Algebraic specification and verification with CafeOBJ

Part 3 – Exploiting AC

Norbert Preining

ESSLLI 2016 Bozen, August 2016

POLYNOMS

Aim

Make CafeOBJ usable for symbolic computation

$$x^4 + 3x^2 - 2x + 3$$

Polynoms

Aim

Make CafeOBJ usable for symbolic computation

$$x^4 + 3x^2 - 2x + 3$$

Techniques used

- associative and commutative rewriting
- reduction strategies,
- parametrized modules ('instances')

Definition of (Commutative) Rings

A *ring* is a set R with two binary operations $+$ and $\cdot$ and one unary operation $-$, satisfying the following axioms:

Definition of (Commutative) Rings

A *ring* is a set R with two binary operations $+$ and $\cdot$ and one unary operation $-$, satisfying the following axioms:

R is an abelian group wrt $+$

- associative: $(a + b) + c = a + (b + c)$
- commutative: $a + b = b + a$
- additive identity: there is $0 \in R$ such that $a + 0 = a$ for all $a \in R$
- additive inverse: $a + (-a) = 0$ for all $a \in R$

Definition of (Commutative) Rings

A *ring* is a set R with two binary operations $+$ and $\cdot$ and one unary operation $-$, satisfying the following axioms:

R is an abelian group wrt $+$

- associative: $(a + b) + c = a + (b + c)$
- commutative: $a + b = b + a$
- additive identity: there is $0 \in R$ such that $a + 0 = a$ for all $a \in R$
- additive inverse: $a + (-a) = 0$ for all $a \in R$

R is a (commutative) monoid wrt $\cdot$

- associative: $a \cdot (b \cdot c) = (a \cdot b) \cdot c$
- (commutative: $a \cdot b = b \cdot a$)
- multiplicative identity: there is $1 \in R$ such that $a \cdot 1 = 1 \cdot a = a$ for all $a \in R$

Definition of (Commutative) Rings

A *ring* is a set R with two binary operations $+$ and $\cdot$ and one unary operation $-$, satisfying the following axioms:

R is an abelian group wrt $+$

- associative: $(a + b) + c = a + (b + c)$
- commutative: $a + b = b + a$
- additive identity: there is $0 \in R$ such that $a + 0 = a$ for all $a \in R$
- additive inverse: $a + (-a) = 0$ for all $a \in R$

R is a (commutative) monoid wrt $\cdot$

- associative: $a \cdot (b \cdot c) = (a \cdot b) \cdot c$
- (commutative: $a \cdot b = b \cdot a$)
- multiplicative identity: there is $1 \in R$ such that $a \cdot 1 = 1 \cdot a = a$ for all $a \in R$

Distributivity of $\cdot$ wrt $+$

- left distributivity: $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$
- right distributivity: $(b + c) \cdot a = (b \cdot a) + (c \cdot a)$

Examples of rings

- $\mathbb{Z}$

Examples of rings

- $\mathbb{Z}$
- $\mathbb{Z}_n$ modular arithmetic, example $\mathbb{Z}_5 = \{\bar{0}, \bar{1}, \bar{2}, \bar{3}, \bar{4}\}$

Examples of rings

- $\mathbb{Z}$
- $\mathbb{Z}_n$ modular arithmetic, example $\mathbb{Z}_5 = \{\bar{0}, \bar{1}, \bar{2}, \bar{3}, \bar{4}\}$
- 2-2 matrices over the reals: $M_2(\mathbb{R}) = \left\{ \left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix} \right) | a, b, c, d \in \mathbb{R} \right\}$

Examples of rings

- $\mathbb{Z}$
- $\mathbb{Z}_n$ modular arithmetic, example $\mathbb{Z}_5 = \{\bar{0}, \bar{1}, \bar{2}, \bar{3}, \bar{4}\}$
- 2-2 matrices over the reals: $M_2(\mathbb{R}) = \left\{ \left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right) | a, b, c, d \in \mathbb{R} \right\}$
(Q: commutative?)

Examples of rings

- $\mathbb{Z}$
- $\mathbb{Z}_n$ modular arithmetic, example $\mathbb{Z}_5 = \{\bar{0}, \bar{1}, \bar{2}, \bar{3}, \bar{4}\}$
- 2-2 matrices over the reals: $M_2(\mathbb{R}) = \left\{ \left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right) | a, b, c, d \in \mathbb{R} \right\}$ (Q: commutative?)
- $\mathbb{Z}[1/n] = \{a/n^b | a \in \mathbb{Z}, b \in \mathbb{N}\}$

Examples of rings

- $\mathbb{Z}$
- $\mathbb{Z}_n$ modular arithmetic, example $\mathbb{Z}_5 = \{\bar{0}, \bar{1}, \bar{2}, \bar{3}, \bar{4}\}$
- 2-2 matrices over the reals: $M_2(\mathbb{R}) = \left\{ \left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right) | a, b, c, d \in \mathbb{R} \right\}$ (Q: commutative?)
- $\mathbb{Z}[1/n] = \{a/n^b | a \in \mathbb{Z}, b \in \mathbb{N}\}$
- $\mathbb{F}[X]$ polynomials over a ring $\mathbb{F}$:

$$\mathbb{F}[X] = p_0 + p_1 X^1 + \cdots + p_m X^m$$

such that p_i are from the ring $\mathbb{F}$ and X^k are formal expressions with $X^0 = 1$ and $X^n X^m = X^{n+m}$.

Specifying (commutative) rings in CafeOBJ

First step: operators!

Where are the sorts and operators for rings?

A *ring* is a set R with two binary operations $+$ and $\cdot$ and one unary operation $-$, satisfying the following axioms:

R is an abelian group wrt $+$

- associative: $(a + b) + c = a + (b + c)$
- commutative: $a + b = b + a$
- additive identity: there is $0 \in R$ such that $a + 0 = a$ for all $a \in R$
- additive inverse: $a + (-a) = 0$ for all $a \in R$

R is a (commutative) monoid wrt $\cdot$

- associative: $a \cdot (b \cdot c) = (a \cdot b) \cdot c$
- (commutative: $a \cdot b = b \cdot a$)
- multiplicative identity: there is $1 \in R$ such that $a \cdot 1 = 1 \cdot a = a$ for all $a \in R$

Distributivity of $\cdot$ wrt $+$

- left distributivity: $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$
- right distributivity: $(b + c) \cdot a = (b \cdot a) + (c \cdot a)$

Sorts and operators for rings

(to be filled in during class)

Sorts and operator definitions in CafeOBJ

Sorts and operator definitions in CafeOBJ

Sort(s)

```
[ Elem ]
```

Sorts and operator definitions in CafeOBJ

Sort(s)

```
 [ Elem ]
```

Operators

```
 op 0r : -> Elem .
 op 1r : -> Elem .
 op _ +r _ : Elem Elem -> Elem .
 op _ *r _ : Elem Elem -> Elem .
 op -r _ : Elem -> Elem .
}
```

Axioms (equations) for rings

Axioms for $+$, commutativity

Axioms (equations) for rings

Axioms for +, commutativity

```
eq a +r b = b +r a .
```

Axioms (equations) for rings

Axioms for +, commutativity

```
eq a +r b = b +r a .
```

Q: What will happen?

Axioms (equations) for rings

Axioms for +, commutativity

```
 eq a +r b = b +r a .
```

Q: What will happen?

```
mod* RING {
 [ Elem ]
 op _ +r _ : Elem Elem -> Elem .
 eq a:Elem +r b:Elem = b + a .
}
open RING .
red a:Elem +r b:Elem .
```

What is the problem?

Operator attributes

To overcome the infinite rewrite problem laid out above, operator attributes are available:
Details see `CafeOBJ> ? operator attr`
Possible attributes:

- `commutative` (or `comm`) – declares the operator as being commutative ($a + b = b + a$)
- `associative` (or `assoc`) – same for associative
- `l-assoc` and `r-assoc` – for left and right associativity
- `idempotence` (or `idem`) – idempotency law $a \star a = a$
- `constr` – declares the operator as constructor
- `id: <const>` defines an identity for the operator
- `prec: <int>` – precedence of the operator in the parsing ('binding strength – the smaller the stronger')
- `strat ( <int list> )` – evaluation strategy

How to use operator attribute?

Instead of writing out the commutativity law, we specify the attribute!

How to use operator attribute?

Instead of writing out the commutativity law, we specify the attribute!

```
mod* RING {
 [ Elem ]
 op _ +r _ : Elem Elem -> Elem { comm } .
}
open RING .
red a:Elem +r b:Elem .
```

How to use operator attribute?

Instead of writing out the commutativity law, we specify the attribute!

```
mod* RING {
  [ Elem ]
  op _ +r _ : Elem Elem -> Elem { comm } .
}
open RING .
red a:Elem +r b:Elem .
```

Q: What will happen?

How to use operator attribute?

Instead of writing out the commutativity law, we specify the attribute!

```
mod* RING {
  [ Elem ]
  op _ +r _ : Elem Elem -> Elem { comm } .
}
open RING .
red a:Elem +r b:Elem .
```

Q: What will happen? – nothing

```
 -- reduce in %RING : (a +r b):Elem
(a +r b):Elem
(0.0000 sec for parse, 0.0000 sec for 0 rewrites + 0 matches)
```

Abelian group

R is an abelian group wrt $+$

- associative: $(a + b) + c = a + (b + c)$
- commutative: $a + b = b + a$
- additive identity: there is $0 \in R$ such that $a + 0 = a$ for all $a \in R$
- additive inverse: $a + (-a) = 0$ for all $a \in R$

Abelian group

R is an abelian group wrt $+$

- associative: $(a + b) + c = a + (b + c)$
- commutative: $a + b = b + a$
- additive identity: there is $0 \in R$ such that $a + 0 = a$ for all $a \in R$
- additive inverse: $a + (-a) = 0$ for all $a \in R$

```
mod* RING {
 [ Elem ]
 op 0r : -> Elem
 op _ +r _ : Elem Elem -> Elem { comm assoc id: 0r }
 op -r _ : Elem -> Elem
 eq (A:Elem +r (- A)) = 0r .
}
```

Does this suffice?

Do we need more equations to reduce/rewrite (all) terms?

```
open RING .
ops a b c : -> Elem .
red a +r ( c +r b ) +r (-r ( b +r a ) ) .
```

Q: What will happen?

Does this suffice?

Do we need more equations to reduce/rewrite (all) terms?

```
open RING .
ops a b c : -> Elem .
red a +r ( c +r b ) +r (-r ( b +r a ) ) .
```

Q: What will happen?

```
%RING> red a +r ( c +r b ) +r (-r ( b +r a ) ) .
-- reduce in %RING : (a +r (c +r (b +r (-r (b +r a))))):Elem
(c):Elem
(0.0040 sec for parse, 0.0000 sec for 1 rewrites + 15 matches
    )
```

Does this suffice?

Do we need more equations to reduce/rewrite (all) terms?

```
open RING .
ops a b c : -> Elem .
red a +r ( c +r b ) +r (-r ( b +r a ) ) .
```

Q: What will happen?

```
%RING> red a +r ( c +r b ) +r (-r ( b +r a ) ) .
-- reduce in %RING : (a +r (c +r (b +r (-r (b +r a))))):Elem
(c):Elem
(0.0040 sec for parse, 0.0000 sec for 1 rewrites + 15 matches
    )
```

Q: Why

Tracing rewriting

```
%RING> set trace on
%RING> red a +r ( c +r b ) +r (-r ( b +r a ) ) .
-- reduce in %RING : (a +r (c +r (b +r (-r (b +r a))))):Elem
1>[1] rule: eq (AC:?Elem +r (A:Elem +r (-r A))) = (AC +r 0r)
   { A:Elem |-> (a +r b), AC:?Elem |-> c }
1<[1] (a +r (b +r ((-r (a +r b)) +r c))):Elem --> (c):Elem

(c):Elem
(0.0000 sec for parse, 0.0000 sec for 1 rewrites + 15 matches
    )
```

Commutative monoid and distributivity

Commutative monoid and distributivity

R *is a (commutative) monoid wrt* $\cdot$

- associative: $a \cdot (b \cdot c) = (a \cdot b) \cdot c$
- (commutative: $a \cdot b = b \cdot a$)
- multiplicative identity: there is $1 \in R$ such that $a \cdot 1 = 1 \cdot a = a$ for all $a \in R$

Commutative monoid and distributivity

R is a (commutative) monoid wrt $\cdot$

- associative: $a \cdot (b \cdot c) = (a \cdot b) \cdot c$
- (commutative: $a \cdot b = b \cdot a$)
- multiplicative identity: there is $1 \in R$ such that $a \cdot 1 = 1 \cdot a = a$ for all $a \in R$

```
op 1r : -> Elem { constr }
op _*r_ : Elem Elem -> Elem { comm assoc id: 1r }
```

COMMUTATIVE MONOID AND DISTRIBUTIVITY

R is a (commutative) monoid wrt $\cdot$

- associative: $a \cdot (b \cdot c) = (a \cdot b) \cdot c$
- (commutative: $a \cdot b = b \cdot a$)
- multiplicative identity: there is $1 \in R$ such that $a \cdot 1 = 1 \cdot a = a$ for all $a \in R$

```
op 1r : -> Elem { constr }
op _*r_ : Elem Elem -> Elem { comm assoc id: 1r }
```

Distributivity of $\cdot$ wrt $+$

- left distributivity: $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$
- right distributivity: $(b + c) \cdot a = (b \cdot a) + (c \cdot a)$

Commutative monoid and distributivity

R is a (commutative) monoid wrt $\cdot$

- associative: $a \cdot (b \cdot c) = (a \cdot b) \cdot c$
- (commutative: $a \cdot b = b \cdot a$)
- multiplicative identity: there is $1 \in R$ such that $a \cdot 1 = 1 \cdot a = a$ for all $a \in R$

```
op 1r : -> Elem { constr }
op _*r_ : Elem Elem -> Elem { comm assoc id: 1r }
```

Distributivity of $\cdot$ wrt $+$

- left distributivity: $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$
- right distributivity: $(b + c) \cdot a = (b \cdot a) + (c \cdot a)$

```
vars A B C : Elem .
eq: (A *r (B +r C)) = (A *r B) +r (A *r C) .
```

Necessary lemma for rings

Lemma $\forall a \in R : a \cdot 0 = 0 \cdot a = 0$

Necessary lemma for rings

Lemma $\forall a \in R : a \cdot 0 = 0 \cdot a = 0$

In CafeOBJ

```
%CRING> red a:Elem *r 0r .
-- reduce in %CRING : (a *r 0r):Elem
(0r *r a):Elem
%CRING>
```

Necessary lemma for rings

Lemma

$$\forall a \in R : a \cdot 0 = 0 \cdot a = 0$$

In CafeOBJ

```
%CRING> red a:Elem *r 0r .
-- reduce in %CRING : (a *r 0r):Elem
(0r *r a):Elem
%CRING>
```

Proof

$$\begin{aligned} a \cdot 0 &= a \cdot 0 + a \cdot 0 - a \cdot 0 \\ &= a \cdot (0 + 0) - a \cdot 0 \\ &= a \cdot 0 - a \cdot 0 \\ &= 0 \end{aligned}$$

Necessary lemma for rings

Lemma

$$\forall a \in R : a \cdot 0 = 0 \cdot a = 0$$

In CafeOBJ

```
%CRING> red a:Elem *r 0r .
-- reduce in %CRING : (a *r 0r):Elem
(0r *r a):Elem
%CRING>
```

Proof

$$\begin{aligned} a \cdot 0 &= a \cdot 0 + a \cdot 0 - a \cdot 0 \\ &= a \cdot (0 + 0) - a \cdot 0 \\ &= a \cdot 0 - a \cdot 0 \\ &= 0 \end{aligned}$$

Additional axiom/equation

```
 eq a:Elem *r 0r = 0r .
```

Adding binary minus and equality

To simply be able to write $a - b$ instead of $a + (-b)$ we introduce a binary minus:

```
op _-r_ : Elem Elem -> Elem
eq (A:Elem -r B:Elem) = ( A +r (-r B) ) .
```

Adding binary minus and equality

To simply be able to write $a - b$ instead of $a + (-b)$ we introduce a binary minus:

```
op _-r_ : Elem Elem -> Elem
eq (A:Elem -r B:Elem) = ( A +r (-r B) ) .
```

For equality we use reducability as equality

```
eq (A:Elem = B:Elem) = (A == B) .
```

Rewrite rules for unary minus

We need to give additional rewrite rules for unary minus to decide equations. We settle on the following normal form:

- minus are pushed into additions
- minus are pulled outside of multiplications

Rewrite rules for unary minus

We need to give additional rewrite rules for unary minus to decide equations. We settle on the following normal form:

- minus are pushed into additions
- minus are pulled outside of multiplications

```
eq (-r (A:Elem +r B:Elem)) = (-r A) +r (-r B) .
eq (-r A:Elem) *r B:Elem = -r (A *r B) .
eq (-r (-r A:Elem)) = A .
```

Putting it all together

```
mod* CRING {
 [ Elem ]
 op 0r : -> Elem { constr }
 op 1r : -> Elem { constr }
 op _ +r _ : Elem Elem -> Elem { comm assoc id: 0r prec: 33 }
      .
 op -r _ : Elem -> Elem { prec: 32 } .
 op _ -r _ : Elem Elem -> Elem { prec: 32 } .
 op _ *r _ : Elem Elem -> Elem { comm assoc id: 1r prec: 31 }
      .

 eq ( A:Elem -r B:Elem ) = ( A +r ( -r B ) ) .
 eq (A:Elem +r (-r A)) = 0r .
 eq (A:Elem *r (B:Elem +r C:Elem)) = (A *r B) +r (A *r C) .
 eq (A:Elem *r 0r) = 0r .
 eq (A:Elem = B:Elem) = (A == B) .
 eq (-r (A:Elem +r B:Elem)) = (-r A) +r (-r B) .
 eq (-r A:Elem) *r B:Elem = -r (A *r B) .
 eq (-r (-r A:Elem)) = A .
}
```

Polynomials

Going back to Polynomials

$\mathbb{F}[X]$ polynomials over a ring $\mathbb{F}$:

$$\mathbb{F}[X] = p_0 + p_1 X^1 + \cdots + p_m X^m$$

such that p_i are from the ring $\mathbb{F}$ and X^k are formal expressions with $X^0 = 1$ and $X^n X^m = X^{n+m}$.

Going back to Polynomials

$\mathbb{F}[X]$ polynomials over a ring $\mathbb{F}$:

$$\mathbb{F}[X] = p_0 + p_1 X^1 + \cdots + p_m X^m$$

such that p_i are from the ring $\mathbb{F}$ and X^k are formal expressions with $X^0 = 1$ and $X^n X^m = X^{n+m}$.

```
mod! POLYNOMIAL ( COEFF :: RING ) {
  pr(INT)
  pr(CRING * { ... }
  [ Elem < Poly ]
  op X^_ : Nat -> Poly
  ...
}
```

Polynomials as ring

The polynomials form a ring, so instead of rewriting the set of axioms for rings, we include the ring algebra and rename sorts and operators:

Polynomials as ring

The polynomials form a ring, so instead of rewriting the set of axioms for rings, we include the ring algebra and rename sorts and operators:

```
pr(CRING * { sort Elem -> Poly,
             op _+r_ -> _+p_,
             op -r_ -> -p_,
             op _-r_ -> _-p_,
             op _*r_ -> _*p_,
             op 0r -> 0p,
             op 1r -> 1p })
```

Polynomials as ring

The polynomials form a ring, so instead of rewriting the set of axioms for rings, we include the ring algebra and rename sorts and operators:

```
pr(CRING * { sort Elem -> Poly,
             op _+r_ -> _+p_,
             op -r_ -> -p_,
             op _-r_ -> _-p_,
             op _*r_ -> _*p_,
             op 0r -> 0p,
             op 1r -> 1p })
```

WARNING Two instances of ring in the algebra of poynomials: one is the ring of polynomials (where the operators are renamed from `+r` to `+p` etc), and one is the ring of coefficients which is a parameter to the module!

Remaining properties (axioms) for polynomials

Properties of the formal terms:

Remaining properties (axioms) for polynomials

Properties of the formal terms:

- $X^0 = 1$
- $X^n X^m = X^{n+m}$
- $rX^n + sX^n = (r+s)X^n$ (plus extra rules for $X^n + sX^n$ etc)

Remaining properties (axioms) for polynomials

Properties of the formal terms:

- $X^0 = 1$
- $X^n X^m = X^{n+m}$
- $rX^n + sX^n = (r+s)X^n$ (plus extra rules for $X^n + sX^n$ etc)

Properties of the computations:

- switch between polynomial and coefficient minus
- identifications of identity elements
- getting rid of superfluous 1

Axioms for polynoms

```
eq (I1 *p I2) = (I1 *r I2) . --ring elem mult.
eq (IP *p 0r) = 0r . -- as with the ring
-- properties of the formal terms
eq ( X^ 0 ) = 1p .
eq ( ( X^ N ) *p ( X^ M ) ) = X^ ( N + M ) .
eq ( I1 *p ( X^ N ) ) +p ( I2 *p ( X^ N ) ) =
   ( I1 +r I2 ) *p ( X^ N ) .
-- switch - from poly to ring
eq -p (I *p IP1) = (-r I) *p IP1 .
-- special treatment of missing coeff
eq ( X^ N ) +p ( I2 *p ( X^ N ) ) =
   ( I2 +r 1r ) *p ( X^ N ) .
eq ( -p ( X^ N ) ) +p ( I2 *p ( X^ N ) ) =
   ( I2 -r 1r ) *p ( X^ N ) .
-- identification of identity elements
eq 1p = 1r .
eq 0p = 0r .
-- getting rid of unnecessary 1
eq (1r *p X^ N) = X^ N .
```

Instantiating polynomials

We need *views* to instantiate polynomials - homomorphisms from the actual algebra to the *pattern algebra*:

Instantiating polynomials

We need *views* to instantiate polynomials - homomorphisms from the actual algebra to the *pattern algebra*:
Example: view the integers as a `CRING`:

Instantiating polynomials

We need *views* to instantiate polynomials - homomorphisms from the actual algebra to the *pattern algebra*:
Example: view the integers as a `CRING`:

```
view INT-AS-CRING from CRING to INT {
 sort Elem -> Int,
 op 0r -> 0,
 op 1r -> 1,
 op _+r_ -> _+_,
 op _*r_ -> _*_,
 op -r_ -> -_,
 op _-r_ -> _-_
}
```

Playing around with polynoms

```
open POLYNOMIAL(COEFF <= INT-AS-CRING) .
red ( 3 *p X^ 2 ) +p ( 5 *p X^ 2 ) .
red 4 *p X^ 2 -p ( 2 *p X^ 2 ) .
red ( 3 *p X^ 1 *p 4 *p X^ 3 ) .
red ( 3 *p X^ 1 *p -4 *p X^ 3 ) .
red ( ( 3 *p X^ 2 +p X^ 1 +p 2 ) *p ( X^ 1 +p 1 ) ) .
red ( ( 3 *p X^ 2 +p X^ 1 +p 2 ) *p ( X^ 1 -p 1 ) ) .
close
```

Rational polynomials

```
view RAT-AS-CRING from CRING to RAT { ... }
```

Rational polynomials

```
view RAT-AS-CRING from CRING to RAT { ... }
```

```
open POLYNOMIAL(COEFF <= RAT-AS-CRING) .
red ( ( 3/2 *p X^ 2 +p X^ 1 +p 2/5 ) *p ( X^ 1 -p 3/2 ) ) .
red ( X^ 3 -p X^ 1 +p 5/3 ) *p ( X^ 2 +p 2/9 *p X^ 1 -p 7/3 )
     .
```

Summary and open questions (preliminary)

Summary and open questions (preliminary)

- renaming of polynomial operators
  nice idea, but breaks rewriting at the moment due to infinite loops

Summary and open questions (preliminary)

- renaming of polynomial operators
  nice idea, but breaks rewriting at the moment due to infinite loops
- manual proof of $a \cdot 0 = 0$
  inverse application of rules, mixture with AC?

Summary and open questions (preliminary)

- renaming of polynomial operators
  nice idea, but breaks rewriting at the moment due to infinite loops

- manual proof of $a \cdot 0 = 0$
  inverse application of rules, mixture with AC?

- completeness of the rewrite systems?

Summary and open questions (preliminary)

- renaming of polynomial operators
  nice idea, but breaks rewriting at the moment due to infinite loops
- manual proof of $a \cdot 0 = 0$
  inverse application of rules, mixture with AC?
- completeness of the rewrite systems?
- AC rewriting and overloading of operators – tricky!

Summary and open questions (preliminary)

- renaming of polynomial operators
  nice idea, but breaks rewriting at the moment due to infinite loops
- manual proof of $a \cdot 0 = 0$
  inverse application of rules, mixture with AC?
- completeness of the rewrite systems?
- AC rewriting and overloading of operators – tricky!
- mathematical practice and formal (absolutely) proofs are different

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.

Behavioral specification

Example: flags in programming languages

Assume we want to specify an abstract notion of flags, that can be realized in various ways (booleans, natural numbers, etc).

Example: flags in programming languages

Assume we want to specify an abstract notion of flags, that can be realized in various ways (booleans, natural numbers, etc).
Necessary operations:

Example: flags in programming languages

Assume we want to specify an abstract notion of flags, that can be realized in various ways (booleans, natural numbers, etc).
Necessary operations:

- raise or set a flag
- lower or clear a flag
- change or switch a flag
- check for a set flag

Example: flags in programming languages

Assume we want to specify an abstract notion of flags, that can be realized in various ways (booleans, natural numbers, etc).
Necessary operations:

- raise or set a flag
- lower or clear a flag
- change or switch a flag
- check for a set flag

Required properties:

Example: flags in programming languages

Assume we want to specify an abstract notion of flags, that can be realized in various ways (booleans, natural numbers, etc).
Necessary operations:

- raise or set a flag
- lower or clear a flag
- change or switch a flag
- check for a set flag

Required properties:

- after raising a flag, checking it returns true
- after lowering a flag, checking it returns false
- after changing a flag, checking it returns the opposite

Example: flags in programming languages

Assume we want to specify an abstract notion of flags, that can be realized in various ways (booleans, natural numbers, etc).
Necessary operations:

- raise or set a flag
- lower or clear a flag
- change or switch a flag
- check for a set flag

Required properties:

- after raising a flag, checking it returns true
- after lowering a flag, checking it returns false
- after changing a flag, checking it returns the opposite

Consequences that should be obtained:

- two times changing a flag returns it to the original state

Example: flags in programming languages

Assume we want to specify an abstract notion of flags, that can be realized in various ways (booleans, natural numbers, etc).
Necessary operations:

- raise or set a flag
- lower or clear a flag
- change or switch a flag
- check for a set flag

Required properties:

- after raising a flag, checking it returns true
- after lowering a flag, checking it returns false
- after changing a flag, checking it returns the opposite

Consequences that should be obtained:

- two times changing a flag returns it to the original state

Q: What do you think?

Possible implementation in CafeOBJ

```
mod* FLAG {
 [ Flag ]
 op raise _ : Flag -> Flag .
 op lower _ : Flag -> Flag .
 op change _ : Flag -> Flag .

 op is-up?_ : Flag -> Bool .
 eq is-up? raise F:Flag = true .
 eq is-up? lower F:Flag = false .
 eq is-up? change F:Flag = not is-up? F .
}
mod! FLAGIMPLEMENTATION ( X :: FLAG ) { }
```

Possible implementation in CafeOBJ

```
mod* FLAG {
 [ Flag ]
 op raise _ : Flag -> Flag .
 op lower _ : Flag -> Flag .
 op change _ : Flag -> Flag .

 op is-up?_ : Flag -> Bool .
 eq is-up? raise F:Flag = true .
 eq is-up? lower F:Flag = false .
 eq is-up? change F:Flag = not is-up? F .
}
mod! FLAGIMPLEMENTATION ( X :: FLAG ) { }
```

What we expect is something like:

```
view FOOBAR-AS-FLAG from FLAG to FOOBAR { ... }
open FLAGIMPLEMENTATION(X <= FOOBAR-AS-FLAG) .
red change-foobar change-foobar F = F .
```

POSSIBLE IMPLEMENTATION IN CafeOBJ

```
mod* FLAG {
 [ Flag ]
 op raise _ : Flag -> Flag .
 op lower _ : Flag -> Flag .
 op change _ : Flag -> Flag .

 op is-up?_ : Flag -> Bool .
 eq is-up? raise F:Flag = true .
 eq is-up? lower F:Flag = false .
 eq is-up? change F:Flag = not is-up? F .
}
mod! FLAGIMPLEMENTATION ( X :: FLAG ) { }
```

What we expect is something like:

```
view FOOBAR-AS-FLAG from FLAG to FOOBAR { ... }
open FLAGIMPLEMENTATION(X <= FOOBAR-AS-FLAG) .
red change-foobar change-foobar F = F .
```

Q: What do you think?

Boolean as flags

First implementation: Booleans

```
mod! BOOLFLAG {
 pr(BOOL)
 ** operators to be used as representations
 ** for flags
 op raise-bool _ : Bool -> Bool .
 op lower-bool _ : Bool -> Bool .
 op change-bool _ : Bool -> Bool .
 op is-up?-bool _ : Bool -> Bool .

 eq raise-bool F:Bool = true .
 eq lower-bool F:Bool = false .
 eq change-bool F:Bool = not F .
 eq is-up?-bool X:Bool = X .
}
```

Boolean as flags

First implementation: Booleans

```
mod! BOOLFLAG {
 pr(BOOL)
 ** operators to be used as representations
 ** for flags
 op raise-bool _ : Bool -> Bool .
 op lower-bool _ : Bool -> Bool .
 op change-bool _ : Bool -> Bool .
 op is-up?-bool _ : Bool -> Bool .

 eq raise-bool F:Bool = true .
 eq lower-bool F:Bool = false .
 eq change-bool F:Bool = not F .
 eq is-up?-bool X:Bool = X .
}
```

Looks fine – or?

Using the implementation

Using an implementation means instantiating the flag implementation module with an actual implementation, and mapping the relevant operators.

Using the implementation

Using an implementation means instantiating the flag implementation module with an actual implementation, and mapping the relevant operators.

```
view BOOL-AS-FLAG from FLAG to BOOLFLAG {
 sort Flag -> Bool,
 op raise_ -> raise-bool_ ,
 op lower_ -> lower-bool_,
 op change_ -> change-bool_,
 op is-up?_ -> is-up?-bool_
}
open FLAGIMPLEMENTATION(X <= BOOL-AS-FLAG) .
```

Using the implementation

Using an implementation means instantiating the flag implementation module with an actual implementation, and mapping the relevant operators.

```
view BOOL-AS-FLAG from FLAG to BOOLFLAG {
 sort Flag -> Bool,
 op raise_ -> raise-bool_ ,
 op lower_ -> lower-bool_,
 op change_ -> change-bool_,
 op is-up?_ -> is-up?-bool_
}
open FLAGIMPLEMENTATION(X <= BOOL-AS-FLAG) .
```

Now let us check whether the double switch property holds:

```
red change-bool change-bool F:Bool = F .
```

Using the implementation

Using an implementation means instantiating the flag implementation module with an actual implementation, and mapping the relevant operators.

```
view BOOL-AS-FLAG from FLAG to BOOLFLAG {
 sort Flag -> Bool,
 op raise_ -> raise-bool_ ,
 op lower_ -> lower-bool_,
 op change_ -> change-bool_,
 op is-up?_ -> is-up?-bool_
}
open FLAGIMPLEMENTATION(X <= BOOL-AS-FLAG) .
```

Now let us check whether the double switch property holds:

```
red change-bool change-bool F:Bool = F .
```

Q: What do you think is the outcome?

Are we happy with that?

Another implementation: Natural numbers

We want to implement flags via natural numbers, and somehow keep track of costs of raising and lowering and changing.

Another implementation: Natural numbers

We want to implement flags via natural numbers, and somehow keep track of costs of raising and lowering and changing.

Our intended operations and semantics are:

- a flag is raised if the counter is even

Another implementation: Natural numbers

We want to implement flags via natural numbers, and somehow keep track of costs of raising and lowering and changing.

Our intended operations and semantics are:

- a flag is raised if the counter is even
- raising the flag multiplies the counter by 2

Another implementation: Natural numbers

We want to implement flags via natural numbers, and somehow keep track of costs of raising and lowering and changing.

Our intended operations and semantics are:

- a flag is raised if the counter is even
- raising the flag multiplies the counter by 2
- lowering the flag multiplies the counter by 2 and adds 1

Another implementation: Natural numbers

We want to implement flags via natural numbers, and somehow keep track of costs of raising and lowering and changing.

Our intended operations and semantics are:

- a flag is raised if the counter is even
- raising the flag multiplies the counter by 2
- lowering the flag multiplies the counter by 2 and adds 1
- changing the flag adds 1

Another implementation: Natural numbers

We want to implement flags via natural numbers, and somehow keep track of costs of raising and lowering and changing.

Our intended operations and semantics are:

- a flag is raised if the counter is even
- raising the flag multiplies the counter by 2
- lowering the flag multiplies the counter by 2 and adds 1
- changing the flag adds 1

Q: Is this a ‘flag’ in our interpretation?

Implementation in CafeOBJ

Implementation in CafeOBJ

```
mod! PNATFLAG {
  [ PNat ]
  op s _ : PNat -> PNat .
  op 0 : -> PNat .
  ...
  eq (N:PNat = M:PNat) = (N == M) .
  ...
  ** operators to be used as representations
  ** for flags
  op raise-pnat _ : PNat -> PNat .
  op lower-pnat _ : PNat -> PNat .
  op change-pnat _ : PNat -> PNat .
  op is-up?-pnat _ : PNat -> Bool .

  eq raise-pnat F:PNat = times2 F .
  eq lower-pnat F:PNat = s times2 F .
  eq change-pnat F:PNat = s F .
  eq is-up?-pnat F:PNat = even F .
}
```

And what about our double switch property?

???

And what about our double switch property?

???

```
view PNAT-AS-FLAG from FLAG to PNATFLAG {
 sort Flag -> PNat,
 op raise_ -> raise-pnat_ ,
 op lower_ -> lower-pnat_,
 op change_ -> change-pnat_,
 op is-up?_ -> is-up?-pnat_
}
open FLAGIMPLEMENTATION(X <= PNAT-AS-FLAG) .
red change-pnat change-pnat N:PNat = N .
close .
```

What went wrong?

Code-wise

```
set trace whole on
%FLAGIMPLEMENTATION(X <= PNAT-AS-FLAG)> -- reduce in %
    FLAGIMPLEMENTATION(X <= PNAT-AS-FLAG) : ((change-pnat (
    change-pnat N)) = N):Bool
[1]: ((change-pnat (change-pnat N)) = N):Bool
---> ((s (change-pnat N)) = N):Bool
[2]: ((s (change-pnat N)) = N):Bool
---> ((s (s N)) = N):Bool
[3]: ((s (s N)) = N):Bool
---> ((s (s N)) == N):Bool
[4]: ((s (s N)) == N):Bool
---> (false):Bool
(false):Bool
(0.0000 sec for parse, 0.0000 sec for 4 rewrites + 4 matches)
```

CODE-WISE

```
set trace whole on
%FLAGIMPLEMENTATION(X <= PNAT-AS-FLAG)> -- reduce in %
    FLAGIMPLEMENTATION(X <= PNAT-AS-FLAG) : ((change-pnat (
    change-pnat N)) = N):Bool
[1]: ((change-pnat (change-pnat N)) = N):Bool
---> ((s (change-pnat N)) = N):Bool
[2]: ((s (change-pnat N)) = N):Bool
---> ((s (s N)) = N):Bool
[3]: ((s (s N)) = N):Bool
---> ((s (s N)) == N):Bool
[4]: ((s (s N)) == N):Bool
---> (false):Bool
(false):Bool
(0.0000 sec for parse, 0.0000 sec for 4 rewrites + 4 matches)
```

But are we interested in the actual value?

What is of interest?

Are we interested in the actual value? – NO! Only in the

observation

whether the flag is raised or not.

What is of interest?

Are we interested in the actual value? – NO! Only in the

observation

whether the flag is raised or not.

In particular, this is the problem

```
eq (N = M) = (N == M) .
```

Definition of equality via ‘syntactic’/‘evaluation-style’ equality.

What is of interest?

Are we interested in the actual value? – NO! Only in the

observation

whether the flag is raised or not.

In particular, this is the problem

```
eq (N = M) = (N == M) .
```

Definition of equality via ‘syntactic’/‘evaluation-style’ equality.

What we want is

```
eq (N = M) = (N and M behave equally) .
```

What is of interest?

Are we interested in the actual value? – NO! Only in the

observation

whether the flag is raised or not.

In particular, this is the problem

```
eq (N = M) = (N == M) .
```

Definition of equality via ‘syntactic’/‘evaluation-style’ equality.

What we want is

```
eq (N = M) = (N and M behave equally) .
```

behavioral rewriting/algebra

First behavioral specification

Standard

```
mod* FLAG {
 [ Flag ]
 op raise _ : Flag -> Flag .
 op lower _ : Flag -> Flag .
 op change _ : Flag -> Flag .
 op is-up?_ : Flag -> Bool .

 eq is-up? raise F:Flag = true .
 eq is-up? lower F:Flag = false .
 eq is-up? change F:Flag = not is-up? F .
}
```

First behavioral specification

Standard

```
mod* FLAG {
 [ Flag ]
 op raise _ : Flag -> Flag .
 op lower _ : Flag -> Flag .
 op change _ : Flag -> Flag .
 op is-up?_ : Flag -> Bool .

 eq is-up? raise F:Flag = true .
 eq is-up? lower F:Flag = false .
 eq is-up? change F:Flag = not is-up? F .
}
```

Behaviour

```
mod* FLAG {
 *[ Flag ]*
 bop raise _ : Flag -> Flag .
 bop lower _ : Flag -> Flag .
 bop change _ : Flag -> Flag .
 bop is-up? _ : Flag -> Bool .

 beq is-up? raise F:Flag = true .
 beq is-up? lower F:Flag = false .
 beq is-up? change F:Flag = not is-up? F .
}
```

First behavioral specification

Standard

```
mod* FLAG {
 [ Flag ]
 op raise _ : Flag -> Flag .
 op lower _ : Flag -> Flag .
 op change _ : Flag -> Flag .
 op is-up?_ : Flag -> Bool .

 eq is-up? raise F:Flag = true .
 eq is-up? lower F:Flag = false .
 eq is-up? change F:Flag = not is-up? F .
}
```

Behaviour

```
mod* FLAG {
 *[ Flag ]*
 bop raise _ : Flag -> Flag .
 bop lower _ : Flag -> Flag .
 bop change _ : Flag -> Flag .
 bop is-up? _ : Flag -> Bool .

 beq is-up? raise F:Flag = true .
 beq is-up? lower F:Flag = false .
 beq is-up? change F:Flag = not is-up? F .
}
```

Changes

- sort definition: `*[ ... ]*`
- operator definition: `bop`
- axiom definition: `beq`

First behavioral specification

Standard

```
mod* FLAG {
 [ Flag ]
 op raise _ : Flag -> Flag .
 op lower _ : Flag -> Flag .
 op change _ : Flag -> Flag .
 op is-up?_ : Flag -> Bool .

 eq is-up? raise F:Flag = true .
 eq is-up? lower F:Flag = false .
 eq is-up? change F:Flag = not is-up? F .
}
```

Behaviour

```
mod* FLAG {
 *[ Flag ]*
 bop raise _ : Flag -> Flag .
 bop lower _ : Flag -> Flag .
 bop change _ : Flag -> Flag .
 bop is-up? _ : Flag -> Bool .

 beq is-up? raise F:Flag = true .
 beq is-up? lower F:Flag = false .
 beq is-up? change F:Flag = not is-up? F .
}
```

Changes

- sort definition: `*[ ... ]*`
- operator definition: `bop`
- axiom definition: `beq`

and above all

- semantics

Running the code

What happens if we run this code through CafeOBJ:

Running the code

What happens if we run this code through CafeOBJ:

```
...
  If you are sure that the proof is correct,
  you can add the following axiom(s):

ceq ceq (hs1:Flag =*= hs2:Flag) = true
    if ((is-up? hs1) == (is-up? hs2)) .
 done.
```

Running the code

What happens if we run this code through CafeOBJ:

```
...
  If you are sure that the proof is correct,
  you can add the following axiom(s):

ceq ceq (hs1:Flag =*= hs2:Flag) = true
    if ((is-up? hs1) == (is-up? hs2)) .
 done.
```

In normal words:

> You can define a kind of equality via the observations `is-up?`.

`=*=` is the behavioral equality

What happened behind the scenes?

The check of congruence comprises of the following:

- the only operator with hidden sort `Flag` as input and a normal sort as output `Bool` is `is-up?`

```
bop is-up? _ : Flag -> Bool .
```

What happened behind the scenes?

The check of congruence comprises of the following:

- the only operator with hidden sort `Flag` as input and a normal sort as output `Bool` is `is-up?`

```
bop is-up? _ : Flag -> Bool .
```

- check for each of the other operators (`raise`, `lower`, `change`) whether the following holds:

```
ceq ( hs1:Flag =*= hs2:Flag ) = true
  if ((is-up? hs1) == (is-up? hs2)) .
```

  where `hs1` and `hs2` are terms starting with the respective operators.

What happened behind the scenes?

The check of congruence comprises of the following:

- the only operator with hidden sort `Flag` as input and a normal sort as output `Bool` is `is-up?`

  ```
  bop is-up? _ : Flag -> Bool .
  ```

- check for each of the other operators (`raise`, `lower`, `change`) whether the following holds:

  ```
  ceq ( hs1:Flag =*= hs2:Flag ) = true
    if ((is-up? hs1) == (is-up? hs2)) .
  ```

  where `hs1` and `hs2` are terms starting with the respective operators.

For example

```
ceq ( (raise f1:Flag) =*= (raise f2:Flag) ) = true
  if ((is-up? (raise f1)) == (is-up? (raise f2))).
```

What happened behind the scenes? – cont

If this check succeeds, one can add the defining equation as suggested, or use

```
set accept =*= proof on
```

What happened behind the scenes? – cont

If this check succeeds, one can add the defining equation as suggested, or use

```
set accept =*= proof on
```

To see the proof carried out:

```
set verbose on
set trace whole on
```

What happened behind the scenes? – cont

If this check succeeds, one can add the defining equation as suggested, or use

```
set accept =*= proof on
```

To see the proof carried out:

```
set verbose on
set trace whole on
```

Then we get:

```
** system already proved "=*=" is a congruence of FLAG

>> adding axiom : ceq (hs1:Flag =*= hs2:Flag) = true
    if ((is-up? hs1) == (is-up? hs2)) .
 done.
```

Hidden Booleans as flag implementation

Let us consider the first implementation of flags via Booleans. Since we need to create an instantiation via a view, the sorts and operators must agree between FLAG and the implementation. Thus, we need something like *hidden Booleans*:

Hidden Booleans (code)

```
mod* BOOLFLAG {
 *[ HBool ]*
 bops htrue hfalse : -> HBool .
 ** basic properties of Booleans
 bop not _ : HBool -> HBool .
 beq not htrue = hfalse .
 beq not hfalse = htrue .
 ** operators for representation
 bop raise-bool _ : HBool -> HBool .
 bop lower-bool _ : HBool -> HBool .
 bop change-bool _ : HBool -> HBool .
 bop is-up?-bool _ : HBool -> Bool .
 ** as before
 beq raise-bool F:HBool = htrue .
 beq lower-bool F:HBool = hfalse .
 beq change-bool F:HBool = not F .
 beq is-up?-bool htrue = true .
 beq is-up?-bool hfalse = false .
 beq is-up?-bool not F:HBool = not is-up?-bool F .
}
```

Instantiating

As before, we need a view to instantiate the FLAGIMPLEMENTATION:

INSTANTIATING

As before, we need a view to instantiate the FLAGIMPLEMENTATION:

```
view BOOL-AS-FLAG from FLAG to BOOLFLAG {
 hsort Flag -> HBool,
 bop raise_ -> raise-bool_ ,
 bop lower_ -> lower-bool_,
 bop change_ -> change-bool_,
 bop is-up?_ -> is-up?-bool_
}
open FLAGTHEORY(X <= BOOL-AS-FLAG) .
red change-bool change-bool F:HBool =*= F .
close .
```

INSTANTIATING

As before, we need a view to instantiate the FLAGIMPLEMENTATION:

```
view BOOL-AS-FLAG from FLAG to BOOLFLAG {
 hsort Flag -> HBool,
 bop raise_ -> raise-bool_ ,
 bop lower_ -> lower-bool_,
 bop change_ -> change-bool_,
 bop is-up?_ -> is-up?-bool_
}
open FLAGTHEORY(X <= BOOL-AS-FLAG) .
red change-bool change-bool F:HBool =*= F .
close .
```

Well, as expected ...

What about the natural numbers?

Let us do the same for the natural numbers: First adapt them to hidden sorts:

What about the natural numbers?

Let us do the same for the natural numbers: First adapt them to hidden sorts:
All as before, only the renaming to hidden counterparts, and a changed definition of equality:

```
mod! HPNAT {
 *[ HPNat ]*
 bop s _ : HPNat -> HPNat .
 bop 0 : -> HPNat .
 bop even _ : HPNat -> Bool .
 bop odd _ : HPNat -> Bool .

 ...
 beq (N:HPNat = M:HPNat) = (N =*= M) .

 ...
}
```

What about the natural numbers?

Let us do the same for the natural numbers: First adapt them to hidden sorts:
All as before, only the renaming to hidden counterparts, and a changed definition of equality:

```
mod! HPNAT {
 *[ HPNat ]*
 bop s _ : HPNat -> HPNat .
 bop 0 : -> HPNat .
 bop even _ : HPNat -> Bool .
 bop odd _ : HPNat -> Bool .

 ...
 beq (N:HPNat = M:HPNat) = (N =*= M) .

 ...
}
```

CafeOBJ duly checks congruence …

Congruence check for HPNAT

With the following operator definitions, which equalities do we have to check under which conditions?

```
bop s _ : HPNat -> HPNat .
bop 0 : -> HPNat .
bop even _ : HPNat -> Bool .
bop odd _ : HPNat -> Bool .
bop times2 _ : HPNat -> HPNat .
bop raise-pnat _ : HPNat -> HPNat .
bop lower-pnat _ : HPNat -> HPNat .
bop change-pnat _ : HPNat -> HPNat .
bop is-up?-pnat _ : HPNat -> Bool .
```

Congruence check for HPNAT

With the following operator definitions, which equalities do we have to check under which conditions?

```
bop s _ : HPNat -> HPNat .
bop 0 : -> HPNat .
bop even _ : HPNat -> Bool .
bop odd _ : HPNat -> Bool .
bop times2 _ : HPNat -> HPNat .
bop raise-pnat _ : HPNat -> HPNat .
bop lower-pnat _ : HPNat -> HPNat .
bop change-pnat _ : HPNat -> HPNat .
bop is-up?-pnat _ : HPNat -> Bool .
```

Obervational operators?

Operators to be checked?

(to be filled in in class)

Instantiation the flag

As before, we need a view to instantiate the `FLAGIMPLEMENTATION`:

Instantiation the flag

As before, we need a view to instantiate the `FLAGIMPLEMENTATION`:

```
view PNAT-AS-FLAG from FLAG to HPNAT {
 hsort Flag -> HPNat,
 bop raise_ -> raise-pnat_ ,
 bop lower_ -> lower-pnat_,
 bop change_ -> change-pnat_,
 bop is-up?_ -> is-up?-pnat_
}
open FLAGTHEORY(X <= PNAT-AS-FLAG) .
red change-pnat change-pnat F:HPNat =*= F .
```

Instantiation the flag

As before, we need a view to instantiate the FLAGIMPLEMENTATION:

```
view PNAT-AS-FLAG from FLAG to HPNAT {
 hsort Flag -> HPNat,
 bop raise_ -> raise-pnat_ ,
 bop lower_ -> lower-pnat_,
 bop change_ -> change-pnat_,
 bop is-up?_ -> is-up?-pnat_
}
open FLAGTHEORY(X <= PNAT-AS-FLAG) .
red change-pnat change-pnat F:HPNat =*= F .
```

Q: What do you expect as outcome?

Summary (preliminary)

- behavioral specification allow for testing of ‘equality’ with respect to a set of observables
- congruence of mixed operators and hidden operators needs to be ensured
- very sensitive to signature changes
- good for abstracting implementation details from intended meaning
- Allows us to see the first specification of flags as correct!