Appears in the Proceedings of CADE'02, LNCS. ©Springer-Verlag

Formal Verification of a Combination Decision Procedure[⋆]

Jonathan Ford and Natarajan Shankar

Computer Science Laboratory
SRI International, Menlo Park CA 94025 USA
{ford, shankar}@csl.sri.com
Phone: (650)859-5272

Abstract. Decision procedures for combinations of theories are at the core of many modern theorem provers such as ACL2, Ehdm, PVS, SIMPLIFY, the Stanford Pascal Verifier, STeP, SVC, and Z/Eves. Shostak, in 1984, published a decision procedure for the combination of canonizable and solvable theories. Recently, Ruess and Shankar showed Shostak's method to be incomplete and nonterminating, and presented a correct version of Shostak's algorithm along with informal proofs of termination, soundness, and completeness. We describe a formalization and mechanical verification of these proofs using the PVS verification system. The formalization itself posed significant challenges and the verification revealed some gaps in the informal argument.

1 Introduction

Decision procedures play an important rôle in a number of areas such as automated deduction, computer-aided verification, and constraint solving. Since bugs in decision procedures can lead to unsound inferences, it is natural to ask if such verification tools can themselves be verified. We present here the first instance of a verified decision procedure for a combination of theories based on Shostak's ideas. Shostak's algorithm [Sho84] for building decision procedures for the union of canonizable and solvable equational theories has been widely used despite the lack of a convincing correctness proof. Recently, Ruess and Shankar [RS01] showed that this algorithm (even with minor flaws corrected [CLS96]) was both nonterminating and incomplete. They gave a corrected version of the algorithm along with informal proofs for termination, soundness, and completeness. We undertook the challenge of formalizing and verifying these informal arguments using the PVS verification system [ORS92]. The results of our verification are presented here along with observations regarding the challenges that we encountered in the formalization and verification process.

⋆ This work was funded by NSF Grant CCR-0082560, DARPA/AFRL Contract F33615-00-C-3043, and NASA Contract NAS1-00079. Sam Owre, Harald Rueß, and John Rushby of SRI provided insightful comments on earlier drafts. We thank the anonymous referees for their constructive criticism.

The correctness of decision procedures has been an important theme in automated reasoning. Several approaches have been developed for using decision procedures to gain efficiency in proof construction without compromising soundness. The LCF approach [GMW79] admits only those decision procedures that can be introduced as *tactics*, which are metalanguage operations for reducing proof goals to subgoals in a way that is justifiable in terms of the primitive inferences of the object logic. Tactics can be hard to define (since they have to mimic proof steps) and inefficient (since they have to generate low-level inference steps). The generation of *proof objects* from finished proofs is another way of ensuring that each proof can be constructed using only the primitive inference steps. The construction of proof objects even from finished proofs can be inefficient in both time and space.

In order to avoid the inefficiency of fully expansive proof generation, a number of researchers have advocated the verification of decision procedures. Boyer and Moore [BM81] introduce a notion of metafunctions, i.e., function definitions in the object logic that could be applied to object logic expressions. They use computational reflection to capture the meanings of these expressions in the object logic and verify the soundness of some simple derived inference rules in this manner. Boyer and Moore [BM79] also verified the semantic correctness of a tautology checker for conditional expressions. Shankar [Sha85] verified both the semantic and proof-theoretic correctness of a tautology checker for propositional logic. Some recent examples of verified decision procedures include a Coq verification of a Gröbner basis algorithm for membership in polynomial ideals by Théry [Thé98], the verification of ordered binary decision diagram (OBDD) operations using PVS by von Henke, Pfab, Pfeifer, and Ruess [vHPPR98], and a similar Coq verification of OBDD operations by Verma and Goubault [VGL00]. Both the algorithm and the theory underlying the combination decision procedure considered here are significantly more complex than these previously verified decision procedures.

The primary contribution of our work is in demonstrating the feasibility of formally verifying complex decision procedures. The variant of Shostak's algorithm we have verified is quite recent and its foundations are not widely understood. Our verification closely follows the published informal proof [RS01] so that we could directly assess its validity. We also used details from an unpublished report that included proofs of some of the lemmas that were given without proof in the published paper. The verification exposed some gaps in the informal argument. We found a monotonicity claim in the informal argument to be false without qualification, but only the qualified form was actually used. A step that is hinted at as being routine, turned out to not be all that obvious. In the algorithm, any solution returned by the solver must contain variables that are either from the given equality or are "fresh". Making the notion of freshness precise, and working with this constraint proved to be one of the major challenges in the formal verification. The verification makes very heavy use of the PVS type system. Our use of PVS types exposed some of the weaknesses in a type propagation feature of the language called *typing judgements*.

Since PVS itself employs Shostak's method (with the incompleteness and non-termination bugs), the validity of this verification might be called into question. However, the Shostak procedure used in PVS is not known to be unsound. Future versions of PVS will employ the ICS decision procedures [FORS01] that are based on the theory verified here. Despite the circularity between the verifier and the verified program, this kind of verification is still quite useful. An unsuccessful proof attempt might reveal significant bugs. A successful verification of the decision procedures could be certified through proof-object generation but subsequently used without the supporting proof objects.

The decision procedure as verified here is not executable, but it is possible to derive a verified, executable version that can be turned into efficient Common Lisp code [Sha99]. The code generated from the verified decision procedure is unlikely to be as efficient as the highly optimized ICS implementation, but it could still be used as a reference procedure that can be invoked when certified results are needed.

We verify both soundness and completeness. The completeness property is crucial. Higher-level simplification routines might diverge or behave erroneously if they incorrectly assume completeness. Due to its complexity and popularity, the verification of Shostak's algorithm is a good case-study for assessing the feasibility of certifying decision procedures.

2 Shostak's Algorithm

We focus here on the verification of a decision procedure for equational theories where terms are constructed from a combination of interpreted and uninterpreted function symbols. There are two basic methods for building decision procedures for combinations of disjoint theories. Nelson and Oppen's method [NO79] combines decision procedures for the individual theories by allowing them to share specific kinds of equality information. Shostak's method [Sho84] extends congruence closure to equational theories that are canonizable and solvable. Nelson and Oppen's method is more generally applicable, but Shostak's method has certain advantages. It is an online algorithm, i.e., processes inputs incrementally, so that the term universe for the input is not known in advance. It also yields a useful function for computing a canonical form respecting the given input equalities.

All formulas are equalities between terms which are constructed from variables by means of n-ary function application for $n \geq 0$. Sequents of the form $T \vdash a = b$ assert the implication between the antecedent equalities in the set T and the consequent equality $a = b$. The basic theory of equality with all function symbols uninterpreted, i.e., without any fixed interpretation, is decidable by means of *congruence closure*. Shostak's algorithm extends the congruence closure decision procedure to handle interpreted operations from a *canonizable* and *solvable* theory. Informally, a theory is canonizable if there is a canonizer operation σ such that $\sigma(a) \equiv \sigma(b)$ exactly when $a = b$ is valid in the theory. It is solvable if there is an operation *solve* such that $\mathit{solve}(a = b)$ either returns $\bot$ when $a = b$ is unsatisfiable, or a solved form S that is equivalent to $a = b$.

Shostak's procedure takes as parameters, a solver *solve* and canonizer σ for a theory such as linear arithmetic. The algorithm verifies a sequent $T \vdash a = b$ by processing the equalities in T to build a solution set S of equalities in solved form, or to return $\bot$ indicating that a contradiction was found in T. If a solution set S is returned, then one can use S and σ to define a canonizer *can* such that $\mathit{can}(S)(f(e))$ returns $\sigma(f(\mathit{can}(S)(e)))$ if f is interpreted. If f is uninterpreted, $\mathit{can}(S)(f(e))$ returns c' for some c equivalent to $f(\mathit{can}(S)(e))$ where $c = c'$ is in S. The conclusion equality $a = b$ can be tested for validity by checking if $\mathit{can}(S)(a) \equiv \mathit{can}(S)(b)$. The operation $\mathit{can}(S)$ is also used for preprocessing the input equalities from T. The preprocessed input equalities are solved and the solution (if any) is composed with the existing value of S. The solution set S is maintained in congruence-closed form so that the right-hand sides of congruent left-hand side terms are merged by solving the equality between them and merging the results into S.

The theory of linear arithmetic is a typical example of a canonizable and solvable theory. A canonizer can be given by means of a function that returns an ordered sum-of-products representation for a given linear polynomial by merging monomials over the same variable into a single monomial. A solver can be given by using algebraic manipulations to isolate a variable on the left-hand side. The Shostak procedure of Ruess and Shankar [RS01] can be illustrated on the sequent

$$f(x-1) - 1 = f(y) + 1,\ y - x + 1 = 0 \vdash \mathit{false},$$

where $+$, $-$, and the numerals are from the theory of linear arithmetic, *false* is an abbreviation for $0 = 1$, and f is an uninterpreted function symbol. Starting with $S \equiv \emptyset$ in the base case, the preprocessing of $f(x-1) - 1 = f(y) + 1$ causes the equality to be placed into canonical form as $-1 + f(-1 + x) = 1 + f(y)$. The solution set S is initialized to contain reflexivity statements for the non-interpreted subterms in the canonicalized input equality as $\{x = x, y = y, f(-1+x) = f(-1 + x), f(y) = f(y)\}$. Solving $-1 + f(-1 + x) = 1 + f(y)$ yields $f(-1+x) = 2+f(y)$, and S is set to $\{x = x, y = y, f(-1+x) = 2+f(y), f(y) = f(y)\}$. No unmerged congruences are detected in S. Next, $y - x + 1 = 0$ is canonized as $1 - x + y = 0$, and solved as $x = 1 + y$. This solution is composed with S to yield $\{x = y + 1, y = y, f(-1 + x) = 2 + f(y), f(y) = f(y)\}$. The congruence between $f(-1 + x)$ and $f(y)$ is detected since the canonical form of $-1 + x$ is y when the solution for x is inserted and the result is canonized by σ. The procedure then tries to merge the respective solutions of $f(-1+x)$ and $f(y)$ by solving $2 + f(y) = f(y)$. The solver returns $\bot$ so that the original sequent is asserted to be valid.

As a second example, one can check that the sequent $f(x-1)-1 = f(y)+1 \vdash g(f(x-1)-2) = g(f(y))$ is valid by computing S to be $\{x = x, y = y, f(-1+x) = 2+f(y), f(y) = f(y)\}$, and verifying $\mathit{can}(S)(g(f(x-1)-2)) \equiv \mathit{can}(S)(g(f(y)))$.

3 Formalizing Shostak's Algorithm in PVS

A brief introduction to PVS is given in Appendix A. The formalization exploits several advanced features of the PVS language including recursive datatypes, predicate subtypes, dependent types, Hilbert's choice operator, and inductive relations. We describe the formalization in sufficient detail so that it can be checked for conformity with the informal arguments [RS01] (abbreviated below as **RS**) and reproduced using some other automated proof checker.[1]

Syntax. Terms are built from a given signature consisting of a set of variables X and function symbols F. A *term* is either a variable x for $x \in X$ or of the form $f(a_1, \dots, a_n)$, where $f \in F$. A term of the form $f(a_1, \dots, a_n)$ is *interpreted* (respectively, *uninterpreted*) if f is interpreted (respectively, uninterpreted). Terms are formalized by means of a recursive datatype `syntax` consisting of a constructor `v` for variables with a natural number index field `index`, and an application constructor `app` with a function symbol field `func` and an arguments field `args` which is formalized as a *dependent* type `[below(arity(func)) -> syntax]` which represents an *array* of `syntax` in the arity of the function symbol of the term. The type `below(num)` for a natural number `num` is the (possibly empty) subrange $0, \dots,$ `num` $- 1$.[2] The function symbol type `funsymbs` is also a datatype consisting of constructors `ifn` and `ufn` for interpreted and uninterpreted function symbols, respectively, each with an `index` field and an `arity` field, and a `thry` (theory) field for interpreted function symbols.

```
funsymbs: DATATYPE                                                    1
  BEGIN
   IMPORTING theories
   ifn(index: nat, arity: nat, thry: TH): ifn?
   ufn(index: nat, arity: nat): ufn?
  END funsymbs

syntax: DATATYPE
  BEGIN
   IMPORTING funsymbs, max_lemmas
   v(index: nat): v?
   app(func: funsymbs,
       args: [below(arity(func)) -> syntax]): app?
  END syntax
```

Since we are admitting just one interpreted theory, we fix a theory `th`. The predicate `thry_func` checks that its argument is an interpreted function symbol

[1] The complete PVS 2.4.1 dump file is available at `ftp://ftp.csl.sri.com/pub/users/shankar/shostak-verification-dump`.

[2] An application could also be formalized in terms of a *list* of arguments whose length is the arity of the function symbol. The array-based formalization has some important advantages. Terms are well-formed by construction thus avoiding the need for cumbersome proof obligations. Operations on terms can be defined by a simple structural recursion without the use of mutual recursion on terms and lists of terms.

from theory th. The type thry_func is the predicate subtype corresponding to the predicate thry_func.

```
 thry_func(ff:funsymbs): bool =                                    2
   ifn?(ff) AND thry(ff) = th
```

The type of equalities is defined as a record type with fields lhs and rhs.

```
   equality: TYPE = [# lhs, rhs: syntax #]                         3
```

The variables a, b, and c are declared to range over terms, aa, bb, and cc range over equalities, and R, S, and T range over lists of equalities.

The set of variables in a term a is defined using datatype recursion as vars(a). Sets are just predicates in the higher-order logic so that a variable x is in the set vars(a) iff vars(a)(x) holds. The set vars(a) can be shown to be finite by structural induction. A term a is well-typed in n for a natural number n, if the index of any variable in a is below n. This is represented by the predicate well_typed?(n)(a) and the corresponding type typed(n). The operation of collecting the set of subterms of a given term is represented by subterm(a). The definitions of these operations are omitted.

Pure Terms. The canonizer and solver are defined for *pure* terms, i.e., terms without uninterpreted function symbols, but then applied to arbitrary terms by treating the uninterpreted subterms as variables. We formalize pure terms by means of a datatype pure that has two classes of variables: v(i) for the ordinary variables indexed by i, and u(a) corresponding to the uninterpreted term a. Function applications for pure terms are typed to contain only interpreted function symbols. It is easy to define an operation abs that converts a term to the corresponding pure term, and its inverse gamma.

```
pure[(IMPORTING theories) th: TH]: DATATYPE WITH SUBTYPES var?, func?  4
BEGIN
   IMPORTING syntax_ops[th]
   v(index: nat): v? : var?
   u(a: uninterpreted): u? :var?
   app(func: thry_func,
       args: [below(arity(func)) -> pure]): app? : func?
END pure
```

Semantics. The semantics for a term a is given by $M[\![a]\!]\rho$ for an *interpretation* M over a domain D such that $M(f)$ yields a mapping from D^n to D for function symbol f of arity n, and an *assignment* ρ mapping variables to values in D. For variables, $M[\![x]\!]\rho = \rho(x)$, and $M[\![f(a_1, \ldots, a_n)]\!]\rho = M(f)(M[\![a_1]\!]\rho, \ldots, M[\![a_n]\!]\rho)$. We say that $M, \rho \models a = b$ iff $M[\![a]\!]\rho = M[\![b]\!]\rho$, and $M \models a = b$ iff $M, \rho \models a = b$ for all assignments ρ over $vars(a = b)$. An equality is *valid* if for all D, M: $M \models a = b$.

The concept of a valid equality requires quantification over all domains D and interpretations M over D. In PVS, such a domain would have to be introduced as the type parameter of a theory. Since PVS does not admit quantification over types, the domain must be given as a subset or a subtype of a fixed type. We take this fixed type to be the set of all terms.[3] This type can be informally shown to be adequate for representating any domain set D for the purposes of equality. The assignment ρ is formalized as a mapping from the set of all variables to the domain D.

In the semantics for pure terms, the domain type `D` is the type of pure terms and a model is a dependent record type consisting of a domain field `mdom` that is a subset of `D`, and a function interpretation field `f` that is a dependent type mapping a function `ff` and an array of argument valuations to a valuation for the application. The type `arity(ff)` is an abbreviation for `below(arity(ff))`.

```
D: TYPE+ = pure                                                        5

model: TYPE = [# mdom : setof[D],
                 f: [ff: thry_func ->
                       [[arity(ff) -> (mdom)] -> (mdom)]] #]
```

Solutions. The "state" of the algorithm is maintained in a solution set S that is just a list of equalities of a special form. The operation `apply(S)(a)` (informally, $S(a)$) is defined recursively to look up the solution for `a` (if any) in `S`.[4]

```
apply(S)(a): RECURSIVE syntax =                                        6
 CASES S OF
  null: a,
  cons(aa, R): IF lhs(aa) = a
                THEN rhs(aa)
                ELSE apply(R)(a)
               ENDIF
 ENDCASES
MEASURE length(S)
```

The operation `replace_vars(S)(d)` (informally, $S[d]$) returns the result of replacing all occurrences of any left-hand side variable from `S` in a pure term `d`, by the corresponding right-hand side. The `replace_vars` operation is extended from pure terms to arbitrary terms as `replace_solvables`. The operation `subst(rho)(d)` (used in $\boxed{7}$) is similar to `replace_vars(S)(d)` but `rho` here is a substitution mapping variables to terms.

Canonizers. A canonizer σ for pure terms from a theory τ is a parameter to the combination decision procedure. A valid canonizer is required to verify validities, i.e., $\models_\tau a = b$ implies $\sigma(a) \equiv \sigma(b)$, and additionally preserve variables, $\sigma(x) = x$ and $vars(\sigma(a)) \subseteq vars(a)$, be idempotent, $\sigma(\sigma(a)) = \sigma(a)$, and leave

[3] The type of closed terms, when nonempty, is also a valid candidate for the domain.

[4] The termination of the recursive definition is justified by the measure `length(S)` which causes the typechecker to generate proof obligations verifying that the measure decreases with each recursive call.

subterms canonical, $\sigma(b) = b$ for any subterm b of $\sigma(a)$. These conditions on a valid canonizer are captured by the predicate `canonizer?(sigma)`. The validity condition is awkward since it uses an oracle $\models_\tau$ for τ-validity. We found a way to replace this condition by the sufficient pair of conditions on σ:

1. σ-substitutivity: $\sigma(\rho[a]) \equiv \sigma(\rho[\sigma(a)])$, for any substitution ρ, and
2. σ-distributivity: $\sigma(f(\sigma(a_1), \ldots, \sigma(a_n))) \equiv \sigma(f(a_1, \ldots, a_n))$.

`canonical?(sigma)(a)` is defined to hold when `sigma(a) = a`.

$\boxed{7}$

```
canonizer?(sigma): bool =
  (   (FORALL d, rho: sigma(subst(rho)(d)) = sigma(subst(rho)(sigma(d))))
  AND (FORALL d: app?(d) IMPLIES
        sigma(app(func(d), LAMBDA (i:arity(func(d))): sigma(args(d)(i))))
        = sigma(d))
  AND (FORALL u   : sigma(u) = u)
  AND (FORALL d, u: vars(sigma(d))(u) IMPLIES vars(d)(u))
  AND (FORALL d   : sigma(sigma(d)) = sigma(d))
  AND (FORALL d, f: sigma(d) = f IMPLIES
        (FORALL (i:arity(func(f))): canonical?(sigma)(args(f)(i)))))
```

The adaptation of the canonizer from pure terms to terms is done through `gamma` and `abs`. The canonizer for arbitrary terms, `sig(a)` (used in $\boxed{9}$ and $\boxed{10}$), is defined as `gamma(sigma(abs(a)))`, where `sigma` is the given canonizer for pure terms. Model M is a *σ-model* if $M \models \sigma(a) = a$ for all a, and $a = b$ is *σ-unsatisfiable* (formalized as the PVS predicate `unsatisfiable`) if $M, \rho \not\models a = b$ for all M and ρ.

Solver. A solver *solve* is another parameter to the algorithm. A valid solver must be such that *solve*$(a = b)$ either returns $\perp$ when $a = b$ is σ-unsatisfiable, or returns a (possibly empty) list S of n equalities of the form $x_i = t_i$ for $1 \leq i \leq n$, where $x_i \in vars(a = b)$ $x_i \not\equiv x_j$ for $i \neq j$, $x_i \notin vars(t_j)$, t_i is canonical ($\sigma(t_i) = t_i$), for $1 \leq x, y \leq n$, and $a = b$ and S are σ-equivalent: for all σ-models M and assignments ρ over the variables in a and b, $M, \rho \models a = b$ iff there is an assignment ρ' extending ρ, over the variables in S, a, and b, such that $M, \rho' \models S$.

The notion of a *solution* for pure term equalities is formalized as the predicate `solve(n, dd, S)` for an index `n`, an equality `dd`, and a solution list `S`. The predicate checks that `dd` is satisfiable, the solution list of equalities `S` is a well-formed solution that is σ-equivalent (formalized as the PVS predicate `sig_equivalent?`) to `dd`. Any variables in `S` not in `dd` must be of index above `n`.

$\boxed{8}$

```
   solve(n, dd, S): bool =
     IF unsatisfiable(dd) THEN
       FALSE
     ELSE
       new_vars_above(n, dd)(S) AND
       check_solution(dd)(S) AND
       sig_equivalent?(dd, S)
     ENDIF
```

A pure term solver is easily extended to one that works on terms. A given solver `solv` is typed so that `solv(m, dd)` returns a dependent record `r` with fields `n` and `s`, where `r‘n` is an index that is at least `m` and `r‘s` is either `bottom` or of the form `up(S)` for a solution list of equalities `S` that is well-typed in `r‘n`.

Canonical Forms. The operation $norm(S)(a)$ (represented as `norm(S)(a)`) for a canonizer `sig`, is informally defined as $\sigma(S[a])$. The definition of `norm` is used to show that if `solve(m, aa, S)` holds, then `norm(S)(lhs(aa)) = norm(S)(rhs(aa))`, and to define the composition of two equality lists `R` and `S` as `R o S`.

```
norm(S)(a): syntax = sig(replace_solvables(S)(a))                    9

o(R, S): RECURSIVE eqlist =
 CASES R OF
  null: S,
  cons(aa, T): cons(eq(lhs(aa), norm(S)(rhs(aa))), T o S)
 ENDCASES
MEASURE length(R)
```

Since composition is defined recursively, its definition includes a termination measure `length(R)` that is used to generate termination proof obligations. The definitions above are used to prove the associativity of composition and the claim: `norm(R o S)(a) = norm(S)(norm(R)(a))`.

The operation `lookup(S)(a)` is defined so that if `a` is a variable, then it returns `apply(S)(a)` which is the formalization of $S(a)$. When a is an application, then `lookup` is defined to scan S till it finds an equality whose left-hand side is of the form $f(a_1, \ldots, a_n)$, where $f(norm(S)(a_1), \ldots, norm(S)(a_n)) \equiv a$.[5]

The canonizer `can(S)(a)` is then defined in terms of the `lookup` operation.

```
can(S)(a): RECURSIVE syntax =                                        10
 CASES a OF
  v(i): apply(S)(a),
  app(ff, args):
   IF intheory?(a) THEN
    sig(app(ff, LAMBDA (i:arity(ff)): can(S)(args(i))))
   ELSE
    lookup(S)(app(ff, LAMBDA (i:arity(ff)): can(S)(args(i))))
   ENDIF
 ENDCASES
MEASURE rank(a)
```

Congruence. Congruence with respect to a solution set S, $f(a_1, \ldots, a_n) \overset{S}{\sim} f(b_1, \ldots, b_n)$, is defined to hold exactly when $norm(S)(a_i) \equiv norm(S)(b_i)$ for $1 \leq i \leq n$. This is captured formally by the predicate `congruent(S)(a, b)`.

[5] This definition of `lookup` is slightly different from that of **RS** which uses $S(a_i)$ instead of $norm(S)(a_i)$. The **RS** definition requires keeping $dom(S)$ subterm-closed, whereas we only require closure under the uninterpreted subterms. Our definition is executable in contrast to the **RS** definition which uses Hilbert's epsilon operator.

```
congruent(S)(a, b): bool =                                         11
  app?(a) AND
  app?(b) AND
  func(a) = func(b) AND
  (FORALL (i:arity(func(a))):
    norm(S)(args(a)(i)) = norm(S)(args(b)(i)))
```

A solution set is *congruence-closed* when the right-hand sides corresponding to any pair of congruent left-hand sides are identical.

```
congruence_closed(S): bool =                                       12
  (FORALL (a,b:(dom(S))): congruent(S)(a, b) IMPLIES
                          apply(S)(a) = apply(S)(b))
```

The solution set that forms the "state" of the algorithm is typed to satisfy the invariants given by the predicate `invariants(S)`. These invariants assert that the left-hand sides of equalities in the solution set S must be variables or uninterpreted terms, the uninterpreted subterms of any equality S must in the domain of S, and any right-hand side term must be canonical, and $S(a)$ and $norm(S)(a)$ must coincide for any $a \in dom(S)$, among other conditions. The predicate `invariant(S)` is used to define a type `above_tinvariants(n)` which ensures that the state is a record `r` consisting of an index `r‘n` and a solution set `r‘s` which is either `bottom` or `up(S)`, where `S` is well-typed in `r‘n` and satisfies `invariants(S)`.

The Main Procedure. The congruence closure operation `cc(r)` successively merges the right-hand sides corresponding to *chosen* congruent pairs of left-hand side terms in the solution set `r‘s`. The operation `merge(m, aa, S)` (used in $\boxed{13}$ and $\boxed{14}$) computes `solv(m, aa)` as a record `r`, returning `bottom` if `r‘s` is `bottom`, and the record `(# n := r‘n, s := S o down(r‘s)#)`, otherwise, where `down(up(R))` is `R`. The return type of `cc` ensures that `cc(r)‘s` is `bottom` when `r‘s` is `bottom` and the `cc(r)‘s` satisfies the invariants spelled out above when it is different from `bottom`. The termination of `cc`, a significant step in the proof, is established by showing that the number of equivalence classes of uninterpreted terms in the domain of `r‘s` decreases with each recursive call. The invariants on the solution set play a crucial role in proving termination.

```
cc(r): RECURSIVE {s : above_tinvariants(r‘n) | bottom?(r‘s)        13
                                       IMPLIES bottom?(s‘s)} =
 CASES r‘s OF
  bottom: tbottom,
  up(T) : IF (NOT congruence_closed(T))
           THEN cc(merge(r‘n, apply(T)(choose(congruent_pair?(T))), T))
           ELSE r
          ENDIF
 ENDCASES
MEASURE cc_rank(r)
```

The `assert(r, aa)` operation places `aa` in canonical form as `aa’`, then expands `r‘s` (if `r‘s` is `up(T)`) with dummy identities for the new subterms in `aa’` as expand(T, aa')`. It then merges `aa'` into this expanded solution set and applies congruence-closure `cc` to the result.

```
assert((r:{r:tinvariants | up?(r`s) IMPLIES                            14
                          congruence_closed(down(r`s))}),
       (aa:typed_equality(r`n))):
   {s:above_tinvariants(r`n) | up?(s`s) IMPLIES
                                congruence_closed(down(s`s))} =
  CASES r`s OF
    bottom: tbottom,
    up(T): cc(merge(r`n, can(T)(aa), expand(T, can(T)(aa))))
  ENDCASES
```

Finally, `process(m, S)` returns a record consisting of a number `n` and a well-typed solution in `n` which may be `bottom`. The type of `process(m, S)` ensures that any solution returned is congruence-closed.

```
process(m, (S:typed_eqlist(m))): RECURSIVE                             15
   {r:above_tinvariants(m) | up?(r`s) IMPLIES
                             congruence_closed(down(r`s))} =
 CASES S OF
  null       : (# n := m, s := up(null)#),
  cons(aa, T): IF up?(process(m, T)`s)
               THEN assert(process(m, T), aa)
               ELSE tbottom
               ENDIF
 ENDCASES
MEASURE length(S)
```

The type and termination proof obligations generated by the PVS typechecker corresponding to the subtype constraints and measures given with the definitions of `process`, `cc`, and other related definitions, ensure the well-typedness and termination of `process`.

4 Verifying Shostak's Algorithm in PVS

The algorithm verifies a sequent $T \vdash a = b$ by computing $S = \mathit{process}(T)$. The sequent is considered valid if either $S = \bot$ or $\mathit{can}(S)(a) \equiv \mathit{can}(S)(b)$. For the soundness of the procedure is established relative to a proof system whose inference rules characterize when a sequent $T \vdash a = b$ is derivable. We prove that the following are equivalent:

1. If $\mathit{process}(T) = S$, then $S = \bot$ or $\mathit{can}(S)(a) \equiv \mathit{can}(S)(b)$.
2. $T \vdash a = b$ is derivable.
3. $T \vdash a = b$ is σ-valid, i.e., valid in all σ-models.

The implication from (1) to (2) is the soundness argument. The implication from (2) to (3) validates the soundness of the proof system with respect to

σ-models. The implication from (3) to (1) establishes the completeness of the decision procedure.

For verifying soundness, we first formally define the class of provable sequents by means of an inductive definition of a predicate `has_proof?(m, T, aa)` for an index `m`, a list of equalities `T`, and an equality `aa`.

```
has_proof?(m,                                                              16
          (T:typed_eqlist(m)),
          (aa:typed_equality(m))): INDUCTIVE bool =
  member(aa, T) OR                                         % Axiom
  lhs(aa) = rhs(aa) OR                                     % Reflexivity
  has_proof?(m, T, eq(rhs(aa), lhs(aa))) OR                % Symmetry
  (EXISTS (a:typed(m)):                                    % Transitivity
    has_proof?(m, T, eq(lhs(aa), a)) AND
    has_proof?(m, T, eq(a, rhs(aa)))) OR
  (LET a = lhs(aa), b = rhs(aa) IN                         % Congruency
    app?(a) AND app?(b) AND
    func(a) = func(b) AND
    (FORALL (i:arity(func(a))):
      has_proof?(m, T, eq(args(a)(i), args(b)(i))))) OR
  (rhs(aa) = sig(lhs(aa))) OR                              % Canonization
  (EXISTS (bb:typed_equality(m)),                          % Solve
          (n:upfrom(m)), (S:typed_eqlist(n)):
    solve(m, bb, S) AND
    has_proof?(m, T, bb) AND
    has_proof?(n, append(T, S), aa)) OR
  (EXISTS (bb:typed_equality(m)):                          % Contradiction
    unsatisfiable(bb) AND
    has_proof?(m, T, bb))
```

The proof soundness theorem below captures the implication from (2) to (3) above. It asserts that any provable sequent is σ-valid since the variable `M` is declared to range over σ-models. It can be proved by the induction scheme generated by the inductive definition of `has_proof?`.

```
proof_soundness: LEMMA                                                     17
  (FORALL m, (T:typed_eqlist(m)), (aa:typed_equality(m)):
    has_proof?(m, T, aa) IMPLIES
      (FORALL M, (rho:assign(M)): satisfies(M, rho)(T, aa)))
```

The following two theorems correspond to the implication between (1) and (2) above. These theorems capture the respective cases of soundness when `process(m, S)` returns a valid solution or a `bottom` value.

```
soundness_1: THEOREM                                                       18
  (FORALL m, (S:typed_eqlist(m)), (a, b:typed(m)):
    up?(process(m, S)`s) AND
    can(down(process(m, S)`s))(a) = can(down(process(m, S)`s))(b)
   IMPLIES has_proof?(m, S, eq(a, b)))
```

```
 soundness_2: THEOREM                                                   19
   (FORALL m, (S:typed_eqlist(m)), (aa:typed_equality(m)):
     bottom?(process(m, S)‘s) IMPLIES
       has_proof?(m, S, aa))
```

Completeness is proved by constructing a canonical σ-model M_R and assignment ρ_R, where $R = \mathit{process}(T) \neq \bot$. The bulk of the proof involves showing that this construction does in fact yield a σ-model satisfying the equalities in T. A crucial property for demonstrating this is confluence which asserts that `can(S)(a) = norm(S)(a)` when `S` is congruence-closed and the uninterpreted terms of `a` are included in `dom(S)`.

```
confluence: LEMMA                                                       20
    invariants(S) AND
    congruence_closed(S) AND
    subset?(U(subterm(a)), dom(S)) IMPLIES
      can(S)(a) = norm(S)(a)
```

Completeness is then proved as the theorem below which formalizes the implication from (2) to (1) above, but it is verified via proof soundness and (3). The theorem states that when the sequent $S \vdash a = b$ is derivable, then either $\mathit{process}(S) = \bot$ or $\mathit{process}(S) = T$ and $\mathit{can}(T)(a) = \mathit{can}(T)(b)$.

```
 completeness: LEMMA                                                    21
   (FORALL m, (S:typed_eqlist(m)), T, (aa:typed_equality(m)):
     up?(process(m, S)‘s) AND
     down(process(m, S)‘s) = T AND
     has_proof?(m, S, aa) IMPLIES
       can(T)(lhs(aa)) = can(T)(rhs(aa)))
```

5 Concluding Observations

Both the formalization and the verification closely follow the informal presentation **RS** [RS01]. There were some areas where **RS** was found to be inadequate or incorrect and where PVS itself was deficient.[6]

RS is terse about the introduction of fresh variables by the *solve* operation. These variables must be fresh with respect to the entire execution of the algorithm or the construction of a proof. Proof transformations like weakening and cut require the variables generated by *solve* to be invariant with respect to a certain kind of renaming.[7] The bookkeeping involved in tracking the well-formedness of terms and equalities up to a given index, occupy a substantial

[6] One minor problem was already noticed prior to this verification attempt. Several of the lemmas in the informal proof regarding the composition of solutions were qualified with the condition that $R \cup S$ be functional, where the appropriate condition is that $R \circ S$ must be functional. This was immaterial for the verification since the definition of composition is in terms of lists and not sets.

[7] A similar renaming problem arises with alpha-renaming in the lambda-calculus and *eigenvariables* in sequent proofs, but the renaming issue is far more complicated

fraction of the effort in both the formalization and proof. PVS has a judgement mechanism that records certain typing relations for use in the typechecker, but we were unable to use it for demonstrating that an expression well-typed in n is also well-typed in any index above n.

Quantification over types, needed to define semantic validity, is not expressible in PVS. We instead restricted the semantic domains to subtypes of the type of terms since any model for terms and equalities is essentially characterized by a partition of the term universe into equivalence classes.

A monotonicity lemma is stated in the informal proof (Lemma 3.12) as: *If* $R \cup S$ *is functional, then if* $R(a) \equiv R(b)$*, then* $(R \circ S)(a) \equiv (R \circ S)(b)$*, for any* a *and* b*.* In addition to the above-mentioned correction to the antecedent, this lemma only holds when a and b are in $dom(R)$. Fortunately, only the weak form of this lemma is actually used.

In the **RS** proof of Lemma 5.11, it is claimed that *it can also be shown that* $can(S'^{+})(a) \equiv can(S')(a)$*, and similarly for* b*.* This claim asserts that padding the solution set S' with reflexivity equalities on the subterms from $can(S')(a)$, does not affect the value of $can(S')(a)$. The claim is in fact valid, but the proof is not all that obvious.

Despite the flaws identified above, the **RS** proofs held up quite well to the rigors of formal scrutiny. We were actually operating from a draft document that contained proofs of lemmas that were given without proof in the published version. Once the formalization challenges were overcome, it was possible to make steady progress in the mechanical verification of the proofs. The procedure as we have defined it is not executable since it uses a choice operator. Further work is needed to derive efficiently executable versions of the verified algorithm while preserving its correctness.

The formalization and proof occupied four months of work with PVS carried out entirely by the first author.[8] The proof involves 68 theories, 120 definitions, 192 TCCs (typing and termination proof obligations), 594 lemmas, and the proof checking time is 2,265 seconds on a 1-Gigahertz Pentium 3. There are roughly 6,200 tokens in the detailed informal presentation as measured by a word count of the text file generated from the LaTeX input. There are approximately 13,000 tokens in the PVS specification, and over 25,000 tokens in the PVS proofs. The proof is highly interactive. We are currently working on improving the degree of mechanization in various ways. The level of effort indicates that the certification of complex decision procedures remains a tough challenge.

References

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

here. The variable indices affect the type and the well-typedness of equalities and proofs so that renaming is not a local operation.

[8] The first author already had prior experience with PVS having used it for two substantial proof developments[FM01b,FM01a].

[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.

[CLS96] David Cyrluk, Patrick Lincoln, and N. Shankar. On Shostak's decision procedure for combinations of theories. In M. A. McRobbie and J. K. Slaney, editors, *Automated Deduction—CADE-13*, volume 1104 of *Lecture Notes in Artificial Intelligence*, pages 463–477, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[FM01a] J. Ford and I. A. Mason. Establishing a General Context Lemma in PVS. In *Proceedings of the 2nd Australasian Workshop on Computational Logic, AWCL'01* , 2001. submitted.

[FM01b] J. Ford and I. A. Mason. Operational techniques in PVS—a preliminary evaluation. In *Proceedings of the Australasian Theory Symposium, CATS '01*, Gold Coast, Queensland, Australia, January–February 2001.

[FORS01] J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: Integrated Canonization and Solving. In G. Berry, H. Comon, and A. Finkel, editors, *Computer-Aided Verification, CAV '2001*, volume 2102 of *Lecture Notes in Computer Science*, pages 246–249, Paris, France, July 2001. Springer-Verlag.

[GMW79] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[NO79] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[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.

[Sha85] N. Shankar. Towards mechanical metamathematics. *Journal of Automated Reasoning*, 1(4):407–434, 1985.

[Sha99] N. Shankar. Efficiently executing PVS. Project report, Computer Science Laboratory, SRI International, Menlo Park, CA, November 1999. Available at `http://www.csl.sri.com/shankar/PVSeval.ps.gz`.

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

[Thé98] Laurent Théry. A certified version of Buchberger's algorithm. In H. Kirchner and C. Kirchner, editors, *Proceedings of CADE-15*, number 1421 in Lecture Notes in Artificial Intelligence, pages 349–364, Berlin, Germany, July 1998. Springer-Verlag.

[VGL00] Kumar Neeraj Verma and Jean Goubault-Larrecq. Reflecting BDDs in Coq. Technical Report 3859, INRIA, Rocquencourt, France, January 2000.

[vHPPR98] Friedrich W. von Henke, Stephan Pfab, Holger Pfeifer, and Harald Rueß. Case studies in meta-level theorem proving. In Jim Grundy and Malcolm Newey, editors, *Proc. Intl. Conf. on Theorem Proving in Higher Order Logics*, number 1479 in Lecture Notes in Computer Science, pages 461–478. Springer-Verlag, September 1998.

A Introduction to PVS

We give a very brief introduction to the PVS language and proof checker. PVS specifications are a collection of theories. A theory can have type or individual parameters that are instantiated when the theory is imported within another theory. A parameterized theory can include constraining assumptions on the parameters. The instances of these assumptions corresponding to the actual parameters are generated as proof obligations when a theory instance is imported.

A theory is a list of declarations of types, constants, and formulas. The expression language of PVS is based on simply typed higher-order logic extended with predicate subtypes, dependent types, and recursive datatypes. PVS types consist of the *base* types `bool` and `real`, and *compound* types constructed as tuples, as in $[\texttt{bool}, \texttt{real}]$, records, as in $[\#\texttt{flag} : \texttt{bool}, \texttt{length} : \texttt{real}\#]$, or function types of the form $[A{\rightarrow}B]$. Predicates over a type A are of type $[A{\rightarrow}\texttt{bool}]$.

Predicate subtypes are a distinctive feature of the PVS higher-order logic. Given a predicate p over A, $\{x : A \,|\, p(x)\}$ (or, (p)) is a predicate subtype of A consisting of those elements of A satisfying p. The type `nzreal` of nonzero real can be defined as $\{\texttt{x} : \texttt{real} \,|\, \texttt{x /= 0}\}$. The type `nat` of natural numbers is a predicate subtype of the type `int` of integers, which in turn is a subtype of the subtype `rat` (of `real`) of rational numbers. Subranges can also be defined as predicate subtypes, and arrays can be typed as functions with subranges as domains, e.g., $[\texttt{below(N)}{\rightarrow}A]$. The PVS typechecker generates proof obligations (called TCCs) corresponding to predicate subtype constraints. Out-of-bounds array accesses generate unprovable TCCs.

Dependent versions of tuple, record, and function types can be constructed by introducing dependencies between different components of the type through predicates. Dependent typing can be used to define a finite sequence (of arbitrary length) as a dependent record consisting of a length and an array of the given length $[\#\texttt{length} : \texttt{nat}, \texttt{seq} : [\texttt{below(length)}{\rightarrow}T]\#]$.

PVS expressions include variables x, constants c, applications $f(a)$, and abstractions `LAMBDA` $(x : T) :\ a$, conditionals `IF` a_1 `THEN` a_2 `ELSE` a_3 `ENDIF`, tuple expressions $(a_1, \ldots, a_n)$, tuple projections $a`i$, record expressions $(\#l_1 \texttt{:=} a_1, \ldots \#)$, record projections $a\texttt{`}l$, and (tuple, record, and function) updates $e[a := v]$.

The definition of a recursive datatype can be illustrated with the list type built from the constructors `cons` and `null`. Theories containing the relevant axioms, induction schemes, and useful datatype operations are generated from the datatype declaration.

```
list [T: TYPE]: DATATYPE                                         1
 BEGIN
  null: null?
  cons (car: T, cdr:list):cons?
 END list
```