Algebraic specification and verification with CafeOBJ

Part 5 – Proving and CITP

Norbert Preining

ESSLLI 2016 Bozen, August 2016

Lab time

The *rank* of a polynomial

$$p = \sum_{k=0}^{n} p_k X^k$$

is the maximum of the exponents of non-zero terms, i.e.,

$$\mathrm{rank}(p) = \max\{k : p_k \neq 0\}$$

Assuming the specification of polynomials from the lecture given. Define an operator and necessary equations so that CafeOBJ can compute arbitrary ranks.
Example: In case in integer polynomials:

```
red rank ( 3 *p X^ 2 +p X^ 1 -p 4 ) .
```

should return 2 because $p_2 = 3$ is the biggest non-zero coefficient.

Lab time II

A *vector space* V over a commutative ring R is a set with two operations, vector addition and scalar multiplication. The elements of V are called *vectors*, the elements of R (the field) *scalars*. The vector addition operators on two vectors, and the scalar multiplication operates on a scalar and a vector. The operations satisfy the following axioms:

- vector addition is associative and commutative
- there is an identity element for the vector addition
- for every vector there is the additive inverse for the vector addition
- scalar multiplication and field multiplication are compatible (a and b are scalars, $\vec{v}$ a vector): $a(b\vec{v}) = (ab)\vec{v}$
- the identity element of the field is multiplicative identity of the scalar multiplication
- scalar multiplication is distributive with respect to *both* scalar addition (addition in the field) and vector addition, that is, $(a+b)\vec{v} = (a\vec{v}) + (b\vec{v})$ and $a(\vec{v}+\vec{w}) = (a\vec{v}) + (a\vec{w})$ where a and b are scalars, and $\vec{v}$ and $\vec{w}$ are vectors.

Lab time II cont

Give a parametrized (parameter is the commutative ring) specification of vector spaces.
Example: With the view `INT-AS-CRING` from the lecture, the following code

```
open VECTORSPACE(SCALAR <= INT-AS-CRING) .
red ( 3 * 2 * (4 + 3) *v (V:Vector +v W:Vector)) .
```

should give

```
((42 *v V) +v (42 *v W)):Vector
```

as output.

Proving

Proof scores

- proofs of properties by reducing them to `true` (e.g.)
- usually written between `open` and `close`
  statements between the two are temporary and are lost after the `close` (temporary module)
- usually several modules plus several blocks of open-close

Proof scores

- proofs of properties by reducing them to `true` (e.g.)
- usually written between `open` and `close`
  statements between the two are temporary and are lost after the `close` (temporary module)
- usually several modules plus several blocks of open-close

Examples

- $x + (-x) = 0$ in group theory
- Associativity of $+$ in PNAT

Group theory

group-theory1.cafe

```
mod* GROUP {
 [ G ]
 op 0 : -> G .
 op _+_ : G G -> G { assoc } .
 op -_ : G -> G .
 var X : G .
 eq[0left] : 0 + X = X .
 eq[neginv] : (- X) + X = 0 .
}
open GROUP .
 op a : -> G .
 red a + ( - a ) .
close
```

Group theory

group-theory1.cafe

```
mod* GROUP {
  [ G ]
  op 0 : -> G .
  op _+_ : G G -> G { assoc } .
  op -_ : G -> G .
  var X : G .
  eq[0left] : 0 + X = X .
  eq[neginv] : (- X) + X = 0 .
}
open GROUP .
  op a : -> G .
  red a + ( - a ) .
close
```

...would be nice – but does not work

Group theory cont.

Why?

Group theory cont.

Why? Let us try to give a proof – can you do it?

Group theory cont.

WHY? Let us try to give a proof – can you do it? Assume we have

$$0 + a = a \tag{1}$$
$$-a + a = 0 \tag{2}$$

$$\begin{aligned}
a + -a &= 0 + a + -a && \text{by (1) right-to-left} \\
&= --a + -a + a + -a && \text{by (2) right-to-left} \\
&= --a + 0 + -a && \text{by (2)} \\
&= --a + -a && \text{by (1)} \\
&= 0 && \text{by (2)}
\end{aligned}$$

Group theory cont.

Why? Let us try to give a proof – can you do it? Assume we have

$$0 + a = a \tag{1}$$

$$-a + a = 0 \tag{2}$$

$$\begin{aligned}
a + -a &= 0 + a + -a && \text{by (1) right-to-left} \\
&= --a + -a + a + -a && \text{by (2) right-to-left} \\
&= --a + 0 + -a && \text{by (2)} \\
&= --a + -a && \text{by (1)} \\
&= 0 && \text{by (2)}
\end{aligned}$$

Why did it not work in CafeOBJ?

Group theory – better proof score

group-theory2.cafe

```
open GROUP .
 op a : -> G .
 start a + ( - a ) .
 apply -.0left at (0) .
 apply -.neginv with X = - a at [1] .
 apply reduce at term .
close
```

Group theory – better proof score

group-theory2.cafe

```
open GROUP .
 op a : -> G .
 start a + ( - a ) .
 apply -.0left at (0) .
 apply -.neginv with X = - a at [1] .
 apply reduce at term .
close
```

Still not there – why?

Group theory – even better proof score

group-theory3.cafe

```
open GROUP .
 op a : -> G .
 start a + ( - a ) .
 apply -.0left at (1) .
 apply -.neginv with X = - a at [1] .
 apply +.neginv with X = a at [2 .. 3] .
 apply reduce at term .
close
```

Group theory – even better proof score

group-theory3.cafe

```
open GROUP .
 op a : -> G .
 start a + ( - a ) .
 apply -.0left at (1) .
 apply -.neginv with X = - a at [1] .
 apply +.neginv with X = a at [2 .. 3] .
 apply reduce at term .
close
```

Where can we go from here?

Group theory – even better proof score

group-theory3.cafe

```
open GROUP .
 op a : -> G .
 start a + ( - a ) .
 apply -.0left at (1) .
 apply -.neginv with X = - a at [1] .
 apply +.neginv with X = a at [2 .. 3] .
 apply reduce at term .
close
```

Where can we go from here?
Prove that 0 is also right inverse

0 IS RIGHT INVERSE

group-theory4.cafe

```
open GROUP .
 op a : -> G .
 -- we have proven the following equation
 -- so we can add it
 eq[invneg] : a + ( - a ) = 0 .
 start a + 0 .
 apply -.neginv with X = a at (2) .
 apply +.invneg at [1 .. 2] .
 apply reduce at term .
 -- and we get a, so (a + 0) = a
close
```

Associativity of $+$ in PNAT

Associativity of $+$

Recall PNAT

```
mod! PNAT {
 [Nat]
 op 0 : -> Nat .
 op s : Nat -> Nat .
 op _+_ : Nat Nat -> Nat .
 vars X Y : Nat
 eq 0 + Y = Y .
 eq s(X) + Y = s(X + Y) .
}
```

Mathematical proof

Assume that $0 + y = y$ and $s(x) + y = s(x + y)$ for all x and y.
How do we show that $(x + y) + z = x + (y + y)$ for all x, y, and z?

Mathematical proof

Assume that $0 + y = y$ and $s(x) + y = s(x + y)$ for all x and y.
How do we show that $(x + y) + z = x + (y + y)$ for all x, y, and z?
Proof by induction:

Induction base

Show that $(0 + y) + z = 0 + (y + z)$

Mathematical proof

Assume that $0 + y = y$ and $s(x) + y = s(x + y)$ for all x and y.
How do we show that $(x + y) + z = x + (y + y)$ for all x, y, and z?
Proof by induction:

Induction base

Show that $(0 + y) + z = 0 + (y + z)$

Induction step

Show that if $(x + y) + z = x + (y + z)$, then also
$(s(x) + y) + z = s(x) + (y + z)$.

FORMAL PROOF IN CafeOBJ

```
mod ADD-ASSOC {
 pr(PNAT)
 -- theorem of constants, denote arbitrary values
 ops x y z : -> Nat .
 op addassoc : Nat Nat Nat -> Bool .
 vars X Y Z : Nat
 eq addassoc(X,Y,Z) = ((X + Y) + Z == X + (Y + Z)) .
}
```

FORMAL PROOF IN CafeOBJ

```
mod ADD-ASSOC {
 pr(PNAT)
 -- theorem of constants, denote arbitrary values
 ops x y z : -> Nat .
 op addassoc : Nat Nat Nat -> Bool .
 vars X Y Z : Nat
 eq addassoc(X,Y,Z) = ((X + Y) + Z == X + (Y + Z)) .
}
```

Induction base

```
open ADD-ASSOC .
 red addassoc(0,y,z) .
close
```

Checking induction base

```
CafeOBJ> set trace whole on
CafeOBJ> open ADD-ASSOC .
%ADD-ASSOC> red addassoc(0,y,z) .
-- reduce in %ADD-ASSOC : (addassoc(0,y,z)):Bool
[1]: (addassoc(0,y,z)):Bool
---> (((0 + y) + z) == (0 + (y + z))):Bool
[2]: (((0 + y) + z) == (0 + (y + z))):Bool
---> ((y + z) == (0 + (y + z))):Bool
[3]: ((y + z) == (0 + (y + z))):Bool
---> ((y + z) == (y + z)):Bool
[4]: ((y + z) == (y + z)):Bool
---> (true):Bool
(true):Bool
(0.000 sec for parse, 4 rewrites(0.000 sec), 12 matches)
%ADD-ASSOC> close
CafeOBJ>
```

Checking induction step

```
CafeOBJ> set trace whole off
CafeOBJ> open ADD-ASSOC .
%ADD-ASSOC> red addassoc(x,y,z) implies
              addassoc(s(x),y,z) .
-- reduce in %ADD-ASSOC : (addassoc(x,y,z) implies addassoc(s
    (x),y,z)):Bool
(true):Bool
(0.000 sec for parse, 11 rewrites(0.000 sec), 50 matches)
%ADD-ASSOC> close
CafeOBJ>
```

End of the proof

Automated Theorem Prover – CITP

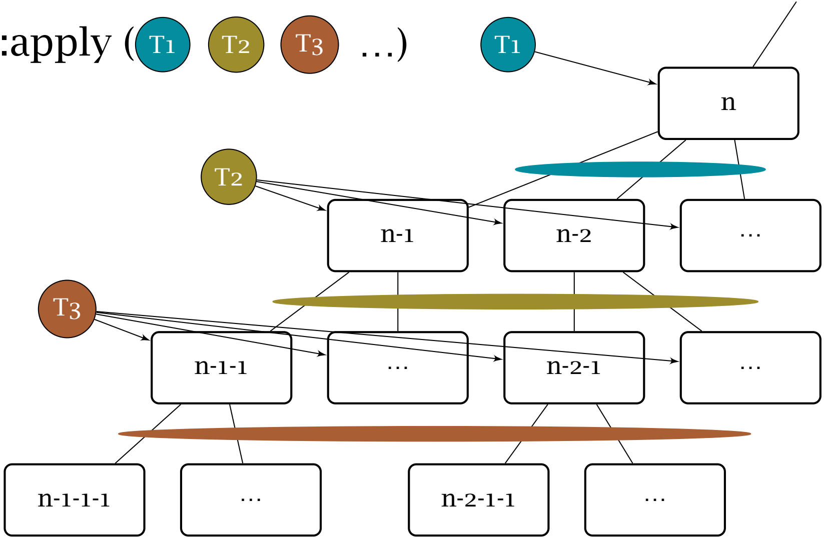
CITP in CafeOBJ

- (semi-)automated theorem prover based on induction
- original version for Maude by Daniel Gaina and Min Zhang
- ported to CafeOBJ by Toshimi Sawada
- manual available in Japanese (but outdated)

Basic steps with CITP

- define the goal to be proven
- apply tactics, either manually or automatically
- aim is to discharge all generated sub-goals

:apply (T1 T2 T3 ...)



Commutativity of Peano addition

Define Peano natural numbers

```
mod! PNAT {
 [ PZero PNzNat < PNat ]
 op 0 : -> PZero {ctor} .
 op s_ : PNat -> PNzNat {ctor} .
 op _+_ : PNat PNat -> PNat .
 eq 0 + N:PNat = N .
 eq s M:PNat + N:PNat = s(M + N) .
}
```

Commutativity of Peano addition

Define Peano natural numbers

```
mod! PNAT {
 [ PZero PNzNat < PNat ]
 op 0 : -> PZero {ctor} .
 op s_ : PNat -> PNzNat {ctor} .
 op _+_ : PNat PNat -> PNat .
 eq 0 + N:PNat = N .
 eq s M:PNat + N:PNat = s(M + N) .
}
```

Then select/open the theory/module and specify the goals:

```
open PNAT .
:goal {
 eq [lemma-1]: M:PNat + 0 = M:PNat .
 eq [lemma-2]: M:PNat + s N:PNat = s(M:PNat + N:PNat) . }
```

Commutativity of Peano addition

Define Peano natural numbers

```
mod! PNAT {
 [ PZero PNzNat < PNat ]
 op 0 : -> PZero {ctor} .
 op s_ : PNat -> PNzNat {ctor} .
 op _+_ : PNat PNat -> PNat .
 eq 0 + N:PNat = N .
 eq s M:PNat + N:PNat = s(M + N) .
}
```

Then select/open the theory/module and specify the goals:

```
open PNAT .
:goal {
 eq [lemma-1]: M:PNat + 0 = M:PNat .
 eq [lemma-2]: M:PNat + s N:PNat = s(M:PNat + N:PNat) . }
```

Give a hint that we are doing induction on `M`, and try auto-mode:

```
:ind on (M:PNat)
:auto
```

Output

```
[si]=> :goal{root}
** Generated 2 goals
[ca]=> :goal{1}
[ca] discharged: eq [lemma-1]: 0 = 0
...
[ip]=> :goal{2-2-1}
[rd]=> :goal{2-2-1}
(consumed 0.0400 sec, including 10 rewrites + 46 matches)
** All goals are successfully discharged.
```

Commutativity of addition

Now add the two lemmas to the theory:

```
mod! PNAT-L {
 inc(PNAT)
 eq [lemma-1]: M:PNat + 0 = M .
 eq [lemma-2]: M:PNat + s N:PNat = s(M + N) .
}
```

Commutativity of addition

Now add the two lemmas to the theory:

```
mod! PNAT-L {
 inc(PNAT)
 eq [lemma-1]: M:PNat + 0 = M .
 eq [lemma-2]: M:PNat + s N:PNat = s(M + N) .
}
```

and try to proof commutativity of addition

```
open PNAT-L .
:goal { eq M:PNat + N:PNat = N:PNat + M:PNat . }
:ind on (M:PNat)
:apply (SI TC RD)
```

Commutativity of addition

Now add the two lemmas to the theory:

```
mod! PNAT-L {
 inc(PNAT)
 eq [lemma-1]: M:PNat + 0 = M .
 eq [lemma-2]: M:PNat + s N:PNat = s(M + N) .
}
```

and try to proof commutativity of addition

```
open PNAT-L .
:goal { eq M:PNat + N:PNat = N:PNat + M:PNat . }
:ind on (M:PNat)
:apply (SI TC RD)
```

Not surprisingly:

```
....
} << proved >>
(consumed 0.0120 sec, including 7 rewrites + 47 matches)
** All goals are successfully discharged.
```

Proofs on lists

Use CITP to prove the following facts:

1. associativity of `@` operation in `NATLIST@`
2. `nil` is right-identity of `@`
3. add reverse operations and show double reverse is identity

Proofs on lists

Use CITP to prove the following facts:

1. associativity of `@` operation in `NATLIST@`
2. `nil` is right-identity of `@`
3. add reverse operations and show double reverse is identity

ad 1.

```
open NATLIST@ .
:goal{eq[@assoc]: (L1:NatList @ L2:NatList) @ L3:NatList
              = L1 @ (L2 @ L3) .}
:ind on (L1:NatList) .
:apply (SI TC RD) .
close
```

Proofs on lists

Use CITP to prove the following facts:

1. associativity of `@` operation in `NATLIST@`
2. `nil` is right-identity of `@`
3. add reverse operations and show double reverse is identity

ad 1.

```
open NATLIST@ .
:goal{eq[@assoc]: (L1:NatList @ L2:NatList) @ L3:NatList
              = L1 @ (L2 @ L3) .}
:ind on (L1:NatList) .
:apply (SI TC RD) .
close
```

ad 2.

```
open NATLIST@ .
:goal{eq[@ri]: L:NatList @ nil = L .}
:ind on (L:NatList) .
:apply (SI TC RD) .
close
```

Available tactics

- `SI` simultaneous induction
- `CA` case analysis after the constructors
- `TC` theorem of constants
- `IP` implication
- `RD` reduction

Additional proof tactics based on case-splitting (not-constructor based):

- `:ctf` case splitting after Boolean values
- `:csp` case splitting after a set of (arbitrary) equations

MORE EXERCISES

Prove `rev1(rev1(L)) = L` and `rev1(L) = rev2(L)` for the following module:

```
mod* NATLISTrev {
 pr(NATLIST@A)
 -- variables
 vars L L1 L2 : NatList
 var E : Nat
 -- one argument reverse operation
 op rev1 : NatList -> NatList .
 eq rev1(nil) = nil .
 eq rev1(E | L) = rev1(L) @ (E | nil) .
 -- two arguments reverse operation
 op rev2 : NatList -> NatList .
 -- auxiliary function for rev2
 op sr2 : NatList NatList -> NatList .
 eq rev2(L) = sr2(L,nil) .
 eq sr2(nil,L2) = L2 .
 eq sr2(E | L1,L2) = sr2(L1,E | L2) .
}
```

Thanks for the attention