma, [] \rightarrow \Gamma}$$

$$\frac{\Gamma, \rho \vDash \varphi \quad \Gamma[i \mapsto \varphi], \Pi \rightarrow \Gamma'}{\Gamma, (i, \rho) :: \Pi \rightarrow \Gamma'}$$

- The proof checking succeeds if from the initial context $\Gamma_{init}$ after consuming the whole proof $\Pi$ we produce a context $\Gamma'$ which contains the empty clause:

$$\Gamma_{init}, \Pi \rightarrow \Gamma' \wedge \exists i\, \Gamma'(i) = ()$$

The key aspect of the proof format is how much information is put in the $\rho$ to facilitate the deduction $\Gamma, \rho \vDash \varphi$ for the checker. A crucial information in

that respect are the dependencies: $\rho$ should explicitely give the information of which parts of $\Gamma$ (i.e. which indexes) are needed in order to derive $\Gamma, \rho \vDash \varphi$. A second information is the method used by the derivation. This information will be referred as the name of the rule in the following.

## 2.2 Explicit syntax

We propose our format as an extension of the SMTLIB format [9].

### 2.2.1 Header

An SMT problem usually comes as a benchmark that has a name and a list of attributes. We just add an extra attribute `proof` that contains the proof. Here is the extension of the grammar:

```
<attribute> ::= logic = <logic>
                | formula = <formula>
                | status = <status>
                | assumption = <assumptions>
                | extrasorts = <sorts>
                | extrafuns = <function>+
                | extrapreds = <pred>+
                | notes = <text>
                | proof = <proof>
<benchmark> ::= benchmark <name> begin <attribute>+ end
```

### 2.2.2 Formulae

The expressions (terms and formulas) manipulated in the proof are just written as SMTLIB expressions, with the added capability of *globally* indexing term and formulae: `#n: F` stores the formula `F` at the index `n` while `#m` refers to the formula at location `m`. Here is the extension of the grammar.

```
<term>        ::= <var> | <numeral> | <rational> | <identifier>
                  | (<symbol> <term>+)
                  | ( ite <formula> <term> <term>)
                  | #<number> (: <term>)?
<atom>        ::= true | false | <var> | <identifier>
                  | (<symbol> <term>+)
<connector>   ::= not | implies | if_then_else
                  | and | or | xor | iff
<quantifier>  ::= exists | forall
<qvar>        ::= ( <var> <sort> )
<formula>     ::= <atom>
                  | ( <connector> <formula>+ )
                  | ( <quantifier> <qvar>+ <formula>)
                  | ( let (<var> <term>) <formula>)
                  | ( flet (<var> <formula>) <formula>)
                  | #<number> (: <formula>)?
```

### 2.2.3 Proof

The proof format is a sequence of lines of the form

```
<num_id>:(<anum_id> <clause> <justification>)
```

where

- `<num_id>` is a numeric identifier;
- `<anum_id>` is the alphanumeric identifier of the rule (see hereunder).
- `<clause>` is a disjunctive list `(L1 ... Lm)` where `L1`,... `Lm` are formulas (`<formula>`). In the following rules, this disjunctive set of formulas may be referred by the previous numeric identifier (`<num_id>`).
- `<justification>` is a hint allowing to derive the clause.

A `<justification>` is either atomic or compound. An atomic `<justification>` corresponds to the instantiation of an axiom and takes as arguments `<clause_id>*` `<param>*` where

- `<clause_id>*` is a (possibly empty) sequence of clause identifiers in the context
- `<param>*` is a (possibly empty) sequence of integers.

The number of those (`<clause_id>` and `<param>`) arguments depends on the rule specified by `<anum_id>`. A compound `justification` is itself a sequence of rules.

The grammar of the proof format is therefore the following

```
<proof>         ::= <deduction>+
<deduction>     ::= <num_id>:(<anum_id> <clause> <justification>)
<clause>        ::= (<formula>*)
<justification> ::= <atomic>
                  | <deduction>*
<atomic>        ::= <clause_id>* <param>*
<clause_id>     ::= <num_id>
<param>         ::= <num_id>
```

The proof format is recursive in that a `<deduction>` can be justified by a list `<deduction>+` of deductions. The inner deductions take place in a new context initialised by the clause. Suppose a compound rule $\rho$ justifying a conflict clause $\varphi$ by a proof $\Pi$. To check the validity of the clause, the proof $\Pi$ is evaluated in a novel context $\Gamma_\varphi$ initialised with the negation of the clause. The clause holds if after evaluating the proof $\Pi$ it is possible to derive the empty clause. This is summarised by the following inference.

$$\frac{\Gamma_\varphi, \Pi \to \Gamma' \quad \exists i, \Gamma'(i) = \texttt{()}}{\Gamma, \rho \vDash \varphi}$$

where $\varphi = \varphi_1 \vee \cdots \vee \varphi_n$ and $\Gamma_\varphi = [1 \mapsto \neg\varphi_1, \ldots, n \mapsto \neg\varphi_n]$.

The whole proof checking process consists in deriving the empty clause `()` from the initial context. The context is a (partial) mapping from identifiers to clauses. An invariant maintained throughout the checking is that all the clauses in the invariants are either tautologies of the logic or logic consequences of the initial formula.

# Chapter 3

# Proof format for SAT

In this section `a`, `b`, `c`, `a_1`, ... `a_n` denote any formulas, and `i`, `j`, `i_1`,... `i_m` denote any strictly positive numbers.

## Input formula

The following rule introduces the starting point of a proof, i.e. the input formula. The input formula is added to the context. If the formula is unsatisfiable, the empty clause will eventually be deduced (using deduction rules presented later) from the input formula and tautologies of the logic

**Rule** `input`: `(input (<formula>))`

*Remark :* the second element of the rule is a disjunctive set of formulas, containing a single formula. Although, it may have been possible to simply give the formula, this form was chosen for uniformity with the other rules.

## The resolution rule

The resolution rule may be used by several modules of SMT solvers, including theory reasoning, the Nelson-Oppen combination framework, and obviously the SAT-solving techniques. More precisely, the resolution rule describes a chain of binary resolutions

**Rule** `resolution`: `(resolution (a_1 ... a_n) i_1 ... i_m)`

The clause `(a_1 ... a_n)` is obtained by chain-resolving the clauses indexed by `i_1`,... `i_m` in the context.

## Tautologies

All the rules in this subsection introduce tautologies in the context. They are useful to define the semantics of Boolean connectors and interpreted symbols. These rules are mainly used in the CNF transformation phase.

### Boolean constants

This two following rules acts as definitions for the `true` and `false` atoms. For instance, the first rule adds the singleton clause `(true)` to the context, and it

is thereafter possible to use the fact that formula `true` is valid.

**Rule** `true`: `(true (true))`

**Rule** `false`: `(false ((not false)))`

### Tautologies for `and`

These rules define the `and` connector.

**Rule** `and_pos`: `(and_pos ((not (and a_1 ... a_n)) a_i) i)`

**Rule** `and_neg`: `(and_neg ((and a_1 ... a_n) (not a_1) ... (not a_n)))`

*Example :* `5:(and_pos ((not (and (p a) (not (p b)))) (p a)) 1)`

By adding this tautology in the context, we state that either the formula `(and (p a) (not (p b)))` is false, or `(p a)` is true. The last argument of the rule is the position of the conjunct (starting from 1).

*Example :* `7:(and_neg ((and (p a) (not (p b))) (not (p a)) (p b)))`

### Tautologies for `or`

**Rule** `or_pos`: `(or_pos ((not (or a_1 ... a_n)) a_1 ... a_n))`

**Rule** `or_neg`: `(or_neg ((or a_1 ... a_n) (not a_i)) i)`

### Tautologies for `xor`

**Rule** `xor_pos1`: `(xor_pos1 ((not (xor a b)) a b))`

**Rule** `xor_pos2`: `(xor_pos2 ((not (xor a b)) (not a) (not b)))`

**Rule** `xor_neg1`: `(xor_neg1 ((xor a b) a (not b)))`

**Rule** `xor_neg2`: `(xor_neg2 ((xor a b) (not a) b))`

### Tautologies for `implies`

**Rule** `implies_pos`: `(implies_pos ((not (implies a b)) (not a) b))`

**Rule** `implies_neg1`: `(implies_neg1 ((implies a b) a))`

**Rule** `implies_neg2`: `(implies_neg2 ((implies a b) (not b)))`

### Tautologies for `equiv`

**Rule** `equiv_pos1`: `(equiv_pos1 ((not (iff a b)) a (not b)))`

**Rule** `equiv_pos2`: `(equiv_pos2 ((not (iff a b)) (not a) b))`

**Rule** `equiv_neg1`: `(equiv_neg1 ((iff a b) (not a) (not b)))`

**Rule** `equiv_neg2`: `(equiv_neg2 ((iff a b) a b))`

### Tautologies for `ite`

**Rule** `ite_pos1`: `(ite_pos1 ((not (if_then_else a b c)) a c))`

**Rule** `ite_pos2`: `(ite_pos2 ((not (if_then_else a b c)) (not a) b))`

**Rule** `ite_neg1`: `(ite_neg1 ((if_then_else a b c) a (not c)))`

**Rule** `ite_neg2`: `(ite_neg2 ((if_then_else a b c) (not a) (not b)))`

## Deductive rules

Together with the previous tautologies, the following deduction rules are mainly used in the CNF transformation phase. Whereas tautologies are used to define the meaning of sub-formulas deep inside the input formula, the following rules help to decompose the input formula at the top-most level.

### Deductive rules for `and`

**Rule** `and`: `(and a_i j i)`

*Remark :* The disjunctive set of formulas at position `j` should be a singleton, containing the sole formula `(and a_1 ... a_n)`. The number `i` explicitly gives the position of the derived formula in the conjunction.

**Rule** `not_and`: `(not_and ((not a_1) ... (not a_n)) i)`

*Remark :* The disjunctive set of formulas at position `i` should be a singleton, containing the sole formula `(not (and a_1 ... a_n))`.

### Deductive rules for `or`

**Rule** `not_or`: `(not_or ((not a_i)) j i)`

*Remark :* The disjunctive set of formulas at position `j` should be a singleton, containing the sole formula `(not (or a_1 ... a_n))`.

**Rule** `or`: `(or (a_1 ... a_n) i)`

*Remark :* The disjunctive set of formulas at position `i` should be a singleton, containing the sole formula `(or a_1 ... a_n)`.

### Deductive rules for `xor`

**Rule** `xor1`: `(xor1 (a b) i)`

**Rule** `xor2`: `(xor2 ((not a) (not b)) i)`

*Remark :* In the two previous rules the disjunctive set of formulas at position `i` should be a singleton, containing the sole formula `(xor a b)`

**Rule** `not_xor1`: `(not_xor1 (a (not b)) i)`

**Rule** `not_xor2`: `(not_xor2 ((not a) b) i)`

*Remark :* In the two previous rules the disjunctive set of formulas at position `i` should be a singleton, containing the sole formula `(not (xor a b))`

### Deductive rules for `implies`

**Rule** `implies`: `(implies ((not a) b) i)`

*Remark :* The disjunctive set of formulas at position `i` should be a singleton, containing the sole formula `(implies a b)`.

**Rule** `not_implies1`: `(not_implies1 (a) i)`

**Rule** `not_implies2`: `(not_implies2 ((not b)) i)`

*Remark :* In the two previous rules the disjunctive set of formulas at position `i` should be a singleton, containing the sole formula `(not (implies a b))`.

### Deductive rules for `equiv`

**Rule** `equiv1`: `(equiv1 ((not a) b) i)`

**Rule** `equiv2`: `(equiv2 (a (not b)) i)`

*Remark :* In the two previous rules the disjunctive set of formulas at position `i` should be a singleton, containing the sole formula `(iff a b)`.

**Rule** `not_equiv1`: `(not_equiv1 (a b) i)`

**Rule** `not_equiv2`: `(not_equiv1 ((not a) (not b)) i)`

*Remark :* In the two previous rules the disjunctive set of formulas at position `i` should be a singleton, containing the sole formula `(not (iff a b))`.

### Deductive rules for `if_then_else`

**Rule** `ite1`: `(ite1 (a c) i)`

**Rule** `ite2`: `(ite2 ((not a) b) i)`

*Remark :* In the two previous rules the disjunctive set of formulas at position `i` should be a singleton, containing the sole formula `(if_then_else a b c)`.

**Rule** `not_ite1`: `(not_ite1 (a (not c)) i)`

**Rule** `not_ite2`: `(not_ite2 ((not a) (not b)) i)`

*Remark :* In the two previous rules the disjunctive set of formulas at position `i` should be a singleton, containing the sole formula `(not (if_then_else a b c))`

# Chapter 4

# Proof Format for EUF

In the Nelson-Oppen combination framework the logic of equality and uninterpreted functions (EUF) allows for different theories to interact by sharing equalities. EUF is therefore a theory at the core of SMT provers. In Section 4.1 we present the current EUF proofs generated by veriT. This is a format that does not put too much burden on the SMT prover. In Section 4.2 we present an alternative format which explicits all the details of the proof and is thus more geared toward a verification by a proof-assistant. It is important to note that both styles are encoded in the very same proof format and thus accredit the versatility of the format. Moreover, we have prototyped a proof post-processing transforming the veriT format into the alternative format.

## 4.1 Current veriT format for EUF

The main characteristics of EUF proofs generated by veriT are that symmetry is implicit and that resolution is used to combine proof arguments. Except resolution, which is a general-purpose proof rule, EUF proofs are made of the following rules.

**Rule** `eq_transitive`:

```
(eq_transitive (
        (not (= x_1 x_2)) ... (not (= x_{n-1} x_n))
        (= x_1 x_n)))
```

**Rule** `eq_congruent`:

```
(eq_congruent ((not (= x_1 y_1)) ... (not (= x_n y_n))
               (= (f x_1 ... x_n) (f y_1 ... y_n))))
```

**Rule** `eq_congruent_pred`:

```
(eq_congruent_pred ((not (= x_1 y_1)) ... (not (= x_n y_n))
                   (not (p x_1 ... x_n)) (p y_1 ... y_n)))
```

## 4.2 Alternative EUF format

For the alternative EUF format, EUF reasoning is explicitly separated from the current proof flow and takes the form of a conflict clause

```
(EUF (F1 ... Fn)  euf_justification)
```

where the justification is a combination of the deduction rules presented below.

**Rule** `euf_reflexive`:`(euf_reflexive ((= x x)))`
*Remarks:* This rule implements the reflexivity axiom of equality.

**Rule** `euf_symmetric`: `(euf_symmetric ((= x y)) i)`
*Remarks:* Index `i` refers to the singleton clause `((= y x))`.

**Rule** `euf_transitive`: `(euf_transitive ((= x z)) i j)`
*Remarks:* Index `i` refers to the singleton clause `((= x y))` for some `y` and index `j` refers to the singleton clause `((= y z))`.

**Rule** `euf_congruent`:

```
(euf_congruent ((= (f x_1 ... x_n) (f y_1 ... y_n))) i j k)
```

*Remarks:* Index `i` refers to the singleton clause `((= (f x_1...x_n) (f z_1...z_n)))` and index `j` refers to the singleton clause `((= z_k t))`. The resulting clause is such that `y_i` is `z_i` for all indexes except `k` for which `y_k` is `t`.

**Rule** `euf_congruent_pred`:

```
(euf_congruent_pred ((p x_1 ... x_n)) i j k)
```

*Remarks* Index `i` refers to the singleton clause `((p y_1 ... y_n))` and index `j` refers to the singleton clause `((= y_k z))`. The resulting clause is such that `x_i` is `y_i` for all indexes except `k` for which `x_k` is `z`.

## 4.3 An example

In Figure 4.1, we give an example of a very simple formula, and the proof output by veriT. Line 1 is the input formula, lines 2 to 7 account for the Conjunctive Normal Form transformation. The remaining lines record the Boolean and theory reasoning leading to the empty clause.

```
(benchmark example
  :logic QF_UF
  :extrafuns ((a U) (b U) (c U) (f U U))
  :extrapreds ((p U))
  :formula (and (= a c) (= b c)
                (or (not (= (f a) (f b)))
                    (and (p a) (not (p b))))))

1:(input ((and (= a c) (= b c)
               (or (not (= (f a) (f b))) (and (p a) (not (p b)))))))
2:(and ((= a c)) 1 1)
3:(and ((= b c)) 1 2)
4:(and ((or (not (= (f a) (f b))) (and (p a) (not (p b))))) 1 3)
5:(and_pos ((not (and (p a) (not (p b))))) (p a)) 1)
6:(and_pos ((not (and (p a) (not (p b))))) (not (p b))) 2)
7:(or ((not (= (f a) (f b))) (and (p a) (not (p b)))) 4)
8:(eq_congruent ((not (= b a)) (= (f a) (f b))))
9:(eq_transitive ((not (= b c)) (not (= a c)) (= b a)))
10:(resolution ((= (f a) (f b)) (not (= b c)) (not (= a c))) 8 9)
11:(resolution ((= (f a) (f b))) 10 2 3)
12:(resolution ((and (p a) (not (p b)))) 7 11)
13:(resolution ((p a)) 5 12)
14:(resolution ((not (p b))) 6 12)
15:(eq_congruent_pred ((not (= b a)) (p b) (not (p a))))
16:(eq_transitive ((not (= b c)) (not (= a c)) (= b a)))
17:(resolution ((p b) (not (p a)) (not (= b c)) (not (= a c))) 15 16)
18:(resolution () 17 2 3 13 14)
```

Figure 4.1: A simple example with its proof.

# Chapter 5

# Proof Format for Arithmetic

In this section we instantiate the proof format for two fragments of arithmetic : difference logic (DL) and linear real arithmetic (LRA). From both the syntactic and semantics point of view DL is a strict subset of LRA. This is reflected in the proof rules used to justify the conflict clauses. The rules for DL are a strict subset of those for LRA.

For our purpose justifications based on Peano axioms are not adequate. They are much too verbose and low level. For instance it would be somewhat silly to justify an equality like `512*512=262144` using the Peano definition of addition and multiplication. Here we follow a pragmatic approach pioneered by Necula for his PCC work [7]. For justifying arithmetic proofs Necula and Lee [8] makes use of the following axiom in order to prove the equality of expressions:

$$\frac{}{E_1 = E_2} \quad \mathbf{arith}(E_1 \simeq E_2)$$

In this axiom $\simeq$ denotes the equality between expressions modulo associativity and commutativity of addition and multiplication. In our proof format this axiom corresponds to the atomic rule `ring`.
*Syntax:* `(ring ((= e_1 e_2)) )`
*Remarks:* Checking the validity of the rule can be done by normalising `e_1` and `e_2` and verifying that the normal forms are equal. There are no restriction on the normal form. For instance common choices are an ordered list of pairs variable coefficient or Horner normal form. This is the latter that is implemented by the `ring` Coq tactic [5].

In the rest of this chapter we introduce three sets of rules. In Section 5.1 we present proof rules for canonising linear formulae. These are deduction rules used to pre-process formulae. In Section 5.2 we present proof rules sufficient for DL and in Section 5.3 we present additional rules needed for LRA.

## 5.1 Canonisation of linear expressions

The purpose of the canonisation rules is to transform arithmetic formulae of the form $e \bowtie e'$ (where $\bowtie \in \{<, \leq, >, =\}$) into equivalent formulae of the form

$e'' \bowtie' 0$ where $\bowtie' \in \{\geq, >, =\}$.

**Rule** `le_0`:`(le_0 ((>= (- e' e) 0)) i)`
*Remarks:* The singleton clause at index `i` has the form `((<= e e'))`.

**Rule** `lt_0`:`(lt_0 ((> (- e' e) 0)) i)`
*Remarks:* The singleton clause at index `i` has the form `((< e e'))`.

**Rule** `ge_0`:`(ge_0 ((>= (- e e') 0)) i)`
*Remarks:* The singleton clause at index `i` has the form `((>= e e'))`.

**Rule** `gt_0`:`(gt_0 ((> (- e e') 0)) i)`
*Remarks:* The singleton clause at index `i` has the form `((> e e'))`.

**Rule** `eq_0`:`(eq_0 ((= (- e e') 0)) i)`
*Remarks:* The singleton clause at index `i` has the form `((= e e'))`.

## 5.2 Difference logic

**Rule** `dl_disequality`: `(dl_disequality ( (< a b) (= a b) (> a b) ) )`
*Remarks:* This is a tautology of arithmetic: when comparing two integers $i$ and $j$ there are three possibilities: $i < j$, $i = j$ or $i > j$.

**Rule** `add_ge_ge`: `(add_ge_ge ((>= (+ e_1 e_2) 0)) i j)`
*Remarks:* Index `i` refers to the singleton clause `((>= e_1 0))` and index `j` refers to the singleton clause `((>= e_2 0))`.

**Rule** `add_ge_gt`: `(add_ge_gt ((> (+ e_1 e_2) 0)) i j)`
*Remarks:* Index `i` refers to the singleton clause `((>= e_1 0))` and index `j` refers to the singleton clause `((> e_2 0))`.

**Rule** `add_ge_eq`: `(add_ge_eq ((>= (+ e_1 e_2) 0)) i j)`
*Remarks:* Index `i` refers to the singleton clause `((>= e_1 0))` and index `j` refers to the singleton clause `((= e_2 0))`.

**Rule** `add_gt_eq`: `(add_gt_eq ((> (+ e_1 e_2) 0)) i j)`
*Remarks:* Index `i` refers to the singleton clause `((> e_1 0))` and index `j` refers to the singleton clause `((= e_2 0))`.

**Rule** `add_eq_eq`: `(add_eq_eq ((= (+ e_1 e_2) 0)) i j)`
*Remarks:* Index `i` refers to the singleton clause `((= e_1 0))` and index `j` refers to the singleton clause `((= e_2 0))`.

**Rule** `neg_ge`: `(neg_ge ((not (>= c 0))))`
*Remarks:* The constant `c` is a negative rational constant. This rule will be used to exhibit a contradiction.

**Rule** `neg_gt`: *Syntax:* `(neg_gt ((not (> c 0))))`
*Remarks:* The constant `c` is a negative rational constant. This rule will be used to exhibit a contradiction.

**Rule** `dl_generic`: `(dl_generic (f_1 ... f_n) justification)`

The disjunctive set of formulas `(f_1 ... f_n)` is a tautology according to the difference logic decision procedure. It can be (optionally) justified using the previous rules for difference logic.

Suppose that the conflict clause `(f_1 ... f_n)` is a minimal conflict clause and is of the form `( (not (>= e_1 0)) ... (not (>= e_n 0)))` where each

`e_n` is of the form `e_n = (+ (- x_n x_1) c_n)` the justification can be automatically derived and takes the following general form:

```
1: (add_ge_ge (>= (+ e_1 e_2) 0) 1 2)
...
1: (add_ge_ge (>= (+ e_1 ... e_n) 0) n-1 n)
2: (ring (= (+ e_1 ... e_n) c))
3: (eq_congruent_pred (
        (not (= (+ e_1 ... e_n) c))
        (not (= 0 0))
        (not (>= (+ e_1 ... e_n) 0))
        (>= c 0)))
4: (neg_ge (not (>= c 0)))
5: (resolution () 1 2 3 4)
```

In the proof the constant `c` is the sum of `c_1+...+c_n`.

## 5.3 Linear Real arithmetic

For DL a justification can be reconstructed syntactically for the conflict clause. For LRA the situation is more complicated. To obtain a contradiction the formulae of the conflict clause have to be multiplied by positive coefficients. The following rules extend the rule of signs to accomodate for multiplications.

**Rule** `cst_ge`: `(pos_ge ((>= c 0)))`
*Remarks:* The constant `c` is a positive rational constant.

**Rule** `mul_ge`: `(mul_ge ((>= (* e_1 e_2) 0)) i j)`
*Remarks:* Index `i` refers to the singleton clause `(>= e_1 0)` and index `j` refers to the singleton clause `((>= e_2 0))`

**Rule** `mul_gt`: `(mul_gt ((>= (* e_1 e_2) 0)) i j)`
*Remarks:* Index `i` refers to the singleton clause `(>= e_1 0)` and index `j` refers to the singleton clause `((> e_2 0))`

**Rule** `mul_eq`: `(mul_eq ((= (* e_1 e_2) 0)) i)`
*Remarks:* Index `i` refers to the singleton clause `((= e_1 0))`.

**Rule** `la_generic`: `(la_generic (f_1 ... f_n) justification)`
*Remarks:* The disjunctive set of formulas `(f_1 ... f_n)` is a tautology according to the linear arithmetic decision procedure.

Suppose that the conflict clause `(f_1 ... f_n)` is a minimal conflict clause and is of the form `( (not (>= e_1 0)) ... (not (>= e_n 0)))`. A justification can follow the following pattern.

```
n+1 : (pos_ge (>= c_1 0))
1   : (mul_ge (>= (* c_1 e_1) 0) n+1 1)
n+1 : (pos_ge (>= c_2 0))
2   : (mul_ge (>= (* c_2 e_2) 0) n+1 2)
2   : (add_ge (>= (+ (* c_1 e_1) (* c_2 e_2)) 0) 1 2)
...
n   : (add_ge (>= (+  (* c_1 e_1) ... (* c_n e_n)) 0) n-1 n)
n+1 : (ring (= (+  (* c_1 e_1) ... (* c_n e_n)) c))
3: (eq_congruent_pred (
```

```
        (not (= (+  (* c_1 e_1) ... (* c_n e_n)) c)
        (not (= 0 0))
        (not (>= (+  (* c_1 e_1) ... (* c_n e_n)) 0))  (>= c 0)))
4: (neg_ge (not (>= c 0)))
5: (resolution () n n+1 3 4)
```

# Chapter 6

# Conclusion and further work

This report presents a first draft of the proof format. The design is a compromise between what can be output by an SMT-solver and what is needed in order for a simple checker to be able to verify a proof. This preliminary version of the proof format is far from being perfect but it is detailed enough so we can start experimenting with it. We strongly believe that the feedback of these experiments will help us improving it.

For the moment, the format may contain too many rules. This was done on purpose to give a very precise and detailed semantic of the proof. We need to evaluate how tractable this is in practice to deal with so many rules. If it happens to be unpractical, we would need to develop solutions in order to reduce this set. A way to go is to increase in the proof format the number of information that is implicit. An interesting alternative could be to add a mechanism to add rules of the fly. In a similar way that the SMTLIB makes it possible to describe new theories, we could extend the format and give the capability to describe new rules and their semantics inside the proof format.

Another thing that needs to be evaluated carefully is the cost of this proof format: how easy it is to produce for the SMT-solver, how easy is it to check a proof in this format. Specially for the case of PCC, we are very interested at evaluating how fast and with how much memory it is possible to check these proofs.

Finally,this format should also be a base for building a set of tools around it. It would be interesting to see how easy it is to write converters to other formats. Also, given a proof, transformers could also be developed so to refactor and optimize it.

# Bibliography


[1] C. Barrett and C. Tinelli. CVC3. In *CAV '07*, volume 4590 of *LNCS*, pages 298–302. Springer-Verlag, 2007.

[2] T. Bouton, D. C. B. de Oliveira, D. Déharbe, and P. Fontaine. veriT: an open, trustable and efficient SMT-solver. In *(CADE'09)*, LNCS. Springer-Verlag, 2009. To appear.

[3] L. M. de Moura and N. Bjørner. Proofs and Refutations, and Z3. In *Proc. of the LPAR 2008 Workshops*, volume 418 of *CEUR Workshop Proceedings*, 2008.

[4] L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS'08*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[5] Benjamin Grégoire and Assia Mahboubi. Proving Equalities in a Commutative Ring Done Right in Coq. In *TPHOLs*, pages 98–113, 2005.

[6] Sean McLaughlin, Clark Barrett, and Yeting Ge. Cooperating theorem provers: A case study combining HOL-Light and CVC Lite. In *PDPAR '05*, volume 144(2) of *ETCS*, pages 43–51. Elsevier, 2006.

[7] George C. Necula. Proof-Carrying Code. In *Proc. of 24th ACM Symp. on Principles of Programming Languages (POPL'97)*, pages 106–119. ACM, 1997.

[8] George C. Necula and Peter Lee. Proof Generation in the Touchstone Theorem Prover. In *CADE-17: Proceedings of the 17th International Conference on Automated Deduction*, pages 25–44, London, UK, 2000. Springer-Verlag.

[9] SMTLIB. http://combination.cs.uiowa.edu/smtlib.

[10] A. Stump and D. L. Dill. Faster Proof Checking in the Edinburgh Logical Framework. In *CADE-18*, pages 392–407. Springer-Verlag, 2002.