Cross-Type Arithmetic Examples

Currently, we can add rationals to rationals, but not rationals to complex numbers:

<table>
<thead>
<tr>
<th>Share Interface</th>
<th>Operators</th>
<th>Cross-type arithmetic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rational(1, 14)</td>
<td>+ Rational(2, 7)</td>
<td>Rational(17, 7)</td>
</tr>
<tr>
<td>ComplexRI(0, 1.0)</td>
<td>* ComplexRI(1, 0.5 * pi)</td>
<td>ComplexRI(1, 0.5 * pi)</td>
</tr>
</tbody>
</table>

**Rational Numbers**

```python
class Rational:  # A rational number represented as a numerator and denominator.
    def __init__(self, numer, denom):  # Greatest common divisor
        g = gcd(numer, denom)
        self.numer = numer // g
        self.denom = denom // g

    def __repr__(self):  # Example:
        return "Rational({0}, {1})".format(self.numer, self.denom)

    def __add__(self, other):  # nx + ny = ax + by
        return Rational(nx + ny, dx * dy)

    def __mul__(self, other):  # nx*ny + ny*dx
        return Rational(nx * dy + ny * dx, dx * dy)
```

**Complex Numbers**

```python
class Complex:  # A rectangular representation.
    def __init__(self, real, imag):  # Example:
        self.real = real
        self.imag = imag

    def __repr__(self):  # Example:
        return "ComplexRV({0}, {1})".format(self.real, self.imag)

    @property
    def magnitude(self):  # Example:
        return sqrt(self.real ** 2 + self.imag ** 2)

    @property
    def angle(self):  # Example:
        return atan2(self.imag, self.real)
```

**Special Method Names**

- `@property` - The method used to define properties
- `__init__` - The method used to initialize an object
- `__add__` - The method used for addition
- `__mul__` - The method used for multiplication
- `__repr__` - The method used for string representation

**Announcements**

- Delayed: Midterm 2 is on Thursday 3/19
- Delayed: Hog contest winners will be announced Friday 3/6 in lecture
- Project 3 due Thursday 3/12 @ 11:59pm
- Quiz 2 due Thursday 3/5 @ 11:59pm
- Last lecture: A function might want to operate on multiple data types
  - Interfaces: collections of messages that have specific behavior conditions
  - Two interchangeable implementations of complex numbers
  - Type dispatching
  - Type coercion
  - What's different? Today's generic functions apply to multiple arguments that don't share a common interface.
**Type Dispatching Analysis**

**Minimal violation of abstraction barriers: we define cross-type functions as necessary.**

**Extensible: Any new numeric type can "install" itself into the existing system by adding new entries to the cross-type function dictionaries.**

**Question:** How many cross-type implementations are required for $m$ types and $n$ operations?

$$m \cdot (m - 1) \cdot n$$
Type Coercion

**Coercion**

Idea: Some types can be converted into other types

Takes advantage of structure in the type system

```python
def rational_to_complex(r):
    r"Return complex equal to rational."
    return Complex(r.numer / r.denom, 0)
```

Question: Can any numeric type be coerced into any other?

Question: Can any two numeric types be coerced into a common type?

Question: Is coercion exact?

---

**Applying Operators with Coercion**

```python
class Number:
    def __add__(self, other):
        x, y = self.coerce(other)
        return x.add(y)

    def coerce(self, other):
        if self.type_tag == other.type_tag:
            return self, other

        elif (self.type_tag, other.type_tag) in self.coercions:
            return (self.coerce_to(other.type_tag), other)

        elif (other.type_tag, self.type_tag) in self.coercions:
            return (self, other.coerce_to(self.type_tag))

    def coerce_to(self, other_tag):
        coercion_fn = self.coercions[(self.type_tag, other_tag)]
        return coercion_fn(self)

coercions = {('rat', 'com'): rational_to_complex}
```

---

**Coercion Analysis**

Minimal violation of abstraction barriers: we define cross-type coercion as necessary

Requires that all types can be coerced into a common type

More sharing: All operators use the same coercion scheme

```plaintext
<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Coerce</th>
<th>Type</th>
<th>Add</th>
<th>Multiply</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complex</td>
<td>Complex</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rational</td>
<td>Rational</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complex</td>
<td>Rational</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rational</td>
<td>Complex</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complex</td>
<td>Rational</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rational</td>
<td>Complex</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

(Demo)