Ein Beispiel zu Tocas

Claus Diem

April 29, 2018

Dieses Dokument wurde mit dem Notebook von jupyter erstellt. Das ging sehr schnell und komfortabel.

python wurde vom Tocas-Verzeichnis aus gestartet, alles weitere wird nun beschrieben.

Zuerst importieren wir Tocas:

```
In [1]: from Tocas import *
```

Wir haben nun einen Ring der ganzen Zahlen:

```
In [2]: Z

Out[2]: Z

In [3]: type(Z)

Out[3]: AbstrakteRinge.Ganzzahlring
```

Z ist also eine Instanz der Klasse Ganzzahlring im Modul AbstrakteRinge.

Mit der Funktion TypBeschreibung von Tocas:

```
In [5]: TypBeschreibung(Z)

Out[5]: 'Ganzzahlring'
```

Wir erzeugen noch einen Ganzzahlring:

```
In [8]: Z2 = Ganzzahlring()
```

Wie zu erwarten gelten Z und Z2 als gleich:

```
In [9]: Z == Z2

Out[9]: True
```

Weitere Ringe muss man zuerst instanziieren.

Zum Beispiel Restklassenringe:

```
In [11]: R = Restklassenring(100)
```

Dieser Ring hat eine Null und eine Eins:

In [14]:
```
R.null
```

Out[14]:
```
[0]_100  in Z/100Z
```

In [15]:
```
R.eins
```

Out[15]:
```
[1]_100  in Z/100Z
```

Wir konstruieren ein weiteres Element:

In [21]:
```
e=RestklassenringElement(3,R)
e
```

Out[21]:
```
[3]_100  in Z/100Z
```

Dieses invertieren wir nun:
Altenativ:

In [33]:
```
R.element(3)
```

Out[33]:
```
[3]_100  in Z/100Z
```

In [22]:
```
e.invers()
```

Out[22]:
```
[67]_100  in Z/100Z
```

In [23]:
```
e*e.invers()
```

Out[23]:
```
[1]_100  in Z/100Z
```

Stimmt!
Das ist dasselbe:

In [25]:
```
e ** -1
```

Out[25]:
```
[67]_100  in Z/100Z
```

Aber wie ist es mit [10]_100 ** -1 ?

In [26]:
```
RestklassenringElement(10,R) ** -1
```

```
---------------------------------------------------------------------------

InvertierungsFehler                       Traceback (most recent call last)

<ipython-input-26-233e542b6eb4> in <module>()
----> 1 RestklassenringElement(10,R) ** -1
```

```
~/daten/mathe/programme/python/tocas-neu/tocas/src/AbstrakteRinge.py in __pow__(self, ex
189
190         if exponent < 0:
--> 191             self = self.invers()
192             exponent = -exponent
193


~/daten/mathe/programme/python/tocas-neu/tocas/src/Restklassenringe.py in invers(self)
178
179         if a != 1:
--> 180             raise InvertierungsFehler(self)
181
182         return u


InvertierungsFehler: [10]_100  in Z/100Z ist nicht invertierbar.
```

Ja, das geht nicht.
Wir können auch ein zufälliges Element in R erzeugen:

```
In [107]: R.random()
```

```
Out[107]: [20]_100  in Z/100Z
```

Jetzt instanziieren wir einen Polynomring über R:

```
In [29]: P = Polynomring(R)
         P
```

```
Out[29]: Z/100Z[x]
```

"Per default" heißt die Variable in der Ausgabe (!) x.

```
In [30]: P.variable
```

```
Out[30]: [1]_100*x  in Z/100Z[x]
```

Ein Polynom:

```
In [32]: P.variable + 3*P.variable ** 4
```

```
Out[32]: [1]_100*x + [3]_100*x^4  in Z/100Z[x]
```

Das kann man auch so erzeugen:

```
In [36]: poly=P.element([0,1,0,0,3])
         poly
```

```
Out[36]: [1]_100*x + [3]_100*x^4  in Z/100Z[x]
```

Die Koeffizienten hiervon sind vom Typ RingTupel:

```
In [37]: poly.koeffizienten
```

```
Out[37]: 5-er Tupel von Elementen aus dem Ring Z/100Z:

         ([0]_100,[1]_100,[0]_100,[0]_100,[3]_100)
```

Dieses hat wieder Koeffizienten:

```
In [39]: poly.koeffizienten.koeffizienten
```

```
Out[39]: [[0]_100  in Z/100Z,
          [1]_100  in Z/100Z,
          [0]_100  in Z/100Z,
          [0]_100  in Z/100Z,
          [3]_100  in Z/100Z]
```

Das ist dann eine ganz normale Liste.

Jetzt erzeugen wir den Polynomring über P. Da die Variable x schon benutzt wird, sollten wir eine andere wählen:

```
In [54]: PP = Polynomring(P,"y")
         PP
```

```
Out[54]: Z/100Z[x][y]
```

Der Name der Variablen wird nur bei der Ausgabe verwendet:

```
In [63]: PPz = Polynomring(P,"z")
         PPz
```

```
Out[63]: Z/100Z[x][z]
```

```
In [64]: PP == PPz
```

```
Out[64]: True
```

```
In [66]: PP
```

```
Out[66]: Z/100Z[x][y]
```

Die Elemente von R liegen nicht in P und die Elemente von P liegen nicht in PP. Aber man man kann sie mit der Eins multiplizieren:

```
In [67]: P.variable
```

```
Out[67]: [1]_100*x  in Z/100Z[x]
```

```
In [68]: P.variable*PP.eins
```

```
Out[68]: [1]_100*x  in Z/100Z[x][y]
```

Oder auch so:

```
In [69]: PP.element(P.variable)
```

```
Out[69]: [1]_100*x  in Z/100Z[x][y]
```

Jetzt erzeugen wir Homomorphismen.
Zuerst ein paar kanonische Homomorphismen:

```
In [72]: a = Homomorphismus(Z,R)
         b = Homomorphismus(R,P)
         c = Homomorphismus(P,PP)
         a,b,c
```

```
Out[72]: (Kanonischer Ringhomomorphismus von Z zu Z/100Z,
          Kanonischer Ringhomomorphismus von Z/100Z zu Z/100Z[x],
          Kanonischer Ringhomomorphismus von Z/100Z[x] zu Z/100Z[x][y])
```

```
In [74]: d = c * b * a
         d
```

```
Out[74]: Verknüpfte Ringhomomorphismen von Z nach Z/100Z[x][y]
```

```
In [75]: d.anwenden(5)
```

```
Out[75]: [5]_100  in Z/100Z[x][y]
```

Wir wollen nun den Homomorphismus PP -> R mit x |-> [3], y |-> [4] konstruieren.
Wir gehen in zwei Schritten vor.

1. Konstruktion des Homomorphismus h: P -> R mit x |-> [3]
2. Konstruktion des gesuchten Homomorpismus PP -> R mit y |-> [4] und h

```
In [80]: h = Homomorphismus(P,R,R.element(3))
         h
```

```
Out[80]: Ringhomomorphismus von Z/100Z[x] zu Z/100Z
         zu x |-> [3]_100
         und zum Homomorphismus auf dem Grundring:
         Kanonischer Ringhomomorphismus von Z/100Z zu Z/100Z
```

```
In [81]: h.anwenden(P.variable)
```

```
Out[81]: [3]_100  in Z/100Z
```

Konstruktion des gesuchten Homomorphismus:

```
In [93]: h2 = Homomorphismus(PP,R,R.element(4),h)
         h2
```

```
Out[93]: Ringhomomorphismus von Z/100Z[x][y] zu Z/100Z
         zu y |-> [4]_100
         und zum Homomorphismus auf dem Grundring:
         Ringhomomorphismus von Z/100Z[x] zu Z/100Z
         zu x |-> [3]_100
         und zum Homomorphismus auf dem Grundring:
         Kanonischer Ringhomomorphismus von Z/100Z zu Z/100Z
```

```
In [94]: h2.anwenden(PP.variable)
```

```
Out[94]: [4]_100  in Z/100Z
```

```
In [98]: h2.anwenden(PP.variable ** 2 + P.variable*PP.eins)
```

```
Out[98]: [19]_100  in Z/100Z
```

Stimmt, denn 4ˆ2 + 3 = 19.
Jetzt verknüpfen wir noch:

```
In [101]: h2*c*b
```

```
Out[101]: Verknüpfte Ringhomomorphismen von Z/100Z nach Z/100Z
```

```
In [102]: (h2*c*b).anwenden(R.element(30))
```

```
Out[102]: [30]_100  in Z/100Z
```

Ja, h2*c*b definiert die Identität auf Z/100Z.
Ebenso wie Homomorphismus(R,R):

```
In [105]: Homomorphismus(R,R)
```

```
Out[105]: Kanonischer Ringhomomorphismus von Z/100Z zu Z/100Z
```

Nur erkennt Tocas das nicht, wiel die Darstellung verschieden ist:

```
In [106]: h2*c*b == Homomorphismus(R,R)
```

```
Out[106]: False
```