Rocket-Fast Proof Checking for SMT Solvers

Michał Moskal

University of Wrocław, Poland

Abstract. Modern Satisfiability Modulo Theories (SMT) solvers are used in a wide variety of software and hardware verification applications. Proof producing SMT solvers are very desirable as they increase confidence in the solver and ease debugging/profiling, while allowing for scenarios like Proof-Carrying Code (PCC). However, the size of typical proofs generated by SMT solvers poses a problem for the existing systems, up to the point where proof checking consumes orders of magnitude more computer resources than proof generation. In this paper we show how this problem can be addressed using a simple term rewriting formalism, which is used to encode proofs in a natural deduction style. We formally prove soundness of our rules and evaluate an implementation of the term rewriting engine on a set of proofs generated from industrial benchmarks. The modest memory and CPU time requirements of the implementation allow for proof checking even on a small PDA device, paving a way for PCC on such devices.

1 Introduction

Satisfiability Modulo Theories (SMT) [14] solvers check satisfiability of a first order formula, where certain function and constant symbols are interpreted according to a set of background theories. These theories typically include integer or rational arithmetic, bit vectors and arrays. Some SMT solvers support only quantifier free fragments of their logics, other also support quantifiers, most often through instantiation techniques. SMT solvers are often based on search strategies of SAT solvers.

The usage of background theories, instantiation techniques and efficient handling of the Boolean structure of the formula differentiates SMT solvers from first-order theorem provers based on resolution. SMT solvers are efficient for larger mostly ground formulas. This makes them good tools for hardware and software verification.

SMT solvers typically either answer that the input formula is unsatisfiable, or give some description of a model, in which the formula might be satisfiable. In terms of software verification the first answer means that the program is correct, while the second answer means, that an assertion might be violated. The model description is used to identify a specific assertion and/or execution trace.

What is troubling is that we are trusting the SMT solver, when it says the program is correct. One problem is that there might be a bug in the SMT solver, whose implementation can be largely opaque to others than the developer.

C.R. Ramakrishnan and J. Rehof (Eds.): TACAS 2008, LNCS 4963, pp. 486–500, 2008.

The other problem is that we might want to provide the evidence of program being correct to someone else, like in Proof-Caring Code [13] scenarios.

It is therefore desirable for an SMT solver to produce the proof of the unsatisfiability of formulas. The problem is that in program verification, the queries are rather huge and so are the proofs. For example formulas in the AUFLIA division of the SMT problem library[1] contain up to 130 000 distinct subterms, with an average of 8 000. The proofs we have generated are on average five times bigger than the formulas. The most complicated proof we have encountered contains around 40 000 basic resolution steps and around 1 000 000 (sub)terms in size. What is worth noting however, is that state of the art SMT solvers are able to check a vast majority of such queries in under a second. As the general expectation is that proof checking should be faster than proof generation, it becomes clear that we need very efficient means of proof checking.

1.1 Contributions

The contributions of this paper are:

- we introduce a simple, yet expressive term rewrite formalism (Sect. 2), and show it is strong enough to encode and check proofs of theory and Boolean tautologies (Sect. 3), and also NNF/CNF conversions with skolemization (Sect. 4),
- we discuss two highly efficient implementations of the proposed rewrite system (Sect. 6). In particular we discuss performance issues (Sect. 6.2) and we describe techniques to help ensure soundness of the rewrite rules (Sect. 6.1).

There are two reasons to use term rewriting as a proof checking vehicle. One is that the term rewriting is a simple formalism, therefore it is relatively easy to reason about the correctness of an implementation of the proof checker. The bulk of soundness reasoning goes at term rewrite rules level, which is much better understood and simpler to reason about than a general purpose (often low level) programming language used to implement a proof checker.

The second reason is memory efficiency, which on modern CPUs is also a key to time efficiency. We encode proof rules as rewrite rules and handle non-local conditions (like uniqueness of Skolem functions) at the meta level, which allows for the rewrite rules to be local. The idea behind the encoding of the proof rules is to take proof terms and rewrite them into the formulas that they prove. This allows for memory held by the proof terms to be immediately reclaimed and reused for the next fragment of the proof tree read from a proof file.

1.2 Proof Search in SMT Solvers

This section gives description of a proof search of an SMT solver based on DPLL and E-matching. It applies to most of the current SMT solvers.

[1] Both the library and this particular division are described further in Sect. 6.2.

In order to check unsatisfiability of a formula, an SMT solver will usually first transform it into an equisatisfiable CNF formula, while simultaneously performing skolemization. Subsequent proof search alternates between Boolean reasoning, theory reasoning, and quantifier instantiation. For the Boolean part we use resolution. Also the final empty clause is derived using resolution. Theory reasoning produces conflict clauses, which are tautologies under respective theories, e.g., $\neg(a > 7) \vee a \geq 6$ or $\neg(c = d) \vee \neg(f(c) = 42) \vee \neg(f(d) < 0)$. Quantifier reasoning is based on instantiating universal quantifiers and producing tautologies like $\neg(\forall x.\, f(x) > 0 \rightarrow P(x)) \vee \neg(f(3) > 0) \vee P(3)$. It can be thought of as just another background theory.

To make this search procedure return a proof, we need proofs of: CNF translation, Boolean tautologies and theory tautologies. By taking these three together, we should obtain a proof that the formula is unsatisfiable.

2 Definitions

Let $\mathcal{V}$ be an infinite, enumerable, set of variables. We use x and y (all symbols possibly with indices) as meta-variables ranging over $\mathcal{V}$. Let Σ be an infinite, enumerable set of function symbols, we use meta-variable f ranging over Σ. We define the set of *terms* $\mathcal{T}$, and the set of *patterns* $\mathcal{P} \subseteq \mathcal{T}$ as follows:

$$\begin{aligned}\mathcal{T} &::= x \mid f(\mathcal{T}_1, \ldots, \mathcal{T}_n) \mid \lambda x.\, \mathcal{T}_1 \mid \mathbf{cons} \cdot (\mathcal{T}_1, \mathcal{T}_2) \mid \mathbf{nil} \cdot () \mid \mathbf{build} \cdot (f, \mathcal{T}_1) \mid \\ &\quad\ \mathbf{apply} \cdot (\mathcal{T}_1, \mathcal{T}_2) \mid \mathbf{fold} \cdot (\mathcal{T}_1) \\ \mathcal{P} &::= x \mid f(\mathcal{P}_1, \ldots, \mathcal{P}_n)\end{aligned}$$

where $n \geq 0$. The notion $s \cdot (...)$ stands for a *special form*, which have particular interpretations in the term rewrite system. We will use $t_1 :: t_2$ as a syntactic sugar for $\mathbf{cons} \cdot (t_1, t_2)$, and $\mathbf{nil}$ for $\mathbf{nil} \cdot ()$.

The set of *free variables* of a term, $FV : \mathcal{T} \rightarrow \mathcal{P}(\mathcal{V})$, is defined as usual:

$$\begin{aligned}FV(x) &= \{x\} \\ FV(f(t_1, \ldots, t_n)) &= \textstyle\bigcup_{1 \leq i \leq n} FV(t_i) \\ FV(\lambda x.\, t) &= FV(t) \setminus \{x\} \\ FV(s \cdot (t_1, \ldots, t_n)) &= \textstyle\bigcup_{1 \leq i \leq n} FV(t_i)\end{aligned}$$

Note that it is also defined on $\mathcal{P}$, as $\mathcal{P} \subseteq \mathcal{T}$. Let $\mathcal{T}(A, B)$ be a set of terms built from function symbols from the set A and variables from the set $B \subseteq \mathcal{V}$ (i.e. if $t \in \mathcal{T}(A, B)$ then $FV(t) \subseteq B$). A *substitution* is a function $\sigma : \mathcal{V} \rightarrow \mathcal{T}$, which we identify with its homomorphic, capture free extension to $\sigma : \mathcal{T} \rightarrow \mathcal{T}$.

A *rewrite rule* is a pair (p, t), where $p \in \mathcal{P}$, $t \in \mathcal{T}$ and $FV(t) \subseteq FV(p)$. Let $\mathcal{R}$ be set of such rewrite rules, such that for distinct $(p, t), (p', t') \in \mathcal{R}$, p and p' do not unify. We define a *normal form* of a term t, with respect to $\mathcal{R}$ as $\boldsymbol{nf}(t)$, with the rules below. Because the function defined below is recursive it is possible for it not to terminate. If the rules below do not result in a single unique normal form for term t, then we say that $\boldsymbol{nf}(t) = \otimes$. If term has $\otimes$ as subterm, it is itself regarded as equal to $\otimes$. In practice this condition is enforced by limiting running time of the proof checker.

$$
\begin{array}{rcl}
\boldsymbol{nf}(x) & = & x \\
\boldsymbol{nf}(f(t_1,\ldots,t_n)) & = & \begin{cases} \boldsymbol{nf}(t\sigma) \\ \quad \text{for } (p,t) \in \mathcal{R} \text{ such that} \\ \quad \exists\sigma.\, p\sigma = f(\boldsymbol{nf}(t_1),\ldots,\boldsymbol{nf}(t_n)) \\ f(\boldsymbol{nf}(t_1),\ldots,\boldsymbol{nf}(t_n)) \\ \quad \text{otherwise} \end{cases} \\
\boldsymbol{nf}(\lambda x.\, t_1) & = & \lambda x.\, \boldsymbol{nf}(t_1) \\
\boldsymbol{nf}(\mathbf{apply} \cdot (\lambda x.\, t_1, t_2)) & = & \boldsymbol{nf}(t_1[x := t_2]) \\
\boldsymbol{nf}(\mathbf{build} \cdot (f, t_1 :: \cdots :: t_n :: \mathbf{nil})) & & \\
\qquad = & = & \boldsymbol{nf}(f(t_1,\ldots,t_n)) \\
\boldsymbol{nf}(\mathbf{build} \cdot (f, t_1)) & = & \mathbf{build} \cdot (f, \boldsymbol{nf}(t_1)) \quad \text{if none of the above apply} \\
\boldsymbol{nf}(s \cdot (t_1,\ldots,t_n)) & = & s \cdot (\boldsymbol{nf}(t_1),\ldots,\boldsymbol{nf}(t_n)) \quad \text{if none of the above apply}
\end{array}
$$

where $t_1[x := t_2]$ denotes a capture free substitution of x with t_2 in t_1[2].

The semantics of the $\mathbf{fold} \cdot (t)$ is not defined above. Its role is to perform theory-specific constant folding on t. Folding is implemented either inside the proof checker or by an external tool called by the proof checker. In this paper we use integer constant folding (for example $\boldsymbol{nf}(\mathbf{fold} \cdot (\mathtt{add}(20, 22))) = 42$).

The signature used throughout this paper can be divided in four categories:

1. logical connectives: `false, implies, and, or, forall, neg`
2. theory specific symbols: `eq, add, leq, minus` and natural number literals (`0, 1, 2, ...`)
3. technical machinery: `lift_known, □, sk`
4. rule names

3 Boolean Deduction

Consider the logical system from Fig. 1. It is complete for Boolean logic with connectives $\rightarrow$ and $\bot$. Three of the derivation rules there ((`mp`), (`absurd`) and (`nnpp`)) fit a common scheme:

$$\frac{\Gamma \vdash \Xi_1(\psi_1,\ldots,\psi_m) \ldots \Gamma \vdash \Xi_n(\psi_1,\ldots,\psi_m)}{\Gamma \vdash \Xi(\psi_1,\ldots,\psi_m)}\ (r)$$

where Ξ_i and Ξ are formulas built from the Boolean connectives and formula meta-variables $\psi_1,\ldots,\psi_m$, while (r) is the name of the rule. We call such rules *standard rules*. Additional Boolean connectives can be handled by adding more standard rules. To encode a standard derivation rule, we use the following rewrite:

$$r(\Box(\Xi_1(x_1,\ldots,x_m)),\ldots,\Box(\Xi_n(x_1,\ldots,x_m)), x_{i_1},\ldots,x_{i_l}) \blacktriangleright \Box(\Xi(x_1,\ldots,x_m))$$

[2] The actual implementation uses de Bruijn indices, so the "capture free" part comes at no cost.

Proof rule	Rewrite rule
$\frac{\Gamma \vdash \psi_1 \to \psi_2 \quad \Gamma \vdash \psi_1}{\Gamma \vdash \psi_2}$ (mp)	$\texttt{mp}(\Box(\texttt{implies}(x_1, x_2)), \Box(x_1)) \blacktriangleright \Box(x_2)$
$\frac{\Gamma \vdash \bot}{\Gamma \vdash \psi}$ (absurd)	$\texttt{absurd}(\Box(\texttt{false}), x) \blacktriangleright \Box(x)$
$\frac{\Gamma \vdash (\psi \to \bot) \to \bot}{\Gamma \vdash \psi}$ (nnpp)	$\texttt{nnpp}(\Box(\texttt{implies}(\texttt{implies}(x, \texttt{false}), \texttt{false}))) \blacktriangleright \Box(x)$
$\frac{\Gamma \cup \{\psi_1\} \vdash \psi_2}{\Gamma \vdash \psi_1 \to \psi_2}$ (assume)	$\texttt{assume}(x_1, x_2) \blacktriangleright \texttt{lift\_known}(\texttt{implies}(x_1, \textbf{apply} \cdot (x_2, \Box(x_1))))$ $\texttt{lift\_known}(\texttt{implies}(x_1, \Box(x_2))) \blacktriangleright \Box(\texttt{implies}(x_1, x_2))$
$\frac{\psi \in \Gamma}{\Gamma \vdash \psi}$ (assumption)	

Fig. 1. A complete system for $\to$ and $\bot$

where x_{i_j} are additional technical arguments, to fulfill the condition that the left-hand side of a rule has to contains all the free variables in the right-hand side and r is a function symbol used to encode this particular rule. Therefore we can model (mp), (absurd) and (nnpp) using the rewrite rules listed in Fig. 1.

We are left with the (assume)/(assumption) pair, which is modeled using lambda expressions. There is no explicit rewrite for (assumption) rule. The term x_2 is expected to be of the form $\lambda y.\, t$, where t is a proof using y in places where (assumption) should be used. This is very similar to the encoding of the IMP-I rule in Edinburgh Logical Framework [10].

We call *restricted* the terms of the form $\Box(...)$, lift_known(...) or sk(...) (the last one is used in the next section). We say that $P \in \mathcal{T}$ is a pre-proof (written ***preproof***(P)), if it does not contain a restricted subterm, or a subterm which is a $s \cdot (...)$ special form.

Lemma 1. *For any pair* (P, σ)*, such that* ***preproof***(P)*,* $\forall x \in \mathcal{V}.\, x\sigma = x \vee \exists \phi.\, x\sigma = \Box(\phi)$ *and* ***nf***$(P\sigma) = \Box(\psi)$*, there exists a derivation* $\Gamma \vdash \psi$ *where* $\Gamma = \{\phi \mid x \in \mathcal{V}, x\sigma = \Box(\phi)\}$*.*

Proof. The proof is by induction on the size of P. Because $\Box(...)$ is not a subterm of P, the head of P must be either:

1. a variable x, in which case $x\sigma$ is $\Box(\psi)$ and the (assumption) rule can be used, since $\psi \in \Gamma$,
2. $P = r(P_1, \ldots, P_n, t_1, \ldots, t_m)$, where a rewrite, obtained from a derivation rule (r), is applicable to:

$$r(\textbf{\textit{nf}}(P_1\sigma), \ldots, \textbf{\textit{nf}}(P_n\sigma), \textbf{\textit{nf}}(t_1), \ldots, \textbf{\textit{nf}}(t_m))$$

We use the induction hypothesis on (P_i, σ), where $\textbf{\textit{nf}}(P_i\sigma) = \Box(\psi_i)$ and build the derivation using the (r) rule.

Proof rule	Rewrite rule
$\frac{}{\Gamma \vdash t = t}$ (eq_refl)	$\texttt{eq\_refl}(x) \blacktriangleright$ $\Box(\texttt{eq}(x, x))$
$\frac{\Gamma \vdash t_1 = t_2 \quad \Gamma \vdash t_2 = t_3}{\Gamma \vdash t_1 = t_3}$ (eq_trans)	$\texttt{eq\_trans}(\Box(\texttt{eq}(x_1, x_2)), \Box(\texttt{eq}(x_2, x_3))) \blacktriangleright$ $\Box(\texttt{eq}(x_1, x_3))$
$\frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash t_2 = t_1}$ (eq_trans)	$\texttt{eq\_symm}(\Box(\texttt{eq}(x_1, x_2))) \blacktriangleright$ $\Box(\texttt{eq}(x_2, x_1))$
$\frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash \psi(t_1) \to \psi(t_2)}$ (eq_sub)	$\texttt{eq\_sub}(\Box(\texttt{eq}(x_1, x_2)), y) \blacktriangleright$ $\Box(\texttt{implies}(\textbf{apply} \cdot (y, x_1), \textbf{apply} \cdot (y, x_2)))$
$\frac{\Gamma \vdash x + x_1 \leq c_1 \quad \Gamma \vdash -x + x_2 \leq c_2}{x_1 + x_2 \leq c_1 + c_2}$ (utvpi_trans)	
$\texttt{utvpi\_trans}(\Box(\texttt{leq}(\texttt{add}(x_1, x_2), x_3)), \Box(\texttt{leq}(\texttt{add}(\texttt{minus}(x_1), y_2), y_3))) \blacktriangleright$ $\Box(\texttt{leq}(\texttt{add}(x_2, y_2), \textbf{fold} \cdot (\texttt{add}(x_3, y_3)))$	

Fig. 2. The equality rules, and an example of an UTVPI rule

3. $P = \texttt{assume}(P_1, \psi)$, which rewrites to $\Box(...)$ in two steps, through the $\texttt{lift\_known}(...)$ (which cannot be used explicitly because $\boldsymbol{preproof}(P)$). $\textbf{apply} \cdot (P_1, \Box(\psi))$ needs to be reduced to $\Box(...)$ for the $\texttt{lift\_known}(...)$ to be applied, so $\boldsymbol{nf}(P_1) = \lambda x.\, P_2$, for some P_2. Because no rule can result in a rewrite to a lambda term (all of the rewrite rules have a term of the form $f(...)$ as their right hand side), then not only $\boldsymbol{nf}(P_1)$, but also P_1 itself needs to start with a lambda binder. Therefore $P_1 = \lambda x.\, P_3$, for some P_3. In this case we use the induction hypothesis on $(P_3, \sigma[x := \Box(\psi)])$, and then use the (assume) rule to construct the implication.

There are no other cases, since no other rewrite rule has $\Box(...)$ as the right-hand side. □

Applying this lemma with $(P, \emptyset)$ gives the theorem.

Theorem 1. *For any P, such that $\boldsymbol{preproof}(P)$ and $\boldsymbol{nf}(P\sigma) = \Box(\psi)$ there exists a derivation $\vdash \psi$.*

Theory Conflicts. Proving theory conflicts clearly depends on the particular theory. Fig. 2 lists rules for the theory of equality. The encoding is the same as for the standard rules from Fig. 1. For arithmetic we currently support the UTVPI fragment [11] of integer linear arithmetic. It consists of inequalities of the form $ax + by \leq c$, where $a, b \in \{-1, 0, 1\}$ and c is an integer. The decision procedure closes set of such inequalities, with respect to a few rules, of the form similar to the one listed in Fig. 2. Again, the encoding is the same as for ordinary deduction rules.

4 Skolemization Calculus

Fig. 3 lists rules for a skolemization calculus. The $\uplus$ is disjoint set union (i.e. $A \cup B$ if $A \cap B = \emptyset$ and undefined otherwise). The intuition behind $S; Q \vdash \psi \rightsquigarrow \phi$

Proof rule	Rewrite rule
$\frac{\emptyset; Q \vdash \neg\psi(f(Q)) \rightsquigarrow \phi}{\{f\}; Q \vdash \neg\forall x.\, \psi(x) \rightsquigarrow \phi}$ (skol)	$\mathtt{sk}(y, \mathtt{skol}(f, y_1), \mathtt{neg}(\mathtt{forall}(x))) \blacktriangleright$ $\mathtt{sk}(y, y_1, \mathtt{neg}(\mathbf{apply} \cdot (x, \mathbf{build} \cdot (f, y))))$
$\frac{S; Q, x \vdash \psi(x) \rightsquigarrow \phi(x)}{S; Q \vdash \forall x.\, \psi(x) \rightsquigarrow \forall x.\, \phi(x)}$ ($\mathtt{skip}_\forall$)	$\mathtt{sk}(y, \mathtt{skip}_\forall(y_1), \mathtt{forall}(x_1)) \blacktriangleright$ $\mathtt{forall}(\lambda x.\, \mathtt{sk}(x :: y, y_1, \mathbf{apply} \cdot (x_1, x)))$
$\frac{}{\emptyset; Q \vdash \psi \rightsquigarrow \psi}$ (id)	$\mathtt{sk}(y, \mathtt{id}, x_1) \blacktriangleright x_1$
$\frac{S_1; Q \vdash \psi_1 \rightsquigarrow \phi_1 \quad S_2; Q \vdash \psi_2 \rightsquigarrow \phi_2}{S_1 \uplus S_2; Q \vdash \psi_1 \wedge \psi_2 \rightsquigarrow \phi_1 \wedge \phi_2}$ ($\mathtt{rec}_\wedge$)	$\mathtt{sk}(y, \mathtt{rec}_\wedge(y_1, y_2), \mathtt{and}(x_1, x_2)) \blacktriangleright$ $\mathtt{and}(\mathtt{sk}(y, y_1, x_1), \mathtt{sk}(y, y_2, x_2))$
$\frac{S_1; Q \vdash \psi_1 \rightsquigarrow \phi_1 \quad S_2; Q \vdash \psi_2 \rightsquigarrow \phi_2}{S_1 \uplus S_2; Q \vdash \psi_1 \vee \psi_2 \rightsquigarrow \phi_1 \vee \phi_2}$ ($\mathtt{rec}_\vee$)	$\mathtt{sk}(y, \mathtt{rec}_\vee(y_1, y_2), \mathtt{or}(x_1, x_2)) \blacktriangleright$ $\mathtt{or}(\mathtt{sk}(y, y_1, x_1), \mathtt{sk}(y, y_2, x_2))$
$\frac{S_1; Q \vdash \neg\psi_1 \rightsquigarrow \phi_1 \quad S_2; Q \vdash \neg\psi_2 \rightsquigarrow \phi_2}{S_1 \uplus S_2; Q \vdash \neg(\psi_1 \vee \psi_2) \rightsquigarrow \phi_1 \wedge \phi_2}$ ($\mathtt{rec}_{\neg\vee}$)	$\mathtt{sk}(y, \mathtt{rec}_{\neg\vee}(y_1, y_2), \mathtt{neg}(\mathtt{or}(x_1, x_2))) \blacktriangleright$ $\mathtt{and}(\mathtt{sk}(y, y_1, \mathtt{neg}(x_1)), \mathtt{sk}(y, y_2, \mathtt{neg}(x_2)))$
$\frac{S_1; Q \vdash \neg\psi_1 \rightsquigarrow \phi_1 \quad S_2; Q \vdash \neg\psi_2 \rightsquigarrow \phi_2}{S_1 \uplus S_2; Q \vdash \neg(\psi_1 \wedge \psi_2) \rightsquigarrow \phi_1 \vee \phi_2}$ ($\mathtt{rec}_{\neg\wedge}$)	$\mathtt{sk}(y, \mathtt{rec}_{\neg\wedge}(y_1, y_2), \mathtt{neg}(\mathtt{and}(x_1, x_2))) \blacktriangleright$ $\mathtt{or}(\mathtt{sk}(y, y_1, \mathtt{neg}(x_1)), \mathtt{sk}(y, y_2, \mathtt{neg}(x_2)))$
$\frac{S_1; Q \vdash \psi_1 \rightsquigarrow \phi_1}{S_1; Q \vdash \neg\neg\psi_1 \rightsquigarrow \phi_1}$ ($\mathtt{rec}_{\neg\neg}$)	$\mathtt{sk}(y, \mathtt{rec}_{\neg\neg}(y_1), \mathtt{neg}(\mathtt{neg}(x_1))) \blacktriangleright$ $\mathtt{sk}(y, y_1, x_1)$

Fig. 3. The skolemization calculus

is that for each model M of $\forall Q.\, \psi$ there exists an extension of M on the symbols from S that satisfies $\forall Q.\, \phi$. We formalize it using second order logic with the following lemma:

Lemma 2. *Let* $Q = \{x_1, \ldots, x_n\}$ *and* $S = \{f_1 \ldots f_n\}$. *If* $S; Q \vdash \psi \rightsquigarrow \phi$ *where* $\psi \in \mathcal{T}(\Sigma, Q)$, $\phi \in \mathcal{T}(\Sigma \uplus S, Q)$, *then* $\models \exists^2 f_1 \ldots \exists^2 f_n.\, \forall x_1, \ldots, x_n.\, \psi \to \phi$.

The key point of the proof is that for the rules of the common form:

$$\frac{S_1; Q \vdash \Xi_1(\psi_1, \ldots, \psi_k) \rightsquigarrow \phi_1 \quad \ldots \quad S_m; Q \vdash \Xi_m(\psi_1, \ldots, \psi_k) \rightsquigarrow \phi_m}{S_1 \uplus \ldots \uplus S_m; Q \vdash \Xi(\psi_1, \ldots, \psi_k) \rightsquigarrow \Xi'(\psi_1, \ldots, \psi_k, \phi_1, \ldots, \phi_m)}\ (r)$$

where ψ_j and ϕ_j range over first order formulas it is enough to show the following:

$$\begin{array}{l}
\forall Q : \mathbf{Type}.\, \forall S_1 \ldots S_n : \mathbf{Type}. \\
\forall \psi_1 \ldots \psi_k : Q \to \mathbf{Prop}. \\
\forall \phi_1 : S_1 \times Q \to \mathbf{Prop}. \ldots \forall \phi_m : S_m \times Q \to \mathbf{Prop}. \\
\quad \bigwedge_{i=1\ldots m} (\exists f_i : S_i.\, \forall x : Q.\, \Xi_i(\psi_i(x), \ldots, \psi_k(x)) \to \phi_i(f_i, x)) \to \\
\quad (\exists f_1 : S_1. \ldots \exists f_m : S_m.\, \forall x : Q.\, \Xi(\psi_1(x), \ldots, \psi_k(x)) \to \\
\qquad \Xi'(\psi_m(x), \ldots, \psi_k(x), \phi_1(f_1, x), \ldots, \phi_m(f_m, x)))
\end{array}$$

which is much like the formula from the lemma, except that there is only one Skolem constant f_i per premise and also there is only one free variable in all the formulas, namely x. However these symbols are of arbitrary type, so they can be thought of as representing sequences of symbols.

We prove such formulas for each rule, the reader can find the proof scripts for Coq proof assistant online [1].

Rewrite encoding. The common form of a rule is encoded as:

$$\begin{array}{l}
\mathtt{sk}(y, r(y_1, \ldots, y_m), \Xi(x_1, \ldots, x_k)) \blacktriangleright \\
\quad \Xi'(x_1, \ldots, x_k, \mathtt{sk}(y, y_1, \Xi_1(x_1, \ldots, x_k)), \ldots, \mathtt{sk}(y, y_m, \Xi_m(x_1, \ldots, x_k)))
\end{array}$$

The first argument of $\texttt{sk}(\dots)$ is the list of universally quantified variables in scope. The second argument is a rule name, along with proofs of premises. The third argument is the formula to be transformed.

The encoding of non-common rules (as well as the common rules used here) is given in the Fig. 3.

Lemma 3. *If* $\boldsymbol{preproof}(P)$, $\boldsymbol{nf}(\texttt{sk}(x_1 :: \cdots :: x_n :: \mathbf{nil}, P, \psi)) = \psi'$, *and for each occurrence of* $\texttt{skol}(f)$ *as a subterm of* P, *the function symbol* f *does not occur anywhere else in* P *nor in* ψ, *then there exists* S, *such that* $S; x_1, \dots, x_n \vdash \psi \rightsquigarrow \psi\psi'$.

Proof. By structural induction over P. □

Theorem 2. *If* $\boldsymbol{preproof}(P)$, $\boldsymbol{nf}(\texttt{sk}(\mathbf{nil}, P, \psi)) = \psi'$, *and for each occurrence of* $\texttt{skol}(f)$ *as a subterm of* P, *the function symbol* f *does not occur anywhere else in* P *nor in* ψ, *and* ψ' *is unsatisfiable then* ψ *is unsatisfiable.*

Proof. By Lemmas 2 and 3. □

5 The Checker

The proof checker reads three files: (1) rewrite rules describing the underlying logic; (2) a query in SMT-LIB concrete syntax; and (3) the proof. The concrete syntax for both the rewrite rules and the proof term is similar to the one used in SMT-LIB. The proof term language includes the following commands:

- **let** $x := t_1$: bind the identifier x to the term $\boldsymbol{nf}(t_1)$
- **initial** t_1 t_2: check if $\texttt{skol}(f)$ is used in t_1 only once with each f, that the f symbols do not occur in t_2, compares t_2 against the query read from the SMT-LIB file and if everything succeeds, binds $\square(\boldsymbol{nf}(\texttt{sk}(\mathbf{nil}, t_1, t_2)))$ to the special identifier `initial`; this command can be used only once in a given proof
- **final** t_1: checks if $\boldsymbol{nf}(t_1) = \square(\texttt{false})$, and if so reports success and exits
- **assert_eq** t_1 t_2: checks if $\boldsymbol{nf}(t_1) = \boldsymbol{nf}(t_2)$ (and aborts if this is not the case)
- **assert_ok** t_1 t_2: checks if $\boldsymbol{nf}(t_1) = \square(\boldsymbol{nf}(t_2))$ (and aborts if this is not the case)
- **print** t_1: prints a string representation of t_1

The last three commands are used to debug the proofs.

The proofs, after initial skolemization, are structured as a sequence of clause derivations, using either resolution, theory conflicts, instantiation or CNF-conversion steps. All these clauses are **let**-bound, until we reach the empty clause. Basically we end up with a proof-tree in natural deduction, deriving the Boolean false constant from the initial formula. The tree is encoded as a DAG, because **let**-bound clauses can be used more than once.

Proof rule	Rewrite rule
$\frac{\Gamma \vdash \psi_1 \wedge \psi_2}{\Gamma \vdash \psi_1}$ ($\texttt{elim}_{\wedge 1}$)	$\texttt{elim}_{\wedge 1}(\Box(\texttt{and}(x_1, x_2))) \blacktriangleright$ $\Box(x_1)$
$\frac{\Gamma \vdash \psi_1 \wedge \psi_2}{\Gamma \vdash \psi_2}$ ($\texttt{elim}_{\wedge 2}$)	$\texttt{elim}_{\wedge 2}(\Box(\texttt{and}(x_1, x_2))) \blacktriangleright$ $\Box(x_2)$
$\frac{\Gamma \vdash \psi_1 \vee \psi_2}{\Gamma \vdash \neg\psi_1 \rightarrow \psi_2}$ ($\texttt{elim}_{\vee}$)	$\texttt{elim}_{\vee}(\Box(\texttt{or}(x_1, x_2))) \blacktriangleright$ $\Box(\texttt{implies}(\texttt{neg}(x_1), x_2))$
$\frac{\Gamma \vdash \neg\psi \quad \Gamma \vdash \psi}{\Gamma \vdash \bot}$ ($\texttt{elim}_{\neg}$)	$\texttt{elim}_{\neg}(\Box(\texttt{neg}(x_1))), \Box((x_1))) \blacktriangleright$ $\Box(\texttt{false})$
$\frac{\Gamma \vdash \psi}{\Gamma \vdash \neg\neg\psi}$ ($\texttt{add}_{\neg\neg}$)	$\texttt{add}_{\neg\neg}(\Box(x_1)) \blacktriangleright$ $\Box(\texttt{neg}(\texttt{neg}(x_1)))$
$\frac{\Gamma \vdash \psi \rightarrow \bot}{\Gamma \vdash \neg\psi}$ ($\texttt{intro}_{\neg}$)	$\texttt{intro}_{\neg}(\Box(\texttt{implies}(x_1, \texttt{false}))) \blacktriangleright$ $\Box(\texttt{neg}(x_1))$
$\frac{\Gamma \vdash \neg\psi \rightarrow \bot}{\Gamma \vdash \psi}$ ($\texttt{elim}_{\neg\rightarrow}$)	$\texttt{elim}_{\neg\rightarrow}(\Box(\texttt{implies}(\texttt{neg}(x_1), \texttt{false}))) \blacktriangleright$ $\Box(x_1)$
$\frac{\Gamma \vdash \forall x.\, \psi(x)}{\Gamma \vdash \psi(t)}$ ($\texttt{inst}$)	$\texttt{inst}(y, \Box(\texttt{forall}(x))) \blacktriangleright$ $\Box(\textbf{apply} \cdot (x, y))$

Fig. 4. Additional rules for the example

All those steps are best described through an example[3]. Fig. 4 lists rules not previously mentioned in this paper, that were used in the proof. The real proof system has more rules. As described in Sect. 6.1, we mechanically check all rules. Our example formula is:

$$P(c) \wedge (c = d) \wedge (\forall x.\, \neg P(x) \vee \neg(\forall y.\, \neg Q(x, y))) \wedge (\forall x.\, \neg Q(d, x))$$

The first step is the initial skolemization:

let $q_1 := \texttt{forall}(\lambda x.\, \texttt{or}(\texttt{neg}(P(x)), \texttt{neg}(\texttt{forall}(\lambda y.\, \texttt{neg}(Q(x, y))))))$
let $q_2 := \texttt{forall}(\lambda x.\, \texttt{neg}(Q(d, x)))$
let $f_{in} := \texttt{and}(P(c), \texttt{and}(\texttt{eq}(c, d), \texttt{and}(q_1, q_2)))$
let $sk := \texttt{rec}_{\wedge}(\texttt{id}, \texttt{rec}_{\wedge}(\texttt{id}, \texttt{rec}_{\wedge}(\texttt{skip}_{\forall}(\texttt{rec}_{\vee}(\texttt{id}, \texttt{skol}(f, \texttt{rec}_{\neg\neg}(\texttt{id})))), \texttt{id})))$
initial $sk\ f_{in}$

Here our expectation, as the proof generator, is that $\forall x.\, \neg P(x) \vee \neg(\forall y.\, \neg Q(x, y))$ will be replaced by $\forall x.\, \neg P(x) \vee Q(x, f(x))$, which we express as:

let $q_3 := \texttt{forall}(\lambda x.\, \texttt{or}(\texttt{neg}(P(x)), Q(x, f(x))))$
let $f_{sk} := \texttt{and}(P(c), \texttt{and}(\texttt{eq}(c, d), \texttt{and}(q_3, q_2)))$
assert_ok `initial` f_{sk}

The first step of the actual proof is a partial CNF-conversion. Our CNF conversion uses Tseitin scheme, which introduces proxy literals for subformulas. This

[3] The proof presented here is not the simplest possible of this very formula. However it follows the steps that our SMT solver does and we expect other SMT solvers to do.

produces equisatisfiable set of clauses, yet the proof maps the proxy literals back to the original subformulas. Then the defining clauses of proxy literals become just basic Boolean facts. We therefore derive clauses of the form $f_{sk} \to \neg\psi \to \bot$, where ψ is one of the conjuncts of f_{sk}, for example:

$$\textbf{let } c_1 := \texttt{assume}(f_{sk}, \lambda f.\, \texttt{assume}(\texttt{neg}(\texttt{eq}(c,d)), \lambda p.\, \texttt{elim}_{\neg}(p, \texttt{elim}_{\wedge 2}(\texttt{elim}_{\wedge 1}(f)))))$$
$$\textbf{assert\_ok } c_1\ \texttt{implies}(f_{sk}, \texttt{implies}(\texttt{neg}(\texttt{eq}(c,d)), \texttt{false}))$$

and similarly we derive:

$$\textbf{assert\_ok } c_0\ \texttt{implies}(f_{sk}, \texttt{implies}(\texttt{neg}(P(c)), \texttt{false}))$$
$$\textbf{assert\_ok } c_2\ \texttt{implies}(f_{sk}, \texttt{implies}(\texttt{neg}(q_3), \texttt{false}))$$
$$\textbf{assert\_ok } c_3\ \texttt{implies}(f_{sk}, \texttt{implies}(\texttt{neg}(q_2), \texttt{false}))$$

Next we instantiate the quantifiers:

$$\textbf{let } c_4 := \texttt{assume}(q_2, \lambda q.\, \texttt{assume}(Q(d, f(c)), \lambda i.\, \texttt{elim}_{\neg}(\texttt{inst}(f(c), q), i)))$$
$$\textbf{assert\_ok } c_4\ \texttt{implies}(q_2, \texttt{implies}(Q(d, f(c)), \texttt{false}))$$
$$\textbf{let } i_1 := \texttt{or}(\texttt{neg}(P(c)), Q(c, f(c)))$$
$$\textbf{let } c_5 := \texttt{assume}(q_3, \lambda q.\, \texttt{assume}(\texttt{neg}(i_1), \lambda i.\, \texttt{elim}_{\neg}(i, \texttt{inst}(c, q))))$$
$$\textbf{assert\_ok } c_5\ \texttt{implies}(q_3, \texttt{implies}(\texttt{neg}(i_1), \texttt{false}))$$

Then we need to clausify i_1:

$$\textbf{let } c_6 := \texttt{assume}(i_1, \lambda i.\, \texttt{assume}(P(c), \lambda o_1.\, \texttt{assume}(\texttt{neg}(Q(c, f(c))), \lambda o_2.\ \texttt{elim}_{\neg}(o_2, \texttt{mp}(\texttt{elim}_{\vee}(i), \texttt{add}_{\neg\neg}(o_1))))))$$
$$\textbf{assert\_ok } c_6\ \texttt{implies}(i_1, \texttt{implies}(P(c), \texttt{implies}(\texttt{neg}(Q(c, f(c)), \texttt{false}))))$$

Then we do some equality reasoning:

$$\textbf{let } c_7 := \texttt{assume}(\texttt{neg}(Q(d, f(c))), \lambda l_n.\, \texttt{assume}(\texttt{eq}(c,d), \lambda e.\, \texttt{assume}(Q(c, f(c)), \lambda l_p.\ \texttt{elim}_{\neg}(l_n, \texttt{mp}(\texttt{eq\_sub}(e, \lambda x.\, Q(x, f(c))), l_p)))))$$
$$\textbf{assert\_ok } c_7\ \texttt{implies}(\texttt{neg}(Q(d, f(c))), \texttt{implies}(\texttt{eq}(c,d), \texttt{implies}(Q(c, f(c)), \texttt{false})))$$

What remains is a pure Boolean resolution. The resolution is realized by assuming the negation of the final clause and then using unit resolution of the assumed literals and some previous clauses, to obtain new literals, and as a last step, the `false` constant. We first resolve c_4 with c_7:

$$\textbf{let } c_8 := \texttt{assume}(q_2, \lambda l_1.\, \texttt{assume}(\texttt{eq}(c,d), \lambda l_2.\, \texttt{assume}(Q(c, f(c)), \lambda l_3.\ \texttt{mp}(\texttt{mp}(\texttt{mp}(c_7, \texttt{intro}_{\neg}(\texttt{mp}(c_4, l_1))), l_2), l_3)))$$
$$\textbf{assert\_ok } c_8 \texttt{implies} q_2, \texttt{implieseq}(c,d), \texttt{implies} Q(c, f(c)), \texttt{false}$$

and finally we derive (also through resolution) the false constant:

$$\textbf{let } kq_2 := \texttt{elim}_{\neg\to}(\texttt{mp}(c_3, \texttt{initial}))$$
$$\textbf{let } kq_3 := \texttt{elim}_{\neg\to}(\texttt{mp}(c_2, \texttt{initial}))$$
$$\textbf{let } kp := \texttt{elim}_{\neg\to}(\texttt{mp}(c_0, \texttt{initial}))$$
$$\textbf{let } ke := \texttt{elim}_{\neg\to}(\texttt{mp}(c_1, \texttt{initial}))$$
$$\textbf{let } kq := \texttt{elim}_{\neg\to}(\texttt{mp}(\texttt{mp}(c_6, \texttt{elim}_{\neg\to}(\texttt{mp}(c_5, kq_3))), kp))$$
$$\textbf{let } c_9 := \texttt{mp}(\texttt{mp}(\texttt{mp}(c_8, kq_2), ke), kq)$$
$$\textbf{final } c_9$$

6 Implementation

We have implemented two versions of the proof checker: one full version in OCaml and a simplified one written in C. Proof generation was implemented inside the Fx7 [1] SMT solver, implemented in the Nemerle programming language. The solver came second in the AUFLIA division of 2007 SMT competition, being much slower, but having solved the same number of benchmarks as the winner, Z3 [6].

An important point about the implementation, is that at any given point, we need to store only terms, that can be referenced by **let**-bound name, and thus the memory used by other terms can be reclaimed. As in our encoding the proof terms actually rewrite to formulas that they prove, there is no need to keep the proof terms around. We suspect this to be the main key to memory efficiency of the proof checker. The C implementation exploits this fact, the OCaml one does not.

Both implementations use de Bruijn [5] indices in representation of lambda terms. We also use hash consing, to keep only a single copy of a given term. We cache normal forms of the terms, we remember what terms are closed (which speeds up beta reductions). Also a local memoization is used in function computing beta reduction to exploit the DAG structure of the term. The rewrite rules are only indexed by the head symbol, if two rules share the head symbol, linear search is used.

All the memoization techniques used are crucial (i.e., we have found proofs, where checking would not finish in hours without them).

The OCaml implementation is about 900 lines of code, where about 300 lines is pretty printing for Coq and Maude formats. The C implementation is 1500 lines. Both implementation include parsing of the proof and SMT formats and command line option handling. The implementations are available online along with the Fx7 prover.

6.1 Soundness Checking

The OCaml version of the checker has also a different mode of operation, where it reads the rewrite rules and generates corresponding formulas to be proven in the Coq proof assistant. There are three proof modes for rules:

- for simple facts about Boolean connectives, arithmetic and equality, the checker generates a lemma and a proof, which is just an invocation of appropriate tactic
- for other generic schemas of proof rules from Sect. 3 and 4, the checker produces proof obligations, and the proofs need to be embedded in the rule descriptions
- for non-generic proof rules, the user can embed both the lemma and the proof in the rule description file, just to keep them close

This semiautomatic process helps preventing simple, low-level mistakes in the proof rules. The checker provides commands to define all these kinds of rules and associated proofs.

Directory	Total	UNSAT	% UNSAT	Fake	% Fake	Fail	% Fail
front_end_suite	2320	2207	95.13%	101	4.35%	12	0.52%
boogie	908	866	95.37%	25	2.75%	17	1.87%
simplify	833	729	87.52%	44	5.28%	60	7.20%
piVC	41	17	41.46%	10	24.39%	14	34.15%
misc	20	16	80.00%	0	0.00%	4	20.00%
Burns	14	14	100.00%	0	0.00%	0	0.00%
RicartAgrawala	14	13	92.86%	0	0.00%	1	7.14%
small_suite	10	8	80.00%	0	0.00%	2	20.00%

Fig. 5. Results on the AUFLIA division of SMT-LIB

6.2 Performance Evaluation

When running Fx7 on a query there are five possible outcomes:

- it reports that the query is unsatisfiable, and outputs a proof
- it reports that the query is unsatisfiable, but because the proof generation is only implemented for the UTVPI fragment of linear arithmetic, the proof is correct only if we assume the theory conflicts to be valid (there is typically a few of them in each of such "fake" proofs)
- it reports the query is satisfiable, timeouts or runs out of memory

Tests were performed on AUFLIA benchmarks from the SMT-LIB [14]. This division includes first order formulas, possibly with quantifiers, interpreted under uninterpreted function symbols, integer linear arithmetic and array theories. They are mostly software verification queries. The machine used was a 2.66GHz Pentium 4 PC with 1GB of RAM, running Linux. The time limit was set to ten minutes.

The results are given in Fig. 5. The "Total" column refers to the number of benchmarks marked unsatisfiable in the SMT-LIB; "UNSAT" refers to the number of cases, where the benchmark was found unsatisfiable and a correct proof was generated; "Fake" is the number of benchmarks found unsatisfiable, but with "fake" proofs; finally "Fail" is the number of cases, where Fx7 was unable to prove it within the time limit. It should be the case that UNSAT + Fake + Fail = Total. The percentages are with respect to the Total.

With the **C implementation**, proof checking a single proof never took more than 7 seconds. It took more than 2 seconds in 4 cases and more than 1 second in 19 cases (therefore the average time is well under a second). The maximal amount of memory consumed for a single proof was never over 7MB, with average being 2MB.

We have also tested the C implementation on a Dell x50v **PDA** with a 624MHz XScale ARM CPU and 32MB of RAM, running Windows CE. It was about 6 times slower than the Pentium machine, but was otherwise perfectly capable of running the checker. This fact can be thought of as a first step on a way to PCC-like scenarios on small, mobile devices. Other devices of similar computing power and, what is more important, RAM amount include most smart phones and iPods.

The **OCaml implementation** was on average 3 times slower than the C version, it also tends to consume more memory, mostly because it keeps all the terms forever (which is because of our implementation, not because of OCaml).

We have also experimented with translating the proof objects into the **Maude syntax** [3]. We have implemented lambda terms and beta reduction using the built-in Maude integers to encode de Bruijn indices and used the standard equational specifications for the first order rules. The resulting Maude implementation is very compact (about 60 lines), but the performance is not as good as with the OCaml or C implementation — it is between 10 and 100 times slower than the OCaml one. It also tends to consume a lot more memory. The reason is mainly the non-native handling of lambda expressions. Beta reductions translate to large number of first order rewrites, which are then memoized, and we were unable to instrument Maude to skip memoization of those.

We have performed some experiments using **Coq metalogic** as the proof checker. We did not get as far as implementing our own object logic. The main obstacle we have found was the treatment of binders. Performing skolemization on a typical input results in hundreds of Skolem functions. When using a higher order logic prover, such functions are existentially quantified and the quantifiers need to be pushed through the entire formula to the beginning. Later, during the proof, we need to go through them to manipulate the formula. This puts too much pressure on the algorithms treating of binders in the higher order prover. In our approach Skolem functions are bound implicitly, so there is no need to move them around. This is especially important in SMT queries, where the vast majority of the input formula is ground and quantified subformulas occur only deep inside the input. We can therefore keep most of the formula binder-free. We were not able to perform any realistic tests, as Coq was running out of memory.

Both Maude and Coq are far more general purpose tools than just proof checkers. However relatively good results with Maude suggest that using a simple underlying formalism is beneficial in proof checking scenarios.

7 Related and Future Work

CVC3 [2] and Fx7 were the only solvers participating in the 2007 edition of the SMT competition to produce formal proofs. The proof generation in CVC3 is based on the LF framework. We are not aware of a published work evaluating proof checking techniques on large industrial benchmarks involving quantifiers.

Formalisms for checking SMT proofs have been proposed in the past, most notably using an optimized implementation [15] of Edinburgh Logical Framework [10]. However even with the proposed optimizations, the implementations has an order of magnitude higher memory requirements than our solution. Also the implementation of the checker is much more complicated.

Recently a Signature Compiler tool has been proposed [16]. It generates a custom proof checker in C++ or Java from a LF signature. We have run our proof checker on a 1:1 translation of the artificial EQ benchmarks from the paper. It is running slightly faster than the generated C++ checker. The memory

requirements of our implementation are way below the size of the input file on those benchmarks. The checkers remain to be compared on real benchmarks involving richer logics and quantifiers.

In context of the saturation theorem provers it is very natural to output the proof just as a sequence of resolution or superposition steps. What is missing here, is the proof of CNF translation, though proof systems has been proposed [8], [7] to deal with that.

Finally, work on integrating SMT solvers as decision procedures inside higher order logic provers include [12], [9], [4]. The main problem with these approaches is that proof generation is usually at least order of magnitude faster than proof checking inside higher order logic prover. The Ergo [4] paper mentions promising preliminary results with using proof traces instead of full proofs with Coq for theory conflicts. It is possible that using traces could also work for CNF conversion and skolemization. Yet another approach mentioned there is verifying the SMT solver itself.

An important remaining problem is the treatment of theory conflicts. One scenario here is to extend the linear arithmetic decision procedure to produce proofs. It should be possible to encode the proofs with just a minor extensions to the rewrite formalism. Another feasible scenario is to use a different SMT solver as a oracle for checking the harder (or all) theory conflicts. This can be applied also to other theories, like bit vectors or rational arithmetic.

8 Conclusions

We have shown how term rewriting can be used for proof checking. The highlights of our approach are (1) time and space efficiency of the proof checker; (2) simplicity of the formalism, and thus simplicity of the implementation; and (3) semiautomatic checking of proof rules. The main technical insight is that the proof rules can be executed locally. Therefore the memory taken by proofs trees can be reclaimed just after checking them and reused for the subsequent fragments of the proof tree.

The author wishes to thank Joe Kiniry, Mikoláš Janota, and Radu Grigore for their help during the work on the system, and Nikolaj Bjørner as well as anonymous TACAS reviewers for his help in getting the presentation of this paper better.

This work was partially supported by Polish Ministry of Science and Education grant 3 T11C 042 30.

References

1. Fx7 web page, `http://nemerle.org/fx7/`
2. Barrett, C., Berezin, S.: CVC Lite: A new implementation of the Cooperating Validity Checker. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 515–518. Springer, Heidelberg (2004)

3. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.F.: Maude: Specification and programming in rewriting logic. Theoretical Computer Science (2001)
4. Conchon, S., Contejean, E., Kanig, J., Lescuyer, S.: Lightweight Integration of the Ergo Theorem Prover inside a Proof Assistant. In: Second Automated Formal Methods workshop series (AFM 2007), Atlanta, Georgia, USA (November 2007)
5. de Bruijn, N.G.: Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. Indag. Math. 34(5), 381–392 (1972)
6. de Moura, L., Bjorner, N.: Efficient E-matching for SMT solvers. In: Proceedings of the 21st International Conference on Automated Deduction (CADE-21), Springer, Heidelberg (to appear, 2007)
7. de Nivelle, H.: Implementing the clausal normal form transformation with proof generation. In: fourth workshop on the implementation of logics, Almaty, Kazachstan, University of Liverpool, University of Manchester, pp. 69–83 (2003)
8. de Nivelle, H.: Translation of resolution proofs into short first-order proofs without choice axioms. Information and Computation 199(1), 24–54 (2005)
9. Fontaine, P., Marion, J.-Y., Merz, S., Nieto, L.P., Tiu, A.: Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 167–181. Springer, Heidelberg (2006)
10. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. In: Proceedings 2nd Annual IEEE Symp. on Logic in Computer Science, LICS 1987, Ithaca, NY, USA, June, 22–25, 1987, pp. 194–204. IEEE Computer Society Press, New York (1987)
11. Harvey, W., Stuckey, P.: A unit two variable per inequality integer constraint solver for constraint logic programming (1997)
12. McLaughlin, S., Barrett, C., Ge, Y.: Cooperating theorem provers: A case study combining HOL-Light and CVC Lite. In: Armando, A., Cimatti, A. (eds.) Proceedings of the 3^{rd} Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR 2005), Edinburgh, Scotland, January 2006. Electronic Notes in Theoretical Computer Science, vol. 144(2), pp. 43–51. Elsevier, Amsterdam (2006)
13. Necula, G.C.: Proof-carrying code. In: Conference Record of POPL 1997: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, France, January 1997, pp. 106–119 (1997)
14. SMT-LIB: The Satisfiability Modulo Theories Library. `http://www.smt-lib.org/`
15. Stump, A., Dill, D.: Faster Proof Checking in the Edinburgh Logical Framework. In: 18th International Conference on Automated Deduction (2002)
16. Zeller, M., Stump, A., Deters, M.: A signature compiler for the Edinburgh Logical Framework. In: Proceedings of International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (2007)