A Theory for Abstract Reduction Systems in PVS

André Luiz Galdino[*]
Departamento de Matemática, Universidade Federal de Goiás
Campus de Catalão, Brazil
galdino@unb.br

and

Mauricio Ayala-Rincón
Instituto de Ciências Exatas, Universidade de Brasília
Brasília D.F., Brazil
ayala@unb.br

Abstract

A theory for Abstract Reduction Systems (ARS) in the proof assistant PVS (Prototype Verification System) is described. Adequate specifications of basic definitions and notions such as reduction, confluence, normal form are given and well-known results proved, which includes non trivial ones such as Noetherian Induction, Newman's Lemma and its generalizations, and Commutation Lemma among others. Although term rewriting proving technologies have been specified in several specification languages and proof assistants, to our knowledge this has not been done in PVS. This makes relevant our ARS specification as the initial step in the formulation of a complete theory for Term Rewriting Systems (TRS) in PVS.

Keywords: Abstract Reduction Systems, Term Rewriting Systems, Automated Theorem Proving, PVS.

1 Introduction

Abstract Reduction Systems (ARS) and Term Rewriting Systems have been specified in several proof assistants, e.g., RRL [9], ACL2 [16], Coq [8], Isabelle [15], Boyer-Moore [17], Otter [5] among others. Term rewriting proving technologies have been shown adequate in several mathematics and computer science fields including theorem proving as well as formal specification and design of computational processes and technologies (i.e., standard and non-standard software and hardware). In particular, we have developed a methodology for specifying reconfigurable hardware over FPGAs using the rewriting-logic programing environment ELAN [2]. These rewriting based hardware specifications are synthesized to commercial reconfigurable hardware by applying the system FELIX [10] and their correctness is verified over the proof assistant PVS after translating the rewriting specification to a corresponding logic theory with the system SAEPTUM [3]. The last mentioned step should be improved by making available a full theory of rewriting methods in PVS, that to our knowledge is not available in this proof assistant.

With this motivation, this paper introduces a PVS theory for dealing with properties of ARSs. Basic ARS notions are adequately specified in such a way that non elementary proof techniques such as Noetherian induction are straightforwardly applicable. To illustrate the adequateness of these specification well-known results such as Newman's, Yokouchi's and commutation Lemma are verified. These specifications are built over PVS theories for sets and relations. In particular Noetherianity is based on the notion of well-founded relations and because of this, after introducing the notion of noetherian relation the principle of noetherian induction should be verified.

The introduced ARS theory should be conceived as a first step in the development of a full TRS theory in PVS. The files of this theory are available at `www.mat.unb.br/∼ayala/TCgroup`.

[*]Author partially supported by the Brazilian Research Council CNPq with a PhD scholarship at the Universidade de Brasília.

2 Brief Introduction to PVS

This section briefly describes the PVS prover used to specify the ARS theory. PVS consists of a specification language integrated with support tools and a proof assistant, that provides an integrated environment for the development and analysis of formal specifications. Only the relevant aspects of PVS are explained here. For more details about the tool, refer to the PVS System Guide [19], the PVS Prover Guide [18] and the PVS Language Reference [14] available at `http://pvs.csl.sri.com`.

The *specification language* of PVS is built on higher-order logic, which supports modularity by means of parameterized *theories*, with a rich type-system, including the notions of subtypes and dependent types. It provides a large set of built-in constructs for expressing a variety of notions. The PVS specifications is organized as a collection of theories, from which the most relevants are collectively referred as the *prelude* [13]. Each theory is composed essentially of *declarations*, which are used to introduce names for types, constants, variables, axioms and formulas, and `IMPORTINGs`, which allow to import the visible names of another theories. Notice that parameterized theories are very convenient since the use of parameters allows more generic specifications, as we can see with the `ars` PVS theory below:

```
ars[T : TYPE] : THEORY
BEGIN

  IMPORTING results_commutation[T],
            modulo_equivalence[T],
            results_normal_form[T],
            newman_yokouchi[T]

end ars
```

`T` is treated as a fixed uninterpreted type. Consequently, when the `ars` theory is invoked by another theory, `T` must be instantiated. For example, the theory of `ars` of set of `term` is just `ars[term]`. Notice that `ars` imports the theories `results_commutation[T]`, `modulo_equivalence[T]`, `results_normal_form[T]` and `newman_yokouchi[T]`.

A important step in PVS specifications is type-checking the theory, which checks for semantic errors, such as undeclared names and ambiguous types. Type-checking may build new files or internal structures such as `TCCs` (type-correctness conditions). These `TCCs` represent *proof obligations* that must be discharged before the theory can be considered type-checked, and its proofs may be postponed indefinitely. Although, the theory is considered *complete* when all `TCCs` and formulas upon which the proof is dependent have been completed.

The PVS *Prover* provides a variety of commands to construct the proofs of the different theorems. It is used interactively and it uses the sequent-style proof representation to display the current proof goal for the proof in progress. The prover maintains a proof tree for the current theorem being proved being the aim of the user to construct a proof tree that is complete, in the sense that all the leaves are recognized as `true`. Each node of the tree is a proof goal that results from the application of a prover command (*rule* or *strategy*) to its parent node. Each proof goal is a sequent consisting of two sequences of formulas called the antecedents (numbered with negative integers) and the consequent (numbered with positive integers) displayed as below:

```
[-1] A₁
[-2] A₂
   ⋮
|-------
 [1] C₁
 [2] C₂
   ⋮
```

3 PVS Strategies Used in the Proofs

Below we describe some PVS prover commands that are commonly used in our specification, and we define some strategies for minimize the size of the proofs. Many of these commands take arguments that control its behavior which are not discussed here. For additional details see [18].

1. `skolem`: This command chooses fresh constant names (universally quantified consequent), and proving "without loss of generality", or unconstrained arbitrary constant when one is known to exist (existentially quantified antecedent). In other words, `skolem` gives new constant names, e.g., for `x` it will give `x!1`, `x!2`, . . . when applied repeatedly.

2. `skeep`: This command is used to introduce Skolem constants by keeping the original names of the quantified variables. See [12] and [20].

3. `flatten`: This command is used to break an antecedent formula that is a conjunction or a consequent formula that is a disjunction into its components.

4. `assert`: This command is used to simplify the proof goal using decision procedures and rewriting.

5. `inst`: This command is used to instantiate a universally quantified antecedent or an existentially quantified consequent formula.

6. `case`: This command generates two subgoal, one where the given boolean expression is assumed to be `true` and the other where it is assumed to be `false`.

7. `expand`: This command expands and simplifies the definitions of the specified functions/predicates at the occurrences.

8. `lemma`: This command is used to pull in a previously proved theorem into the current proof goal instantiated as specified by the user.

Other useful rules can be found in [18], e.g., `replace`, `prop`, `split` and `decompose-equality`. PVS also provides a simple language to combine sequences of commonly used proof steps into strategies [1]. These strategies can then be used as prover commands. In many proofs, it is necessary to use the same sequence of proof steps. Thus, to facilitate and to minimize the proofs we turned some commonly used sequences of proof commands into strategies. Some of them are discussed below.

In some proofs, it was necessary to firstly, `expand` the definition of *joinable*; afterward, to introduces `skolem` constants and finally, to apply disjunctive simplification (`flatten`). The strategy `join-skolem` accomplishes this.

```
(defstep join-skolem (var1 fnum)
  (then (expand "joinable?" fnum) (skolem * var1) (flatten))
  "Expanding joinable?, Skolemizing, and
   Applying disjunctive simplification.")
```

Another useful strategy is `expand-closure` that expands the definition of closure relation according to input `closure` which can be either `rtc` (Reflexive Transitive Closure) or `ec` (Equivalence Closure) or `rc` (Reflexive Closure) or `sc` (Symmetric Closure). This strategy uses another one called `expand-um` which expands the definitions of `union` and `member`.

```
(defstep expand-closure (closure fnum)
  (if (equal closure ‘rtc)
      (then (expand "RTC" fnum )
            (expand "IUnion" fnum))
      (if (equal closure ‘ec)
          (then (expand "EC" fnum)
                (expand "RTC" fnum)
                (expand "IUnion" fnum))
          (if (equal closure ‘rc)
              (then (expand "RC" fnum)
                    (expand-um fnum))
              (if (equal closure 'sc)
                  (then (expand "SC" fnum)
                        (expand-um fnum))
                  (skip)))))
  "Expanding the definition of ~A.")
```

Few other commonly used sequences of proof steps were turned into strategies too.

4 Specifying ARS in PVS

We briefly present the standard definitions of ARS and some properties [4] and then we present their specification in PVS.

An *Abstract Reduction System* (ARS) is a pair $(A, \to)$, where the *reduction* $\to$ is a binary relation on the set A, i.e., $\to \subseteq A \times A$. In this paper we consider some arbitrary but fixed ARS $(A, \to)$. We treated, in PVS, the set A as a fixed uninterpreted type `T`, and the reduction $\to$ as a binary relation `R` on `T` defined as predicate `PRED: TYPE = [[T,T] -> bool]`. So the relation `R(x,y)` means `x` reduces to `y`, and `y` is called a *reduct* of `x`.

To specify some of the central notions of ARS such as *confluence* and *termination*, first, it is necessary to adequately speficify several closure relations:

Abstract definition	PVS specification
$\to^0 := \{(x,x) \mid x \in A\}$	identity
$\to^{i+1} := \to^i \circ \to$	$(i+1)$-fold composition, $i \geq 1$
$\to^= := \to \cup \to^0$	reflexive closure (`RC`)
$\to^+ := \bigcup_{i>0} \to^i$	transitive closure (`TC`)
$\to^* := \to^+ \cup \to^0$	reflexive transitive closure (`RTC`)
$\leftarrow := \to^{-1}$	converse
$\leftrightarrow := \to \cup \leftarrow$	symmetric closure (`SC`)
$\leftrightarrow^* := (\leftrightarrow)^*$	equivalence closure (`EC`)

`RC`, `TC`, `RTC`, `SC`, and `EC` were defined in the PVS theory `relations_closure` in the same way that Alfons Geser does in the PVS theory for closure operators (`PVS/lib/sets_lemmas/closure_ops`). We just changed the names of the definitions and we proved some additional properties. For example, `RTC` is defined using the `iterate` function which allows us to obtain inductive proofs on the length of derivations:

```
    RTC(R): reflexive_transitive = IUnion(LAMBDA n: iterate(R, n))
```

Then the additional properties are proved:

```
    R_subset_RTC: LEMMA subset?(R, RTC(R))

    iterate_RTC: LEMMA FORALL n: subset?(iterate(R, n), RTC(R))
```

4.1 Confluence

For all $x, y, z \in A$ a relation $\to$ is called

1. *confluent* iff $y \ {}^*\!\!\leftarrow x \to^* z$ implies that y and z are *joinable*, i.e., iff there is a $r \in A$ such that $y \to^* r \ {}^*\!\!\leftarrow z$.
2. *Church-Rosser* iff $x \leftrightarrow^* y$ implies that x and y are joinable.
3. *semi-confluent* iff $y \leftarrow x \to^* z$ implies that y and z are joinable.

These and other notions such as *local confluent*, *strongly confluent*, *diamond property*, *normal form*, *normalizing* and *commutation* are specified in the PVS theory `ars_terminology` as follow:

```
ars_terminology[T: TYPE]: THEORY
BEGIN

  IMPORTING relations_closure[T]

    R, R1, R2: VAR PRED[[T, T]]
   x, y, z, r: VAR T
 ...
 joinable?(R)(x,y): bool = EXISTS z: RTC(R)(x,z) & RTC(R)(y, z)

 church_rosser?(R): bool = FORALL x, y: EC(R)(x,y) => joinable?(R)(x,y)

 semi_confluent?(R): bool = FORALL x, y, z: R(x,y) & RTC(R)(x,z) =>
                                                    joinable?(R)(y,z)

 confluent?(R): bool = FORALL x, y, z: RTC(R)(x,y) & RTC(R)(x,z) =>
                                                    joinable?(R)(y,z)

 commute?(R1,R2): bool = FORALL x, y, z: RTC(R1)(x,y) & RTC(R2)(x,z) =>
                                 EXISTS r: RTC(R2)(y,r) & RTC(R1)(z,r)
 ...
 END ars_terminology
```

Some basic results involving confluence are specified and proved in the PVS theory `results_confluence`. For example, the equivalence between Church-Rosser and confluence, and the *commutative union lemma* which tells us that for commutative relations union preserves confluence are specified as:

```
CR_iff_Confluent: THEOREM church_rosser?(R) <=> confluent?(R)

Commutative_Union_Lemma: LEMMA confluent?(R1) & confluent?(R2) &
                            commute?(R1,R2) => confluent?(union(R1, R2))
```

4.2 Termination

A relation $\rightarrow$ is called *terminating* or *noetherian* iff there is no infinite descending chain $a_0 \rightarrow a_1 \rightarrow \cdots$. In other words, $\rightarrow$ is *noetherian* iff $\leftarrow$ is well-founded.

As it is well-known many results involving termination are proved by *Noetherian induction*, that is: let P be some property of elements of A. Then to prove $P(x)$ for all $x \in A$, it suffices to prove $P(x)$ under the assumption that $P(y)$ holds for all successors $y \in A$ of x.

In the PVS theory `noetherian` below, we defined noetherian relation based on the notion of well-founded relation (it simplify proofs) and we proved the principle of Noetherian induction. To prove this principle we used the lemma `wf_induction`, with suitable substitutions, which expresses the principle of *well-founded induction* and can be found in the PVS prelude theory [13] as well as the notions of *well-founded* relations.

```
noetherian[T: TYPE]: THEORY
BEGIN

  IMPORTING ars_terminology[T],
            sets_aux@well_foundedness[T]

     P: VAR PRED[T]
     R: VAR PRED[[T, T]]
  x, y: VAR T

 noetherian?(R): bool = well_founded?(converse(R))
 ...
 noetherian_induction: LEMMA
```

```
             (FORALL (R: noetherian, P):
               (FORALL x:
                  (FORALL y: TC(R)(x, y) IMPLIES P(y))
                      IMPLIES P(x))
            IMPLIES
              (FORALL x: P(x)))

END noetherian
```

4.3 Modulo Equivalence

Considering a reduction relation R, together with an equivalence relation Eq we defined the notions of *reduction modulo equivalence* [7] and we proved the generalization of Newman's Lemma:

```
modulo_equivalence[T: TYPE] : THEORY
BEGIN

  IMPORTING noetherian[T]

  R, S: VAR  PRED[[T, T]]
    Eq: VAR equivalence
  x, y,
  z, w,
  u, v: VAR T
...
  joinable_m?(R, Eq)(x,y) : bool = EXISTS u,v: RTC(R)(x,u) &
                                               Eq(u,v) & RTC(R)(y,v)

  local_confluent_m?(R, Eq) : bool = FORALL x, y, z: R(x,y) & R(x,z) =>
                                                     joinable_m?(R, Eq)(y,z)

  confluent_m?(R, Eq) : bool = FORALL x, y, z, w: RTC(R)(x,z)  &
                                                  Eq(x,y) &
                                                  RTC(R)(y,w) =>
                                                  joinable_m?(R, Eq)(z,w)

  locally_coherent?(R, Eq, S) : bool = symmetric?(S) &
                                       FORALL  x, y, z: R(x,y) &
                                       S(x,z) => joinable_m?(R, Eq)(y,z)

...

van_oostrom94: LEMMA diamond_property?(RTC(R) o Eq) => confluent_m?(R, Eq)


newman_lemma_general: THEOREM noetherian?(R) =>
                              (local_confluent_m?(R, Eq) &
                               locally_coherent?(R, Eq, Eq) <=>
                                                    confluent_m?(R, Eq))
END modulo_equivalence
```

5 PVS Theory Organization, Proof Examples and Proof Summary

5.1 PVS Theory Organization

Below we show the organization of the PVS theories which compound the ars theory and we give a brief description of each one (see Figure 1).

1. `relations_closure`: This theory contains the definitions of closure of a relation and some properties.
2. `ars_terminology`: This theory contains some terminology of ARS such as *unique normal form*, *reducible* and *sucessor*, and notions of confluence and commutation.
3. `results_confluence`: This theory contains some results about confluence such as *strong confluent implies semi-confluent.*
4. `results_commutation`: This theory contains some results about commutation such as *Commutation lemma.*
5. `results_normal_form`: This theory contains some results involving normal form such as *a relation is normalizing and confluent iff every element has a unique normal form.*
6. `noetherian`: This theory contains the definition of *convergent reduction* and noetherian relation and the Noetherian induction lemma.
7. `newman_yokouchi`: This theory contains the specification of *Newman's lemma* and *Yokouchi's lemma.*
8. `modulo_equivalence`: This theory contains the notions of reduction modulo equivalence and, for example, the proof of the generalization of Newman's Lemma.

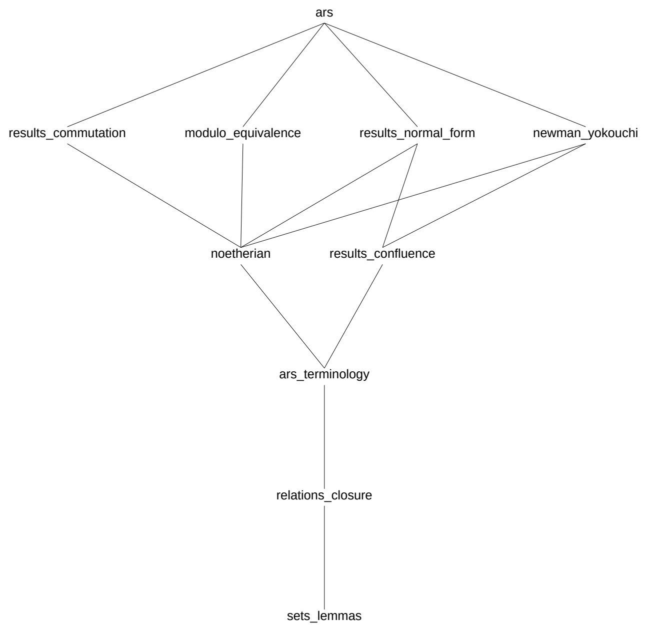

Figure 1: `ars` theory dependencies

5.2 Proof Examples

The PVS proofs are available as part of the `ars` theory at `www.mat.unb.br/~ayala/TCgroup` and detailed explanations of the PVS proofs of Newmann's and Yokouchi's lemmas are available in [6].

To prove the commutation lemma:

```
Commutation_Lemma: THEOREM strong_commute?(R1,R2) => commute?(R1,R2)
```

we use the sequence of commands (`skeep`), (`expand`), (`skolem`), (`lemma`), (`inst`), and (`assert`). The command `lemma` is used to invoke the following lemma:

```
 commute_and_iterate_two: LEMMA FORALL (n,m: nat): strong_commute?(R1,R2)
                        & iterate(R1, n)(x,y) & iterate(R2, m)(x,z) =>
                                   EXISTS r: RTC(R2)(y,r) & RTC(R1)(z,r)
```

This lemma is proved by induction on `m` by applying the command (`induct "m"`), and by invoking the lemma `commute_and_iterate_one` which is proved by induction too.

Now, we present details of the proof of Newman's lemma that since the inductive proof given by Huet in [7] is considered a classical benchmark for proof in higher-order logic as discussed for instance in [5].

Newman's lemma states that an abstract reduction system is confluent if it is local confluent and noetherian and is specified in the `ars` theory as:

```
Newman_lemma: THEOREM noetherian?(R) => (confluent?(R) <=> local_confluent?(R))
```

When the PVS prover is invoked the proof tree starts off with a root node having no antecedent and the theorem to be proved as the sole consequent:

```
  |-------
{1}   FORALL (R: PRED[[T, T]]):
        noetherian?(R) => (confluent?(R) <=> local_confluent?(R))
```

The universally quantified variable is skolemized and disjuntive simplification is applied using the command `skeep`, and then the command `split` yields (splitting the consequent formula) two subgoals: The first subgoal is

```
[-1]  noetherian?(R)
  |-------
{1}   confluent?(R) IMPLIES local_confluent?(R)
```

This subgoal is proved immediately from the definitions of `confluent?(R)` and `local_confluent?(R)`. Firstly, by applying the disjuntive simplification (`flatten`), then the definitions of `local_confluent?` and `confluent?` are expanded (`expand*`):

```
 {-1}  FORALL (x: T), (y: T), (z: T):
        RTC(R)(x, y) & RTC(R)(x, z) => joinable?(R)(y, z)
  |-------
 {1}   FORALL (x: T), (y: T), (z: T): R(x, y) & R(x, z) => joinable?(R)(y, z)
```

Next, applying (`skeep`) and invoking the lemma `R_subset_RTC` which establish that `R` $\subseteq$ `RTC`. Finally, by applying the strategy (`expand-sm`), that expands the definitions of `subset?` and `member`, and doing the convenient instantiation using the command (`inst`).

The second and truly interesting subgoal is

```
[-1]  noetherian?(R)
  |-------
{1}   local_confluent?(R) IMPLIES confluent?(R)
```

As it is well-known, this result is obtained by Noetherian induction using the predicate: $P(x) = \forall y, z.\ y \; {}^*\!\!\leftarrow x \rightarrow^* z$ implies that y and z are joinable.

After applying the disjuntive simplification (`flatten`), we invoke the lemma `noetherian_induction` instantiated with the predicate

```
(LAMBDA (a: T): (FORALL (b,c: T): RTC(R)(a,b) AND RTC(R)(a,c) IMPLIES joinable?(R)(b,c)))
```

Next, by applying the command (`split`) we obtain two subgoals. As we can see below, the first one is obvious and its proof is obtained expanding `confluent?`, and applying the sequence of commands `(skeep)`, `(inst)`, used with suitable substitutions, and `(assert)`.

```
[-1]  FORALL (x: T):
        FORALL (b, c: T):
          RTC(R)(x, b) AND RTC(R)(x, c) IMPLIES joinable?(R)(b, c)
  |-------
[1]   confluent?(R)
```

The second subgoal requires to prove $P(x)$ under the assumption $P(y)$ for all y such that $x \rightarrow^{+} y$:

```
[-1]  local_confluent?(R)
[-2]  noetherian?(R)
  |-------
[1]   FORALL (x: T):
        (FORALL (y: T):
           TC(R)(x, y) IMPLIES
            (FORALL (b, c: T):
               RTC(R)(y, b) AND RTC(R)(y, c) IMPLIES joinable?(R)(b, c)))
         IMPLIES
         (FORALL (b, c: T):
            RTC(R)(x, b) AND RTC(R)(x, c) IMPLIES joinable?(R)(b, c))
```

By applying the sequence of commands `skeep`, `expand-closure`, and `skolem` we obtain

```
[-1]  FORALL (y: T):
        TC(R)(x, y) IMPLIES
         (FORALL (b, c: T):
            RTC(R)(y, b) AND RTC(R)(y, c) IMPLIES joinable?(R)(b, c))
[-2]  iterate(R, i)(x, b)
{-3}  iterate(R, j)(x, c)
[-4]  local_confluent?(R)
[-5]  noetherian?(R)
  |-------
[1]   joinable?(R)(b, c)
```

Now, after some simplifications and manipulations we obtain the result. Below we present a small part of proof tree.

```
    ...
     (case-replace "i = 0" :hide? t)
     (("1"
       (hide-all-but (-2 -3 1))
       (expand "joinable?")
       (inst 1 "c")
       (expand* "RTC" "IUnion")
       (split)
       (("1" (expand "iterate" -1) (replaces -1) (inst 1 "j"))
        ("2" (inst 1 "0") (expand "iterate" 1) (propax))))
      ("2"
       (case-replace "j = 0" :hide? t)
     ...
           (("1"
             (assert)
             (decompose-equality)
             (decompose-equality)
             (inst -2 "(x, b)")
             (inst -1 "(x, c)")
             (replaces -1)
             (replaces -1)
```

```
              (expand "o")
        ...
```

Above in the proof tree the commands (`case-replace "i = 0" :hide?  t`) and (`case-replace "j = 0" :hide?  t`) contemplate the cases `x = b` or `x = c`.

Another example of results available in the theory and obtained by Noetherian induction is the Yokouchi lemma which is specified as:

```
Yokouchi_lemma: THEOREM ( noetherian?(R) & confluent?(R) &
         diamond_property?(S) & (FORALL x, y, z: (S(x,y) & R(x,z)) =>
           (EXISTS (u: T): RTC(R)(y,u) & (RTC(R) o S o RTC(R))(z,u))) )
                                => diamond_property?(RTC(R) o S o RTC(R))
```

5.3 Proof Summary

We present below a proof summary for all theories involved with `ars` that is obtained by using the PVS tool ProofLite [11]. The PVS development consists of 65 lemmas specified in 790 lines (25094 bytes) and 7169 lines (481178 bytes) of proofs.

```
 Proof summary for theory noetherian
    R_is_Noet_iff_TC_is..................proved - complete   [shostak](0.58 s)
    noetherian_induction.................proved - complete   [shostak](0.50 s)

 Proof summary for theory results_confluence
    Joinable_implies_Equiv...............proved - complete   [shostak](0.06 s)
    reduct_transitive....................proved - complete   [shostak](0.06 s)
    semi_and_iterate.....................proved - complete   [shostak](0.27 s)
    Confl_implies_Semi...................proved - complete   [shostak](0.06 s)
    Semi_implies_CR......................proved - complete   [shostak](0.06 s)
    CR_iff_Confluent.....................proved - complete   [shostak](0.20 s)
    strong_and_iterate...................proved - complete   [shostak](0.20 s)
    Str_Confl_implies_Semi_Confl.........proved - complete   [shostak](0.10 s)
    Strong_Confl_implies_Confl...........proved - complete   [shostak](0.02 s)
    DP_implies_StC.......................proved - complete   [shostak](0.07 s)
    R1_Confl_iff_R2_Confl................proved - complete   [shostak](0.14 s)
    R1_equal_R2..........................proved - complete   [shostak](0.15 s)
    R2_Str_Confl_implies_R1_Confl........proved - complete   [shostak](0.08 s)
    Confluence_Commute...................proved - complete   [shostak](0.11 s)
    R1_R2_RTC_R1_R2......................proved - complete   [shostak](0.32 s)
    Commutative_Union_Lemma..............proved - complete   [shostak](0.14 s)

 Proof summary for theory newman_yokouchi
    Newman_lemma.........................proved - complete   [shostak](0.70 s)
    Yokouchi_lemma_ax1...................proved - complete   [shostak](0.90 s)
    Yokouchi_lemma.......................proved - complete   [shostak](1.12 s)

 Proof summary for theory results_normal_form
    NF_doesnot_rewrite...................proved - complete   [shostak](0.17 s)
    NF_implies_RTC.......................proved - complete   [shostak](0.05 s)
    NFs_implies_Equal....................proved - complete   [shostak](0.05 s)
    Norm_and_Confl_implies_UNF...........proved - complete   [shostak](0.07 s)
    Normalizing_and_Confl................proved - complete   [shostak](0.18 s)
    Normal_Confl_iff_UNF.................proved - complete   [shostak](0.11 s)
    Noetherian_implies_normalizing.......proved - complete   [shostak](0.13 s)
    Convergent_UNF.......................proved - complete   [shostak](0.02 s)
    Noet_and_Confl_iff_UNF...............proved - complete   [shostak](0.02 s)
    Convergent_iff_eqNF..................proved - complete   [shostak](0.23 s)
 Proof summary for theory modulo_equivalence
    van_oostrom94........................proved - complete   [shostak](0.19 s)
    newman_lemma_general.................proved - complete   [shostak](1.04 s)

 Proof summary for theory results_commutation
    Local_Comu_and_Noeth.................proved - complete   [shostak](0.69 s)
    commute_and_iterate_one..............proved - complete   [shostak](0.21 s)
    commute_and_iterate_two..............proved - complete   [shostak](0.25 s)
    Comutation_Lemma.....................proved - complete   [shostak](0.05 s)

Grand Totals: 65 proofs, 65 attempted, 65 succeeded (12.92 s)
```

6 Conclusions and Future Work

The PVS theory `ars` specifies adequately basic notions of the theory of Abstract Reduction Systems. On the one hand `ars` is built over the PVS theory for binary relations being the closures specified in terms of "iteration" of the binary relations. In this way inductive proofs on the length of derivations are possible. On the other hand, the notion of noetherianity is specified in terms of the notion of well-founded relations which allows us to adequately formulate and verify the principle of noetherian induction necessary for proving several properties of ARSs.

Our intention specifying the `ars` theory was not to exhaustively include proofs of all well-known results of the theory of ARSs, but instead to give the essential mechanisms for expressing and mechanically proving all these results. Adequability of our specification is made evident by the presentation of elegant proofs of well-known results over ARSs such as the Newman's and Yokouchi's lemmas [6]. Also it should be stressed here that although `ars` does not advance the state of the art in the formalization of mathematics since specifications of Abstract Reductions Systems and even of Term Rewriting Systems are available since the development of the Rewriting Rule Laboratory (RRL) in the 1980s [9], it is of practical interest since the availability of rewriting proving technologies are essential in a modern proof assistants as PVS.

As current work `ars` is being extended to a more elaborated PVS theory for full Term Rewriting Systems that is of interest to verify the correction of concrete rewriting based specifications of computational objects as mentioned in the introduction. By this extension rewriting strategies and new tactic-based techniques will be available in PVS in a natural manner.

References

[1] Archer, M., Vito, B. D., and Muñoz, C. Developing user strategies in PVS: A tutorial. In *Proceedings of Design and Application of Strategies/Tactics in Higher Order Logics STRATA'03* (NASA LaRC,Hampton VA 23681-2199, USA, September 2003), NASA/CP-2003-212448.

[2] Ayala-Rincón, M., Llanos, C. H., Jacobi, R. P., and Hartenstein, R. W. Prototyping time- and space-efficient computations of algebraic operations over dynamically reconfigurable systems modeled by rewriting-logic. *ACM Trans. Design Autom. Electr. Syst. 11*, 2 (2006), 251–281.

[3] Ayala-Rincón, M., and Sant'Ana, T. M. SAEPTUM: Verification of ELAN Hardware Specifications using the Proof Assistant PVS. In *19th Symp. on Integrated Circuits and System Design* (2006), ACM Press, pp. 125–130.

[4] Baader, F., and Nipkow, T. *Term Rewriting and* All That. Cambridge University Press, 1998.

[5] Bezem, M., and Coquand, T. Neman's Lemma - a Case Study in proof automation and geometric logic. *Bull. of the European Association for Theoretical Computer Science 79*, 86-100 (2003).

[6] Galdino, A., and Ayala-Rincón, M. Verification of Newman's and Yokouchi Lemmas in PVS. Tech. rep., Departamento de Matemática, Universidade de Brasília, 2007. Available: `www.mat.unb.br/∼ayala/publications.html`.

[7] Huet, G. Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems. *Journal of the Association for Computing Machinery 27(4)* (1980), 797–821.

[8] Huet, G. Residual Theory in λ-calculus: A Formal development. *Jornal of Functional Programming 4(3)* (1994), 371–394.

[9] Kapur, D., and Zhang, H. An overview of Rewrite Rule Laboratory (RRL). In *Proc. Third Int. Conf. on Rewriting techniques and Applications, Chapel-Hill, NC* (April 1989), N. Dershowitz, Ed., vol. 355 of *LNCS*, Springer.

[10] Morra, C., Becker, J., Ayala-Rincón, M., and Hartenstein, R. W. FELIX: Using Rewriting-Logic for Generating Functionally Equivalent Implementations. In *15th Int. Conference on Field Programmable Logic and Applications - FPL 2005* (2005), IEEE CS, pp. 25–30.

[11] Muñoz, C. Batch proving and proof scripting in PVS. Report NIA Report No. 2007-03, NASA/CR-2007-214546, NIA-NASA Langley, National Institute of Aerospace, Hampton, VA, February 2007.

[12] Muñoz, C., and Mayero, M. Real automation in the field. ICASE Interim Report 39 NASA/CR-2001-211271, NASA Langley Research Center, NASA Langley Research Center, December 2001.

[13] Owre, S., and Shankar, N. The PVS Prelude Library. Tech. rep., SRI-CSL-03-01, Computer Science Laboratory, SRI International, Menlo Park, CA, March 2003. Available: `http://pvs.csl.sri.com/`.

[14] Owre, S., Shankar, N., Rushby, J. M., and Stringer-Calvert, D. W. J. *PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, Nov. 2001.

[15] Rasmussen, O. The church-rosser theorem in isabelle: A proof porting experiment. Tech. Rep. 364, University of Cambridge, Computer Laboratory, March 1995.

[16] Ruiz-Reina, J. L., Alonso, J. A., Hidalgo, M. J., and Martín-Mateos, F. Formalizing Rewriting in the ACL2 Theorem Prover. *Lecture Notes in Computer Science 1930* (2001).

[17] Shankar, N. A Mechanical Proof of the Church-Rosser theorem. *Journal of the ACM 35* (1988), 475–522.

[18] Shankar, N., Owre, S., Rushby, J. M., and Stringer-Calvert, D. W. J. *PVS Prover Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, Nov. 2001.

[19] SRI International, Computer Science Laboratory. *PVS System Guide - Versão 2.4*. 333 Ravenswood Avenue Menlo Park CA 94025 USA, December 2001.

[20] Vito, B. D. *Manip User's Guide, Version 1.1*. NASA Langley Research Center, Hampton, Virginia, February, 18 2003.