Genericity

Lecture 9

Genericity

- A generic class is a class that accepts type parameterization.
- It is a type pattern
- The use of generic parameter in class definitions means that only a single specification that is not type specific can be written.

Genericity

- For example, a Stack is best defined with a generic parameter because the behaviour exhibited by an instance of the stack object is not affected by the types of the objects contained in it.

Stack

```
final class Stack of X ^=
abstract
   var
    stack : seq of X
interface
   function top : X
    pre #stack > 0
    ^= stack.head;
```

Stack

```
schema !push(y : X)
  post
   stack! = seq of X{y} ++ stack;


schema !pop
 pre #stack > 0
 post
  stack! = stack.tail;
```

Stack

```
    build{}
        post stack! = seq of X{};
end;
```

Exercise

Specify a generic Stack with an upper size limit of N, N>0.

Queue

```
final class Queue of X ^=
 abstract
  var
     queue : seq of X,
     size : int;
   invariant #queue <= size
```

Queue

```
interface
 function head : X
   pre #queue > 0
   ^= queue.head;


 schema !join(y : X)
   pre #queue < size
   post
     queue! = queue ++ seq of X{y};
```

Queue

```
schema !leave
  pre #queue > 0
  post
   queue! = queue.tail;
 build{s : int}
    pre s > 0
    post queue! = seq of X{},
           size! = s;
end;
```

Relation

```
final class Relation of (X,Y)
  require X has operator = (arg) end,
          Y has operator = (arg) end
  ^=
  abstract
      var rel : set of pair of (X,Y);
```

Relation

```
interface
   operator # : nat
    ^= #rel;

   // domain restriction
   function domRes(a:set of X) :
                                  set of pair of (X,Y)
    ^= those k :: rel :- k.x in a;
```

Relation

```
//domain subtraction
function domSub(a:set of X) :
                                set of pair of(X,Y)
 ^= those k :: rel :- k.x ~in a;


//range restriction
function ranRes(a : set of Y) :
                                set of pair of (X,Y)
 ^= those k :: rel :- k.y in a;
```

Relation

```
// range subtraction
function ranSub(a : set of Y) :
                                set of pair of (X,Y)
 ^= those k :: rel :- k.y ~in a;
```

Relation

```
function dom : set of X
 ^= for t :: rel yield t.x;

function ran : set of Y
 ^= for t :: rel yield t.y;

 function empty : bool
 ^= #rel = 0;
```

Relation

```
schema !append(a:X,b:Y)
  post
     rel! = rel.append(pair of (X,Y){a,b});


schema !append(t:pair of(X,Y))
  post
    rel! = rel.append(t);
```

Relation

```
schema !remove(a:X,b:Y)
  post
    rel! = rel.remove(pair of (X,Y){a,b});


schema !remove(a : set of X)
  post
   rel! = domSub(a);
```

Relation

```
    build{}
        post rel! = set of pair of(X,Y){};
    build{t: set of pair of(X,Y)}
        post rel! = t;
end;
```

Genericity

```
import "Relation.pd";
class RelationTest ^=
abstract
    var
     data : Relation of (int,string);
interface
    schema !insert(p:pair of(int,string))
      post data!append(p);
    function getNames : set of string
     ^= data.ran;
    build{}
          post data! = Relation of(int,string){};
end;
```