Interactive Proof Tools

Summer Term 2003

Hans de Nivelle

`http://www.mpi-sb.mpg.de/ nivelle`

Patrick Maier

`http://www.mpi-sb.mpg.de/ maier`

What Are Interactive Proof Tools?

Semi-automated interactive theorem provers

- Prover offers commands to prove nontrivial theorems step by step.
- User has to select appropriate command (and provide arguments).
- Each step that the prover offers is logically sound.
- Granularity of individual steps varies from **substitute a term for a variable** to **try to prove theorem automatically**.

What Are They Good For?

- Formalizing mathematics:
  - rigorous development of textbook mathematics
- Formalizing hardware/software/protocols:
  - rigorous verification of properties, e. g., partial correctness, termination, secrecy, authentication, ...
- Formalizing semantics of programming languages:
  - properties of the semantics derivable
  - verification of interpreters, compilers
  - Ideally, the proof of the existence of a compiler **is** the compiler (proofs as programs).

Existing Interactive Proof Tools

Non-comprehensive list of current interactive theorem provers

- Prototype Verification System (PVS)
- Coq
- Isabelle
- Twelf

many more systems (some outdated but most have a long life)

- ELF (logical framework, predecessor to Twelf?)
- LCF (computable functions, predecessor to Isabelle?)
- HOL (higher order logic, predecessor to Isabelle?)
- nqthm (Boyer-Moore prover/ACL2, set-theory + induction)

What We Treat In This Course

Theory behind theorem provers based on higher order logic:

- Deductive calculi:
  - natural deduction, sequent calculus,
  - intuitionistic vs classical logic.
- Inductive definitions, datatypes.
- Typed lambda calculus vs higher order logic:
  - Basics (substitution, reductions, unification).
  - Formulas as types, proofs as programs (Curry-Howard-Iso).
  - Type checking vs proof checking.
  - Program synthesis from specifications.

What We Treat In This Course

Practical application of PVS and Coq to

- verification of (small) functional programs,
- verification of (simple) protocols.

The main focus of the course is on the general theory and on the (in)capabilities of interactive theorem provers. In particular, the course is **not**

- a thorough tutorial on the use of PVS, or
- a course on verification.

Overview over PVS

PVS consists of

- a specification language:
  - higher order logic + recursive definitions,
  - rich type system.
- an interactive theorem prover:
  - powerful automatic components, e. g., linear arithmetic.
- a library of existing specifications (theories):
  - lots of usefuls lemmas.

PVS by Examples — Natural Numbers I

Want to prove monotonicity of addition in the natural numbers $\mathbb{N}$, i. e., $\forall k, m, n \in \mathbb{N} : m \leq n \Rightarrow m + k \leq n + k$.

- Corresponing PVS theory:

```
nat_xmpls: THEORY
BEGIN
  k, m, n: VAR nat
  linear: THEOREM m <= n IMPLIES m + k <= n + k
END nat_xmpls
```

- Proving theorem *linear*: `prove`, `x-prove`.

- Solved by builtin decision procedure (`grind`).

PVS by Examples — Natural Numbers II

Want to prove $\forall m, n \in \mathbb{N} : m \leq n \Rightarrow m^2 \leq n^2$, i. e., in PVS:

```
quadratic: THEOREM m <= n IMPLIES m * m <= n * n
```

Sequent to prove:

```
  |-------
{1}   FORALL (m, n: nat): m <= n IMPLIES m * m <= n * n
```

Rule `(grind)` does not prove sequent.

PVS by Examples — Natural Numbers II

```
  |-------
{1}   FORALL (m, n: nat): m <= n IMPLIES m * m <= n * n
```

Step 1: Eliminate quantifiers with rule `(skolem!)`

New sequent:

```
  |-------
{1}   m!1 <= n!1 IMPLIES m!1 * m!1 <= n!1 * n!1
```

Constants `m!1` and `n!1` are called **Skolem** constants.

```
   |-------
{1}   m!1 <= n!1 IMPLIES m!1 * m!1 <= n!1 * n!1
```

Step 2: Eliminate implication with rule `(flatten)`

New sequent:

```
{-1}  m!1 <= n!1
   |-------
{1}   m!1 * m!1 <= n!1 * n!1
```

Formula `{-1}` called **antecedent**, formula `{1}` called **succedent**. Reading of sequent $\Gamma \vdash \Delta$: Conjunction of antecedent formulas (in Γ) implies disjunction of succedent formulas (in Δ).

PVS by Examples — Natural Numbers II

```
{-1}  m!1 <= n!1
  |-------
{1}   m!1 * m!1 <= n!1 * n!1
```

Step 3: Apply prelude lemma `le_times_le_pos` with rule `(rewrite "le_times_le_pos")`, where lemma `le_times_le_pos` is

$$\forall x, z \in \mathbb{R}_{\geq 0}\ \forall y, w \in \mathbb{R} : x \leq y \wedge z \leq w \Rightarrow xz \leq yw$$

`Q.E.D.`

PVS by Examples — Natural Numbers II

Other proofs of $\forall m, n \in \mathbb{N} : m \leq n \Rightarrow m^2 \leq n^2$.

- using a library of rewrite lemmas (containing `le_times_le_pos`):
  1. `(auto-rewrite-theory "real_props")`
  2. `(grind)`
- by induction on n:
  1. `(induct "n")`
  2. `(grind)`
  3. `(grind)`
- with a more automation:
  1. `(induct-and-simplify "n")`

PVS by Examples — Recursive Functions

Recursive definition of $\sum_{i=0}^{n} i$ in PVS:

```
sum(n): RECURSIVE nat =
  (IF n = 0 THEN 0 ELSE n + sum(n - 1) ENDIF)
  MEASURE (LAMBDA n: n)
```

- `sum` is function from `nat` to `nat`.
- `MEASURE` is function from arguments of `sum` (here `nat`) to a well-founded domain (here `nat`).
- Termination of `sum` ensured because `MEASURE` is strictly decreasing for each recursive call (is checked automatically).

PVS by Examples — Recursive Functions

Prove that $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$, i. e., in PVS:

```
sum_form: THEOREM sum(n) = (n * (n + 1))/2
```

Proof: `(induct-and-simplify "n")`

```
Q.E.D.
```

PVS by Examples — Rec. Functions over Lists I

Appending and reversing lists:

```
t: TYPE
xs, ys, zs: VAR list[t]

app(xs,ys): RECURSIVE list[t] =
  (IF null?(xs) THEN ys ELSE cons(car(xs),app(cdr(xs),ys)) ENDIF)
  MEASURE (LAMBDA xs,ys: length(xs))

rev(xs): RECURSIVE list[t] =
  (IF null?(xs) THEN null
                ELSE app(rev(cdr(xs)),cons(car(xs),null)) ENDIF)
  MEASURE (LAMBDA xs: length(xs))
```

Prove that `rev` is an involution, i. e., `rev(rev(xs)) = xs`.

PVS by Examples — Rec. Functions over Lists I

We want `rev_rev:  THEOREM rev(rev(xs)) = xs`

Proof: `(induct-and-simplify "xs" :rewrites ("rev_app"))`

Uses `rev_app:  LEMMA rev(app(xs,ys)) = app(rev(ys),rev(xs))`

Proof:

1. `(induct-and-simplify "xs" :rewrites ("app_null2" "app_assoc"))`
2. `(grind)`

This proof uses the lemmas `app_null2` and `app_assoc`.

PVS by Examples — Rec. Functions over Lists I

Two remaining lemmas:

- `null` is a right identity to `app`.

  ```
  app_null2:  LEMMA app(xs, null) = xs
  ```

  Proof: `(induct-and-simplify "xs")`

- `app` is associative.

  ```
  app_assoc:  LEMMA app(app(xs,ys),zs) = app(xs,app(ys,zs))
  ```

  Proof: `(induct-and-simplify "xs")`

No further lemmas required, try command `prove-theory`.

PVS by Examples — Rec. Functions over Lists II

Defining recursive 'functions' through axioms:

```
t: TYPE
x: VAR t
xs, ys, zs: VAR list[t]
app: [list[t], list[t] -> list[t]]
rev: [list[t] -> list[t]]

app_null: AXIOM app(null,ys) = ys
app_cons: AXIOM app(cons(x,xs),ys) = cons(x,app(xs,ys))

rev_null: AXIOM rev(null) = null
rev_cons: AXIOM rev(cons(x,xs)) = app(rev(xs),cons(x,null))

AUTO_REWRITE+ app_null, app_cons, rev_null, rev_cons
```

PVS by Examples — Rec. Functions over Lists II

Due to the `AUTO_REWRITE` declarations, proofs of the lemmas `app_null2`, `app_assoc`, `rev_app` and the theorem `rev_rev` can be reused.

Pitfalls with `AUTO_REWRITE`:

- Depending on the set of rewrite rules and the rewrite strategy, automatic rewriting
  - may not terminate, or
  - may terminate with unexpected results.
- Beware of conflicts arising between implicitly installed rewrite rules (e. g., through a `RECURSIVE` declaration) and explicitly (e. g., through `AUTO_REWRITE`) installed rewrite rules.

PVS by Examples — Rec. Functions over Lists II

What is the difference between recursive definitions and axioms?

- A recursive definition **defines** a function (which satisfies the given equation), thus the function **exists**.
- Axioms merely postulate the existence of a function which satisfies the equation, yet such functions **need not exist**.