Couples, Relations and Functions

Lecture 7

Couples

A couple is a pair of values. Couples are enclosed by round brackets and separated by a comma. Examples are, (1,2), (a,b), (John, Mary).

Constructor:

```
pair of (type, type){..}
pair of (int,int){a,b};
pair of (Person, Person){p1,p2};
```

Couples

Selector

Given p : pair of (X,Y){a,b}

p.x = a and p.y = b

Couples

```
const p1 : pair of(int,int)
^= pair of (int,int){5,6};
const p2 : pair of(int,int)
^= pair of (int,int){5,6};
const p3 : pair of(int,int)
^= pair of (int,int){3,6};
const p4 : pair of(int,int)
^= pair of (int,int){3,7};
```

Couples

```
property assert p1 = p2;
property assert p1 ~~ p2 = same@rank;
property assert ~(p1 < p3);
property assert p3 < p4;
property assert p1.x = p2.x & p1.y = p2.y;
property assert p1.x > p3.x;
```

for .. yield

for id :: collection **yield** expression

for those id :: collection :- predicate **yield** expression

The collection must be a set, bag or sequence and the result is a set, bag or sequence whose type is the same as the expression. In the second form only those values that satisfy the predicate are chosen.

for..yield

Example

Given a set `s = {1,2,3,4,5,6,7}`

```
for x :: s yield 2*x
```

returns the set {2,4,6,8,10,12,14}

```
for those x :: s :- x%2 = 0 yield 2*x
```

returns the set {4,8,12}

for .. yield

Question 2 Worksheet 5

Calculate total occupancy in hotel could be written as follows:

```
function totalOccupancy : nat
 pre exists r :: rooms :- r.numOccupants > 0
 ^= + over(for x :: rooms yield
                          x.numOccupants);
```

Relations

A relation is a set of couples.

A = {(1,2), (1,3), (2,4),(2,6), (3,5)}
B = {(John, Mary), (Donal,Pat)}

Domain of A

{1,2,3}

Range of A

{2,3,4,6,5}

Relations

Constructor

```
set of pair of (type,type){..}
set of pair of(int,int){..}
```

Given Point ^= pair of (int, int);

```
set of Point;
```

Relations

Example

Specify a class called `IntRelation` that models a relation of `int` couples. The class should have functions that retrieve the domain, range, and cardinality of the relation. Functions to check if a given couple is an element of the relation and check if a given set of couples are contained in the relation should be given. The class should also have functions that specify domain subtraction, domain restriction and range subtraction and restriction.

Relations

```
class IntRelation ^=
abstract
   var
    data : set of pair of (int,int);
interface
..
build{}
        post  data! = set of pair of (int,int){};
 build{f : set of pair of (int,int)}
     post  data! = f;
end;
```

Relations

The domain is the set of x ordinates

```
function dom : set of int
 ^= for a :: data yield a.x;
```

The range is the set of y ordinates

```
function ran : set of int
 ^= for a :: data yield a.y;
```

Relations

Given f : IntRelation such that

data = {(1,2),(3,5),(3,6),(5,7)}

f.dom = set of int{1,3,5}
f.ran = set of int{2,5,6,7}

Cardinality is given by

function size : nat ^= #data;

f.size = 4

Relations

isElement returns true if a,b is an element of the relation; false otherwise

```
function isElement(a :int,b:int): bool
  ^= pair of(int,int){a,b} in data;
```

contains returns true if f is a sub set of the relation; false otherwise

```
function contains(f : set of pair of(int,int)): bool
 ^= f <<= data;
```

Relations

Given a relation **r** and a set **s**, domain restriction equals the set of couples **t** such that

```
forall c :: t :- c.x in s
```

```
function domRestrict(s : set of int) :
                                set of pair of (int,int)
 ^= those  c :: data :- c.x in s;
```

Relations

Given f : IntRelation such that

data = {(1,2),(3,5),(3,6),(5,7),(6,2)}

and s = set of int{1,5}

f.domRestrict(s) = {(1,2),(5,7)}

Relations

Given a relation **r** and a set **s**, range restriction equals the set of couples **t** such that

```
forall c :: t :- c.y in s
```

```
function ranRestrict(s : set of int) : set of
pair of (int,int)
 ^= those  c :: data :- c.y in s;
```

Relations

Given f : IntRelation such that

data = {(1,2),(3,5),(3,6),(5,7),(6,2)}

and s = set of int{2,5}

f.ranRestrict(s) = {(1,2),(3,5),(6,2)}

Relations

Given a relation **r** and a set **s**, domain subtraction equals the set of couples **t** such that

```
forall c :: t :- c.x ~in s
```

```
function domSubtract(s : set of int) : set
of pair of (int,int)
 ^= those  c :: data :- c.x ~in s;
```

Relations

Given f : IntRelation such that

data = {(1,2),(3,5),(3,6),(5,7),(6,2)}

and s = set of int{1,3}

f.domSubtract(s) = {(5,7),(6,2)}

Relations

Given a relation **r** and a set **s**, range subtraction equals the set of couples **t** such that

```
forall c :: t :- c.y ~in s
```

```
function ranSubtract(s : set of int) : set of
pair of (int,int)
 ^= those  c :: data :- c.y ~in s;
```

Relations

Given f : IntRelation such that

data = {(1,2),(3,5),(3,6),(5,7),(6,2)}

and s = set of int{2,5}

f.ranSubtract(s) = {(3,6),(5,7)}

Relations

Schemas to add a new couple and a set of couples.

```
schema !add(a :int, b:int)
    post
     data! = data.append(pair of (int,int){a,b});

   schema !add(f : set of pair of (int,int))
     post
       data! = data ++ f;
```

Relations

Schemas to remove a set of couples and remove a set of domain values.

```
schema !remove(f : set of pair of (int,int))
 post
   data! = data -- f;

schema !removeFromDomain(s : set of int)
 post
   data! = data -- domSubtract(s);
```

Relations

Schema to remove a set of values from the range.

```
schema !removeFromRange(s : set of int)
 post
   data! = data -- ranSubtract(s);
```

YearGroup

A class of students can take any combination of the followng subjects: English, Maths, History, French, German. There are no restrictions on the number of subjects each student can take and there is no limit to the class size.

Specify a class called `YearGroup` that models students and the subjects they are studying.

YearGroup

```
class Student ^= string;

class Subject ^= enum English, Maths,
   History, French, German end;
```

Studies is a couple mapping a student to a subject

```
class Studies ^= pair of(Student,Subject);
```

YearGroup

```
class YearGroup ^=
abstract
   var
    // data is a relation
    data : set of Studies;
interface
..
build{}
        post ?;
end;
```

YearGroup

The list of students in the year is given by:

```
function students : set of Student
  ^= for s :: data yield s.x;
```

The list of subjects currently being taken by students is given by:

```
function subjects : set of Subject
  ^= for sub :: data yield sub.y;
```

YearGroup

List the subjects taken by a given student.

```
function getSubjects(st : Student): set of Subject
 pre st in students
 ^= (for those s :: data :- s.x = st yield s.y );
```

YearGroup

List students taking a particular subject

```
function listSubject(sub : Subject) : set of
Student
 ^= (for those s :: data :- s.y = sub yield
      s.x);
```

YearGroup

Add a student together with a single subject

```
schema !add(st : Student, sub : Subject)
 post
  data! = data ++ set of Studies{pair of
                    (Student,Subject){st,sub}};
```

Alternative post

```
data! = data.append(pair of (Student,
Subject) {st,sub});
```

YearGroup

Add a student together with a list of subjects

```
schema !add(st:Student, sub : set of
   Subject)
     post
        data! = data ++ (for x :: sub yield pair of
   (Student,Subject){st,x});
```

Functions

A function is a relation where each element in the domain has at most one image in the range

R1 = {(1,2),(2,3),(4,5),(5,6)}
R2 = {(1,2),(1,3),(2,3),(4,5)}
R3 = {(1,2),(2,2),(3,2),(4,2)}

R1 and R3 are functions but R2 violates the rule because it contains the couples (1,2) and (1,3)

Functions

Functions are modelled in Perfect by the map class

Constructors

```
map of (X->Y){..}
map of (int->int){1->2, 3->5}
map of (string->string){"pat"->"teacher"}
```

Functions

```
build{p: set of pair of (X,Y)}
  pre forall x, y::p :- x = y | x.x ~= y.x
  post ?
  assert self'.pairs = p;

build{a: seq of pair of (X,Y)}
  pre forall x, y::a :- x = y | x.x ~= y.x
  post ?
  assert self'.pairs = a.ran;
```

Functions and Schemas

returns cardinality of the set of couples

A++B, where A and B of type map, returns the union of both maps

A -- B, where A of type map and B a set, returns those couples in A that are not contained in B.

A ** B, where A of type map and B a set, returns those couples in A that are contained in B

Functions

`x in A` returns true if `x` is an element of the domain of `A`; otherwise false

```
operator (a: X) in : bool
  ^= exists x :: pairs :- x.x = a;
```

`A.pairs` returns the set of couples in `A`

Functions

A.dom returns the set of domain elements in *A*

A.ran returns the set of range elements in *A*, duplicates removed

A.ranb returns the bag of range elements in *A*

A.empty returns true if *A* is the empty map; false otherwise

Functions

A.append(b), where A of type map and b : pair of (X,Y), returns a map with couple b appended to A

```
function append(a: pair of (X,Y)): map of (X->Y)
  pre a.x in self ==> a.y = self[a.x]
  ^= map of (X->Y){pairs.append(a)};
```

Functions

```
function append(a:X->b:Y):
                                        map of (X->Y)
  pre a.x in self ==> a.y = self[a.x]
  ^= map of (X->Y){pairs.append(a)};
```

Note the pre-condition in both functions

Functions

```
function remove(a: X): map of (X->Y)
  ^= map of(X->Y){those x::pairs :-x.x ~= a};


selector [](a: X): Y
  pre a in dom
  ^= ?
  assert result = (that x::pairs :- x.x = a).y;
```

The selector means that it is possible to reference a range value using the domain value as an index

Functions

It is also possible to use indexing to modify an element of a mapping. For example, given a map A of integer couples such that

A = {1->2,3->5}, then

A[3]! = 7,

gives the map {1->2, 3->7}

MapTest

```
class MapTest ^=
abstract
   var
   data : map of (int->int);
interface
..
build{}
        post data! = map of (int->int){1->2,3->5};
```

MapTest

```
build{a : set of pair of (int,int)}
      pre forall x::a :-
            forall y::a :- (y = x) | ~(y.x = x.x)
      post data! = map of (int->int){a};
end;
```

MapTest

```
function dom : set of int
 ^= data.dom;

function ranb : bag of int
 ^= data.ranb;

function isElement(a : int, b : int) : bool
 ^= exists c :: data.pairs :- c.x = a & c.y = b;
```

MapTest

```
function domRestrict(s : set of int) : set of
pair of (int,int)
 ^= those c :: data.pairs :- c.x in s;

function domainRestrict(s : set of int) :
map of (int->int)
 ^= data ** s;
```

MapTest

```
function domSubtract(s : set of int) :
 map of (int->int)
     ^= data -- s;

schema !add(a : int, b : int)
 pre a in data.dom ==> b = data[a]
 post
   data! = data.append(a->b);
```

MapTest

```
schema !remove(a : set of int)
   post
     data! = data -- a;

schema !remove(a : int)
 post
  data! = data.remove(a);
```

Books Example

Specify a simple database of books where each book has an associated unique key. A book has a single attribute title.

Books Example

```
class Book ^=
abstract
   var
    title : string;
    invariant #title > 0;
interface
   function title;
   build{t : string}
       pre #t > 0
       post title! = t;
end;
```

Books Example

```
class Key ^=
abstract
   var key : string;
   invariant #key > 0;
interface
   function key;
   build{a : string}
       pre #a > 0
        post key! = a;
end;
```

Books Example

```
import "Book.pd", "Key.pd";
class Books ^=
abstract
   var
    data : map of (Key->Book);
interface
   function data;
   function keys : set of Key
    ^= data.dom;
```

Books Example

```
function books : bag of Book
 ^= data.ranb;

function empty : bool
 ^= data.empty;

schema !add(k:Key,b:Book)
 pre k ~in data.dom
 post data! = data.append(k->b);
```

Books Example

```
schema !delBook(k:Key)
 pre k in data.dom
 post data! = data.remove(k);

build{}
      post data! = map of (Key->Book){};
```

Books Example

```
property (a: Key, b:Book)
    pre a ~in data.dom
     assert ~(self after it!add(a,b)).empty;

ghost schema !addToEmptyThenRemove(a:
Key,b:Book)
     pre empty, a ~in data.dom
     post !add(a,b) then !delBook(a)
     assert data'.empty;
end;
```