Rewriting, Inference, and Proof*

Appears in the Proceedings of the 8th Workshop on Rewriting Logic and its Applications, 2010

Natarajan Shankar[1]

Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA
{shankar}@csl.sri.com
URL: http://www.csl.sri.com/~shankar/

Abstract. Rewriting is a form of inference, and one that interacts in several ways with other forms of inference such as decision procedures and proof search. We discuss a range of issues at the intersection of rewriting and inference. How can other inference procedures be combined with rewriting? Can rewriting be used to describe inference procedures? What are some of the theoretical challenges and practical applications of combining rewriting and inference? How can rewriters, decision procedures, and their combination be certified? We discuss these problems in the context of our ongoing effort to use PVS as a metatheoretic framework to construct a proof kernel for justifying the claims of theorem provers, rewriters, model checkers, and satisfiability solvers.

Rewriting is a versatile framework that can be used as a programming notation, a modeling formalism, and as an inference method. It is a crucial component of any effective interactive theorem prover. Rewriting can also be used as framework for prototyping and reasoning about inference procedures. A rewriter is itself a very powerful inference procedure and certifying the claims made by rewriters can be quite challenging. We explore several themes centered around rewriting, inference, and proof. We describe the combination of rewriting and decision procedures employed by SRI's Prototype Verification System (PVS) [ORSvH95] in its simplifier. This inference rule is built into PVS and hence its soundness cannot be taken for granted. We present an architecture for justifying the soundness of such complex inference procedures based on the use of verified reference checkers. We review some of the progress in developing this architecture. We also show how rewriting can be used to define such reference checkers.

There is a long history of work in rewriting in the context of theorem proving. Woody Bledsoe [Ble77] advocated it as a human-oriented method for automated proof search. The Boyer-Moore family of theorem provers [BM79,BM88,KMM00] are well

* This research was supported NSF Grants CSR-EHCS(CPS)-0834810 and CNS-0917375. Sam Owre commented on earlier drafts of the paper, and the participants at the 2010 Workshop on Rewriting Logic and Applications, particularly José Meseguer and Peter Ölveczky, offered valuable feedback and advice.

known for induction, but rewriting is one of its big strengths. The Rewrite Rule Laboratory [KZ88] supports both explicit and implicit induction within a rewriting framework. The OBJ family of systems [GW88,GKM+87] employed rewriting as an algebraic specification language. Maude [CDE+99] and ELAN [BKK+96] are descendants of OBJ that support extremely fast rewriting within an expressive rewriting logic framework.

Rewriting also plays a crucial role in the interactive proof assistant of PVS. It is used within the PVS simplifier in conjunction with decision procedures and simplification rules. The simplifier is around 4000 lines of Common Lisp code, and it relies on decision procedures that also run to nearly 4000 lines of code. PVS uses external decision procedures including a BDD package and the Yices SMT solver, but the simplifier is easily the most complicated of the built-in inference procedures. We describe this procedure and examine the challenge of certifying results that are claimed by the simplifier.

Our approach to certifying the results of untrusted inference procedures is developed within the *Kernel of Truth* (KoT) project at SRI. The approach is captured by Figure 1. At the bottom, there is a small, trusted proof kernel, and at the top, we have the untrusted inference procedures. We rely on *verified* reference checkers to check the claims made by untrusted inference procedures [Sha08] relative to the trusted proof kernel. In the KoT approach, we do not verify the verifier, but we instead check verification claims with a verified checker. The approach is driven by the idea that checking is always easier than solving, and checkers are usually easier to verify than solvers. Furthermore, the certificates that are checked by the checkers need not be formal proofs and can be customized to specific classes of problems. The verification of the checker demonstrates the existence of a formal proof corresponding to a valid certificate, but the actual proof need not be explicitly constructed. By using verified checkers, the untrusted procedures can be optimized for speed while avoiding the overhead of proof instrumentation and generation. The untrusted procedures can provide hints or certificates to the verified checkers. The verified checkers are expected to be simple and might therefore perform slower than the untrusted procedures, but this is acceptable since the validation of untrusted results and certificates is done offline. The hints provided by the untrusted procedures should also make it more efficient to check the resulting claims. The verification of the checkers can be performed by the untrusted inference tools, since for these specific claims, we can break the circular dependency by generating and checking the kernel-level formal proofs.

Our KoT approach should be contrasted with existing techniques for building trusted inference tools. The LCF approach used by systems such as Coq [The09], HOL [GM93], Isabelle [Pau94], and Nuprl [CAB+86], relies on proof generation as a way of validating claims. Though proofs are constructed using tactics that generate subgoals from goals, the application of these tactics are valid only when there is a proof of the goal from the subgoals. Proof generation imposes an engineering and performance overhead. Some SAT solvers have been instrumented to generate resolution proofs, and though these proofs can get quite large, the overhead of generating and checking them is quite modest at about 2 to 12% for generation with a checking time that is significantly smaller than the solving time [ZM03]. However, it should be noted that these resolution

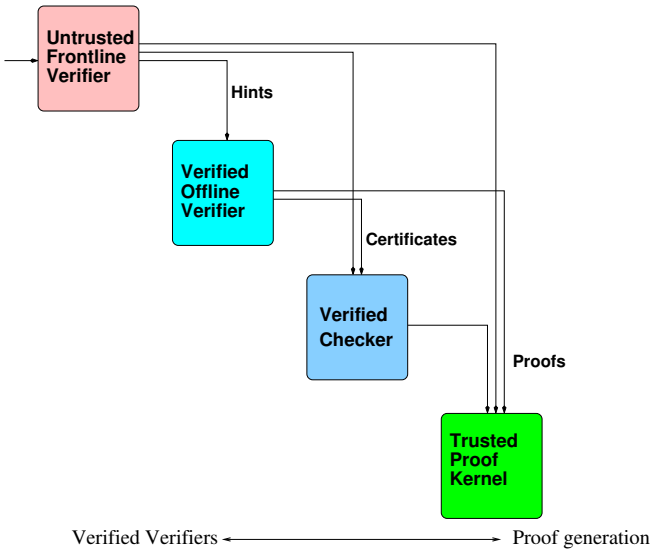

Fig. 1. The Kernel of Truth Architecture

proofs are actually certificates and not formal proofs, and not all inference procedures are similarly amenable to proof generation.

Some systems use reflection to verify and apply decision procedures. In computational reflection [BM81], the logic is used to formalize a syntactic fragment such as arithmetic or propositional logic and to verify an inference procedure on this representation. A meaning function is used to connect this syntax to formulas in the logic. Computational reflection does not require any external devices, but it does require that the inference procedure be executable. It is also possible to verify inference procedures non-reflectively using a different trusted or untrusted inference tool.

Recently, Davis [Dav09] has completed a dissertation where he has verified a fairly sophisticated theorem prover, a simplified version of ACL2, in a reflective manner, by defining 11 layers of proof checkers of increasing sophistication. The most sophisticated of these, level 11, includes mechanisms for induction, rewriting, and simplification. This level 11 proof checker is used to define proof checkers at levels 1 to 11, and to show that proofs at level $i + 1$ can be justified by proofs at level i, for $1 \leq i \leq 10$. These correctness proofs at level 11 can then be translated to level 1 proofs.

Our KoT approach has some similarities to the approach used by Davis, but we focus on building verified checkers for certificates and not on verifying the inference procedures themselves. The main reason for this is that high-end inference tools evolve rapidly. Even if verification were feasible, it would be hard to keep up with the changes to these tools. In contrast, the certificates generated by these tools can be fairly stable. Checking these certificates can be much more efficient than solving the original inference problem. Many different verification tools can share the same certificate format so that the investment in verifying the checkers can be amortized over these multiple uses.

Our KoT approach smoothly accomodates the entire spectrum from proof generation to verification so that some inference procedures can be justified by generating proofs corresponding to their claims, and others by verification, but in most cases we would use the middle option of generating certificates that are checked by verified checkers.

In Section 1, we present a brief overview of PVS. Section 2 describes the PVS simplifier which combines rewriting, simplification, and decision procedures. In Section 3, we describe the concept of inference systems as an abstract framework for proving the soundness and completeness of inference procedures. The Kernel of Truth framework is outlined in Section 4 where we describe a kernel proof checker that is being developed within PVS to serve as a reference proof system to justify the correctness of various specialized checkers. Checkers for rewriting and resolution are presented in Section 5, and the current status of the project and future work are covered in Section 6.

1 Brief Background on PVS

The Prototype Verification System (PVS) is a comprehensive framework for interactive and automated verification based on higher-order logic [ORSvH95] where variables can range not only over individuals, as in first-order logic, but also over functions, functions of functions, and so on. Higher-order logic uses types to avoid paradoxes due to self-application. Types are built from base types such as the Booleans `bool` and the real numbers `real`. The type [`A`→`B`] represents the type of functions with domain type `A` and range type `B`. For example, [A→`bool`] represents the type of predicates over the type A, and we abbreviate this as `PRED[`A`]` or as `set[`A`]`. The type $[A_1, \ldots, A_n]$ represents the type of n-tuples where the i'th element has type A_i for $1 \leq i \leq n$. In addition, PVS has predicate subtypes which are of the form $\{x : T | e\}$ which contains the elements x of T satisfying e. With this, we can define subtypes for subranges, rational numbers, integers, even numbers, prime numbers, ordering relations, and order-preserving maps. For example, the subtype of even numbers can be defined as {`i : int | EXISTS (j : int) 2*j = i`}. With predicate subtypes, typechecking and theorem proving become interdependent since the demonstration that an expression like $6 + 4$ is an even number now requires a proof. The PVS typechecker generates proof obligations corresponding to predicate subtypes called type correctness conditions (TCCs). It also makes use of typing judgements to incorporate forward chaining rules such as the assertion that the sum of even numbers is an even number. PVS also has dependent types such as $[x : A \rightarrow B]$, where the range type B can depend on the domain element x. For example, if B is the type *multiples*(x) containing the integer multiples of x, then the dependent type $[x : int \rightarrow$ *multiples*$(x)]$ contains those functions on the integers that map each integer x to some multiple of x. Arrays are just functions. Update expressions can be applied to update the value of a function, record, or tuple at a specific index, field, or position, respectively. The PVS language has other features like parametric theories and recursive and corecursive datatypes. The PVS language and its type system can be used to embed other methodologies that require the generation of proof obligations, for example, Hoare logic [Hoa69,HJ00] or the B method [Abr96,Muñ99].

Almost all of the PVS language is executable. The only non-executable parts are equality on infinite higher-order types, which includes quantification on infinite do-

mains. The PVS code generator detects when it is safe to evaluate updates destructively and is able to generate efficient code in these cases.

PVS also has an interactive proof checker that builds on various automated procedures for decision procedures, binary decision diagrams, satisfiability modulo theories, and rewriting. The proof checker uses a sequent representation for proof goals such that each proof step either completes the proof of a subgoal or generates new subgoals. Proof strategies are built from primitive inference steps using a strategy language. Strategies can be defined to execute complex patterns of proof steps, like induction followed by simpification and rewriting. The PVS simplifier which combines rewriting, simplification, and the use of decision procedures is described in the next section.

2 Combining Rewriting, Simplification, and Decision Procedures

A simplifier is a crucial part of an interactive proof assistant. It must ensure that the formula as presented to the user should have the expected simplifications applied. Many simplifications such as eliminating multiplication by 0, cancelling common factors in a fraction, and applying distributive laws, are quite natural. Others, such as beta reduction, are usually, but not always, a good idea. Decision procedures need to be employed during simplification. They can be used to propagate known information so that an expression of the form $i = j \Rightarrow A[i := v](j) \neq v$ can be simplified to `FALSE`. With decision procedures, simplification now becomes contextual. For example, the context $i = j$ is used in simplifying the expression $A[i := v](j) \neq v$. It also makes sense to integrate rewriting into the simplifier. Many simplifications can be expressed using rewriting. Conversely, rewriting also exploits simplification since we can assume that the expression being rewritten is in simplified form. Also, conditions in the application of conditional rewrite rules can often be discharged by simplification.

The PVS simplifier employs an inside-out strategy where sub-expressions are simplified before the expression is analyzed. The simplifier also carries a context for the decision procedure that is incrementally extended with new assertions. For example, when simplifying the branches of a conditional expression, the condition is asserted positively in the `THEN` branch, and negatively in the `ELSE` branch. The ground decision procedures can be used to decide if a given formula (that is, a boolean expression) is true or false (or not known to be either) with respect to the current context and relative to theories such as those of equality over uninterpreted function symbols and linear arithmetic. In a sequent of the form $a_1 \ldots a_m \vdash b_1 \ldots b_n$, the a_i are simplified and recorded as being true, and the b_i are simplified and recorded as being false. The simplifications are described below. The recording process can yield a refutation in which case the sequent has been proved. The ground decision procedure is a Shostak combination [Sho84,RS01,SR02] of the theory of equality with uninterpreted function symbols, quantifier-free integer and real linear arithmetic equalities and inequalities, array and function updates, and tuples and records.

The simplifier performs a range of simplifications. One set of simplifications applies to redexes as in the following examples (drawn from the PVS Prover Guide [SORSC99]).

1. Lambda redex: `(lambda x : x * x)(2)` $\longrightarrow$ 2 * 2

2. Record redex : `b((# a:= 1, b:= 2, c:= 3 #))` $\longrightarrow$ `2`
3. Tuple redex : `proj_2((1, 2, 3))` $\longrightarrow$ `2`
4. Function update redex: For function `f`,

$$\texttt{(f WITH [(i) := 3])(i)} \Longrightarrow \texttt{3}$$
$$\texttt{(f WITH [(0) := 3])(1)} \Longrightarrow \texttt{f(1)}$$

5. Record update redex: For record `r`,

$$\texttt{a(r WITH [(a) := 3])} \Longrightarrow \texttt{3}$$
$$\texttt{a(r WITH [(b) := 2])} \Longrightarrow \texttt{a(r)}$$

6. Cotuple redex:

$$\texttt{in\_2(out\_2(x))} \Longrightarrow \texttt{x}$$
$$\texttt{out\_2(in\_2(x))} \Longrightarrow \texttt{x}$$
$$\texttt{in\_2?(in\_2(x))} \Longrightarrow \texttt{TRUE}$$
$$\texttt{in\_1?(in\_2(x))} \Longrightarrow \texttt{FALSE}$$

7. Datatype redex: `car(cons(1, null))` $\Longrightarrow$ `1`
8. Recognizer redex:

$$\texttt{cons?(null)} \Longrightarrow \texttt{FALSE}$$
$$\texttt{cons?(cons(1, null))} \Longrightarrow \texttt{TRUE}$$

9. Subtype redex: `even?(i)` $\Longrightarrow$ `TRUE`, if `even?` is one of the subtype predicates in the type of `i`.

Several of the above simplifications can be expressed as rewrite rules, but some like record and tuple reduction are generic across records and tuple types and require a family of rewrite rules depending on the actual type of the record or tuple. The second of the function update reductions requires a disequality to be established in the general case, and is therefore not directly representable as a rewrite rule. The remaining simplifications are summarized in brief. The PVS prover guide [SORSC99] contains more details.

A second set of simplifications addresses arithmetic expressions which are placed into an ordered sum-of-products form while grouping similar monomials and eliminating multiplication by 0 or 1. These simplifications yield a normal form for polynomials.

A third set of simplifications applies to Boolean expressions involving conjunction, disjunction, implication, equivalence, and negation.

A fourth set of simplifications applies to conditional and case expressions to prune infeasible branches and merge equivalent branches. For conditional expressions, the test is added to the context when simplifying the `THEN` branch, and its negation is added when simplifying the `ELSE` branch.

A final set of simplifications applies to quantified expressions. Note that when the body of a lambda-expression or a quantified expression is simplified, the type constraints on the bound variables are assumed, i.e., added to the context. Examples of simplifications applied to quantified expressions are

```
      (EXISTS x: x = 5) ⟹ TRUE
(EXISTS x, y, z: x = y + z AND f(x, y, z))
                          ⟹ (EXISTS y, z: f(y + z, y, z))
 (EXISTS (x: T): TRUE) ⟹ TRUE
(FORALL (x: T): FALSE) ⟹ FALSE
```

The last two simplifications only happen when the type `T` is known to be nonempty.

Rewriting. The PVS simplifier uses conditional rewriting to simplify expressions with respect to definitions and lemmas that have been installed by the user. A conditional rewrite rule triggers only when the conditions *simplify* to `TRUE`. Some rewrite rules such as recursive definition might loop if applied unconditionally. Therefore, if the right-hand side expression of a recursive definition is a conditional expression, then the top-level condition must simplify to `TRUE` or `FALSE` in order for the rewrite to occur. This mix of decision procedures and rewriting means that context comes into play in simplifying these conditions. Since the instantiable variables in a rewrite rule can have type constraints, proof obligations are generated corresponding to these type constraints on the actual instantiation for these variables. These proof obligations must also be discharged by the simplifier. The type constraints on the instantiable variables are therefore treated as conditions. Matching on the left-hand side of the rewrite rules also uses decision procedures to, for example, match a term a with $i + 3$, where i is a natural number, when it is possible to demonstrate that a is at least 3 in the given context. Decision procedures can also be used to check if a pattern of the form $f(x, x)$ matches an instance of the form $f(a, b)$ where $a = b$ is known in the context. Pattern matching is also lifted to the higher-order level through the use of Miller's higher-order patterns [Mil90] which turns out to be very useful for rewrite rules involving higher-order operations such as `map` and `reduce`.

3 Rewriting and Inference Systems

We now present *inference systems* as an abstract framework for presenting and reasoning about inference procedures [SR02,Sha05,dMDS07,Sha09]. Inference systems can be represented using rewriting logic. An inference system is a triple $\langle \Psi, \Lambda, \vdash \rangle$ consisting of a set Ψ of inference states, a mapping Λ from an inference state to a formula, and a binary inference relation $\vdash$ between inference states. For each formula ϕ, there must be at least one state ψ such that $\Lambda(\psi) = \phi$. There is a special unsatisfiable inference state $\bot$. Given an input formula, the inference system is used to construct a sequence of logical states $\psi_0 \vdash \ldots \vdash \bot$ where the input formula is represented by the first state. The inference relation $\vdash$ must be

1. **Conservative:** If $\psi \vdash \psi'$, then $\Lambda(\psi)$ and $\Lambda(\psi')$ must be equisatisfiable.
2. **Progressive:** For any subset S of Ψ, there is a state $\psi \in S$ such that there is no $\psi' \in S$ where $\psi \vdash \psi'$.
3. **Canonizing:** If $\psi \in \Psi$ is irreducible, that is, there is no ψ' such that $\psi \vdash \psi'$, then either $\psi \equiv \bot$ or $\Lambda(\psi)$ is satisfiable.

Res	$\dfrac{K, k \vee \kappa_1, \overline{k} \vee \kappa_2}{K, k \vee \kappa_1, \overline{k} \vee \kappa_2, \kappa_1 \vee \kappa_2}$ $\kappa_1 \vee \kappa_2 \notin K$ $\kappa_1 \vee \kappa_2$ is not tautological
Contrad	$\dfrac{K}{\bot}$ if $p, \neg p \in K$ for some p

Fig. 2. Inference System for Ordered Resolution

Inference systems are presented in the form of inference rules. For example, the inference system for ordered resolution on a set of ordered clauses (deleting tautologies, i.e., clauses containing a literal and its negation) is given in Figure 2.

In a number of cases, inference rules can be given as rewrite rules. In the case of resolution, the rewrite rules operate on an inference state that is a set of clauses. One rewrite rule adds a new clause obtained by resolving two clauses, and the other rewrite rule detects a contradiction.

Inference systems can be given for a variety of decision procedures including SAT and SMT solvers. These inference systems can also be used to construct satisfying assignments when the input formulas are satisfiable, and proofs when the input formulas are unsatisfiable. Theory solvers for theories such as equality, arithmetic, and arrays can also be expressed as inference systems. While some of these require specialized data structures like hash tables and linked pointer structures, it is possible to prototype such solvers within a rewriting system like Maude.

4 A Kernel of Truth

Some applications, particularly safety-critical and security-critical ones, need the claims made using complex inference tools to be certified. For inference, the expected standard for certification is that of a proof. We have already noted that it is possible to construct inference procedures that are proof generating. In many cases, the overhead of proof generation is not significant, although the representation of these proofs can become large. State-of-the-art inference tools are constantly being modified and improved, and proof generation is an added burden. For example, in the case of the PVS simplifier described in Section 2, we would have to combine proofs from many different sources.

We outline a lighter approach to certifying inference claims. In our approach, we use a kernel proof checker as the reference standard. In our case, we use PVS to define a proof checker for first-order logic with the axioms of ZFC. Such a proof checker can be defined in about 500 lines of PVS. Though this proof checker is executable, we mostly use it to demonstrate the existence of proofs that are, in the usual case, not explicitly constructed. The point of the reference proof checker is to demonstrate the correctness of other checkers. These checkers can range from reference implementations of inference procedures to those that check certificates for specific classes of problems. We illustrate this with checkers for resolution proofs and certificates from rewriting.

The reference proof checker has some notable features. One, it uses one-sided sequents. This is mainly to reduce the size and complexity of the proof calculus. The kernel itself could be used to justify the correctness of a two-sided sequent calculus.

The system we use is quite similar to the one given in Shoenfield [Sho67]. Second, it uses two kinds of function and predicate symbols. *Interpreted* function and predicate symbols are used for defined operations such as those for equality, set membership, and arithmetic. Uninterpreted function and predicate symbols are used as schematic operations. These can be substituted by lambda-expressions of the appropriate arity. Such lambda-expressions do not have a first-class status in the logic, but are merely used to instantiate the schematic operators. Schematic operators have several uses. They can be used to introduce eigenvariables corresponding to the sequent rule for universal quantification as schematic constants. Uninterpreted predicates of arity 0 can serve as propositional atoms. Uninterpreted functions and predicate can also be used to capture schematic axioms and theorems. For example, the comprehension axiom scheme of set theory can be written as

$$\forall y.\exists z.\forall x.(x \in z \iff x \in y \wedge p(x)),$$

where p is a schematic predicate. Here, p can be replaced by a lambda-expression of the form $\lambda w.A$, that contains no free variables (but may contain schematic function and predicate symbols) to yield $\forall y.\exists z.\forall x.x \in z \iff x \in y \wedge A[x/w]$. Similarly, the replacement axiom scheme can be written as

$$\forall w.(\forall x \in w.\exists! y.q(x,y,w)) \Rightarrow \exists z.\forall y.(y \in z \iff \exists x \in w.q(x,y,w)),$$

where q is a schematic predicate.

With this proof checker, we can also define the concept of an LCF-style *tactic*. A *theorem* is a sequent that has a proof. A tactic then is an operation that maps a *conclusion* sequent $\vdash \Delta$ to a list of premise sequents $\vdash \Delta_1, \ldots, \vdash \Delta_n$ such that the conclusion is a theorem if the premise sequents are theorems. This concept of a tactic can be given as a type in PVS.

The basic judgement is given by a one-sided sequent of the form $\vdash A_1, \ldots, A_n$. We have a contraction rule that allows $\vdash \Gamma$ to be derived from $\vdash \Delta$ when $\Delta \subseteq \Gamma$. The basic propositional connectives are negation and disjunction, and these are used to define the other connectives. The propositional proof rules are shown in Figure 3 and the quantifier rules are shown in Figure 4.

As noted earlier, the language admits schematic function and predicate symbols that can be instantiated. For n-ary uninterpreted function symbol f, let $\Delta[\lambda\overline{x}.s/f]$, where $\overline{x}$ represents the sequence $x_1, \ldots, x_n$, be the result of replacing each subterm $f(t_1, \ldots, t_n)$ in Δ with $s[t_1/x_1, \ldots, t_n/x_n]$. Similarly, for n-ary uninterpreted predicate symbol p, let $\Delta[p \leftarrow \lambda\overline{x}.A]$ be the result of replacing each subformula $p(t_1, \ldots, t_n)$ by $A[t_1/x_1, \ldots, t_n/x_n]$ while renaming bound variables in A as needed to avoid variable capture. We then have a function instantiation rule that allows $\vdash \Delta[\lambda\overline{x}.s/f]$ to be derived from $\vdash \Delta$, and the predicate instantiation rule allows $\vdash \Delta[\lambda\overline{x}.s/f]$ to be derived from $\vdash \Delta$.

For equality, we have inference rules corresponding to reflexivity and congruence. The rules for transitivity and symmetry can be derived from reflexivity and predicate instantiation.

Ax	$\dfrac{}{\vdash A, \neg A, \Delta}$
$\neg\neg$	$\dfrac{\vdash A, \Delta}{\vdash \neg\neg A, \Delta}$
$\vee$	$\dfrac{\vdash A, B, \Delta}{\vdash A \vee B, \Delta}$
$\neg\vee$	$\dfrac{\vdash \neg A, \Delta \quad \vdash \neg B, \Delta}{\vdash \neg(A \vee B), \Delta}$
Cut	$\dfrac{\vdash A, \Delta \qquad \vdash \neg A, \Delta}{\vdash \Delta}$

Fig. 3. A Sequent Calculus for Propositional Logic

$\exists$	$\dfrac{\vdash A[t/x], \Delta}{\vdash \exists x.A, \Delta}$
$\neg\exists$	$\dfrac{\vdash \neg A[c/x], \Delta}{\neg\exists x.A, \Delta}$

Fig. 4. Sequent proof rules for quantification. The schematic constant c must not occur in Δ.

5 A Verified Checker for Rewriting and other Inference Procedures

We now describe some ongoing work on building a certified checker for rewriting. Here, we rely only on the first-order logic part of the kernel checker. A proof system for certifying rewriting is given by Rosu, Eker, Lincoln, and Meseguer [RELM03]. We describe a checker for rewriting that we are currently verifying. A term is either a variable or an n-ary function symbol, with $0 \leq n$, applied to a sequence of n terms. The free variables $vars(s)$ of a term s is the set of all the variables that occur in the term. A path π is a a finite sequence of natural numbers. Given a term s and a path π, the subterm $s|_\pi$ of s at π is defined as s itself, when π is empty, and as $s_i|_{\pi'}$, where $\pi \equiv i, \pi'$ and $s \equiv f(s_1, \ldots, s_n)$. The result of replacing the subterm $s|_\pi$ by a term t is represented by $s|_{\pi \leftarrow t}$. We restrict our attention to unconditional rewrite rules of the form $\forall \overline{x}.l = r$, where $\overline{x}$ is a sequence of distinct variables that contains all and only the variables in $vars(l)$ and $vars(r) \subseteq vars(l)$. The rewriter takes a set of rewrite rules and applies them to rewrite an expression e to e'. We represent the certificate as a sequence of triples $\langle \tau_1, \ldots, \tau_m \rangle$, where each triple τ_i of the form $\langle R_i, \pi_i, \sigma_i \rangle$ consists of a rewrite rule R_i, a path π_i, and a substitution σ_i. Such a sequence of triples is a valid certificate for the claim $e = e'$ if there is a sequence of terms $e_0, \ldots, e_m$ such that $e \equiv e_0$ and $e' \equiv e_m$, and $e_{i+1} \equiv e_i|_{\pi_{i+1} \leftarrow \sigma_{i+1}(r_{i+1})}$, where l_{i+1} and r_{i+1} are the left-hand and right-hand sides of the rewrite rule R_{i+1} and $e_i|_{\pi_{i+1}} \equiv \sigma_{i+1}(l_{i+1})$ for $0 \leq i < m$.

The checker for such a certificate has to check the validity conditions above. We define $rwcheck(R, \langle \tau_1, \ldots, \tau_m \rangle, e, e')$ to check that $\langle \tau_1, \ldots, \tau_m \rangle$ yields a sequence of replacements from e to e' using the rewrite rules in R. We then have to prove the

metatheorem that whenever $rwcheck(R, \langle \tau_1, \ldots, \tau_m \rangle, e, e')$ holds, there is a formal proof of $\vdash \neg R, e = e'$, where $\neg R$ is the formula-wise negation of each element of R. Once this metatheorem is proved, there is no need to actually generate this proof. For a rewrite step from e_i to e_{i+1}, the formal justification goes as follows. First, we establish the instantiation rule where we can derive $\vdash \sigma(l = r)$ from $\vdash \forall \overline{x}.l = r$, where σ maps each variable in $\overline{x}$ is mapped to a ground term, i.e., a term with no free variables. The proof of $e = e'$ then follows by applying congruence to $\vdash \sigma_{i+1}(l_{i+1} = r_{i+1})$ to derive $\vdash e_i = e_{i+1}$, and then transitivity to establish $\vdash e_0 = e_m$ from the sequents $\vdash e_i = e_{i+1}$, for $0 \leq i < m$.

In the more general situation of proof development in first-order and higher-order logic, rewriting can occur within the body of a quantification. The validity checker for certificates requires a more powerful metatheorem that uses the *equality theorem* [Sho67] that justifies the replacement of one term by a provably equal one within a formula.

Other specialized proof calculi can be similarly justified using certificate formats that are validated by verified checkers. For example, the proof system for resolution can be justified as follows. We have to show that when a set of clauses K yields a refutation, then the negation of the formula corresponding to K is provable. For some applications, we have to represent the derivation of clauses and not just refutational proofs. Each resolution step where a clause κ is derived from the clauses κ_1 and κ_2 is represented by the proof of the sequent $\vdash \neg\kappa_1, \neg\kappa_2, \kappa$. This sequent is easily proved from Ax, $\vee$, and $\neg\vee$ steps. A complete resolution refutation from a set of clauses K is represented by the proof of the sequent $\vdash \neg K$. Such a proof is constructed from those of the individual resolution inferences using the Cut rule. The resolution calculus is used as a kernel for justifying a SAT solver as a verified checker.

In addition to certificates and proof formats, the KoT kernel can also be used as a foundation for other logics. For example, a proof calculus for a modal logic or a higher-order logic can be justified in terms of its set-theoretic semantics. Once this is done, we can use the proof checker for the new logic as a kernel for other checkers. We plan to use this to justify proof calculi for various higher-order logics (including PVS) as well as modal, temporal, and program logics. We also plan to develop certificate checkers for SAT and SMT solvers, model checkers, and program analyzers.

6 Conclusions

Modern implementations of inference procedures like rewriting and propositional and theory satisfiability are extremely sophisticated and not easily amenable to formal verification. This makes it difficult to certify the results obtained by these procedures. It is even more difficult to certify results that are obtained by a combination of these tools. We address the challenge of certifying results from untrusted tools by relying on verified checkers that can be used to validate certificates generated by these tools. Our *Kernel of Truth* approach takes a middle ground between proof generation, where the untrusted tools are required to generate formal proofs, and verification, where only verified inference tools are used. Both these extreme cases are feasible within the KoT framework, but we also allow the more practical alternative of verifying checkers that

use other logics or representations of certificates. Our approach is also similar to *translation validation* [PSS98] where each source-to-binary translation by a compiler is verified. These individual translations are often easily verified, whereas the verification of an entire compiler is a monumental task [SC98]. The KoT approach similarly avoids directly verifying inference procedures in favor of the checking the individual claims made by them.

The idea of verifying the inference steps of PVS within PVS itself might seem circular and vacuous. However, this is the one instance where we exploit the availability of explicit formal proofs to break the circularity. This is done by using the KoT framework itself to generate the complete formal proof for the verification of the correctness of any checkers. This proof can be checked by the kernel proof checker so that we need not trust the inference procedures used by PVS.

As mentioned at the beginning, the work we have described is still at a very early stage. We have defined the kernel proof checker for first-order logic described in Section 4 and have used it to prove some basic metatheorems. In earlier work from 2007 with Marc Vaucher, we also verified a sophisticated SAT solver using PVS. With Andrei Dan and Antoine Toubhans, we have developed a verified certificate checker for the proof traces generated by PicoSAT [Bie08]. A similar verified checker has been developed by Darbari, Fischer, and Marques-Silva [DFMS10]. We are working toward verified certificate checkers for rewriting, satisfiability modulo theories (SMT), and simplifiers. We hope to eventually develop a range of certificate checkers so that inference tools like PVS, Yices [DdM06], and SAL [dMOR$^+$04] can be instrumented to generate checkable certificates.

The approach of checking the results of a computation with a verified checker is not restricted to inference tools. Result checking can be applied to a wide range of computations where there is a specific correctness claim associated with the output. Such an approach has been already been advocated by Mehlhorn [Meh03] in what he calls the *Reliable Algorithmic Software Challenge*. Generating efficiently checkable certificates both for inference and non-inference procedures is itself an interesting challenge. At the Workshop on Rewriting Logic and Applications, José Meseguer posed the challenge of certifying that a unifier is the most general unifier. It is easy to certify that a substitution is in fact a unifier, but it is often important to know that it is in fact the most general one. There are many ways to approach such challenges. One approach is to verify a unification algorithm[Pau84]. Another approach is to demonstrate that no generalization of the unifier is a valid unifier. A third approach is for the unification procedure to generate a trace that demonstrates how any solution to the unification problem can be transformed into an instance of the unifier, but generating and checking such traces is still a problem. A similar challenge arises with graph algorithms where we must demonstrate that a path is indeed the shortest path or that a target vertex is unreachable. In earlier work [Sha10], we showed how fixpoints can be used to construct efficient certificates for such graph search algorithms. The KoT project is a response to Mehlhorn's challenge for the specific case of inference procedures but we are also interested in extending it to a larger class of computations. Compared to the goal of verifying software, the Kernel of Truth framework has the more limited ambition of checking computations in a verifiable manner.

References

[Abr96] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[Bie08] Armin Biere. PicoSAT essentials. *JSAT*, 4(2-4):75–97, 2008.

[BKK+96] P. Borovansky, C. Kirchner, H. Kirchner, P. E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In *Proc. of the First Int. Workshop on Rewriting Logic*, volume 4. Elsevier, 1996.

[Ble77] W. W. Bledsoe. Non-resolution theorem proving. *Artificial Intelligence*, 9:1–36, 1977.

[BM79] R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, New York, NY, 1979.

[BM81] R. S. Boyer and J S. Moore. Metafunctions: Proving them correct and using them efficiently as new proof procedures. In R. S. Boyer and J S. Moore, editors, *The Correctness Problem in Computer Science*. Academic Press, London, 1981.

[BM88] R. S. Boyer and J. S. Moore. *A Computational Logic Handbook*. Academic Press, New York, NY, 1988.

[CAB+86] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall, Englewood Cliffs, NJ, 1986. Nuprl home page: http://www.cs.cornell.edu/Info/Projects/NuPRL/.

[CDE+99] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. The Maude system. In P. Narendran and M. Rusinowitch, editors, *Proceedings of the 10th International Conference on Rewriting Techniques and Applications (RTA-99)*, pages 240–243, Trento, Italy, 1999. Springer-Verlag LNCS 1631.

[Dav09] Jared Curran Davis. *A Self-Verifying Theorem Prover*. PhD thesis, Computer Science Department, The University of Texas at Austin, December 2009.

[DdM06] Bruno Dutertre and Leonardo de Moura. The Yices SMT solver. http://yices.csl.sri.com/, 2006.

[DFMS10] Ashish Darbari, Bernd Fischer, and Joao Marques-Silva. Industrial-strength certified sat solving through verified sat proof checking. In *Int. Colloq. on Theoretical Aspects of Computing (ICTAC), 2010*, 2010. To appear.

[dMDS07] Leonardo de Moura, Bruno Dutertre, and Natarajan Shankar. A tutorial on satisfiability modulo theories. In Werner Damm and H. Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007*, volume 4590 of *Lecture Notes in Computer Science*, pages 20–36. Springer-Verlag, 2007.

[dMOR+04] Leonardo de Moura, Sam Owre, Harald Rueß, John Rushby, N. Shankar, Maria Sorea, and Ashish Tiwari. SAL 2. In Rajeev Alur and Doron Peled, editors, *Computer-Aided Verification, CAV '2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 496–500, Boston, MA, July 2004. Springer-Verlag. SAL home page: http://sal.csl.sri.com/.

[GKM+87] J. Goguen, C. Kirchner, A. Megrelis, J. Meseguer, and T. Winkler. An introduction to OBJ3. In S. Kaplan and J.-P. Jouannaud, editors, *Conditional Term Rewriting Systems, 1st International workshop, Orsay, France*, volume 308 of *Lecture Notes in Computer Science*, pages 258–263. Springer-Verlag, July 1987.

[GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, Cambridge, UK, 1993. HOL home page: http://www.cl.cam.ac.uk/Research/HVG/HOL/.

[GW88] Joseph A. Goguen and Timothy Winkler. Introducing OBJ. Technical Report SRI-CSL-88-9, Computer Science Laboratory, SRI International, Menlo Park, CA, August 1988.

[HJ00] Marieke Huisman and Bart Jacobs. Java program verfication via a hoare logic with abrupt termination. In Tom Maibaum, editor, *Fundamental Approaches to Software Engineering, FASE 2000*, volume 1783 of *Lecture Notes in Computer Science*, pages 284–303, Berlin, Germany, March/April 2000. Springer-Verlag.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–583, 1969.

[KMM00] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*, volume 3 of *Advances in Formal Methods*. Kluwer, 2000.

[KZ88] D. Kapur and H. Zhang. RRL: A rewrite rule laboratory. In E. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction (CADE)*, volume 310 of *Lecture Notes in Computer Science*, pages 768–769, Argonne, IL, May 1988. Springer-Verlag.

[Meh03] Kurt Mehlhorn. The reliable algorithmic software challenge RASC. In Klaus Jansen, Marian Margraf, Monaldo Mastrolilli, and José D. P. Rolim, editors, *Experimental and Efficient Algorithms, Second International Workshop, WEA 2003, Ascona, Switzerland, May 26-28, 2003, Proceedings*, volume 2647 of *Lecture Notes in Computer Science*, page 222. Springer, 2003.

[Mil90] Dale Miller. An extension to ML to handle bound variables in data structures: Preliminary report. In *Informal Proceedings of the Logical Frameworks BRA Workshop*, Nice, France, June 1990. Available as UPenn CIS technical report MS-CIS-90-59.

[Muñ99] César Muñoz. PBS: Support for the B-method in PVS. Technical Report SRI-CSL-99-1, Computer Science Laboratory, SRI International, Menlo Park, CA, February 1999.

[ORSvH95] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995. PVS home page: http://pvs.csl.sri.com.

[Pau84] L. C. Paulson. Verifying the unification algorithm in LCF. Technical Report 50, University of Cambridge Computer Laboratory, 1984.

[Pau94] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994. Isabelle home page: http://www.cl.cam.ac.uk/research/hvg/Isabelle/.

[PSS98] Amir Pnueli, M. Siegel, and Eli Singerman. Translation validation. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 151–166, 1998.

[RELM03] Grigore Rosu, Steven Eker, Patrick Lincoln, and José Meseguer. Certifying and synthesizing membership equational proofs. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME*, volume 2805 of *Lecture Notes in Computer Science*, pages 359–380. Springer, 2003.

[RS01] Harald Rueß and Natarajan Shankar. Deconstructing Shostak. In *16th Annual IEEE Symposium on Logic in Computer Science*, pages 19–28, Boston, MA, July 2001. IEEE Computer Society.

[SC98] David W. J. Stringer-Calvert. *Mechanical Verification of Compiler Correctness*. PhD thesis, University of York, Department of Computer Science, York, England, March 1998. Available at http://www.csl.sri.com/ dave_sc/papers/thesis.html.

[Sha05] Natarajan Shankar. Inference systems for logical algorithms. In R. Ramanujam and Sandeep Sen, editors, *FSTTCS 2005: Foundations of Software Technology and*

Theoretical Computer Science, volume 3821 of *Lecture Notes in Computer Science*, pages 60–78. Springer Verlag, 2005.

[Sha08] Natarajan Shankar. Trust and automation in verification tools. In Sungdeok (Steve) Cha, Jin-Young Choi, Moonzoo Kim, Insup Lee, and Mahesh Viswanathan, editors, *6th International Symposium on Automated Technology for Verification and Analysis (ATVA 2008)*, volume 5311 of *Lecture Notes in Computer Science*, pages 4–17. Springer-Verlag, October 2008.

[Sha09] Natarajan Shankar. Automated deduction for verification. *ACM Comput. Surv*, 41(4):20:1–56, 2009.

[Sha10] Natarajan Shankar. Fixpoint and search in pvs. In Peter Müller, editor, *Advanced Lectures on Software Engineering*, volume 6029 of *Lecture Notes in Computer Science*, pages 140–161. Springer-Verlag, 2010.

[Sho67] J. R. Shoenfield. *Mathematical Logic*. Addison-Wesley, Reading, MA, 1967.

[Sho84] Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, January 1984.

[SORSC99] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Prover Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[SR02] Natarajan Shankar and Harald Rueß. Combining Shostak theories. In Sophie Tison, editor, *International Conference on Rewriting Techniques and Applications (RTA '02)*, volume 2378 of *Lecture Notes in Computer Science*, pages 1–18, Copenhagen, Denmark, July 2002. Springer-Verlag.

[The09] The Coq Development Team. The Coq proof assistant reference manual version 8.2. Technical report, INRIA, February 2009.

[ZM03] Lintao Zhang and Sharad Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *DATE*, pages 10880–10885. IEEE Computer Society, 2003.