61A Lecture 18

Monday, October 10

The Independence of Data Types

Data abstraction and class definitions keep types separate

Some operations need to cross type boundaries

How do we add a complex number and a rational number together?

Rational numbers as numerators & denominators

Complex numbers as two-dimensional vectors

There are many different techniques for doing this!

Complex Numbers: the Rectangular Representation

```python
class ComplexRI(object):
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag
    @property
    def magnitude(self):
        return (self.real ** 2 + self.imag ** 2) ** 0.5
    @property
    def angle(self):
        return atan2(self.imag, self.real)
    def __repr__(self):
        return 'ComplexRI({0}, {1})'.format(self.real, self.imag)
def add_complex(z1, z2):
    return ComplexRI(z1.real + z2.real, z1.imag + z2.imag)
```

Rational Numbers: Now with Classes

Rational numbers represented as a numerator and denominator

```python
class Rational(object):
    def __init__(self, numer, denom):
        g = gcd(numer, denom)
        self.numer = numer // g
        self.denom = denom // g
    def __repr__(self):
        return 'Rational({0}, {1})'.format(self.numer, self.denom)
def add_rat(z1, z2):
    return Rational(z1.numer * z2.denom + z2.numer * z1.denom, z1.denom + z2.denom)
def add_complex_and_rational(z1, z2):
    return ComplexRI(z1.real + z2.real, z2.imag + z2.imag)
```

Generic Functions, Continued

A function might want to operate on multiple data types

Last time:
- Polymorphic functions using message passing
- Interfaces: collections of messages with a meaning for each
- Two interchangeable implementations of complex numbers

Today:
- An arithmetic system over related types
- Type dispatching instead of message passing
- Data-directed programming
- Type coercion

What’s different? Today’s generic functions apply to multiple arguments that don’t share a common interface

Rational Numbers, Now with Classes

```python
Generic Functions, Continued

A function might want to operate on multiple data types

Last time:
- Polymorphic functions using message passing
- Interfaces: collections of messages with a meaning for each
- Two interchangeable implementations of complex numbers

Today:
- An arithmetic system over related types
- Type dispatching instead of message passing
- Data-directed programming
- Type coercion

What’s different? Today’s generic functions apply to multiple arguments that don’t share a common interface

Rational Numbers, Now with Classes

```python
class Rational(object):
    def __init__(self, numer, denom):
        g = gcd(numer, denom)
        self.numer = numer // g
        self.denom = denom // g
    def __repr__(self):
        return 'Rational({0}, {1})'.format(self.numer, self.denom)
def add_rat(z1, z2):
    return Rational(z1.numer * z2.denom + z2.numer * z1.denom, z1.denom + z2.denom)
def add_complex_and_rational(z1, z2):
    return ComplexRI(z1.real + z2.real, z2.imag + z2.imag)
```

Type Dispatching

Define a different function for each possible combination of types for which an operation (e.g., addition) is valid

```python
def iscomplex(z):
    return type(z) in (ComplexRI, ComplexMA)
def isrational(z):
    return type(z) == Rational

def add_complex_and_rational(z1, z2):
    return ComplexRI(z1.real + z2.real, z2.imag + z2.imag)
def add_by_typeDispatching(z1, z2):
    if iscomplex(z1) and iscomplex(z2):
        return add_complex(z1, z2)
    elif isrational(z1) and isrational(z2):
        return add_complex_and_rational(z1, z2)
    elif iscomplex(z1) and isrational(z2):
        return add_complex_and_rational(z1, z2)
    elif isrational(z1) and iscomplex(z2):
        return add_complex_and_rational(z2, z1)
    else:
        return add_rational(z1, z2)
```
Tag-Based Type Dispatching

**Idea:** Use dictionaries to dispatch on type

```python
def type_tag(x):
    return type_tag.tags[type(x)]

type_tag.tags = {'ComplexRI': 'com', 'ComplexMA': 'com', 'Rational': 'rat'}

def add(z1, z2):
    types = (type_tag(z1), type_tag(z2))
    return add.implementations[types](z1, z2)
```

Declarative that ComplexRI and ComplexMA should be treated uniformly

```python
add.implementations = {}
add.implementations[('com', 'com')] = add_complex
add.implementations[('com', 'rat')] = add_complex_and_rational
add.implementations[('rat', 'com')] = add_rational_and_complex

lambda r, z: add_complex_and_rational(z, r)
```

Type Dispatching Analysis

**Minimal violation of abstraction barriers:** we define cross-type functions as necessary, but use abstract data types

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

```python
def add(z1, z2):
    types = (type_tag(z1), type_tag(z2))
    return add.implementations[types](z1, z2)
```

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

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

1. Add, subtract, multiply, divide

Data-Directed Programming

**Idea:** One dispatch function for (operator, types) pairs

```python
def apply(operator_name, x, y):
    tags = (type_tag(x), type_tag(y))
    key = (operator_name, tags)
    return apply.implementations[key](x, y)
```

Demo

Applying Operators with Coercion

1. Attempt to coerce arguments into values of the same type
2. Apply type-specific (not cross-type) operations

```python
def coerce_apply(operator_name, x, y):
    tx, ty = type_tag(x), type_tag(y)
    if tx != ty:
        if (tx, ty) in coercions:
            tx, y = tx, coercions[(tx, ty)](x)
        elif (ty, tx) in coercions:
            ty, y = tx, coercions[(ty, tx)](y)
        else:
            return 'No coercion possible.'
    key = (operator_name, tx)
    return coerce_apply.implementations[key](x, y)
```

Demo
Coercion Analysis

Minimal violation of abstraction barriers: we define cross-type coercion as necessary, but use abstract data types

Requires that all types can be coerced into a common type

More sharing: All operators use the same coercion scheme

<table>
<thead>
<tr>
<th>Arg 1</th>
<th>Arg 2</th>
<th>Add</th>
<th>Multiply</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complex</td>
<td>Complex</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rational</td>
<td>Rational</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complex</td>
<td>Rational</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rational</td>
<td>Complex</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<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>Rational</td>
<td></td>
<td>Complex</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rational</td>
<td>Complex</td>
<td></td>
<td>Rational</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>