Computer-Supported Program Verification with PVS

Wolfgang Schreiner
Wolfgang.Schreiner@risc.uni-linz.ac.at

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
http://www.risc.uni-linz.ac.at

1. **An Overview of PVS**

2. Specifying Arrays

3. Verifying the Linear Search Algorithm

The PVS Prototype Verification System

- Integrated environment for developing and analyzing formal specs.
  - SRI (Software Research Institute) International, Menlo Park, CA.
  - Developed since 1993, current version 3.2 (November 2004).
  - Core system is implemented in Common Lisp.
  - Emacs-based frontend with Tcl/Tk-based GUI extensions.
  - Not open source, but Linux/Intel executables are freely available.
  - http://pvs.csl.sri.com
- PVS specification language.
  - Based on classical, typed higher-order logic.
  - Used to specify libraries of theories.
- PVS theorem prover.
  - Collection of basic inference rules and high-level proof strategies.
  - Applied interactively within a sequent calculus framework.
  - Proofs yield proof scripts for manipulating and replaying proofs.

Applied e.g. in the design of flight control software and real-time systems.

Theorem Proving in PVS

PVS combines aspects of interactive “proof assistants” with aspects of automatic “theorem provers”.

- Human control of the higher levels of proof development.
  - Provides a fairly intuitive interactive user interface.
    - In contrast to provers with a command-line interface only.
  - Supports an expressive specification language with a rich logic.
    - In contrast to provers supporting e.g. only first-order predicate logic.
- Automation of the the lower levels of proof elaboration.
  - Includes various decision procedures.
    - Propositional logic, theory of equality with uninterpreted function symbols, quantifier-free linear integer arithmetic with equalities and inequalities, arrays and functions with updates, model checking.
  - Supports various proof strategies and allows to define own strategies.
    - Induction over various domains, term rewriting, heuristics for proving quantified formulas, etc.

PVS is a proof assistant to some, a theorem prover to others.

Usage of PVS

For a first overview, see the "PVS System Guide".

- Develop a theory.
  - Declarations/definitions of types, constants, functions/predicates.
  - Specifies axioms (assumed) and other formulas (to be proved).
  - Theory may import from and export to other theories.
- Parse and type-check the theory.
  - Creates type-checking conditions (TCCs).
  - Need to be proved (now or later).
  - Proofs of other formulas assume truth of these TCCs.
- Prove the formulas in the theory.
  - Human-guided development of the proof.
  - Proof steps are recorded in a proof script for later use.
    - Continuing or replaying or copying proofs.
- Generate documentation.
  - Theories and proofs in PostScript, LaTeX or HTML.

Sophisticated status and change management for large-scale verification.

Developing a Theory

PVS uses the Emacs editor as its frontend.

- Starting PVS.

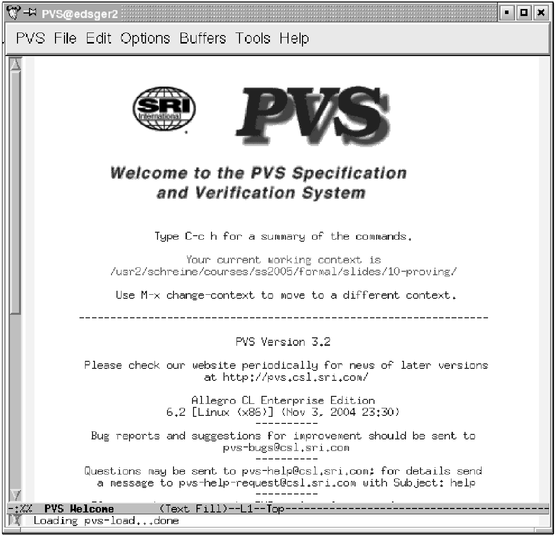
  `pvs` [*filename*.pvs] `&`

  - Each PVS session operates in a context ($\approx$ directory).
  - Files can be created in the context or imported from another context.
- Finding a PVS file or creating a new one.
  - `C-`*`key`*: Ctrl + *key*, `M-`*`key`*: Alt + *key* (Meta = Alt).

`C-x C-f`	Find an existing PVS file.
`M-x nf`	Create a new PVS file.
`M-x imf`	Import an existing PVS file from another context.

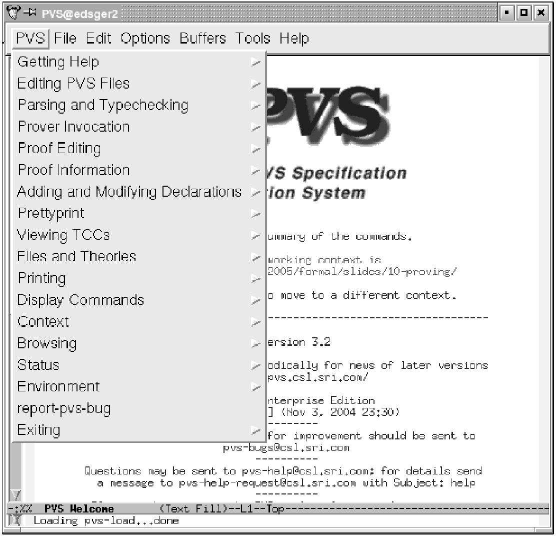
File editing as in Emacs (`C-h m` for help on the PVS mode); most commands can be also invoked from the menu bar.

PVS Startup



PVS@edsger2
PVS File Edit Options Buffers Tools Help
SRI International
PVS
Welcome to the PVS Specification
and Verification System
Type C-c h for a summary of the commands.
Your current working context is
/usr2/schreine/courses/ss2005/formal/slides/10-proving/
Use M-x change-context to move to a different context.
PVS Version 3.2
Please check our website periodically for news of later versions
at http://pvs.csl.sri.com/
Allegro CL Enterprise Edition
6.2 [Linux (x86)] (Nov 3, 2004 23:30)
Bug reports and suggestions for improvement should be sent to
pvs-bugs@csl.sri.com
Questions may be sent to pvs-help@csl.sri.com; for details send
a message to pvs-help-request@csl.sri.com with Subject: help
-:%% PVS Welcome (Text Fill)--L1--Top
Loading pvs-load...done

PVS Menu Bar


PVS@edsger2
PVS File Edit Options Buffers Tools Help
Getting Help
Editing PVS Files
Parsing and Typechecking
Prover Invocation
Proof Editing
Proof Information
Adding and Modifying Declarations
Prettyprint
Viewing TCCs
Files and Theories
Printing
Display Commands
Context
Browsing
Status
Environment
report-pvs-bug
Exiting
pvs-bugs@csl.sri.com
Questions may be sent to pvs-help@csl.sri.com; for details send
a message to pvs-help-request@csl.sri.com with Subject: help
PVS Welcome (Text Fill)--L1--Top
Loading pvs-load...done

A PVS Theory

```
% Tutorial example from PVS System Guide
sum: THEORY
  BEGIN

    % function/predicate parameter or formula variable
    n: VAR nat

    % recursive function definitions need a termination "measure"
    sum(n): RECURSIVE nat =
      (IF n = 0 THEN 0 ELSE n + sum(n-1) ENDIF)
      MEASURE (LAMBDA n: n)

    % A formula (all the same: THEOREM, LEMMA, PROPOSITION, ...)
    closed_form: THEOREM
      sum(n) = n * (n+1)/2

  END sum
```

See the "PVS Language Reference".

Parsing and Type-Checking a Theory

- Basic commands:

`M-x pa`	Parse (syntax-check) the PVS file.
`M-x tc`	Type-check PVS file and generate TCCs.
`M-x tcp`	Type-check PVS file and prove TCCs.
`M-x tccs`	View status of TCCs.

- Generated TCCs:

```
% Subtype TCC generated (at line 8, column 36) for  n - 1
    % expected type  nat
  % proved - complete
sum_TCC1: OBLIGATION FORALL (n: nat): NOT n = 0 IMPLIES n - 1 >= 0;

% Termination TCC generated (at line 8, column 32) for  sum(n - 1)
  % proved - complete
sum_TCC2: OBLIGATION FORALL (n: nat): NOT n = 0 IMPLIES n - 1 < n;
```

Proving the TCCs often proceeds fully automatically.

Proving a Formula

- For each formula F, PVS maintains a proof tree.
  - Each node of the tree denotes a proof goal.
    - Logical sequent: $A_1, A_2, \ldots \vdash B_1, B_2, \ldots$.
    - Interpretation: $(A_1 \wedge A_2 \wedge \ldots) \Rightarrow (B_1 \vee B_2 \vee \ldots)$
  - Initially the tree consists of the root node $\vdash F$ only.
- The overall task is to expand the tree to completion.
  - Every leaf goal shall denote an obviously true formula.
    - Either the consequent $B_1, B_2, \ldots$ of the goal is true,

      Consequent is empty or some B_i is true.
    - Or the antecedent $A_1, A_2, \ldots$ of the goal is false.

      Some A_i is false.
  - In each proof step, a proof rule is applied to a non-true leaf goal.
    - Either the goal is recognized as true and thus the branch is completed,
    - Or the goal becomes the parent of a number of children (subgoals).

      The conjunction of subgoals implies the parent goal.

```
{-1}   A1
[-2]   A2
        ⋮
-----------
{1}    B1
[2]    B2
        ⋮
```

Proving a Formula

- Running a Proof:

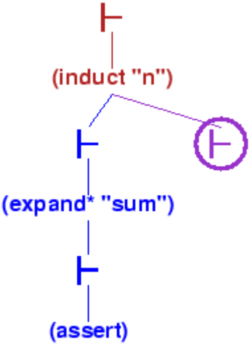
`M-x pr`	Start proof of formula
`M-x xpr`	Start proof with graphics
`M-x redo-proof`	Rerun previous proof
`M-x show-proof`	Show proof in text view
`M-x x-show-proof`	Show proof in graphics view
`M-x display-proofs-formula`	Show all proofs of formula

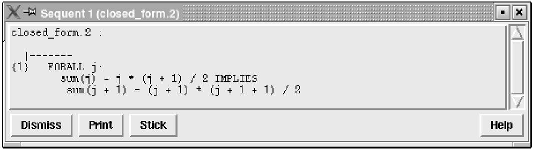
- Prover commands: `Rule?` *command*

`M-p`	Toggle back in command history ("previous")
`M-n`	Toggle forward in command history ("next")
`C-c C-c`	Interrupt current proof step
`(postpone)`	Switch to next open goal
`q`	Quit current proof attempt

While in proof mode, still files can be edited.

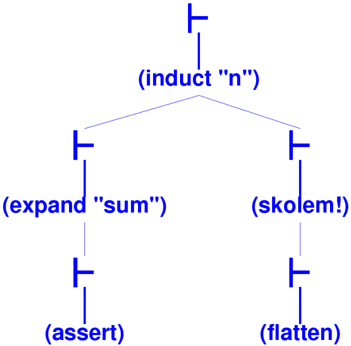
Proof in Graphics View





The circled ⊢ symbol denotes the current proof situation; by clicking on any ⊢ symbol, the corresponding proof situation is displayed.

Proof in Graphics View



Visual representation of a proof script.

Proof in Text View

```
closed_form :
  |-------
{1}   FORALL (n: nat): sum(n) = n * (n + 1) / 2

Rerunning step: (induct "n")
Inducting on n on formula 1,
this yields  2 subgoals:
closed_form.1 :
  |-------
{1}   sum(0) = 0 * (0 + 1) / 2

Rerunning step: (expand* "sum")
Expanding the definition(s) of (sum),
this simplifies to:
closed_form.1 :
  |-------
{1}   0 = 0 / 2

Rerunning step: (assert)
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of closed_form.1.
```

Proof in Text View

```
closed_form.2 :

  |-------
{1}   FORALL j:
        sum(j) = j * (j + 1) / 2 IMPLIES
         sum(j + 1) = (j + 1) * (j + 1 + 1) / 2

Rerunning step: (skolem!)
Skolemizing,
this simplifies to:
closed_form.2 :

  |-------
{1}   sum(j!1) = j!1 * (j!1 + 1) / 2 IMPLIES
       sum(j!1 + 1) = (j!1 + 1) * (j!1 + 1 + 1) / 2

Rerunning step: (flatten)
Applying disjunctive simplification to flatten sequent,
this simplifies to:
```

Proof in Text View

```
closed_form.2 :

{-1}  sum(j!1) = j!1 * (j!1 + 1) / 2
  |-------
{1}   sum(j!1 + 1) = (j!1 + 1) * (j!1 + 1 + 1) / 2

Rerunning step: (expand "sum" +)
Expanding the definition of sum,
this simplifies to:
closed_form.2 :

[-1]  sum(j!1) = j!1 * (j!1 + 1) / 2
  |-------
{1}   1 + sum(j!1) + j!1 = (2 + j!1 + (j!1 * j!1 + 2 * j!1)) / 2

Rerunning step: (assert)
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of closed_form.2.

Q.E.D.
```

Automatic Version of the Proof


```
closed_form :

  |-------
{1}   FORALL (n: nat): sum(n) = n * (n + 1) / 2

Rerunning step: (induct-and-simplify "n")
sum rewrites sum(0)
  to 0
sum rewrites sum(1 + j!1)
  to 1 + sum(j!1) + j!1
By induction on n, and by repeatedly rewriting and simplifying,
Q.E.D.

Run time  = 0.62 secs.
Real time = 1.56 secs.
```

Generating Documentation

- Basic commands:

`M-x ltt`	Create LaTeX for theory
`M-x ltv`	View LaTeX for theory
`M-x ltp`	Create LaTeX for last proof
`M-x lpv`	View LaTeX for last proof
`M-x html-pvs-file`	Create HTML for PVS file

sum: THEORY
BEGIN

n: VAR nat

sum(n): RECURSIVE nat $=$ (IF $n = 0$ THEN 0 ELSE $n + \text{sum}(n-1)$ ENDIF)
MEASURE (λ n: n)

closed_form: THEOREM sum(n) $= n \times (n+1)/2$

END sum

Generating Documentation

Verbose proof for `closed_form`.

`closed_form`:

$$\begin{array}{l|l} \hline \{1\} & \forall\ (n\colon \mathrm{nat})\colon\ \mathrm{sum}(n) = n \times (n+1)/2 \end{array}$$

Inducting on n on formula 1,

...

Expanding the definition of sum,

`closed_form.2`:

$$\begin{array}{l|l} \{\text{-}1\} & \mathrm{sum}(j') = j' \times (j'+1)/2 \\ \hline \{1\} & 1 + \mathrm{sum}(j') + j' = 2 + j' + j' \times j' + 2 \times j'/2 \end{array}$$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `closed_form.2`.

Q.E.D.

PVS Prover Commands

For details, see the "PVS Prover Guide".

- Powerful proving strategies.
  - Induction proofs: `induct-and-simplify`.
    - Combination of `induct` and repeated simplification.
  - Simple non-induction proofs: `grind`.
    - Definition expansion, arithmetic, equality, quantifier reasoning.
  - Manual quantifier proofs: `skosimp*`
    - Skolemization (`skolem!`): "let x be arbitrary but fixed".
    - Repeated simplification, if necessary starts with skolemization again.
- Installing additional rewrite rules for simplification procedures.
  - Most general: `install-rewrites`
    - Install declarations as rewrite rules to be used by `grind`.
  - More special: `auto-rewrite`, `auto-rewrite-theory`.

Try the high-level proving strategies first.

PVS Prover Commands

- Propositional formula manipulation:
  - `flatten`: remove from consequent implications and disjunctions, from antecedents conjunctions.
    - Example: to prove $A \Rightarrow B$, we assume A and prove B.
    - No branching: current goal is replaced by single new goal.
  - `split`: split in consequent conjunctions and equivalences, in antecedent disjunctions and implications, split `IF` in both.
    - Branching: current goal is decomposed into multiple subgoals.
  - `lift-if`: move `IF` to the top-level.
    - Example: $f(\texttt{IF } p \texttt{ THEN } a \texttt{ ELSE } b) \rightsquigarrow \texttt{IF } p \texttt{ THEN } f(a) \texttt{ ELSE } f(b)$.
    - Often required for further applications of `flatten` and `split`.
  - `case`: split proof into multiple cases.
    - Example: to prove A, we prove $B \Rightarrow A$ and $\neg B \Rightarrow A$.
    - Creative step: human introduces new assumption B.

Typical performed in the middle of a proof.

PVS Prover Commands

- Definition expansion.
  - `expand`: expand definition of some function or predicate.
    - Creative step: human tells to "look into definition".
- Quantifier manipulation.
  - `inst`: instantiate universal formula in antecedent or existential formula in consequent.
    - Example: We know $\forall x : A$. Thus we know $A[t/x]$.
    - `inst-cp` leaves original formula in goal for further instantiations.
    - Creative step: human introduces instantiation term t.
- Introduction of new knowledge.
  - `lemma`: add to antecedent (an instance of) a formula.
    - Formula declared in some theory is separately proved.
    - Creative step: human tells which lemma to apply.
  - `extensionality`: add to antecedent extensionality axiom for a particular type.
    - Axiom describes how to prove the equality of two objects of this type.
    - Creative step: human tells to switch "object level".

Here PVS needs human control (but may also use automatic heuristics).

1. An Overview of PVS

2. Specifying Arrays

3. Verifying the Linear Search Algorithm

Arrays as an Abstract Datatype

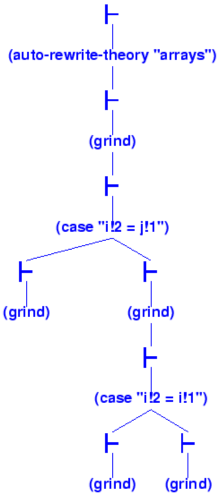
```
arrays[elem: TYPE+]: THEORY
BEGIN
  arr: TYPE+

  new:     [nat -> arr]
  length: [arr -> nat]
  put:     [arr, nat, elem -> arr]
  get:     [arr, nat -> elem]

  a, b: VAR arr
  n, i, j: VAR nat
  e: VAR elem

  length1: AXIOM
    FORALL(n): length(new(n)) = n

  length2: AXIOM
    FORALL(a, i, e):
      0 <= i AND i < length(a) IMPLIES
        length(put(a, i, e)) =
        length(a)
```

```
get1: AXIOM
  FORALL(a, i, e):
    0 <= i AND i < length(a) IMPLIES
      get(put(a, i, e), i) = e

get2: AXIOM
  FORALL(a, i, j, e):
    0 <= i AND i < length(a) AND
    0 <= j AND j < length(a) AND
    i /= j IMPLIES
      get(put(a, i, e), j) =
      get(a, j)

equality: AXIOM
  FORALL(a, b): a = b IFF
    length(a) = length(b) AND
    FORALL(i):
      0 <= i AND i < length(a)
      IMPLIES get(a,i) = get(b,i)

END arrays
```

An Expected Array Property

```
test[ elem: TYPE+ ]: THEORY
  BEGIN
    IMPORTING arrays[elem]

    a: VAR arr
    i, j: VAR nat
    e, e1, e2: VAR elem

    commutes: LEMMA
      FORALL(a, i, j, e):
         0 <= i AND i < length(a) AND
         0 <= j AND j < length(a) AND
         i /= j IMPLIES
          put(put(a, i, e1), j, e2) =
          put(put(a, j, e2), i, e1)
  END test
```

Proving the Property `commutes`



Only manual insertion of case distinctions necessary.

Arrays as Functions

```
arrays[elem: TYPE+]: THEORY
BEGIN
  arr: TYPE = [ nat, [nat -> elem] ]

  a,b: VAR arr
  n, i, j: VAR nat
  e: VAR elem

  anyelem: elem
  anyarray: arr

  new (n): arr =
    (n, (lambda n: anyelem))

  length(a): nat = a‘1

  put(a, i, e): arr =
    IF i < a‘1
      THEN (a‘1, a‘2 WITH [(i) := e])
      ELSE anyarray ENDIF
```

```
get(a, i): elem =
  IF i < a‘1
    THEN a‘2(i) ELSE anyelem ENDIF

length1: THEOREM ...
length2: THEOREM ...
get1: THEOREM ...
get2: THEOREM ...

equality: THEOREM
  FORALL(a, b): a = b IFF
    length(a) = length(b) AND
    FORALL(i):
      0 <= i AND i < length(a)
      IMPLIES get(a,i) = get(b,i)

unassigned: AXIOM
  FORALL(a, i):
    i >= a‘1
    IMPLIES a‘2(i) = anyelem
```

```
END arrays
```

Proving the Properties

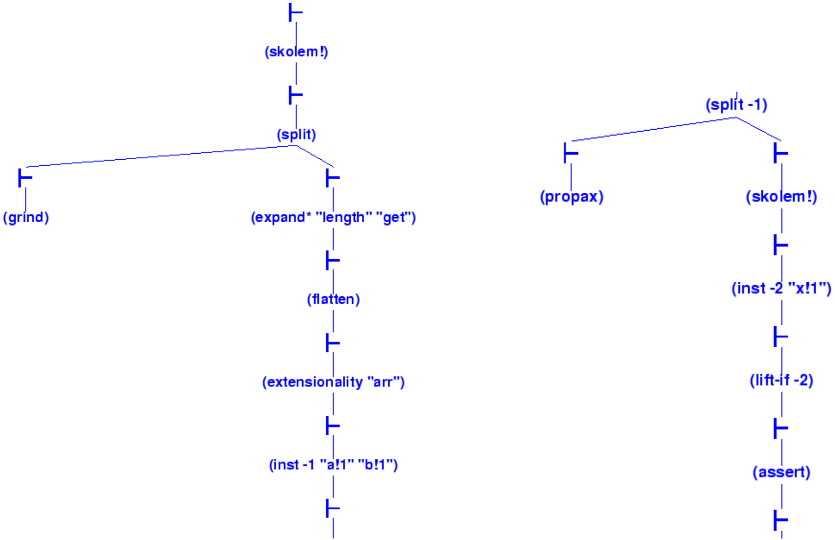
- `length1` and `length2`:

- `get1` and `get2`:

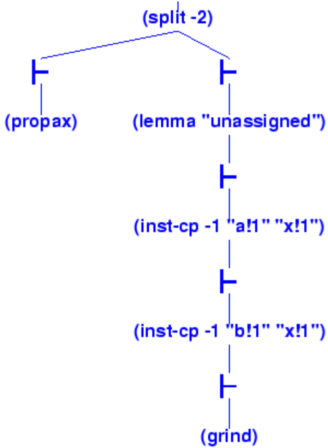
- `commutes`:

Completely automatic.

Proving the Properties: `equality`



Proving the Properties: `equality`



Manual proof control for *one* direction of the proof; this direction depends on additional lemma.

1. An Overview of PVS

2. Specifying Arrays

3. **Verifying the Linear Search Algorithm**

Linear Search

$$\{olda = a \wedge oldx = x \wedge n = \text{length}(a) \wedge i = 0 \wedge r = -1\}$$
$$\textbf{while } i < n \wedge r = -1 \textbf{ do}$$
$$\quad \textbf{if } a[i] = x$$
$$\quad\quad \textbf{then } r := i$$
$$\quad\quad \textbf{else } i := i + 1$$
$$\{a = olda \wedge$$
$$\quad ((r = -1 \wedge \forall i : 0 \leq i < \text{length}(a) \Rightarrow a[i] \neq x) \vee$$
$$\quad\ (0 \leq r < \text{length}(a) \wedge a[r] = x \wedge \forall i : 0 \leq i < r : a[i] \neq x))\}$$

By application of the rules of the Hoare calculus, we generate the necessary verification conditions.

Verification Conditions

$$
\begin{aligned}
&\mathit{Input} :\Leftrightarrow \mathit{olda} = a \wedge \mathit{oldx} = x \wedge n = \text{length}(a) \wedge i = 0 \wedge r = -1\\
&\mathit{Output} :\Leftrightarrow a = \mathit{olda} \wedge\\
&\quad ((r = -1 \wedge \forall i : 0 \le i < \text{length}(a) \Rightarrow a[i] \neq x) \vee\\
&\quad\ (0 \le r < \text{length}(a) \wedge a[r] = x \wedge \forall i : 0 \le i < r : a[i] \neq x))\\
&\mathit{Invariant} :\Leftrightarrow \mathit{olda} = a \wedge \mathit{oldx} = x \wedge n = \text{length}(a) \wedge\\
&\quad 0 \le i \le n \wedge \forall j : 0 \le j < i \Rightarrow a[j] \neq x \wedge\\
&\quad (r = -1 \vee (r = i \wedge i < n \wedge a[r] = x))
\end{aligned}
$$

$$
\begin{aligned}
&A :\Leftrightarrow \mathit{Input} \Rightarrow \mathit{Invariant}\\
&B_1 :\Leftrightarrow \mathit{Invariant} \wedge i < n \wedge r = -1 \wedge a[i] = x \Rightarrow \mathit{Invariant}[i/r]\\
&B_2 :\Leftrightarrow \mathit{Invariant} \wedge i < n \wedge r = -1 \wedge a[i] \neq x \Rightarrow \mathit{Invariant}[i+1/i]\\
&C :\Leftrightarrow \mathit{Invariant} \wedge \neg(i < n \wedge r = -1) \Rightarrow \mathit{Output}
\end{aligned}
$$

The verification conditions A, B_1, B_2, and C have to be proved.

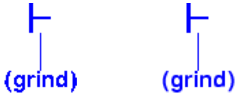
Specifying the Verification Conditions

```
linsearch[elem: TYPE+]: THEORY
BEGIN
  IMPORTING arrays[elem]

  a, olda: arr
  x, oldx: elem
  i, n: nat
  r: int

  j: VAR nat

  Input: bool =
    olda = a AND oldx = x AND n = length(a) AND i = 0 AND r = -1

  Output: bool =
    a = olda AND
    ((r = -1 AND
        (FORALL(j): 0 <= j AND j < length(a) IMPLIES get(a,j) /= x)) OR
     (0 <= r AND r < length(a) AND get(a,r) = x AND
        (FORALL(j): 0 <= j AND j < r IMPLIES get(a,j) /= x)))
```

Specifying the Verification Conditions

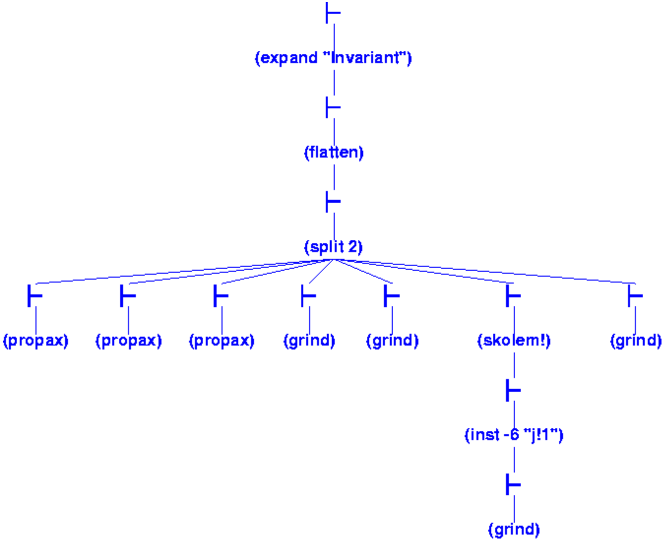
```
  Invariant(a: arr, x: elem, i: nat, n: nat, r: int): bool =
    olda = a AND oldx = x AND n = length(a) AND
    0 <= i AND i <= n AND
    (FORALL (j): 0 <= j AND j < i IMPLIES get(a,j) /= x) AND
    (r = -1 OR (r = i AND i < n AND get(a,r) = x))

  A: THEOREM
    Input IMPLIES Invariant(a, x, i, n, r)

  B1: THEOREM
    Invariant(a, x, i, n, r) AND i < n AND r = -1 AND get(a,i) = x
      IMPLIES Invariant(a, x, i, n, i)

  B2: THEOREM
    Invariant(a, x, i, n, r) AND i < n AND r = -1 AND get(a,i) /= x
      IMPLIES Invariant(a, x, i+1, n, r)

  C: THEOREM
    Invariant(a, x, i, n, r) AND NOT(i < n AND r = -1)
      IMPLIES Output
END linsearch
```

Proving the Verification Conditions: `A/B1`

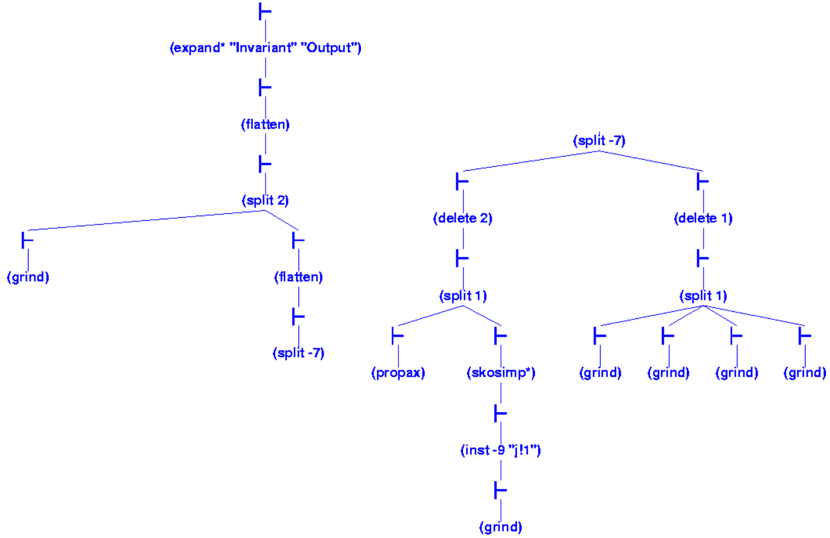


The simple ones.

Proving the Verification Conditions: B2



Proving the Verification Conditions: C



Summary

So what does this experience show us?

- Parts of a verification proof can be handled quite automatically:
  - Those that depend on skolemization, propositional simplification, expansion of definitions, rewriting, and linear arithmetic only.
  - Manual case splits may be necessary.
- More complex proofs require manual control.
  - Manual instantiation of universally quantified formulas.
  - Manual application of additional lemmas.
  - Proofs of existential formulas (not shown).

PVS can do the essentially simple but usually tedious parts of the proof; the human nevertheless has to provide the creative insight.

Other Proving Systems

- Coq: `http://coq.inria.fr`
  - LogiCal project, INRIA, France.
  - Formal proof management system (aka "proof assistant").
  - "Calculus of inductive constructions" as logical framework.
  - Decision procedures, tactics support for interactive proof development.
- Isabelle/HOL: `http://isabelle.in.tum.de`
  - University of Cambridge and Technical University Munich.
  - Isabelle: generic theorem proving environment (aka "proof assistant").
  - Isabelle/HOL: instance that uses higher order logic as framework.
  - Decisions procedures, tactics for interactive proof development.
- Theorema: `http://www.theorema.org`
  - Research Institute for Symbolic Computation (RISC), Linz.
  - Extension of computer algebra system Mathematica by support for mathematical proving.
  - Combination of generic higher order predicate logic prover with various special provers/solvers that call each other.