An example type hierarchy

Paul Cockshott

February 2, 2009

Contents

Preface

This gives an example of a type hierarchy for Java that is fairly rigorously defined by algebraic means. It is intended as a crib which students doing MscIT can extend for their practical exercise.

Chapter 1

Atomic types

1.1 Universal

This is the Superclass to which all other classes belong - we assume that basic arithmetic is available on all types The universal type is the supertype that contains all of the sub-types derived from atoms, tuples and sets.

Its Σ (set of operations) is $\{\leftarrow = > < \leq \geq + - \times \div$ max min show #$\}$. The class Universal has a single pre-given member `undefinedValue`, which is returned when any operation returns an undefined result.

These operations are inherited by or overriden by equivalent operators in all subtypes of universal. The reason for universal having so mUniversal operations is that we want to be able to construct tuples whose elements are drawn from the class universal. If we want to perform arithmetic on tuples, the basic arithmetic operation must be defined on all members of the class universal.

Beneath the universal class we have atomics, tuples and sets.

1.2 Operations on Unversal

Operator	Type	Explanation
$a < b$	(Universal⊗Universal $\rightarrow \mathbf{B}$)	Returns 1 if a is less than b, see section **??**
$a = b$	(Universal⊗Universal $\rightarrow \mathbf{B}$)	Returns 1 if a equals b
$z \leftarrow a + b$	(Universal⊗ Universal $\rightarrow Universal$)	z is the algebraic sum of a and b
$z \leftarrow a + +b$	(Universal⊗ Universal $\rightarrow Universal$)	z is the concatenation of a and b unlike addition, not a comm
$z \leftarrow a - b$	(Universal⊗ Universal $\rightarrow Universal$)	z is the algebraic difference of a and b
$z \leftarrow a \times b$	(Universal⊗ Universal $\rightarrow Universal$)	z is the algebraic product of a and b
$z \leftarrow a \div b$	(Universal⊗ Universal $\rightarrow Universal$)	z is the quotient of a and b
$z \leftarrow \neg a$	(Universal$\rightarrow \mathbf{B}$)	Negation: $z = 1 \Leftrightarrow a = 0$
$z \leftarrow -a$	(Universal→Universal)	Unary minus: $z =^{-} 1 \times a$
$z \leftarrow \#x$	(Universal→$\mathbf{R}$)	z is the absolute value of x

```
package strathclyde.cs.relational;
import strathclyde.cs.relational.atomic.*;
public interface  Universal{
  public static final Universal undefinedValue = new Numeric(Double.NaN);
```

The implementaion of the undefined value makes use of the format NaN, short for Not a Number, which is defined in the IEEE floating point standard as the undefined value.

```
  public abstract Universal plus (Universal b);
```

implements $+$

```
  public abstract Universal concat (Universal b);
```

implements $++$

```
  public abstract Universal minus (Universal b);
```

implements $-$

```
  public abstract Universal times (Universal b);
```

implements $\times$

```
  public abstract Universal divide (Universal b);
```

implements $\div$

```
  public  Universal max (Universal b);
  public  Universal min (Universal b);

  public abstract boolean lessthan (Universal b);
```

implements $<$ note that this returns a boolean not a universal, a boolean can be converted to a universal using the class Bool which implements universal.

1.3 Numeric

The Δ of type *Number* is the union of all the Δs below it. The implementation will when possible, use the most compact binary representation available for numeric values. Its Σ is inherited from universal and extended by $\{$ main, inf, sup $\}$. These are used for compatibbility with triplex arithmetic their meanings of these are defined below.

Numerics will be held approximately using IEEE floating point representation when they are existing as single objects. Collections of numerics will try to compress these to minimum bit lengths.

1.3.1 Operations on Numbers

Operator	**Type**	**Explanation**
$a < b$	(Number$\otimes$Number $\rightarrow \mathbf{B}$)	Returns 1 if a is less than b
$a = b$	(Number$\otimes$Number $\rightarrow \mathbf{B}$)	Returns 1 if a equals b
$z \leftarrow a + b$	(Number$\otimes$ Number $\rightarrow$ *Number*)	z is the algebraic sum of a and b
$z \leftarrow a - b$	(Number$\otimes$ Number $\rightarrow$ *Number*)	z is the algebraic difference of a and b
$z \leftarrow a \times b$	(Number$\otimes$ Number $\rightarrow$ *Number*)	z is the algebraic product of a and b
$z \leftarrow a \div b$	(Number$\otimes$ Number $\rightarrow$ *Number*)	z is the quotient of a and b

```
import strathclyde.cs.relational.*;
import strathclyde.cs.relational.structured.*;

public class Numeric extends Number implements Universal,Interval

{
  public static final  double epsilon =2.22044604925e-16;

  protected double value;
  public Numeric(){value = Double.NaN;}
  public Numeric(double x) ...
  public Numeric(float x) ...
  public Numeric(int x){ value = (double)x; }
  public byte byteValue(){return (byte)value;}
  public double doubleValue(){return value;}
  public float floatValue()...
  public int intValue()...
  public long longValue()...
  public short shortValue()...
```

```
  public int hashCode()...
```

The next set of methods are used to make the type Numeric compatible with the type Triple, for interval arithmetic

```
  public double inf(){
    if (value>0)
      return value - value * epsilon;
    else if (value<=0)
      return value + value * epsilon;
    throw new RuntimeException("Error in inf  of Triplex");
  }
  //=====================================================
  public double main(){return value;}
  public double sup(){
    if (value>0)
      return value + value * epsilon;
    else if (value<=0)
      return value - value * epsilon;
    throw new RuntimeException("Error in sup of Triplex");
  }
  //=====================================================

  public  Universal  plus(Universal b)
  {
/* commented out initially, uncomment this when you get to implement triple
   if(b instanceof Triple) return (new Triple(this)).plus(b);
   else
*/
    if (b instanceof Numeric ) return new Numeric(value+((Numeric)b).value);
    else return undefinedValue;
  }
  public Universal concat(Universal b){ return new Text(toString()+b);}
  public Universal   minus( Universal b) ...
  public  Universal   times( Universal b) ...
  public  Universal   divide( Universal b) ...
  public  boolean   equals( Object b)...
  public  boolean   lessthan( Universal  b)...
  public  Universal max (Universal b)...
  public  Universal min (Universal b)...
  public String toString (){
    if (Double.isNaN(value)) return "?";
    return value + "";
  }
}
```

1.4 Interval

Interval is an interface shared by Numeric and Triple types allowing mixed mode arithmetic between them.

```
package strathclyde.cs.relational.atomic;
public interface Interval{
  public double inf();
  public double main();
  public double sup();
}
```

1.5 Bool

The class Bool is a subclass of universal that extends the Java class number. It does this to allow booleans to be treated as a special case of numbers in the range 0 to 1. What follows is an abridged look at a source implementation. Things for you to fill in are indicated with ...

```
public class Bool extends Number
implements Universal
{
  public boolean value;
  public static final Bool False = new Bool(false);
  public static final Bool True = new Bool(true);
  public byte byteValue()...
  public double doubleValue()...
  public float floatValue()...
  public int intValue()...
  public long longValue()...
  public short shortValue()...
  public int hashCode()...
  public Bool(boolean v)...
  public boolean equals(Object b){
    if(b instanceof Number){
      return  (intValue()==((Number)b).intValue());
    } else {
      return false;
    }
  }

  public boolean lessthan(Universal b)...
  }
  public Universal plus (Universal b)...
```

following the mathematician Boole this is treated as or

```
  public Universal concat(Universal b){return new Text(toString()+b);}
  public Universal times (Universal b)...
```

Following the standard boolean approach multiplication is equivalent to the and function special care has to be taken with undefined values.

```
  public Universal minus (Universal b)...
  public Universal divide (Universal b)...
  //*****************************************
  public  Universal max (Universal b)...
  public  Universal min (Universal b)...
  //*****************************************
  public String toString (){return value + "" ;  }
}
```

1.6 Subscriptable

Subscriptable [1] is an abstract interface that is inherited by all those classes that can be subscripted by universals - tuples, vectors, relations with primary keys. */

```
package strathclyde.cs.relational;

public interface Subscriptable
{
public Universal subscript(Universal i);
public Universal subscript(int i);
}
```

1.7 Text

Text which is the same as strings of characters are an atomic data type.

```
public class Text extends Structured implements Subscriptable
{
  protected String thetext;

  public Text(String s){ thetext=s;}
```

[1]* Part of Mbase implemented in Java author Paul Cockshott Started Aug 1997 Copyright (c) University of strathclyde

Operations inherited from Universal

```
  public Universal concat(Universal b){return new Text(thetext+b.toString());}
public Universal plus(Universal b){return concat(b);}
public  Universal max (Universal b){return(lessthan(b)?b:this);}
public  Universal min (Universal b){return (lessthan(b)?this:b);}
public boolean lessthan(Universal b)
{ return (b instanceof Text?thetext.compareTo(((Text)b).thetext)<0:false);}
public Universal times(Universal b){return undefinedValue;}
public Universal minus(Universal b){return undefinedValue;}
public Universal divide(Universal b){return undefinedValue;}
```

Operations inherited from structured

```
  public Enumeration members(){return new stupit(this);}
public   int arity(){ return thetext.length();}
public  double cardinality(){return arity();}

public Structured image(Moperator op)
{
  int i;
  String res ="";
  for(i=0;i<thetext.length(); i++)
    res=res+charof(op.apply(subscript(i)));
  return new Text(res);
}
```

Operations inherited from Subscriptable

```
public Universal subscript(int I)
{return (I<arity()&&I>=0 ? subscript(I):undefinedValue );}
public Universal subscript(Universal i)
{
  if(i instanceof Numeric)
  {
    int I = ((Numeric)i).intValue();
    return subscript(I);
  }
  if (i instanceof Tuple)
  {
    Tuple T=(Tuple)i;
    int j,N=T.arity();
    String t="" ;
    for(j=0;j<N;j++)
      t=t+charof(subscript(T.subscript(j)));
```

```
      return new Text(t);
    }
    return undefinedValue;
  }
```

Operations inherited from the java Object class

```
  public String toString()...
  public int hashCode()...
  public boolean equals(Object b){
    return((b instanceof Text)?  ...
                        :false);
  }
```

A private procedure used in the subscript function

```
  private char charof(Universal a)
  {
    if(a==undefinedValue) return '\0';
    if(a instanceof Numeric) return (char)((Numeric)a).intValue();
    return '\0';
  }
}
 class stupit implements Enumeration{
  int i;
  Text t;
  public boolean hasMoreElements(){return i<t.arity();}
  public Object nextElement(){return t.subscript(i++);}
  public stupit(Text t1){i=0;t=t1;}
}
```

Chapter 2

Structured

Structured data in the form of vectors, tuples, sets, relations etc is supported.

The crucial principle applied in achieving parallelism is the provision of higher order functions or functionals which allow lower level functions to applied to all elements of structured values. The meanings of these higher order functions are discussed in section **??**. The higher order functions in the Σ of all structured types are in addition to what they inherit from universal: $\{\mathcal{MR}$ apply cardinality $\}$ For all structured types the implementation provides a method `applytomembers` that applies a procedure to each element of the structured type. This iterator can then be used to implement operations like reduce in a type independent way. At the level of `tstructured` the iterator is defined as an abstract method that is to be overridden by lower level methods.

2.1 Operations on structured data

Operator	**Type**	**Explanation**
$z \leftarrow \circ\mathcal{R}\S$	((Any$\otimes$Any $\rightarrow Any$)$\otimes$Structured $\rightarrow Any$)	if $x = a,b,c$ are some structured collection and$\circ$ a dyadic operator $z = ((a \circ b) \circ c)$
$z \leftarrow F\mathcal{M}x$	((Any$\rightarrow$Any)$\otimes$Structured $\rightarrow Structured$)	e.g. $F\mathcal{M}\langle a,b,c\rangle \rightarrow \langle Fa,Fb,Fc\rangle$ see 2.1
$z \leftarrow \|\|x$	(Struct$\rightarrow$Number)	z is the cardinality of x

```
public abstract class Structured implements Universal
{

  public  abstract int arity();
  public  abstract double cardinality();
  public abstract Structured image(Moperator op);
```

form image of set under operator

```
public Universal reduce(Doperator op)
```

reduce the set using a dyadic operator

```
{
  Universal temp;
  Enumeration n=members();

  if (n. hasMoreElements())
  { temp=(Universal)(n.nextElement());
  while(n.hasMoreElements())
    temp=op.apply(temp,(Universal)n.nextElement());
  return temp;
  }
  else return undefinedValue;
}
public int hashCode()
```

Hash code formed for structure S with n elements $\{S_1, S_2, ...S_n\}$ is given by $H_S(n)$ where

$$H_S(0) = hash(S_1)$$
$$H_S(i) = 37 \times H_S(i-1) + hash(S_i)$$

```
{
  int temp;
  Enumeration n=members();

  if (n. hasMoreElements())
  { temp=(n.nextElement()).hashCode();
  while(n.hasMoreElements())
    temp=(n.nextElement()).hashCode()+temp*37;
  return temp;
  }
  else return 1;
}
public Universal concat(Universal b){
```

defined as + for completeness

```
  return plus(b);
}
  public abstract Enumeration members();
```

```
}
```

2.2 Tuples

Tuples are structured values whose components may not be of the same type.

```
class tupit implements Enumeration{
  int i;
  Tuple t;
  public boolean hasMoreElements(){return i<t.arity();}
  public Object nextElement(){return t.field[i++];}
  public tupit(Tuple t1){i=0;t=t1;}
}
public class Tuple extends Structured
implements Subscriptable
{
  public Universal field[];
  // ------------ Generators
  public Tuple(int n){field = new Universal [n];}

  public Tuple(Universal  u[])
  {
    field = u ;
  }

  public Tuple(String s[])
```

A constructor to form a tuple from a vector of strings the individual strings must either be numbers or else they are treated as uninterpreted text strings.

```
  {

    int i,n= s.length;
    Universal [] u= new Universal [n];

    for(i=0;i<n;i++){
      try {u[i]=new Numeric(Double.valueOf(s[i]).doubleValue());}
      catch (NumberFormatException e){
        u[i]=new Text(s[i]);
      }
    }
```

```
  field=u;
}
private static String[] split(String s,char delim) ...
```

function to split a string s into a vector of strings using a delimiter character

```
public Tuple(String s, char delim)
```

A constructor that splits a string using the delimiter into fields and then builds a tuple. Suitalble for parsing comma or tab separated relational database files.

```
{

  this(split(s,delim));
}


// ------------------------ from structured
public double cardinality(){ return (float)( field.length);}

public Enumeration members(){return new tupit(this);}
```

Used to implement reduce and Image

```
public Structured image(Moperator op)
{
  Tuple n= new Tuple(arity());
  int i;
  for(i=0;i<arity();i++)
  {  n.field[i]= op.apply(field[i]);
  }
  return n;
}
```

2.2.1 Subscription

A tuple may be subscripted. If

$$t = \langle a, b, c, ... \rangle$$

then

$$t_1 = a, t_2 = b, t_3 = c$$

etc. Subscription of a tuple by a tuple of integers in the appropriate range yields the tuple formed by using the elements of the second tuple as subscripts

$$\langle a,b,c,d\rangle_{\langle 1,3,2\rangle} = \langle a,c,b\rangle$$

Subscription can bein either by numbers or tuples.

```
// ---------------------- from subscriptable
public Universal subscript(int I)
```

useful for call within java when one knows the fields that one is to subscript.

```
{return( (I>=0 && I<arity()) ? field[I]: undefinedValue);}
public Universal subscript(Universal i)
```

This is useful within an interpreter to allow an arbitrary number or tuple to be used for subscription. Numbers are rounded to integers before subscription. Uses the previous function to do its dirty work.

```
{
  if(i instanceof Numeric) return subscript( ((Numeric)i).intValue());
  if (i instanceof Tuple)
  {
    Tuple T=(Tuple)i;
    int N=T.arity();
    Tuple t = new Tuple(N);
    int j;
    for(j=0;j<N;j++)t.field[j]=subscript(T.field[j]);
    return t;
  }
  return undefinedValue;
}
// ---------------------- from Universal
```

2.2.2 Concatenation

Tuples may be concatenated, for notation of this we use the ++ symbol. thus

$$\langle i,j,...,n\rangle ++ \langle p,q,...r\rangle = \langle i,j,...n,p,q,...,r\rangle$$

```
public Universal concat(Universal b )
{
  if(b instanceof Tuple){
    Tuple bt=(Tuple)b;
```

```
    int n = arity()+bt.arity();
    int i;
    Universal [] u = new Universal [ n ];
    for (i=0;i<n;i++)
      if(i>=arity())u[i]=bt.field[i-arity()];
    else u[i]= field[i];
    return new Tuple(u);
  }
  else{
    int i,n=arity();
    Universal [] u = new Universal [n+1];
    for(i=0;i<n;i++) u[i]=field[i];
    u[n]=b;
    return new Tuple(u);
  }

}
```

2.2.3 Injection

Binary operations between tuples are treated as the binary operation distributed over corresponding elements of the two tuples.

Thus

$$\langle 1,2,3,4\rangle + \langle 1,3,5\rangle = \langle 2,5,8,5\rangle$$

etc.

We can have tuples containing tuples

$$\langle 1, \langle' joe', 3\rangle\rangle$$

for which of course the injected binary operators are well defined

$$\langle 1, \langle' joe', 3\rangle\rangle + \langle 2, \langle' fish', 100\rangle\rangle = \langle 3, \langle' joefish', 103\rangle\rangle$$

```
public Universal  minus(Universal b )
{    int i; Tuple t;
if (b instanceof Tuple) {
  int l= Math.max(field.length,((Tuple)b).field.length);
  //int l= Math.min(field.length,((Tuple)b).field.length);
  t=new Tuple(l);
  for (i=0 ; i<l;i++)  {
    t.field[i]=
      (field[i%field.length].minus(((Tuple)b).field[i % ((Tuple) b).arity()]));
  }
  return t;
}
```

```
else return undefinedValue;
}
public Universal  plus(Universal b )...

public Universal  times(Universal b )...
public Universal  divide(Universal b )...

public boolean  equals(Object b )
```

Equal if have same number of fields and corresponding ones are equal

```
{ int i; boolean t;

if (b instanceof Tuple ){
  t=arity() == ((Tuple)b).arity();
  for (i=0 ;i<arity();i++)
    t= t && (field[i].equals(((Tuple)b).field[i]));
  return t;
}else  return false;
}
public boolean  lessthan(Universal b )...
```

Tuple $a < b$ follows string comparison rules.

```
//*****************************************
public  Universal max (Universal b){
  if (lessthan(b)){
    return b;
  } else {
    return this;}
}
public  Universal min (Universal b){
  if (lessthan(b)){
    return this;
  } else {
    return b;}
}
//*****************************************
public int arity(){ return field.length;}
```

Return number of fields in the tuple

```
public String toString()
```

Return the fields comma separated and with the whole square bracketed

```
  {
    String r="[";
    int i=0;

    for(i=0;i< arity(); i++)
    { if (i>0)r= r.concat(",");
    r=r.concat(field[i].toString());
    }
    return r.concat("]");
  }
}
```

2.3 Sets

A *set* follows the normal semantics of set theory. It provides the base class for relations. **You are to implement sets yourself in a manner consitent with the rest of the type hierarchy and following the algebraic specification below.**

2.3.1 Operations on Sets

Operator	Type	Explanation
$a < b$	(Set⊗Set → **B**)	Returns 1 if a is proper subset of b $(\{a\} < \{b,c\}) \Rightarrow (a = b) or (a = c)$ $b \neq a, \{a\} < \{b\} = false$
$?A$	(Set→Any)	Choice, returns a random member of set. $(?A) \in A$
$a = b$	(Set⊗Set → **B**)	Returns 1 if a equals$b$
$z \leftarrow a + b$	(Set⊗ Set → Set)	z is the union a and b $\{a\} + \{b\} = \{a,b\}$
$z \leftarrow a - b$	(Set⊗ Set → Set)	z is the set difference of a and b $(x+y) - y = x - y$ $\{a,b\} - \{a\} = \{b\}$
$z \leftarrow a \times b$	(Set⊗ Set → Set)	z is the intersection of a and b $(\{a\} \times \{a\}) = \{a\}$ $x \times (y \times z) = (x \times y) \times z$ $(\{a\} \times \{b\}) = (\{b\} \times \{a\})$
$z \leftarrow a \div b$	(Set⊗ Set → Set)	z is the quotient of a and b thus $a = b \times z$
$x \bullet a$	(Atom⊗ Set → Set)	set insertion $(a \bullet \{a\}) = \{a\}$ $(a \bullet \{\}) = \{a\}$ $(\circ \bullet x) = x$ $(a \bullet \{b\}) = b \bullet \{a\}$
$x \in a$	(Atom⊗ Set → **B**)	tests membership $x \in (x \bullet y) = true$
$A\|f$	(Set⊗(Any→**B**) → Set)	selection $A\|f = \{a\|a \in A, f(a)\}$
$z \leftarrow \|x$	(Set→**R**)	z is the cardinality of x

2.4 Weighted or fuzzy sets

If we wish to be able to represent uncertain knowledge, then it is thus convenient to provide a type of set that models the axioms of probability theory. Probability theory defines itself in terms of the following axioms[1]: Let E be a collection of elements $\alpha\beta\gamma$... called elementary events, and $\mathbf{F}_E$ a set of subsets of E; the elements of the set $\mathbf{F}_E$ being random events.

1. **F** is a field of sets.

[1] Kolmogorov, "Foundations of Probability", New York, 1950.

2. $\mathbf{F}$ contains E.

3. $\forall A \in \mathbf{F}_E, \exists P_E(A) \in \mathbf{R}$ where $P_E(A)$ is the probability of A.

4. $P_E(E) = 1$

5. If $A \times B = \{\}$ then
$$P_E(A+B) = P_E(A) + P_E(B)$$

If we associate with our sets S a function $\mathbf{P}_S(A)$ with the properties above then we have a model for probability theory[2].

We need to extend the definitions of the basic set operations to take probabilities into account.

```
import strathclyde.cs.relational.*;
import java.util.Random;
public abstract class Wset extends Set
{
 protected double multiplier;
 public Wset(){super();card=0;multiplier=1;}
```

The variable m is a multiplier used to compute the probability of an element from its weight. It is a useful scaling factor used to normalise probabilities to be equal to 1.

```
 public String toString()
```

Returns a space separted string of elements bracketed thus { } with each element preceded by its probabilty.

```
{
    String r="Weighted{";
    Enumeration i;

    for(i=members();i.hasMoreElements(); )
    {    Universal ui=(Universal)i.nextElement()   ;
      r=r.concat(" "+ui+":"+P(ui) );

    }
    return r.concat("} x "+cardinality());
}

protected abstract double W(Universal b);
```

[2]This approach differs from that of Dey and Sarkar in *A Probabalistic Relational Model*, (ACM TODS, Vol 21, No. 3, Sept 1996). In their model the sum of the probablilities of the tuples in a relation do not sum to 1.

This returns the current weight of an element in the set. Weights are proportional to probabilities but do not sum to 1.

```
protected abstract void  addmember(Universal b, double w);
```

Adds a new member with a given weight, if a member already exists then the previous weight is replaced by the new weight. After executing `addmember(b,w)` we have $W(b) = w$.

```
protected abstract void submember(Universal b, double w);
```

Removes a member and part of its weight, the new weight $W'(b)$ is either zero or the difference between the old weight and w

$$W'(b) = \mathbf{max}(W(b) - w, 0)$$

```
public double P(Universal u){return multiplier*W(u);}


public Universal choice()
```

If we interpret the basic choice operator $?$ as returning all members of a set with equal probability then there is an obvious extension to weighted sets. For large numbers of choices, the proportion of times that $(?E) \in A$ will tend to $\mathbf{P}_E(A)$. That is to say, the probability function associated with the elements of the set specifies the probability that elements will be chosen randomly from the set.

```
{    if (cardinality()==0) return Universal.undefinedValue;
    Random r= new Random();
   while(true)
   {  Enumeration i;
      Universal e;
      for(i=members();i.hasMoreElements();){
         if(r.nextFloat()<P(e=(Universal)i.nextElement()))
            return e;
      }

   }
}
public Set join(Doperator comb,Doperator sim, Set b)
```

Let r, s be sets. Let $p = r(\oplus \S \approx)s$ where

- $\S$ is the join functional,

- $\oplus$ is some dyadic combining operator of type (Any$\otimes$Any $\rightarrow Any$)
- $\approx$ is some similarity operator of type (Any$\otimes$Any $\rightarrow 0..1$), two tuples t,u are said to be similar if $(t \approx u) > 0$.

then we can say that p contains an element corresponding to every pair of elements in x,y that are similar.

$$\forall i \in r, \forall j \in s | i \approx j, \exists k \in p \tag{2.1}$$

each such $k = i \oplus j$ is the result of applying the combining operator to the pairs of similar elements.

Let us define the conditional similarity C_{ab} of two weighted sets a,b to be :

$$C_{ab} = \sum_{i \in a} \sum_{j \in b} (i \approx j) \times \mathbf{P}_a(i) \times \mathbf{P}_b(j) \tag{2.2}$$

This will be a number in the range 0..1. We can use it to define the probabilities associated with elements of a joined set. Thus in (**??**) and (2.1) we have

$$\mathbf{P}_p(k) = \frac{\sum_{\forall i,j | k=(i \oplus j)} (i \approx j) \times \mathbf{P}_r(i) \times \mathbf{P}_s(j)}{C_{rs}} \tag{2.3}$$

Note that

- as it should, the probabilities of the elements sum to unity: $\sum_{a \in p} \mathbf{P}_p(a) = 1$,
- equation (2.3) generalises equation (2.1).

Taking the definition of conditional similarity C_{ab} given in (2.2), the cardinality of joined sets is given by

$$\|p = C_{rs} \times \|r \times \|s \tag{2.4}$$

```
{
   Wset s = (Wset)EmptySet();
   Enumeration i,j;
   double C=Double.MIN_VALUE; // conditional similarity
   for (i=members();i.hasMoreElements();){
      Universal ui = (Universal)i.nextElement();
      double Pi= P(ui);
      for(j=b.members();j.hasMoreElements();){
         Universal uj = (Universal )j.nextElement();
         double Ws,Pj = b.P(uj);
         Number compare = (Number)sim.apply(ui,uj);// similarity
         C+= (Ws=Pj*Pi* compare.doubleValue());
         if (compare.doubleValue()>0){
            Universal nm=comb.apply(ui,uj); // new member
            (s).addmember(nm,s.W(nm)+Ws);
         }
      }
   }
```

```
   // Set multiplier to reciprocal of conditional similarity
   s.multiplier= 1/C;
   // set cardinality of s
   s.card =( cardinality()*b.cardinality())*C;
   return s;
}
public Universal times(Universal b)
```

The product of two sets is their intersection. Intersection of two sets is an operation defined in terms of the join functional, the equality operator and an operator **left** that always returns its left argument. thus for sets r, s:

$$r \times s = r(\mathbf{left} \S =)s \tag{2.5}$$

Consider two departments represented as unweighted (equiprobable) sets:

- *sales*= {Jo,Mo, Flo,Sue} with cardinality 4.
- *marketing*= {Lou, Sue, Mo} with cardinality 3.

under the normal interpretation of set theory their intersection would have cardinality 2.

The conditional similarity of the two sets is given by summing the product of the probabilities of occurence of Sue and Mo in the two sets $= 2 \times (\frac{1}{4} \times \frac{1}{3}) = \frac{2}{12} = \frac{1}{6}$.

The cardinality is then the products of the individual set cardinalities by the conditional similarity $= \frac{3 \times 4}{6} = 2$ as expected.

Consider now an example using weighted sets.

- *bombings*= {England$_{0.2}$,Ulster$_{0.8}$} with cardinality 10, representing IRA bombings over the last 3 years.
- *meteors* = {Scotland$_{0.3}$,Ulster$_{0.1}$, Wales$_{0.1}$, England$_{0.5}$} with cardinality 2 representing probable meteorite falls over the next 3 years.

The intersection of the two sets would represent the probabilities that areas would have an IRA bombing in the last 3 years and a meteorite fall in the next 3 years.

The conditional similarity is given as $\sum\{\text{England}_{0.2\times0.5}+\text{Ulster}_{0.1\times0.8}\} = 0.18$.

The cardinality is given as $2 \times 10 \times 0.18 = 3.6$.

The resulting set is

$$3.6 \times \{England_{0.55556}, Ulster_{0.44444}\}$$

we can interpret this as there being 3.6 expected pairs of bombs and meteorites hitting the same country.

```
{
   if(b instanceof Set)
   {  return join(new Left(),new Equals(),(Set)b);
```

```
   } else return undefinedValue;
}

public Universal  plus(Number weight,Universal b)
{
   double t,re,c1,c2,r;

   r=((Numeric)weight).doubleValue();
   if((r>1)||(r<0) ) return undefinedValue;
   if(b instanceof Set)
   {
      Set b1=(Set)b;
      Wset e= (Wset)EmptySet();
      Enumeration [] n={members(),b1.members()};

      Universal u;
      int i;
      for(i=0;i<2;i++){
         for(;n[i].hasMoreElements();)
         {
            Universal ui=(Universal)n[i].nextElement();
            e.addmember(ui,e.P(ui)+(i==0?r*P(ui):(1-r)*b1.P(ui)));
         }
      }
      e.multiplier=1;                          // unitary scaling of weights
      t=cardinality()+b1.cardinality();   // total cardinality
      re=   cardinality()/t;             // equilibrium partionioning
      c1=   (re/r)*t;                   // two estimates of cardinality
      c2= ( (1-re)/(1-r))*t;
      e.card = (c1<c2?c1:c2);             // chose minimum
      return e;
   } else return undefinedValue;
}
```

Union of sets

Disjoint Union

Consider the union operation, and more specifically the union operation on disjoint sets A, B. We know by axiom (5) of probability theory that probabilities of disjoint sets should be additive *within the context of some set F_E to which these constitute subsets*. Thus if we let $C = A + B$ we know that

$$\mathbf{P}_C(C) = 1 = \mathbf{P}_C(A) + \mathbf{P}_C(B) \tag{2.6}$$

and that if $A_i < A, B_j < B$ then

$$\mathbf{P}_C(A_i + B_j) = \mathbf{P}_C(A_i) + \mathbf{P}_C(B_j) \tag{2.7}$$

However we do not yet have enough information to derive $\mathbf{P}_C()$ from $\mathbf{P}_A()$ and $\mathbf{P}_B()$.

If we make the assumption that

$$\mathbf{P}_C(A_i) = (\mathbf{P}_C(A)) \times (\mathbf{P}_A(A_i)) \tag{2.8}$$

then we have an interpretation of the probability function over the union set in terms of the conditional probability of A_i given that A occurs. Thus given a real number $r : [0,1)$ as an additional parameter to + , as in $+_r$, we can define a new union operator that has a consistent interpretaion:

$$A \times B = \{\}, C = A +_r B, A_i \subset A \Rightarrow \mathbf{P}_C(A_i) = r\mathbf{P}_A(A_i) \tag{2.9}$$

From 2.9 we can deduce that

$$B_j \subset B \Rightarrow \mathbf{P}_C(B_j) = (1-r)\mathbf{P}_B(B_j) \tag{2.10}$$

Clearly it is necessary to have a similar parameterisation of the insertion operator

$$A \bullet_r b = A +_r \{b\} \tag{2.11}$$

It is desirable to have an interpretation of the simple unparameterised + operator that retains its original meaning. That is to say, it must preserve the meaning of the choice operator. Unparameterised union, must result in a set all of whose members are equally probable. This can be achieved if

$$a + b = a +_{r_e} b, r_e = \frac{\|a}{\|a + \|b} \tag{2.12}$$

that is to say, the sets are given union probabilities in proportion to their cardinalities.

2.4.1 Cardinality

The cardinality of the union of two sets is normally defined by the equation

$$A \times B = \{\} \Rightarrow \|(A + B) = (\|A) + (\|B) \tag{2.13}$$

For weighted sets, this is no longer sufficient. Instead we want to model situations like the following: a department has 3 employees, $\{$ John who is there full time, Robert and Gordon who each work half time $\}$. However, the number of employees who are there at any one time is 2. If we randomly select one of the employees present we will get John 50% of the time, Robert 25% and Gordon 25% of the time.

This probability distribution can be obtained by parameterising the + operator with 0.5.

$$\{John_{0.5}, Robert_{0.25}, Gordon_{0.25}\} = \{John\} +_{0.5} \{Robert, Gordon\}$$

Is there a treatment of cardinality that is consistent with our requirements?

If we define r_e as in eqn(2.12), and let $t = (\|A) + (\|B)$, then we can define the cardinality of weighted unions as follows:

$$A \times B = \{\}, \|(A +_r B) = \min(\frac{r_e}{r}t, \frac{1-r_e}{1-r}t) \tag{2.14}$$

This definition of cardinality reduces to the normal definition in the case of unweighted union. In the example above we get:

$$\|(\{John\} +_{0.5} \{Robert, Gordon\}) = \min(\frac{1/3}{1/2} \times 3, \frac{2/3}{1/2} \times 3) = 2$$

Generalised Union

Having dealt with cardinality and disjoint union we can now take on the problem of union between weighted sets with non-null intersections.

Suppose we have two departments *sales* =$\{\text{John}_{1/3}, \text{Robert}_{2/3}\}$ and *marketing*= $\{\text{John}_{1/5}, \text{Alice}_{2/5}, \text{Gordon}_{2/5}\}$. *Sales* set has cardinality 1.5, *marketing* has cardinality 2.5. We can interpret this as John working half of his time in each department whilst Robert, Alice and Gordon work full-time in their respective departments. The union of the two departments *sales-and-marketing= sales + marketing* should have a cardinality 4, as four people work in it full-time, and each person should have equal weight. We can generalise eqn(2.9) to:

$$C = A +_r B, A_i \subset A, A_i \subset B \Rightarrow \mathbf{P}_C(A_i) = r\mathbf{P}_A(A_i) + (1-r)\mathbf{P}_B(A_i) \tag{2.15}$$

In the example of the *sales and marketing* department,

sales+marketing = sales$+_{3/8}$*marketing*

by the normal equiprobable interpretation of union. Johns probability of being randomly selected from the joint department is then $\frac{3}{8} \times \frac{1}{3} + \frac{5}{8} \times \frac{1}{5} = \frac{1}{4}$ as desired.

```
public Universal plus(Universal b)
{ double t,re;
  if(b instanceof Set){
     t=cardinality()+((Set)b).cardinality();
     re = cardinality()/t;
     return plus( new Numeric(re),b);
  }
  else return undefinedValue;
}
```

The inherited plus operation is implemented in terms of the addition of sets using the equiprobable ratios as in eqn(2.12).

```
}
```

2.5 Relations

A *Relation* is a set all of whose elements are tuples, all of which are of the same arity and type. It inherits all of its operations other than projection and join from sets. A weighted relation similarly inherits all of its operations from weighted sets. We will treat weighted relations as the general form with unweighted relations being treated as the special case of a relation for which all tuples have equal weights.

Relations add to the set operations : Projection and Compression.

You should implement relations

2.5.1 Relational projection

Let r be a relation over tuples of type $\langle t_1, t_2, t_3, ..., t_n \rangle$ with t_i being types, then if

$$x = r \textbf{ proj } \langle j, k, ... \rangle \tag{2.16}$$

with $j, k, ..$ integers in range $1..n$. Then x is a set of tuples of type $\langle t_j, t_k, ... \rangle$ and

$$\forall a = \langle p, q, ... \rangle \in x, \exists i \in r | i_j = p, i_k = q, ..., \ etc \tag{2.17}$$

and

$$\forall z \in r, \exists s \in x | s_1 = z_j, s_2 = z_k, ..., \ etc \tag{2.18}$$

where in the above, the set membership operator '$\in$' is interpreted as true if the probability of occurence of the tuple in the relation is non zero:$a \in B \Rightarrow \mathbf{P}_B(a) > 0$.

2.5.2 Probabilities under projection

When projection occurs the probabilities of the projected tuples are the sum of the probabilities of the tuples that are projected onto them. Thus in (2.16),(2.17)

$$\mathbf{P}_x(a) = \sum \mathbf{P}_r(i) \forall i \in r | i_j = p, i_k = q, ..., \ etc \tag{2.19}$$

2.5.3 Cardinalities under projection

Since a weighted relation is taken to encode two things

1. A probability density function over a population characterised by the fields of the tuples,
2. The size of the population - the cardinality of the relation

it follows that the population should be invariant under projection, in that projection is equivalent to ignoring certain attributes of the population. Thus in (2.16), $\|x = \|r$.

Chapter 3

Operators

The whole package contains classes for operators which I will distribute to you as a packed jar file. You do not have to either implement these or use them, but I give them to you for interest sake.