Proc. Intl. Conf. on Theorem Proving in Higher Order Logics (TPHOLs'98).
Springer LNCS, Vol 1479, pp. 461–478, Sept. 1998. © Springer-Verlag.

Case Studies in Meta-Level Theorem Proving*

Friedrich W. von Henke, Stephan Pfab, Holger Pfeifer, Harald Rueß**

Universität Ulm
Fakultät für Informatik
D-89069 Ulm/Donau
{vhenke,pfab,pfeifer}@informatik.uni-ulm.de
ruess@csl.sri.com

Abstract. We describe an extension of the Pvs system that provides a reasonably efficient and practical notion of reflection and thus allows for soundly adding formalized and verified new proof procedures. These proof procedures work on representations of a part of the underlying logic and their correctness is expressed at the object level using a computational reflection function. The implementation of the Pvs system has been extended with an efficient evaluation mechanism, since the practicality of the approach heavily depends on careful engineering of the core system, including efficient normalization of functional expressions. We exemplify the process of applying meta-level proof procedures with a detailed description of the encoding of cancellation in commutative monoids and of the kernel of a BDD package.

1 Introduction

Many of the proof techniques we discover in the process of proving theorems with mechanized support are potentially useful for improving the overall performance of the underlying proving system. Thus a mechanism that allows users of such a system to incorporate specialized or even new automated proof techniques into the system would be helpful. It is crucial, however, that system modifications do not alter soundness of the system at hand, since uncontrolled insertion of unverified new proof procedures can be entirely fatal to the usefulness of such a system. A mechanized theorem prover is said to be *soundly extensible* if a meta language can be employed to extend reasoning capabilities and if these extensions can be shown to be correct with respect to some correctness criterion. There are basically two different mechanisms for soundly extending theorem provers, namely tactics and reflective systems.

Tactics have first been introduced in Edinburgh Lcf [13], and since then successfully applied to extend mechanical theorem provers in a sound way [10, 13,

* This work has been partially supported by the Deutsche Forschungsgemeinschaft (DFG) project *Verifix* and by the Deutscher Akademischer Austauschdienst (DAAD).

** Current affiliation: SRI International

12, 20, 21]. A *tactic* is a function written in a procedural meta-language (mostly some version of ML) that splits a goal into a set of subgoals, and provides a *justification* to ensure soundness of each tactic invocation. This reduction step corresponds to backwards application of rules in the sense that the given goal may be inferred by basic rules from the subgoals. Tactics and strategies built from them do not have to be proved correct since a *safety kernel* of the basic tactic mechanism assures that these proof search procedures may fail but will never produce incorrect proofs. In effect, each successful tactic invocation is expanded into a proof of the original goal from the proofs of all computed subgoals, using the primitive inference rules of the underlying logic.

It has been observed in the past, however, that tactics may not be the most appropriate technique for constructing proofs of many facts that are expressible as meta-level statements [1, 3, 4, 16]. An example may help to illustrate this. Consider proving the equality of two terms that contain some associative commutative operator. A tactic that solves such a task must chain together appropriate instances of lemmas and rules. Using the analogy of tactic invocation with application of a meta-lemma, a tactic re-computes a proof object in terms of primitive rules for every instance of this *meta-lemma*, instead of taking the natural approach of simply instantiating its proof. This results in sometimes rather messy construction of tactics and obliges the tactic writer always to concern himself with generating proofs in terms of basic inference rules; this can increase the intellectual effort involved in constructing theorem-proving procedures. Moreover, all programming knowledge is implicitly contained in the tactic code. This complicates maintenance and modification of deductive systems based on the tactics approach; furthermore, even though the user of a tactic may know, or be able to verify, that a given tactic will construct a correct proof of a given goal, the implementation must still execute the tactic and verify the result. Finally, it may be hard to achieve the *efficiency* required to complete large verifications at reasonable cost using the LCF approach [6].

An alternative to purely procedural tactics is to encode theorem-proving methods as verifiable meta-functions in a *self-referential* system. Such a system is able to refer to (parts of) itself; it consists of a base system, the so-called *object level*, an internal representation of (parts of) itself, the *meta level*, and a *reflection* mechanism that expresses the relationship between the object level and its corresponding meta-level encoding. In such a framework it is possible to make formal statements about the behavior of meta-functions and verify their correctness. The main advantage of this approach lies in the fact that meta-theoretic results, once proven, can be used without further justification. Consider again the associative-commutative example mentioned above. A natural approach is to view the left-hand and right-hand sides of an equation as trees and to check whether their fringes are permutations of each other.

There are basically two different paradigms of encoding theorem-proving capabilities as verifiable meta-functions. *Computational reflection* uses an interpreter to associate meta-level representations with the values they denote [8, 15, 23]. In this approach meta-level representations are used to make the syntactic

structures of object-level entities amenable to inspection and manipulation. Consequently, computational reflection frameworks do not permit statements about provability and the existence of proofs. The other approach can be termed *deductive reflection*. The idea here is to encode a meta-level *provability predicate*, say Pr, for a certain subset of the object-level theory. This predicate Pr is used to reduce the provability of a goal ϕ to the provability of $Pr('\phi')$; here, $'\phi'$ is the meta-level representation of the object-level entity ϕ. In a *deductive reflection* system the transitions between the meta-level and the object level are established by *deductive reflection rules*. These rules can usually be shown to be admissible in the object-level calculus by proving the *correctness* and *completeness* of the meta-level encoding, and, consequently, the extended reflective system is a *conservative extension* of the base-level system. While such a conservative approach can not give a *reflection principle* in the logicians' sense,[1] it allows a single system to simulate a large amount of meta-reasoning and gives the assurance that the resulting system remains consistent.

Sound extension by means of a self-referential system has several pragmatic advantages over tactics. First, theorem-proving procedures are ordinary programs of the object language that examine and manipulate representations of object-language expressions. Second, verified meta-programs do not have to deal explicitly with justifications. Instead, the procedures are justified by separate correctness arguments. Correctness is established *once and for all* and its proof "reused" every time the procedure is called. In contrast to LCF tactics, proofs are not re-computed but merely instantiated. This is especially important, since formal proofs of interesting developments in both mathematics and computer science tend to be rather large objects. Last but not least, for the very same reasons as for any other piece of software, formalized proof procedures permit building up and modifying libraries in a controlled and mathematical way, and formalized and (mechanically) proved properties of the procedures help in understanding their effects.

Despite these apparent advantages of verified meta-programs over tactics, reflective systems have had almost no impact on the design of theorem proving systems and sound extension by means of reflection is not yet ready to push back the boundaries of what is feasible in mechanized theorem proving [14]. There are two main reasons for this failure. First, although the underlying techniques of reflection needed to guarantee sound extension are well understood it is a towering (and rather thankless) task to build a reflective system: starting from a small kernel system enriched with some notion of reflection, a plethora of verified proof procedures must be added in the bootstrapping process in order to provide a useful initial system. Second, the practicality of reflective systems relies heavily on a notion of fast execution of the encoded proof procedures, since the basic idea of reflection is to replace deduction at the object level by meta-level computation. Unfortunately, many existing theorem proving systems

[1] The logicians' use of *reflection* is a way of extending theories by adding axioms and rules which are not derivable in a conservative extension of the system under consideration [17, 25].

— like HOL, COQ, LEGO, PVS, all of which are based on some sort of type theory — do not support efficient evaluation, although their underlying calculi clearly include the notions of evaluation and normalization. Consequently, a successful reflective system can be built only by taking the "programming" part of theorem proving systems serious and treating it as full-fledged programming language.

The purpose of this paper is to demonstrate by examples that existing theorem provers can be extended to provide a useful and practical notion of reflection in order to soundly extend theorem proving capabilities. Our starting point is the PVS system. We first extend the implementation of this system by an efficient evaluation mechanism to get *Lisp*-like speed of function evaluation. This extended system forms a suitable basis for verifying formalized proof procedures and applying them in the proof process. Proof procedures work on representations of a—rather small—part of the underlying logic and correctness thereof is expressed at the object level using a computational reflection function. Typically, showing correctness of such a meta function involves showing that the function is equality-preserving or that the target expression is a refinement of the source expression. We distinguish three phases of applying formalized proof functions. In the *reification* phase, PVS strategies are used to compute representations of some parts of the current proof goal; the subsequent *normalization* and *reflection* phases rely exclusively on evaluation. We exemplify this process with a detailed description of the encoding of cancellation in commutative monoids and applications of the resulting proof procedure. In a similar manner, the kernel of a BDD package has been encoded as a meta function that can be used as a decision procedure for propositional logic.

The remainder of the paper is structured as follows. Section 2 provides an informal introduction to the PVS system. The implementation of an evaluation mechanism for functional expressions of PVS is described in Sect. 3. Section 4 discusses in detail the encoding, verification, and application of a proof procedure for cancelling in Abelian groups, while Sect. 5 gives a tour through our encodings of a BDD package. Section 6 closes with a comparison to related work and final remarks.

2 Formal Background

The PVS system combines an expressive specification language with an interactive proof checker; see [19] for an overview. This section provides a brief description of the PVS language and prover, and introduces some of the concepts needed in this paper. More details can be found in [11].

The PVS specification language builds on classical typed higher-order logic with the usual base types, `bool`, `nat`, `int`, among others, the function type constructor `[D → R]`, and the product type constructor `[A,B]`. The type system of PVS is augmented with *dependent types* and *abstract data types*. In PVS, predicates over some type `A` are, as usual, boolean-valued functions on `A`, and `pred[A]` is an abbreviation for the function type `[A → bool]`. A distinctive feature of the PVS specification language are *predicate subtypes*: the subtype `{x:D | P(x)}`

consists of exactly those elements of type `D` satisfying predicate `P`. The expression `(P)` is an abbreviation for the predicate subtype {`x:D | P(x)`}. Predicate subtypes are used for explicitly constraining domains and ranges of operations in a specification and for defining partial functions. In general, type-checking with predicate subtypes is undecidable; the type-checker generates proof obligations, so-called *type correctness conditions* (TCCs), if satisfaction of the restricting predicate cannot immediately be derived. A large number of TCCs are discharged by specialized proof strategies; a Pvs expression is not considered to be fully type-checked until all generated TCCs have been proved. Pvs specifications are packaged as *theories* that can be parametric in types and constants.

Consider, for example, the theory `alist` in 1 for implementing association lists.[2] This theory is parameterized with respect to two nonempty types `D` and `R` and defines an abstract datatype `maybe` with two constructors `yes` and `no`.

```
alist [D,R : TYPE+] : THEORY                                        1
BEGIN
  maybe : DATATYPE
  BEGIN
    yes(arg:R) : yes?
    no         : no?
  END maybe

  alist : TYPE = [D→maybe]

  v : VAR D;  c : VAR R;  a : VAR alist

  empty         : alist = λv: no
  update(a,v,c) : alist = a WITH [v := yes(c)]
  insert(a,v,c) : alist =
    CASES a(v) OF
      yes(y) : a,
      no     : update(a,v,c)
    ENDCASES
END alist
```

Furthermore, `alist` is defined to be the type of functions with domain `D` and codomain `maybe`, the clause `v : VAR D` declares the type of variable `v` to be `D`, the expression `a WITH [v := yes(c)]` denotes the update of function `a` at position `v` with value `yes(c)`, and the insert operation is defined through a simple case split such that an alist is updated at position `v` unless a binding for `v` already exists.

A theory can use the definitions and theorems of another theory by *importing* it. Parameterized theories can be imported in either of two ways: as an instance of the theory by providing actual values for the formal parameters, or uninstantiated. In the latter case, all possible instantiations of the imported theory may

[2] To increase the readability of PVS specifications the syntax has liberally been modified by replacing some ASCII codings with a more familiar mathematical notation.

be used, and ambiguities can be resolved by qualification. A built-in *prelude* and loadable *libraries* provide standard specifications and proved facts for a large number of theories.

Proofs in PVS are presented in a sequent-calculus style. The atomic commands of the PVS prover include induction, quantifier instantiation, conditional rewriting, simplification using arithmetic and equality decision procedures and type information, and propositional simplification. PVS has an LCF-like strategy language for combining inference steps into more powerful proof strategies. The strategy combinator `(then <strat-list>)`, for example, successively applies a list of strategies, while `(spread <strat> <strat-list>)` applies the strategy `<strat>` to the current goal and then pairs the strategies in `<strat-list>` with the subgoals. Furthermore, the strategy language includes the constructs `if` for branching and `let` for binding variables in the body of a strategy to the results of Lisp computations. The most comprehensive strategies manage to generate many proofs fully automatically.

3 Efficient Evaluation

A subset of the PVS specification language can be considered as an executable, functional programming language; it includes all kinds of operations on expressions of basic types, several forms of conditionals (`IF`, `CASES`, `COND`, `TABLE`), total recursive functions (by means of measure recursion), and homomorphic functionals [27] on abstract datatypes. Although the PVS prover provides basic strategies like `beta`—which is based on an (inefficient) implementation of a substitution calculus—for executing these functional programs, it has been observed many times that efficient evaluation has the potential of speeding up and automating many proofs, e.g. simulation proofs. For this reason we have extended PVS to considerably improve on the speed of computing normal forms. Hereby, we use the idea of *inverse evaluation* by Berger and Schwichtenberg [5] and compute a normal form for a PVS expression in three successive steps: first, an expression is translated into the corresponding Lisp program, second, the Lisp program is executed using Lisp's evaluation function *eval*, and, finally, the result of Lisp evaluation is translated back to a corresponding PVS expression. In this way, we obtain Lisp-like execution speed for normalizing PVS expressions, and we may readily use Lisp compilers to produce efficient machine code for PVS functions. In addition to normal programming languages constructs, the evaluator also handles uninterpreted constant and function symbols; conditions involving uninterpreted symbols may be decided by calling the PVS prover; these features, however, are not used in the sequel. For the purpose of this paper it suffices to presuppose an efficient symbolic evaluator for normalizing PVS expression that faithfully implements the reduction relation of the underlying PVS logic (including abstract datatypes); technical details can be found in [22].

As interface to the efficient evaluator described above, we have extended the PVS prover with a new basic strategy `(NORM)`. This strategy takes as argument a PVS expression `e` and replaces it with its corresponding normalized expression.

4 Cancellation in Commutative Monoids

Cancellation in the particular structure $(\mathbb{Z}, +, 0)$ has already been considered in the context of COMPUTATIONAL LOGIC [7] by Boyer and Moore [8]. They develop a meta function *cancel* that reduces, for example, the equation

$$(a + i) + (b + k) = j + (k + (i + x))$$

to

$$a + b = j + x$$

The main step in the *cancel* algorithm is to compute the fringes of both sides of the source equation, to delete all common terms, and to compute the simplified target equation by means of a meaning function that associates syntactic representations of equations with an equation. Boyer and Moore's *cancel* function is restricted to the particular structure $(\mathbb{Z}, +, 0)$, but the abstraction mechanisms of PVS permit generalizing their development to arbitrary commutative monoids satisfying the left cancellation property.

The development of the cancel function is carried out in a generic theory `cancel` (see 2) that takes the components of a commutative monoid (`A`, $\bullet$, `e`) as arguments. Here, $\bullet$ is a binary operator on `A` and `e` is the identity; the semantic constraints on the theory parameters are stated in the assumption part of the theory.

```
cancel [A:TYPE, •:[A,A→A], e:A] : THEORY                    2
BEGIN
  ASSUMING
    A: ASSUMPTION associative?(•)
    C: ASSUMPTION commutative?(•)
    I: ASSUMPTION left_identity?(•)(e)
    L: ASSUMPTION ∀(x,y,z:A): x•y = x•z ⇒ y = z
  ENDASSUMING
```

The first step is to define types `trm` and `equality` (see 3) for representing terms built up from the binary operator $\bullet$ and equations, respectively. These (meta-level) representations simply make the internal structure of equations on terms of type `A` explicit and permit inspecting and manipulating the structure of equalities on expressions of type `A`.

```
vars: TYPE = nat        % Infinite supply of meta variables   3

trm : DATATYPE
BEGIN
  mk_neutral               : neutral?
  mk_var(name : vars)      : var?
  mk_app(left,right : trm) : app?
END trm

equality : TYPE = [# lhs, rhs : trm #]
```

In the next step, we encode denotation functions for computing object-level terms from corresponding representations of types `equality` and `trm`. These functions require, besides a representation, a context that associates elements of type `vars` with terms of the object level. Here, a context `c` of a term `t` is represented as a function of type `alist: TYPE = [vars → A]`, with the additional requirement that all variables occurring in `t` also occur in `c`.

4
```
x : VAR vars;  c  : VAR alist;     v : VAR A;
t : VAR trm;   eq : VAR equality;

contains_all_vars?(t)(c) : bool =
  ∀x: occurs?(x,t) ⇒ ∃v: c(x) = yes(v)

TrmContext(t) : TYPE = (contains_all_vars?(t))

contains_all_vars?(eq)(c) : bool =   % Note the use of overloading here
  contains_all_vars?(lhs(eq))(c)∧contains_all_vars?(rhs(eq))(c)

EqContext(eq) : TYPE = (contains_all_vars?(eq))
```

Now it is a simple matter to define the (overloaded) functions ⟦`t,c`⟧ and ⟦`eq,c`⟧ to compute denotations of representations of terms and equations, respectively, with respect to a context `c`.

5
```
⟦⟧(t:trm,c:TrmContext(t)) : RECURSIVE A =        % Reflection
  CASES t OF
    mk_neutral     : e,
    mk_var(x)      : arg(c(x)),
    mk_app(t₁,t₂)  : ⟦t₁,c⟧ • ⟦t₂,c⟧
  ENDCASES MEASURE t BY ≪

⟦⟧(eq:equality,c:EqContext(eq)) : bool =
  ⟦lhs(eq),c⟧ = ⟦rhs(eq),c⟧
```

Recall that cancellation works by computing fringes from terms. These fringes can be represented conveniently by bags. We omit here the realization of bags in terms of lists. In the following, we use $\mathtt{b}_1 \cup \mathtt{b}_2$ to denote the union of two bags, $.\setminus.$ for the difference and $.\cap.$ for the intersection of bags; the definitions of the functions are omitted. The functions `fringe` and `tree` in 6 are simple conversions between terms and bags.

```
IMPORTING bags[vars]                                                    6

fringe(t:trm) : RECURSIVE Bag =
  CASES t OF
    mk_neutral    : null,
    mk_var(x)     : cons(x,null),
    mk_app(t1,t2) : fringe(t1)∪fringe(t2)
  ENDCASES MEASURE t BY ≪

tree(b:Bag) : RECURSIVE trm =
  CASES b OF
    null      : mk_neutral,
    cons(x,l) : mk_app(mk_var(x),tree(l))
  ENDCASES MEASURE b BY ≪
```

Now we have collected all the ingredients to encode the `cancel` function in 7. Cancellation works by computing two bags of argument terms of $\bullet$ from the left hand side and the right hand side of the equality under consideration. Now, common terms are cancelled in both bags, simplified terms are computed from these bags, and the target equality is formed from these simplified terms.

```
cancel(eq:equality) : equality =                                        7
    LET b1      = fringe(lhs(eq)),
        b2      = fringe(rhs(eq)),
        common  = b1∩b2,
        new_lhs = b1∖common,
        new_rhs = b2∖common
    IN
       (# lhs := tree(new_lhs), rhs := tree(new_rhs) #)
```

Theorem `preserves_eq` in 8 states that the function `cancel` preserves equality, i.e. the denotations of some source equality `eq` and the corresponding target equality `cancel(eq)` are equivalent.

```
preserves_eq: THEOREM                                                   8
  ∀(eq:equality,c:EqContext(eq)): ⟦eq,c⟧ = ⟦cancel(eq),c⟧

END cancel
```

The proof of the correctness theorem follows closely the one described by Boyer and Moore [8]. The main step is accomplished using lemma `meaning_difference` in 9, which essentially states that splitting a term into two arbitrary parts preserves the denotation.

```
meaning_difference : LEMMA                                              9
  ∀(b1,b2:Bag, c:TrmContext(tree(b2))):
    subbag?(b1,b2) ⇒ ⟦tree(b2),c⟧ = ⟦tree(b1),c⟧ • ⟦tree(b2∖b1),c⟧
```

Theorem `preserves_eq` is used to define the strategy `cancel` in Fig. 1. This strategy takes three strings `typ`, `op`, and `e`, selects the formula `fml` of the current proof sequent (line 3), and type checks the argument strings (lines 4 and

5). If these operations are successful and `fml` is indeed an equality (line 6) then the call to function `quote-eqn` (see Appendix A) yields a representation `rep` for `fml` of type `equality` and an association list for all argument terms of the operator `op1` in `fml` (lines 7–9). This information is used to instantiate theorem `preserves_eq` (line 10), followed by a call to the normalization strategy `NORM` (see Sect. 3) and term rewriting with this normalized equality. Finally, the newly introduced equation is hidden from the current sequent (lines 11-13). The PVS prover generates a type correctness condition for the instantiation of `preserves_eq`, since the association list computed by `quote-eqn` must be a context for `rep`. This TCC, however, is easily proved by unfolding the definitions involved (lines 14 and 15).

```
 1: (defstep cancel (typ op e &optional (fnum 1))
 2:   (let ((sforms (s-forms (current-goal *ps*)))
 3:         (fml   (formula (car (select-seq sforms (list fnum)))))
 4:         (op1   (typecheck (pc-parse op 'expr)))
 5:         (e1    (typecheck (pc-parse e 'expr))))
 6:     (if (not (equality? fml)) (skip)
 7:       (let ((rep-alist  (quote-eqn fml op1 e1))
 8:             (rep        (first rep-alist))
 9:             (alist      (second rep-alist)))
10:         (spread (lemma "preserves_eq" :subst ("c" alist "eq" rep))
11:                 ((then (NORM -1)
12:                        (replace -1)
13:                        (hide -1))
14:                  (then (auto-rewrite "update" "insert" ...)
15:                        (reduce))))))) ...)
```

Fig. 1. Defined strategy for cancellation.

Consider, for example, simplification of the following proof goal, where `f` is some uninterpreted function, by means of the cancellation strategy.

```
  |-------
{1}    x + (f(y) + z) = (z + f(u)) + (z + f(y))
```

Applying the strategy `(cancel "real" "+" "0")` yields the simplified sequent

```
  |-------
{1}    x = f(u) + z
```

together with four TCCs corresponding to the assumptions on the parameters of theory `cancel` (see 2) instantiated with the actual parameters `real`, `+`, and `0`. One way to avoid repetitive generation of identical TCCs when applying the cancel strategy is to import an instance of the cancel theory—e.g. `IMPORTING`

`cancel[real, +, 0]`. This causes the PVS type checker to suppress TCCs corresponding to theory instantiation, since it is able to detect that identical TCCs have already been generated; in those cases, calls to the `cancel` strategy yield exactly one subgoal, namely the simplified equation.

If the basic strategy `NORM` has been implemented correctly, the strategy `cancel` is "correct by construction" since it obeys the PVS prover interface; in particular, Lisp functions like `quote-eqn` can not alter soundness of the system as long as they do not update internal prover structures. The major difference between PVS strategies and LCF-like tactics is that in PVS there is no safety kernel to guarantee that some defined strategy conforms to the specified interface.[3]

5 Ordered Binary Decision Diagrams

Ordered Binary Decision Diagrams (OBDDs) represent Boolean functions in a form that is both canonical and compact for many practical cases. They have found widespread use in CAD applications such as formal verification, logic synthesis, and test generation, since OBDDs are manipulated by efficient graph algorithms. Here, we encode OBDDs together with a core library of verified OBDD manipulations in PVS and use these functions as a verified decision procedure for propositional logic. Our OBDD implementation follows the description in [2].

An OBDD is a rooted, directed acyclic graph with one or two terminal nodes labeled 0 and 1 of out-degree zero, a set of non-terminal nodes of out-degree two with one outgoing edge labeled *low* and the other *high*, a variable name attached to each non-terminal node such that on all paths from the root to the terminal nodes these variables respect a given linear order. Furthermore, no non-terminal node has identical 0 and 1-successor.

OBDDs may be represented by a function `tab` that maps node indices to nodes (see 10) and satisfies the additional properties concerning variable ordering and reduction of common successors as mentioned above. The first two entries in the OBDD table are reserved for the terminal nodes `0` and `1`, whereas the remaining indices point to non-terminal nodes. The latter are elements of type `Node` comprising triples of variable names (encoded as numbers with the ordering on natural numbers) and the two successor nodes. Technically, the function `tab` is defined as the inverse of a function `lookup`, which computes for every node a unique node index by a twofold application of Cantor's diagonalization technique. Since `lookup` is strictly monotonically increasing in all of its arguments the corresponding graph is assured to be acyclic.

[3] It is straightforward, however, to write such a checker.

```
bdd : THEORY                                                          10
BEGIN

  name      : TYPE = nat
  index     : TYPE = nat
  nodeindex : TYPE = upfrom(2)
  Node      : TYPE = [# variable: name, low: index, high: index #]

  l,h,i : VAR index;  n : VAR nodeindex;  v : VAR name;

  pairing(l,h)  : index     = ((l+h)*(l+h+1))/2 + l
  lookup(v,l,h) : nodeindex = 2 + pairing(pairing(v,l),h)

  tab(n) : Node =
    LET T = inverse(lookup)(n) IN
      (# variable:=PROJ_1(T), low:=PROJ_2(T), high:= PROJ_3(T) #)

  leaf?(i) : bool = i < 2
  rank(i)  : nat  = IF leaf?(i) THEN 0 ELSE 1+variable(tab(i)) ENDIF

  ordered?(i) : RECURSIVE bool =
    IF leaf?(i) THEN TRUE
      ELSE LET t = tab(i) IN
             (leaf?(low(t)) ∨ variable(t) > variable(tab(low(t))))
             ∧ (leaf?(high(t)) ∨ variable(t) > variable(tab(high(t))))
             ∧ ordered?(low(t)) ∧ ordered?(high(t))
    ENDIF MEASURE rank(i)

  reduced?(i:(ordered?)) : RECURSIVE bool =
    IF leaf?(i) THEN TRUE
      ELSE low(tab(i))≠high(tab(i))
           ∧ reduced?(low(tab(i))) ∧ reduced?(high(tab(i)))
    ENDIF MEASURE rank(i)

  OBDD : TYPE = (reduced?)
```

Given an environment ρ of type `[name → bool]` for associating variable names with some boolean expression, one can easily define the meaning $[\![\mathtt{b}]\!](\rho)$ of an OBDD `b` by recursively computing the corresponding *if-then-else* normal form.

```
  [ ](b:OBDD)(ρ:[name→bool]): RECURSIVE bool =                        11
    IF      b = 0 THEN FALSE
      ELSIF b = 1 THEN TRUE
      ELSIF ρ(variable(tab(b)))
         THEN [high(tab(b))](ρ)
      ELSE    [low(tab(b)) ](ρ)
    ENDIF MEASURE rank(b)
```

We have encoded in Pvs a number of fundamental functions for building up OBDDs. The `apply` function (see [9,2]), for example, combines two source

OBDDs into a target OBDD according to the binary Boolean operation $\bullet$; this fact is expressed and formally verified in theorem `apply_correct`.

```
  • : VAR [bool, bool -> bool];  b,l,h : VAR OBDD                          12

  decr?(v)(b) : bool = leaf?(b) ∨ v > variable(tab(b))

  makenode(v,(l,h:(decr?(v)))): {b | rank(l)≤rank(b)∧rank(h)≤rank(b)}
  = IF l = h THEN l ELSE lookup(v,l,h) ENDIF

  apply(•,l,h) : RECURSIVE {b | rank(b)≤rank(l)∨rank(b)≤rank(h)} =
    IF leaf?(l)∧leaf?(h) THEN
      bool2bit(bit2bool(l)•bit2bool(h))
    ELSIF leaf?(l)∧¬leaf?(h) THEN
        makenode(variable(tab(h)),
                 apply(•,l,low(tab(h))), apply(•,l,high(tab(h))))
    ELSIF ¬leaf?(l)∧leaf?(h) THEN
        makenode(variable(tab(l)),
                 apply(•,low(tab(l)),h), apply(•,high(tab(l)),h))
    ELSIF variable(tab(l)) = variable(tab(h)) THEN
        makenode(variable(tab(l)),
                 apply(•,low(tab(l)),low(tab(h))),
                 apply(•,high(tab(l)),high(tab(h))))
    ELSIF variable(tab(l)) < variable(tab(h)) THEN
         makenode(variable(tab(h)),
                  apply(•,l,low(tab(h))), apply(•,l,high(tab(h))))
    ELSE makenode(variable(tab(l)),
                  apply(•,low(tab(l)),h), apply(•,high(tab(l)),h))
    ENDIF MEASURE rank(l) + rank(h)

  apply_correct : THEOREM
    ⟦apply(•,l,h)⟧(ρ) = ⟦l⟧(ρ) • ⟦h⟧(ρ)

END bdd
```

The OBDD encodings can be used to construct a verified—and reasonably efficient—procedure for deciding validity (or unsatisfiability) of some propositional formula `p` by means of computational reflection. Since the overall structure is a straightforward variant of the application of cancellation in Sect. 4, we restrict ourselves to a rough outline of this procedure; details of the Pvs definitions can be found in Appendix B.

First, a specialized tactic computes a syntactic representation `rep` of `p` together with an environment ρ for associating variable names with the arguments of the Boolean operators in `p` such that `p` = ⟦`rep`⟧(ρ). The meaning ⟦`rep`⟧(ρ) of representation `rep` (with respect to ρ) is computed by recursing on `rep`. Second, compute an OBDD `build(rep)` by recursively applying the OBDD constructor `apply`. Theorem `apply_correct` in 12 is used to prove the correctness of this procedure:

$$[\![\texttt{rep}]\!](\rho) = [\![\texttt{build(rep)}]\!](\rho)$$

Third, depending on whether the original formula `p` is valid or unsatisfiable, the evaluation of the right hand side of the equation above yields `TRUE` or `FALSE`, respectively. In all other cases one has the choice of replacing `p` with an equivalent *if-then-else* normal form or leaving `p` unchanged.

It is not hard to see how this procedure can be expressed—as a variant of the definition of the cancellation strategy in Sect. 4—as a defined strategy `bddprop`. We have applied this strategy to numerous examples. Figure 2, for example, states the run times[4] for deciding various pigeon hole formulas (with n holes and $n+1$ pigeons). These numbers indicate that our verified decision procedure is efficient enough to be used for a number of problems that occur in practice.

n	1	2	3	4	5	6	7	8	9	10	11	12
sec.	1.48	7.83	23.73	20.62	39.84	68.59	154.93	151.81	184.09	293.57	418.18	489.49

Fig. 2. Deciding pigeon hole formulas

6 Conclusions

The main thesis of this paper is that current theorem provers like Pvs can readily be extended to provide a reasonably efficient and practical notion of reflection in order to soundly extend theorem proving capabilities. To substantiate this claim, we have extended the Pvs system by verified proof procedures—such as cancellation of equations and a decision procedure based on OBDDs—and demonstrated how to apply these procedures through (Pvs) tactics and computational reflection. More precisely, the proof procedures are functions encoded in the underlying logic that work on representations of parts of this logic itself. Functional proof procedures are applied by sequencing a number of Pvs strategies to compute a representation from the current proof goal, to compute the normal form of the proof procedure applied to this representation, and to compute a corresponding simplified term by means of computational reflection. The practicality of such an extension depends heavily on efficient normalization of functional expressions.

The overall architecture of computational reflection is essentially a re-casting of computational reflection as described in [8]. Our mechanism, however, is much more flexible—and therefore more widely applicable—in that we can abstract meta theorems with respect to classes of structures. Moreover, our approach allows for defining different representations, lifting of terms/formulas to these

[4] Run times on a Sparc Ultra-II as reported by Pvs. Disturbances in the expected monotonicity are due to garbage collection.

representations, and suitable correctness criteria; there are examples for which equivalence reasoning is not appropriate and the result of the computational reflection process should rather be some "refinement" of the original formula.

The concept of reflection has been used in several proof systems and has been applied to various examples; see [14] for a good survey. In particular, the extraction mechanism of Coq has recently been used to translate proofs into executable Caml programs [6, 26].

Much work remains to be done. Specialized OBDD packages written in C are an order of magnitude faster and more effective than the encodings given in this paper. This is mainly due to aggressive use of tabulation techniques, specialized garbage collection algorithms, and variable reordering techniques. There is no conceptual reason, however, not to extend the verified OBDD package described above with more advanced features like efficient garbage collection. Another desirable goal would be to encode and verify complete OBDD packages and symbolic model-checkers inside Pvs, thereby eliminating the error-prone task of connecting external (and unreliable) decision procedures. Similarly, it seems feasible to internalize and formally verify algorithms for combining decision procedures such as those described in [24, 18].

Acknowledgements. We would like to thank S. Owre from SRI International for invaluable help with implementing the symbolic evaluator and integrating it with Pvs. We also thank the anonymous reviewers for their helpful comments.

References

1. S.F. Allen, R.L. Constable, D.J. Howe, and W.E. Aitken. The Semantics of Reflected Proof. In *Proc. 5th Annual IEEE Symposium on Logic in Computer Science*, pages 95–105. IEEE CS Press, 1990.
2. H.R. Anderson. An Introduction to Binary Decision Diagrams. Available at: `ftp.id.dtu.dk/pub/hra`, September 1994.
3. D.A. Basin. Beyond Tactic Based Theorem Proving. In J. Kunze and H. Stoyan, editors, *KI-94 Workshops: Extended Abstracts*. Gesellschaft für Informatik e.V, 1994. 18. Deutsche Jahrestagung für Künstliche Intelligenz, Saarbrücken.
4. D.A. Basin and R.L. Constable. Metalogical Frameworks. Technical Report TR 91-1235, Department of Computer Science, Cornell University, September 1991.
5. U. Berger and H. Schwichtenberg. An Inverse of the Evaluation Functional for Typed λ-calculus. In *Proceedings, Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211, Amsterdam, The Netherlands, 15–18 July 1991. IEEE Computer Society Press.
6. S. Boutin. Using Reflection to Build Efficient and Certified Decision Procedures. In M. Abadi and T. Ito, editors, *Theoretical Aspects of Computer Software*, volume 1281 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.
7. R.S. Boyer and J.S. Moore. *A Computational Logic*. Academic Press, New York, 1979.
8. 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*, chapter 3. Academic Press, 1981.

9. R.E. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.
10. R.L. Constable, S.F. Allen, and H.M. Bromley et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice–Hall, 1986.
11. J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A Tutorial Introduction to PVS. Presented at WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida, April 1995.
12. M.J.C. Gordon and T.F. Melham. *Introduction to HOL : A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.
13. M.J.C. Gordon, A.J.R. Milner, and C.P. Wadsworth. *Edinburgh LCF: a Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1979.
14. J. Harrison. Metatheory and Reflection in Theorem Proving: A Survey and Critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK, 1995.
15. D.J. Howe. *Automating Reasoning in an Implementation of Constructive Type Theory*. PhD thesis, Cornell University, 1988. Available as technical report TR 88-925 from the Department of Computer Science, Cornell University.
16. T.B. Knoblock and R.L. Constable. Formalized Metareasoning in Type Theory. In *Proceedings of LICS*, pages 237–248. IEEE, 1986. Also available as technical report TR 86-742, Department of Computer Science, Cornell University.
17. G. Kreisel and A. Lévy. Reflection Principles and Their Use for Establishing the Complexity of Axiomatic Systems. *Zeitschrift für math. Logik und Grundlagen der Mathematik*, Bd. 14:97–142, 1968.
18. G. Nelson and D.C. Oppen. Simplification by Cooperating Decision Procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.
19. S. Owre, J. Rushby, N. Shankar, and F. 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.
20. L.C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*. Number 2 in Cambride Tracts in Theoretical Computer Science. Cambridge University Press, 1987.
21. L.C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.
22. S. Pfab. Efficient Symbolic Evaluation of Formal Specifications and Its Interrelationship with Theorem Proving. Master's thesis, Universität Ulm, Fakultät für Mathematik, January 1998.
23. H. Rueß. Computational Reflection in the Calculus of Constructions and Its Application to Theorem Proving. In J. R. Hindley P. de Groote, editor, *Proceedings of Typed Lambda Calculus and Applications (TLCA'97)*, volume 1210 of *Lecture Notes in Computer Science*, pages 319–335. Springer-Verlag, April 1997.
24. R.E. Shostak. Deciding Combinations of Theories. *Journal of the ACM*, 31(1):1–12, 1984.
25. C. Smorynski. *Self-Reference and Modal Logic*. Springer-Verlag, 1985.
26. C. Sprenger. A Verified Model Checker for the Modal μ-Calculus in Coq. In B. Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
27. F. von Henke. An Algebraic Approach to Data Types, Program Verification, and Program Synthesis. In *Mathematical Foundations of Computer Science, Proceedings*, volume 45 of *Lecture Notes in Computer Science*. Springer-Verlag, 1976.

A Auxiliary Functions for Cancel Strategy

```
(defun argument1 (app) (first (exprs (argument app))))
(defun argument2 (app) (second (exprs (argument app))))

(defun quote-eqn (eqn op e &key (modinst ""))
  "Compute a representation together with an association list for a
   equality."
  (multiple-value-bind (rep1 trms1)
      (quote-trm (argument1 eqn) op e :modinst modinst)
    (multiple-value-bind (rep2 trms2)
        (quote-trm (argument2 eqn) op e :trms trms1 :modinst modinst)
      (let* ((rep
              (format nil "(# lhs := ~a, rhs := ~a #):~a.equality"
                      rep1 rep2 modinst))
             (trms  (union trms1 trms2 :test #'tc-eq))
             (alist (generate-alist (reverse trms) :modinst modinst)))
        (list rep alist)))))

(defun quote-trm (trm op e &key trms (modinst ""))
  "Compute a representation together with an association list for
   argument terms of an equation."
  (cond ((tc-eq trm e)
         (values (format nil "~a.mk_neutral" modinst) trms))
        ((and (application? trm) (tc-eq (operator trm) op))
         (let ((arg1 (argument1 trm))
               (arg2 (argument2 trm)))
           (multiple-value-bind (rep1 trms1)
               (quote-trm arg1 op e :trms trms :modinst modinst)
             (multiple-value-bind (rep2 trms2)
                 (quote-trm arg2 op e :trms trms1 :modinst modinst)
               (let ((rep (format nil "~a.mk_app(~a,~a)"
                                  modinst rep1 rep2)))
                 (values rep trms2))))))
        (t (let* ((new-trms (adjoin trm trms :test #'tc-eq))
                  (pos (position trm (reverse new-trms) :test #'tc-eq))
                  (rep (format nil "~a.mk_var(~a)" modinst pos)))
             (values rep new-trms)))))


(defun generate-alist (l &key (acc "empty") (count 0) (modinst ""))
  "Compute an association list for the  list of terms t1,...,tn
   of the form (insert(...insert(insert(empty, t1, 0), t2, 1),...))."
  (if (null l) acc
    (let ((newacc (format nil "~a.insert(~a,~a,~a)"
                          modinst acc count (car l))))
      (generate-alist (cdr l) :acc newacc :count (1+ count)
                      :modinst modinst))))
```

B Auxiliary Definitions for BDD Package

Datatype for representing Boolean expressions:

```
BExpr : DATATYPE                                                    13
BEGIN
  mk_true                          : true?
  mk_false                         : false?
  mk_var(i : name)                 : variable?
  mk_not(arg : BExpr)              : negation?
  mk_and(left,right : BExpr)       : conjunction?
  mk_or(left,right : BExpr)        : disjunction?
  mk_implies(left,right : BExpr)   : implication?
END BExpr
```

The meaning of representation `rep` with respect to an environment ρ:

```
⟦⟧(rep:BExpr)(ρ:[name → bool]) : RECURSIVE bool =                   14
  CASES rep OF
    mk_false        : FALSE,
    mk_true         : TRUE,
    mk_var(x)       : ρ(x),
    mk_not(v)       : ¬⟦v⟧(ρ),
    mk_and(v,w)     : ⟦v⟧(ρ) ∧ ⟦w⟧(ρ),
    mk_or(v,w)      : ⟦v⟧(ρ) ∨ ⟦w⟧(ρ),
    mk_implies(v,w) : ⟦v⟧(ρ) ⇒ ⟦w⟧(ρ)
  ENDCASES MEASURE rep BY ≪
```

Function `build` computes an OBDD for a represention `rep` by recursively applying the OBDD constructor `apply`.

```
build(rep:BExpr): RECURSIVE OBDD =                                  15
  CASES rep OF
    mk_false        : 0,
    mk_true         : 1,
    mk_var(v)       : lookup(v,0,1),
    mk_not(v)       : apply(λx,y: ¬x, build(v), 0),
    mk_and(v,w)     : apply(∧, build(v), build(w)),
    mk_or(v,w)      : apply(∨, build(v), build(w)),
    mk_implies(v,w) : apply(⇒, build(v), build(w)),
  ENDCASES MEASURE rep BY ≪
```