FME '96 Tutorial: An Introduction to Some Advanced Capabilities of PVS*

Sam Owre and John Rushby
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

1 Introduction

The specification language of PVS builds on classical higher-order logic and the simple theory of types. PVS extends simple type theory with dependent types and predicate subtypes, structures specifications into parameterized theories, and adds a construction for freely generated abstract data types. These extensions provide a surprisingly rich set of capabilities, and the purpose of this tutorial is to introduce some of them to you. The strategy mechanism of the PVS theorem prover likewise allows rather powerful capabilities to be built on the basic theorem proving resources of propositional calculus, quantifier reasoning, decision procedures, and rewriting; some of these capabilities are also described here.

This tutorial is intended to supplement, not replace, the standard PVS tutorials [6, 38], and the other examples available from our web site at `http://www.csl.sri.com/pvs/examples/`. In particular, you are recommended to work through the various "phone book" examples at the beginning of the WIFT Tutorial [6] and the "airline reservation," "noninterference," and "gcd" examples in the "less elementary" Tutorial [38] before continuing with this one (the specification and proof files can be downloaded from the URL mentioned above, and from those given with the references).

*Development of PVS was funded by SRI International.

This tutorial will eventually cover the following topics.

- Predicate subtypes and type judgements
- Dependent types and type conversions
- Inductively defined relations
- Model checking

Only the first bullet is currently documented here; the others will become available soon and can be downloaded in updates to this document from `http://www.csl.sri.com/fme96-tutorial.html`. Lectures and demonstrations based on this material cover all the topics, and some additional ones including datatypes with subtypes, and proof strategies.

1.1 Background on PVS

PVS is an environment for specification and automated verification developed in the Computer Science Laboratory of SRI International under [29–31]. PVS is made freely available under license from SRI. Implementations are currently available for Sun (Sparc) and IBM (RS6000) workstations. PVS uses Emacs, Tcl/Tk, and LaTeX to provide its interface functions. These and other requirements for running the system, instructions for downloading and installing the system, and links to much other information (including the mirror sites at the Universities of Paris VI, York, and Ulm) are available at URL `http://www.csl.sri.com/pvs.html`.

PVS has been applied to critical algorithms for fault tolerance in aircraft flight control systems [9,12,21–23,31,36,37,39,41], to software requirements and design for manned [5,8,13,14] and interplanetary [1,24] spacecraft, to real-time systems [16,42,43], to distributed systems [7] and communications protocols [15,17], to multimedia collaborations [33], to software development steps [10,40], and to hardware design [2,3,19,25–28,35,44,45] and synthesis [32,34].

Because their purpose is to introduce features of the PVS language and system, the examples in this and other tutorials are of a rather trivial nature. For more realistic examples, you are recommended to examine some of the references above. Many of those papers can be retrieved via URL `http://www.csl.sri.com/fm.html`.

2 Predicate Subtypes, Judgements, Recursion, and Proofs by Induction

PVS extends simple type theory by providing ***predicate subtypes***, which allow a predicate to characterize a subtype of a given existing type. For example, here is a specification of the subtype of the natural numbers consisting of those that are even.

```
  even_nat: TYPE = {i:nat | EXISTS (j:nat): i = 2*j}
```

We can define functions on such subtypes. For example, here is a recursive function that returns the natural number that is half the value of its even argument.

```
  ee: VAR even_nat

  half(ee): RECURSIVE nat =
     IF ee=0 THEN 0 ELSE half(ee-2) + 1 ENDIF
  MEASURE ee
```

Recursive functions are defined like ordinary functions, except that they require the keyword `RECURSIVE`, and must be supplied with a `MEASURE` that can be used to demonstrate termination. We will see more complex measures later, but the simplest is just an expression involving the arguments to the function. PVS will require that we show that the value of this expression decreases across recursive calls. PVS does this by generating the following termination TCC when we typecheck the specification.

```
% Termination TCC generated (line 10) for  half
  % unchecked
half_TCC2: OBLIGATION (FORALL (ee): NOT ee = 0 IMPLIES ee - 2 < ee)
```

This TCC is easily discharged by the default proof strategy used by `M-x tcp`. (Remember, TCCs can be examined by `M-x tcc` or `M-x ppe`.)

Typechecking this specification also causes the following TCC to be generated.

```
% Subtype TCC generated (line 10) for  ee - 2
  % unchecked
half_TCC1: OBLIGATION
   (FORALL (ee):
      NOT ee = 0 IMPLIES ee - 2 >= 0 AND EXISTS (j: nat): ee - 2 = 2 * j)
```

The explanation for this TCC is that the argument to the recursive call is `ee - 2`, and PVS cannot be sure that this is an `even_nat`, as required for the argument to `half`. In fact, since we have not defined a subtraction operator on the `even_nat`s, it is not obvious that there is any interpretation at all for the expression `ee - 2`.

However, because `ee` is an `even_nat`, it is also a `nat` (since the former is a subtype of the latter), and thus an `int` (since `nat` is a subtype of `int`). There is a subtraction operator defined for the `int`s, so PVS interprets `ee - 2` as an expression on integers. The value of the expression is therefore an integer, but it appears in a context where an `even_nat` is required. PVS allows a value of a supertype to appear where one of a subtype is required, provided the value can be shown, in the context of a left-to-right reading, to satisfy the defining predicate(s) of the subtype concerned. Here, the context is `NOT ee = 0` (since otherwise we would not be in the `ELSE` branch of the `IF`) and the defining predicates concerned are those for `nat` (specified in the PVS prelude as `{i:int | i >= 0}`), and `even_nat`. This explains the two conjuncts of the TCC above.

The default proof strategy does not succeed in proving this TCC because its simple-minded technique for guessing substitutions guesses wrong. We therefore have to prove it by hand. We begin the proof by placing the cursor on the TCC concerned in the buffer produced by `M-x tcc`, and giving the command `M-x prove`. The default strategy is a variant on `grind`. Since this failed to find the correct substitution, we give the command `(grind :if-match nil)`, which tells PVS not to guess substitutions. This produces the following sequent.

```
Trying repeated skolemization, instantiation, and if-lifting,
this simplifies to:
half_TCC1 :
{-1}    j!1 >= 0
{-2}    ee!1 = 2 * j!1
  |-------
{1}    2 * j!1 = 0
{2}    EXISTS (j: nat): 2 * j!1 - 2 = 2 * j

Rule?
```

We can see that the correct substitution is `j!1 - 1`, so we give the command `(inst + "j!1-1")` (the `+` tells PVS which formula we are talking about—we could also say `2`, but proofs are more robust if they do not mention specific formula numbers) and then finish off the proof with `(assert)`.

2.1 Simple Proofs by Induction

Now that we have discharged the TCCs for this specification, we should check the definition of `half` by attempting to prove a few plausible "challenge" theorems. The most natural one is to prove that `half` does indeed halve its argument.

```
  half_halves: THEOREM 2 * half(ee) = ee
```

Since we are dealing with a recursive function, it is natural to expect that this theorem will require a proof by induction. PVS has a large repertoire of induction schemes defined in its prelude, but we do not have one for inducting over `even_nat`s. A good exercise for more advanced users is to define and prove such an induction scheme, but in this case we can do things more simply by first defining a lemma that uses natural numbers.

```
  i: VAR nat
  half_twice: LEMMA  half(2 * i) = i
```

This generates the following TCC (necessary to ensure `2 * i` is an `even_nat`), which is proved by the default strategy.

```
% Subtype TCC generated (line 14) for  2 * i
  % untried
half_twice_TCC1: OBLIGATION
     (FORALL (i: nat): EXISTS (j: nat): 2 * i = 2 * j)
```

We can now turn to proving `half_twice` by induction. The most powerful of the standard PVS induction strategies is `induct-and-simplify`, which can usually figure out the correct induction scheme and then applies `grind`-like techniques to the various cases. Here, we can tell PVS to apply this strategy to the variable `i` with the command `(induct-and-simplify "i")`. This produces the following result.

```
half rewrites half(0)
  to  0
half rewrites half(2 + 2 * j!1)
  to  1 + half(2 * j!1)
By induction on i, and by repeatedly rewriting and simplifying,
this simplifies to:
half_twice :

  |-------
{1}    2 * i!1 = 2 * i!2
{2}    half(2 * i!2) = i!2

Rule?
```

Careful examination of formula `1` reveals that two different Skolem constants are involved (`i!1` and `i!2`); the formula would be true if the same one appeared on both sides of the equality. This situation is usually due to a faulty guess by the heuristic instantiator. We could do the instantiation by hand by giving the `:if-match nil` argument to `induct-and-simplify` (as we did earlier for `grind`) but another approach is to instruct PVS to generate all instantiations (from a single matching formula) using the command `(induct-and-simplify "i" :if-match all)`. This command succeeds in proving the lemma. Users who would like to examine more of the details should try the "`$`" form of the command, `(induct-and-simplify$ "i"`

`:if-match all)`, then examine the proof generated with `M-x edit-proof` or one of its variants (e.g., `M-x x-step-proof`). Alternatively, you can test your skill by attempting the proof by hand after invoking the induction step by `(induct "i")`.

Now that `half_twice` is proved, we can return to `half_halves`. We need to get this into a form where we can apply `half_twice`. Starting the proof and doing `(skolem!)` we gives us the following sequent.

```
Skolemizing,
this simplifies to:
half_halves :

  |-------
{1}    2 * half(ee!1) = ee!1

Rule?
```

Now `ee!1` is an `even_nat`, so we know that it can be written in the form `2*j` for some `j`, but how do we communicate this to PVS? The answer is that we need to invoke the *type predicate* for `ee!1` using the command `(typepred "ee!1")` to produce the following sequent.

```
Adding type constraints for  ee!1,
this simplifies to:
half_halves :

{-1}    EXISTS (j: nat): ee!1 = 2 * j
  |-------
[1]    2 * half(ee!1) = ee!1
```

(We could have got here directly with the single command `(skolem-typepred)`.) Next, we skolemize again.

```
Rule? (skolem!)

Skolemizing,
this simplifies to:
half_halves :

{-1}    ee!1 = 2 * j!1
  |-------
[1]    2 * half(ee!1) = ee!1

Rule?
```

(We could have got here directly by `(repeat (skolem-typepred))`, or by `(grind)`.) If we now replace the `ee!1` in formula `1` with `2 * j!1` (from the right hand side of formula `-1`) using the command `(replace -1 :hide? t)` (the `:hide t` hides the formula `-1` after it has been used), we get the following sequent in which the conclusion formula is in a form where we can use `half_twice`.

```
Replacing using formula -1,
this simplifies to:
half_halves :

  |-------
{1}    2 * half(2 * j!1) = 2 * j!1

Rule?
```

The proof can then be finished off by the command `(rewrite "half_twice")`.

2.2 Type Judgements

Now let us try another small challenge example.

```
  test: THEOREM half(ee + ee) = ee
```

Typechecking this produces another TCC of the type we have been seeing frequently in this example.

```
% Subtype TCC generated (line 21) for  ee + ee
  % unfinished
test_TCC1: OBLIGATION
    (FORALL (ee: even_nat): EXISTS (j: nat): ee + ee = 2 * j)
```

This is needed to ensure that `ee + ee` is an `even_nat`. We can imagine that many such TCCs might arise in a large specification and it would be useful if we could dispose of them all once and for all by telling PVS that `+` is closed on the `even_nat`s. One way to do this is to overload `+` with the following function definition.

```
  +(p, q: even_nat): even_nat = p + q
```

(Note, since this definition is not declared `RECURSIVE`, the `+` in the right hand side of the definition is ordinary natural number addition.) This new function definition leads to the following TCC, but that for `ee + ee` is no longer generated.

```
% Subtype TCC generated (line 12) for  p + q
  % untried
plus_TCC1: OBLIGATION
      (FORALL (p, q: even_nat): EXISTS (j: nat): p + q = 2 * j);
```

The problem with this approach is that it introduces a new function and overloads an existing symbol with an additional interpretation. What we really want to do is to inform the PVS typechecker that the *standard* `+` happens to have the property that it is closed on the `even_nat`s. This is exactly what is accomplished by the following PVS `JUDGEMENT` declaration.

```
  JUDGEMENT + HAS_TYPE [even_nat, even_nat ->  even_nat]
```

This generates the following TCC once and for all, and the TCC for `ee + ee` is no longer generated.

```
% Subtype TCC generated for  x1 + x2
  % untried
plus_TCC1: OBLIGATION
   (FORALL (x1: even_nat, x2: even_nat): EXISTS (j: nat): x1 + x2 = 2 * j)
```

The command `(grind :if-match nil)` produces the following sequent for this TCC.

```
Trying repeated skolemization, instantiation, and if-lifting,
this simplifies to:
plus_TCC1 :

{-1}    j!2 >= 0
{-2}    j!1 >= 0
{-3}    x1!1 = 2 * j!1
{-4}    x2!1 = 2 * j!2
  |-------
{1}    EXISTS (j: nat): 2 * j!1 + 2 * j!2 = 2 * j

Rule?
```

And the proof can then be finished off by the commands `(inst + "j!1+j!2")` and `(assert)`.

Some of the other TCCs that we have seen can be suppressed by adding the following judgements.

```
  JUDGEMENT 2 HAS_TYPE even_nat
  JUDGEMENT * HAS_TYPE [nat,      even_nat ->  even_nat]
  JUDGEMENT * HAS_TYPE [even_nat, nat      ->  even_nat]
```

You should try proving the TCCs generated by these, and also the theorem `test`.

Now let us introduce the type `even_int`, and define the function `ihalf` in terms of `half`.

```
  even_int: TYPE = {k: int | EXISTS (j: int): k = 2*j}
  ii: VAR even_int

  ihalf(ii): int = IF ii >= 0 THEN half(ii) ELSE -half(-ii) ENDIF
```

The TCCs from the definition of `ihalf` are left as an exercise. A plausible challenge for this definition is the following (recall that `ee` is an `even_nat`).

```
  another_test: THEOREM 2 * ihalf(ee) = ee
```

We may be surprised to find that the following TCC is generated.

```
% Subtype TCC generated (line 33) for  ee
  % untried
another_test_TCC1: OBLIGATION
      (FORALL (ee: even_nat): EXISTS (j: int): ee = 2 * j)
```

Since `ihalf` expects an `even_int`, `ee` is an `even_nat`, and `nat` is a subtype of `int`, it might seem obvious that `ee` is also an `even_int`. A little thought should convince you, that this is not so, however. The `even_int`s and the `nat`s are independent subtypes of the integers, and the `even_nat`s are a subtype of the `nat`s. The fact that the two `even` subtypes have similar-looking defining predicates is not enough to ensure that the `even_nat`s are also a subtype of the `even_int`s. If we want to establish this relation between the `even_nat`s and `even_int`s, we must explicitly specify it and discharge the corresponding proof obligation. The specification is accomplished by the following `JUDGEMENT` declaration, which is of a different kind than the ones seen previously (there are just the two kinds: `subtype_of` and `has_type`).

```
  JUDGEMENT even_nat SUBTYPE_OF even_int
```

This declaration produces the following TCC (whose proof is left as an exercise), and `another_test` no longer generates a TCC.

```
% Judgement subtype TCC generated for  x
  % untried
SUBTYPE_TCC1: OBLIGATION (FORALL (x: even_nat): EXISTS (j: int): x = 2*j)
```

Type judgements are used extensively in the PVS prelude in developing the hierarchy of numeric types and their operators.

2.3 Introducing Lemmas During Proof

The function `ihalf` requires an `even_int` as its argument, and cannot be applied to an arbitrary integer. We could define a generalized function `ghalf` that first "rounds" down its argument if it is not even, and then uses `ihalf`. The function we want to write will look like the following, where `even_int?` is the predicate that characterizes the `even_int` subtype.

```
  ghalf(n: int): int = IF even_int?(n) THEN ihalf(n) ELSE ihalf(n-1) ENDIF
```

Unfortunately, this predicate is not generated automatically by the subtype declaration for `even_int`. To get what we want, we need to declare the predicate first, and construct the subtype from it as follows.

```
  even_int?(k: int): bool = EXISTS (j: int): k = 2*j
  even_int: TYPE = (even_int?)
```

A predicate in parentheses denotes the subtype characterized by that predicate; the second of these two declarations therefore simply defines `even_int` as a synonym for `(even_int?)`. With these constructions, the definition for `ghalf` is accepted, and generates the following TCC.

```
% Subtype TCC generated (line 41) for  n - 1
  % unchecked
ghalf_TCC1: OBLIGATION
      (FORALL (n: int): NOT even_int?(n) IMPLIES even_int?(n - 1))
```

This is a generally useful property, so rather than prove it in-line, we will break it out as an explicit lemma.

```
  even_prop: THEOREM FORALL (k:int): even_int?(k) OR even_int?(k-1)
```

The proof of this theorem is quite interesting. Basically, it requires a proof by induction—but we cannot induct over the integers, so we will need to do a comparable induction over the naturals and then relate the negative and non-negative integers separately to that induction. We begin by skolemizing.

```
Rule? (SKOSIMP)
Skolemizing and flattening,
this simplifies to:
even_prop :

  |-------
{1}    even_int?(k!1)
{2}    even_int?(k!1 - 1)
```

To make progress, we now need to appeal to a lemma similar to `even_prop`, but defined on the `nat`s so that we can use proof by induction. We could break off the present proof, introduce this new lemma into the specification text, and restart, but that would be rather disrupting. Alternatively, we could modify the specification text mid-proof with the `M-x add-decl` command, but that would introduce into the specification text a lemma that is only of interest internally to the present proof. The best solution here is to introduce the lemma by means of the `case` command.

```
Rule? (CASE "FORALL (n:nat): even_int?(n) OR even_int?(n-1)")
Case splitting on
   FORALL (n:nat): even_int?(n) OR even_int?(n-1),
this yields  2 subgoals:
even_prop.1 :

{-1}    FORALL (n: nat): even_int?(n) OR even_int?(n - 1)
  |-------
[1]    even_int?(k!1)
[2]    even_int?(k!1 - 1)
```

The `case` creates two branches: in one (this one) we are allowed to assume the formula from the `case` command; in the other we will have to prove it. Thus, we have created a purely local lemma. In this branch we need a further case-split according to whether `k!1` is negative or not.

```
Rule? (CASE "k!1>=0")
Case splitting on
   k!1>=0,
this yields  2 subgoals:
even_prop.1.1 :

{-1}    k!1 >= 0
[-2]    FORALL (n: nat): even_int?(n) OR even_int?(n - 1)
  |-------
[1]    even_int?(k!1)
[2]    even_int?(k!1 - 1)
```

This branch is trivially discharged with `(inst?)` and `(assert)`, and we are then presented with the other.

```

This completes the proof of even_prop.1.1.
even_prop.1.2 :

[-1]    FORALL (n: nat): even_int?(n) OR even_int?(n - 1)
  |-------
{1}    k!1 >= 0
[2]    even_int?(k!1)
[3]    even_int?(k!1 - 1)

Rule?
```

The substitution required here is less obvious and we specify it explicitly.

```
Rule? (INST - "-k!1")
Instantiating the top quantifier in - with the terms:
 -k!1,
this yields  2 subgoals:
even_prop.1.2.1 :

{-1}    even_int?(-k!1) OR even_int?(-k!1 - 1)
  |-------
[1]    k!1 >= 0
[2]    even_int?(k!1)
[3]    even_int?(k!1 - 1)
```

The commands `(then (expand "even_int?")(prop)(skosimp))` generate two cases (from the disjunction in formula `-1`) and PVS presents us with the first of these.

```
even_prop.1.2.1.1 :

{-1}    -k!1 = 2 * j!1
  |-------
[1]    k!1 >= 0
[2]    EXISTS (j: int): k!1 = 2 * j
[3]    EXISTS (j: int): k!1 - 1 = 2 * j

Rule?
```

The obvious step here is to instantiate formula `2`.

```
Rule? (INST 2 "-j!1")
Instantiating the top quantifier in 2 with the terms:
 -j!1,
this simplifies to:
even_prop.1.2.1.1 :

[-1]    -k!1 = 2 * j!1
  |-------
[1]    k!1 >= 0
{2}    k!1 = 2 * -j!1
[3]    EXISTS (j: int): k!1 - 1 = 2 * j
```

This branch is then finished off by `(assert)` and we are presented with the other branch.

```
This completes the proof of even_prop.1.2.1.1.

even_prop.1.2.1.2 :

{-1}    -k!1 - 1 = 2 * j!1
  |-------
[1]    k!1 >= 0
[2]    EXISTS (j: int): k!1 = 2 * j
[3]    EXISTS (j: int): k!1 - 1 = 2 * j

Rule?
```

Here we should instantiate formula 3.

```
Rule? (INST 3 "-j!1-1")
Instantiating the top quantifier in 3 with the terms:
 -j!1-1,
this simplifies to:
even_prop.1.2.1.2 :

[-1]    -k!1 - 1 = 2 * j!1
  |-------
[1]    k!1 >= 0
[2]    EXISTS (j: int): k!1 = 2 * j
{3}    k!1 - 1 = 2 * (-j!1 - 1)
```

Again, `(assert)` finishes this branch of the proof, and we are presented with a TCC branch.

```
This completes the proof of even_prop.1.2.1.2.
This completes the proof of even_prop.1.2.1.

even_prop.1.2.2 (TCC):

  |-------
{1}    -k!1 >= 0
[2]    k!1 >= 0
[3]    even_int?(k!1)
[4]    even_int?(k!1 - 1)
Rule?
```

The Emacs command `M-x explain-tcc` causes PVS to describe the source of this TCC.

```
Subtype TCC
-k!1 >= 0
generated by typechecking
-k!1 : nat
```

That is, it came from our instantiation of `-k!1` (an integer) for a variable of type `nat`. This TCC branch is discharged trivially by `(assert)`, and we are finally presented with the second of the top-level branches.

```
This completes the proof of even_prop.1.2.2.
This completes the proof of even_prop.1.2.
This completes the proof of even_prop.1.

even_prop.2 :

  |-------
{1}    FORALL (n: nat): even_int?(n) OR even_int?(n - 1)
[2]    even_int?(k!1)
[3]    even_int?(k!1 - 1)

Rule?
```

Formulas `2` and `3` are now irrelevant, so we remove them by `(hide 2 3)`, and then do `(induct "n")`. This gives us two branches (the base case and the inductive step) and we are first presented with the inductive step.

```
Inducting on n,
this yields  2 subgoals:
even_prop.2.1 :

  |-------
{1}    even_int?(0) OR even_int?(0 - 1)
Rule?
```

This branch is easily discharged by the commands `(expand "even_int?")`, `(inst + "0")`, and `(assert)`, and we are presented with the inductive step.

```
This completes the proof of even_prop.2.1.

even_prop.2.2 :

  |-------
{1}    (FORALL (j: nat):
         even_int?(j) OR even_int?(j - 1)
             IMPLIES even_int?(j + 1) OR even_int?(j + 1 - 1))

Rule?
```

At this point the command `(grind :if-match nil)` does quite a bit of work for us.

```
even_int? rewrites even_int?(j)
  to  EXISTS (j_0: int): j = 2 * j_0
even_int? rewrites even_int?(j - 1)
  to  EXISTS (j_1: int): j - 1 = 2 * j_1
even_int? rewrites even_int?(1 + j)
  to  EXISTS (j_2: int): 1 + j = 2 * j_2
Trying repeated skolemization, instantiation, and if-lifting,
this simplifies to:
even_prop.2.2 :

{-1}    integer_pred(j!2)
{-2}    j!1 >= 0
{-3}    j!1 - 1 = 2 * j!2
  |-------
{1}    EXISTS (j_2: int): 1 + j!1 = 2 * j_2
```

And we can finish off the proof with `(inst + "j!2+1")` and `(assert)`.

```
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of even_prop.2.2.
This completes the proof of even_prop.2.

Q.E.D.
```

A dual property to `even_prop` is the following.

```

even_prop2: THEOREM forall (k:int): NOT (even_int?(k) AND even_int?(k-1))
```

This also has quite an interesting proof. We begin with `(skosimp)`, `(expand "even_int?")`, and `(skosimp*)`.

```
Repeatedly Skolemizing and flattening,
this simplifies to:
even_prop2 :

{-1}    k!1 = 2 * j!1
{-2}    k!1 - 1 = 2 * j!2
  |-------

Rule?
```

We now need to establish a contradiction between the formulas `-1` and `-2`. These kinds of Diophantine properties (i.e., sets of linear equalities for which we need to establish whether or not there is a solution in integers) are quite tricky to prove in PVS (because its arithmetic capabilities are not quite a complete decision procedure on the integers). The best approach is to establish or refute a simpler case that contains only a single formula, or that contains explicit non-integral terms. Here, we begin by replacing the left side of `-1` by its right hand side in `-2` and hiding `-1`.

```
Rule? (replace -1 :hide? t)
Replacing using formula -1,
this simplifies to:
even_prop2 :

{-1}    2 * j!1 - 1 = 2 * j!2
  |-------

Rule?
```

At this point `(assert)` is able to finish the proof. Sometimes it is necessary to flush the data structure maintained by the arithmetic decision procedures with `(assert :flush? t)` in order to do such proofs. (The data structures are used to accumulate the arithmetic facts discovered during a proof. Usually, the more facts that are known the better. In cases such as this one, however, it is better to focus on just the facts local to the given sequent.) On other occasions it may be necessary to introduce explicit non-integral elements by, for example `(case "j!1-1/2=j!2")` (deduced by dividing formula `-1` by 2), in which the nonexistence of integer solutions is made obvious to the prover.

Supplied with these properties of `even_int?`, it is easy to discharge the TCC from `ghalf`. The following challenge is slightly less easy, and is left as an exercise.

```
  still_another: THEOREM ghalf(5) =2
```

2.4 Strong Induction

Before we leave this example, let us examine a definition for a halving function similar to those we have seen before, but generalized to take any natural number argument.

```
  nhalf(i): RECURSIVE nat =
      IF i = 0 THEN 0 ELSIF i = 1 THEN 0 ELSE nhalf(i - 2) + 1 ENDIF
        MEASURE i
```

This definition differs from `ghalf` in that it does not coerce its argument to be even. A property of interest here is the following.

```
  twice_nhalf: THEOREM 2*nhalf(i) = i OR 2*nhalf(i) = i-1
```

To prove this directly (i.e., not resorting any properties proved earlier), it is clear that we will need proof by induction. However, if we use the default induction strategy **(induct "i")**, then go to the inductive step branch (with **(postpone)**) and skolemize, we find the following sequent.

```
Repeatedly Skolemizing and flattening,
this simplifies to:
twice_nhalf.2 :

{-1}    2 * nhalf(j!1) = j!1 OR 2 * nhalf(j!1) = j!1 - 1
  |-------
{1}    2 * nhalf(j!1 + 1) = j!1 + 1
{2}    2 * nhalf(j!1 + 1) = j!1 + 1 - 1

Rule?
```

The usual way forward in an inductive proof is to expand instances of the recursive function below the line and try to match them with instances above the line. This will not work here, however, since expanding **nhalf(j!1 + 1)** will lead to **nhalf(j!1 - 1)** and this does not match what is above the line.

The source of the problem here is that the default induction strategy uses simple induction (in which we assume the property for n and prove it for $n + 1$), and we need strong induction (in which we assume the property *for all* k less than n, and prove it for n). We can specify the induction scheme to use by specifying its name as an argument to the induction commands. Strong induction is the formula named `NAT_induction` in the PVS prelude, so we restart the proof as follows.

```
Rule? (INDUCT-AND-SIMPLIFY "i" :NAME "NAT_induction" :IF-MATCH NIL)
By induction on i, and by repeatedly rewriting and simplifying,
this simplifies to:
twice_nhalf :

{-1}    (FORALL (k: nat):
          k < j!1 IMPLIES 2 * nhalf(k) = k OR 2 * nhalf(k) = k - 1)
  |-------
{1}    2 * nhalf(j!1) = j!1
{2}    2 * nhalf(j!1) = j!1 - 1
```

Using `induct-and-simplify` rather than just `induct` takes care of some of the obvious steps, but the **:if-match nil** argument stops it finding incorrect instantiations. We specify the correct instantiation with **(inst - "j!1-2")**.

```
Instantiating the top quantifier in - with the terms:
 j!1-2,
this yields  2 subgoals:
twice_nhalf.1 :

{-1}    j!1 - 2 < j!1
          IMPLIES 2 * nhalf(j!1 - 2) = j!1 - 2
            OR 2 * nhalf(j!1 - 2) = j!1 - 2 - 1
  |-------
[1]    2 * nhalf(j!1) = j!1
[2]    2 * nhalf(j!1) = j!1 - 1

Rule?
```

This branch can now be finished off with `(expand "nhalf" +)` (to expand just the instances below the line), and `(smash)`. We are left with a TCC branch (necessary to show that our instantiation `j!1-2` is a natural number).

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

twice_nhalf.2 (TCC):

  |-------
1    j!1 - 2 >= 0
[2]    2 * nhalf(j!1) = j!1
[3]    2 * nhalf(j!1) = j!1 - 1

Rule?
```

The same combination of `(expand "nhalf" +)` and `(smash)` finishes this branch and the entire proof.

References

[1] Yoko Ampo and Robyn Lutz. Evaluation of software safety analysis using formal methods. In *Workshop for Foundation of Software Engineering (FOSE)*, Hamana-Ko, Japan, December 1995. In Japanese.

[2] R. W. Butler and J. A. Sjogren. Hardware proofs using EHDM and the RSRE verification methodology. NASA Technical Memorandum 100669, NASA Langley Research Center, 1988.

[3] Victor A. Carreño and Paul S. Miner. Specification of the IEEE-854 floating-point standard in HOL and PVS. In *HOL95: Eighth International Workshop on Higher-Order Logic Theorem Proving and Its Applications*, Aspen Grove, UT, September 1995. Category B proceedings, available from `http://lal.cs.byu.edu/lal/hol95/Bprocs/indexB.html`.

[4] Costas Courcoubetis, editor. *Computer-Aided Verification, CAV '93*, volume 697 of *Lecture Notes in Computer Science*, Elounda, Greece, June/July 1993. Springer-Verlag.

[5] Judith Crow and Ben L. Di Vito. Formalizing Space Shuttle software requirements. In *First Workshop on Formal Methods in Software Practice (FMSP '96)*, pages 40–48, San Diego, CA, January 1996. Association for Computing Machinery.

[6] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A tutorial introduction to PVS. Presented at WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida, April 1995. Available, with specification files, from `http://www.csl.sri.com/wift-tutorial.html`.

[7] Frank S. de Boer and Marten van Hulst. Local nondeterminism in asynchronously communicating processes. In FME [11].

[8] Ben L. Di Vito. Formalizing new navigation requirements for NASA's space shuttle. In FME [11].

[9] Ben L. Di Vito and Ricky W. Butler. Formal techniques for synchronized fault-tolerant systems. In C. E. Landwehr, B. Randell, and L. Simoncini, editors, *Dependable Computing for Critical Applications—3*, volume 8 of *Dependable Computing and Fault-Tolerant Systems*, pages 163–188. Springer-Verlag, Vienna, Austria, September 1992.

[10] Axel Dold. Representing, verifying and applying software development steps using the PVS system. In V. S. Alagar and Maurice Nivat, editors, *Algebraic Methodology and Software Technology, AMAST'95*, number 936 in Lecture Notes in Computer Science, pages 431–445, Montreal, Canada, July 1995. Springer-Verlag.

[11] *Formal Methods Europe FME '96*, Lecture Notes in Computer Science, Oxford, UK, March 1996. Springer-Verlag. To appear.

[12] Li Gong, Patrick Lincoln, and John Rushby. Byzantine agreement with authentication: Observations and applications in tolerating hybrid and link faults. In *Dependable Computing for Critical Applications—5*, pages 79–90, Champaign, IL, September 1995. IFIP WG 10.4, preliminary proceedings.

[13] David Hamilton, Rick Covington, and John Kelly. Experiences in applying formal methods to the analysis of software and system requirements. In *WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques*, pages 30–43, Boca Raton, FL, 1995. IEEE Computer Society.

[14] David Hamilton, Rick Covington, and Alice Lee. An experience report on requirements reliability engineering using formal methods. In *Sixth International Conference on Software Reliability Engineering, ISSRE '95*, pages 52–57, Toulouse, France, 1995. IEEE Computer Society.

[15] Klaus Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In FME [11].

[16] Jozef Hooman. Correctness of real time systems by construction. In Langmaack et al. [20], pages 19–40.

[17] Jozef Hooman. Verifying part of the ACCESS.bus protocol using PVS. In P. S. Thiagarajan, editor, *15th Conference on the Foundations of Software Technology and Theoretical Computer Science*, number 1026 in Lecture Notes in Computer Science, Bangalore, India, December 1995. Springer-Verlag.

[18] Steven D. Johnson, editor. *CHDL '95: 12th Conference on Computer Hardware Description Languages and their Applications*, Chiba, Japan, August 1995. Proceedings published in a single volume jointly with ASP-DAC '95, CHDL '95, and VLSI '95, IEEE Catalog no. 95TH8102.

[19] J. Joyce, E. Liu, J. Rushby, N. Shankar, R. Suaya, and F. von Henke. From formal verification to silicon compilation. In *IEEE Compcon*, pages 450–455, San Francisco, CA, February 1991.

[20] H. Langmaack, W.-P. de Roever, and J. Vytopil, editors. *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, Lübeck, Germany, September 1994. Springer-Verlag.

[21] Patrick Lincoln and John Rushby. Formal verification of an algorithm for interactive consistency under a hybrid fault model. In Courcoubetis [4], pages 292–304.

[22] Patrick Lincoln and John Rushby. A formally verified algorithm for interactive consistency under a hybrid fault model. In *Fault Tolerant Computing Symposium 23*, pages 402–411, Toulouse, France, June 1993. IEEE Computer Society.

[23] Patrick Lincoln and John Rushby. Formal verification of an interactive consistency algorithm for the Draper FTP architecture under a hybrid fault model. In *COMPASS '94 (Proceedings of the Ninth Annual Conference on Computer Assurance)*, pages 107–120, Gaithersburg, MD, June 1994. IEEE Washington Section.

[24] Robyn Lutz and Yoko Ampo. Experience report: Using formal methods for requirements analysis of critical spacecraft software. In *Proceedings of the 19th Annual Software Engineering Workshop*, pages 231–248, Greenbelt, MD, December 1994. NASA Goddard Space Flight Center.

[25] Steven P. Miller and Mandayam Srivas. Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods. In *WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques*, pages 2–16, Boca Raton, FL, 1995. IEEE Computer Society.

[26] Paul S. Miner. Verification of fault-tolerant clock synchronization systems. NASA Technical Paper 3349, NASA Langley Research Center, Hampton, VA, November 1993.

[27] Paul S. Miner. Defining the IEEE-854 floating-point standard in PVS. NASA Technical Memorandum 110167, NASA Langley Research Center, Hampton, VA, June 1995.

[28] Paul S. Miner, Shyamsundar Pullela, and Steven D. Johnson. Interaction of formal design systems in the development of a fault-tolerant clock synchronization circuit. In *13th Symposium on Reliable Distributed Systems*, pages 128–137, Dana Point, CA, October 1994. IEEE Computer Society.

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

[30] S. Owre, N. Shankar, and J. M. Rushby. *User Guide for the PVS Specification and Verification System.* Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993. Three volumes: Language, System, and Prover Reference Manuals; A new edition for PVS Version 2 is expected in early 1996.

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

[32] P. Sreeranga Rajan. Transformations in high level synthesis: Specification and verification. Technical Report NL-TN 118/94, Philips Research Laboratories, Eindhoven, The Netherlands, April 1994. Revised version available as SRI Technical Report SRI-CSL-94-10, October 1994.

[33] Sreeranga Rajan, P. Venkat Rangan, and Harrick M. Vin. A formal basis for structured multimedia collaborations. In *Proceedings of the 2nd IEEE International Conference on Multimedia Computing and Systems*, pages 194–201, Washington, DC, May 1995. IEEE Computer Society.

[34] Sreeranga P. Rajan. Correctness of transformations in high level synthesis. In Johnson [18], pages 597–603.

[35] H. Ruess, M. K. Srivas, and N. Shankar. Modular verification of SRT division. In Rajeev Alur and Tom Henzinger, editors, *Computer-Aided Verification, CAV '96*, Lecture Notes in Computer Science, New Brunswick, NJ, July 1996. Springer-Verlag. To appear.

[36] John Rushby. A fault-masking and transient-recovery model for digital flight-control systems. In Jan Vytopil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Kluwer International Series in Engineering and Computer Science, chapter 5, pages 109–136. Kluwer, Boston, Dordecht, London, 1993. An earlier version appeared in [46, pp. 237–257].

[37] John Rushby. A formally verified algorithm for clock synchronization under a hybrid fault model. In *Thirteenth ACM Symposium on Principles of Distributed Computing*, pages 304–313, Los Angeles, CA, August 1994. Association for Computing Machinery.

[38] John Rushby and David W. J. Stringer-Calvert. A less elementary tutorial for the PVS specification and verification system. Technical Report SRI-CSL-95-10, Computer Science Laboratory, SRI International, Menlo Park, CA, June 1995. Available, with specification files, from `http://www.csl.sri.com/csl-95-10.html`.

[39] John Rushby and Friedrich von Henke. Formal verification of algorithms for critical systems. *IEEE Transactions on Software Engineering*, 19(1):13–23, January 1993.

[40] N. Shankar. Steps towards mechanizing program transformations using PVS. *Science of Computer Programming*, 1996. To appear.

[41] Natarajan Shankar. Mechanical verification of a generalized protocol for Byzantine fault-tolerant clock synchronization. In Vytopil [46], pages 217–236.

[42] Natarajan Shankar. Verification of real-time systems using PVS. In Courcoubetis [4], pages 280–291.

[43] Jens U. Skakkebæk and N. Shankar. Towards a Duration Calculus proof assistant in PVS. In Langmaack et al. [20], pages 660–679.

[44] Mandayam K. Srivas and Steven P. Miller. Applying formal verification to a commercial microprocessor. In Johnson [18], pages 493–502.

[45] Mandayam K. Srivas and Steven P. Miller. Formal verification of the AAMP5 microprocessor. In Michael G. Hinchey and Jonathan P. Bowen, editors, *Applications of Formal Methods*, Prentice Hall International Series in Computer Science, chapter 7, pages 125–180. Prentice Hall, Hemel Hempstead, UK, 1995.

[46] J. Vytopil, editor. *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 571 of *Lecture Notes in Computer Science*, Nijmegen, The Netherlands, January 1992. Springer-Verlag.