Lecture 4

Specifying with Sets

Set Theory

Intuitively a set is an unordered collection of elements that does not allow duplicates.

This means that two sets are equal if and only if they contain the same elements.

Set Theory

```
const s1 : set of int
 ^= set of int{1,2,3};
const s2 : set of int
 ^= set of int{3,2,1};
const s3 : set of int
 ^= set of int{1,2,3,2};
property assert s1 = s2;
property assert s1 = s3;
```

Set Theory

Set Membership

x in A returns true if x is an element of set A; false otherwise

```
property assert 1 in set of int{1,2,3};
property assert 4 ~in set of int{1,2,3};
```

Empty set

The empty set is a set with no elements. Denoted by {}

```
x in {} <==> false
```

The function `a.empty` returns `true` if a is the empty set; `false` otherwise.

Cardinality

The cardinality of a set is the number of elements in the set. It is denoted by `#s`

```
property assert
#set of int{1,2,3} = #set of int{1,2,3,1};
#set of int{} = 0;
```

Subset

A is a subset of B iff all elements in A are contained in B. Denoted by <<=

```
A <<= B <==> x in A => x in B
```

```
property assert
    set of int{1,2} <<= set of int{1,2,3};
```

Note: Proper subset denoted by A << B

Intersection

A intersection B is the set of elements common to both A and B. Denoted by **

A ** B <==> x in A & x in B

```
property assert
set of int{1,3,5} ** set of int{1,5,7} = set of int{1,5};
```

Union

A union B is the set of elements contained in A and B. Denoted by ++

A++B <==> x in A | x in B

```
property assert
set of int{1,3,5} ++ set of int{1,5,7} = set of
 int{1,3,5,7};
```

Set difference

A -- B is the set of elements contained in A that are not in B. Denoted by --

A -- B <==> x in A & x ~in B

```
property assert
 set of int{1,3,5} -- set of int{1,5,7} = set of
 int{3};
```

append

The function append adds an element to an existing set and returns a new set.

```
function append(a:X):set of X
 satisfy result >>= self,
          forall x :: result :- x = a | x in self;
```

Example

```
set of int{1,2,3}.append(4) =
                          set of  int{1,2,3,4};
```

remove

The function remove removes an element from an existing set and returns a new set.

```
function remove(a: X): set of X
    satisfy result <<= self,
     a ~in result,
     self = result | self = result.append(a);
```

Example

```
set of int{1,2,3}.remove(3) = set of int{1,2};
```

max

The function max returns the maximum element from a given set.

```
function max: X
    pre ~empty
    satisfy result in self,
    ~(exists x::self :- (result ~~ x) =
                                below@rank);
```

Example

```
    set of int{1,2,3,4}.max = 4;
```

min

The function min returns the minimum element from a given set.

```
function min: X
    pre ~empty
    satisfy result in self,
    ~(exists x::self :- (x ~~ result) =
                                below@rank);
```

Properties of sets

{} <<= A	Empty set is a subset of every set
A <<= A	Every set is a subset of itself
A ** {} = {}	A intersection empty set equals empty set
A ** A = A	A intersection A equals A
A ++ {} = A	A union empty set equals A
A ++ A = A	A union A equals A
A -- {} = A	A take away empty set equals A
A -- A = {}	A take away A equals empty set

Properties of sets

Laws of Distribution

A ** (B ++ C) = (A ** B) ++ (A ** C)

A ++ (B ** C) = (A ++ B) ** (A ++ C)

Universal Set

Universal Set

If every set X under discussion is a sub-set of some set U, then U is called the universal set.

Given a universal set U and a set A, such that A<<=U, we define the complement of A as

compA = U -- A

Constructing large sets

In Perfect it is possible to list the elements in a set as part of the constructor

e.g. `set of int{1,2,3,4,5,6,7,8};`

But not possible to use shorthand notation.

To construct large sets use recursive definition

Constructing large set of int

```
function setConstructor(n : nat) : set of int
 decrease n
 ^= ([n = 0]:
     set of int{},
    []:
     set of int{n} ++ setConstructor(n-1)
    );
```

Universal set

```
const U : set of int
    ^= setConstructor(50);

const D : set of int
    ^= those x :: U :- x % 2 = 0;

const E : set of int
    ^= those x :: U :- x % 2 ~= 0;
```

Universal set

```
const compD : set of int
  ^= U -- D;

property assert forall x :: D :- x ~in E;
property assert D ** E = set of int{};
property assert D ++ compD = U;
```

Sample problem

Specify a class called `SetEx` that has a single set of integer values as attribute.

The data set must always contain at least one non-negative value.

Solution

```
class SetEx ^=
abstract
   var
        data : set of int;
   invariant
       #data > 0  & (forall c :: data :- c >= 0);
```

Constructor

```
build{}
    post data! = set of int{1};

build{a : set of int}
 pre #a >0 & (forall x :: a :- x >= 0)
 post data! = a;
```

Functions

```
function isElementOf(x : int) : bool
    ^= x in data;

function contains(a : set of int) : bool
    ^= a <<= data;
```

Schemas

```
schema !insert(x : int)
      pre x >=0
      post data! = data.append(x);

schema !insertSet(a : set of int)
     pre (forall c :: a :- c >= 0)
     post data! = data ++ a;
```

Schemas

```
schema !remove(x : int)
    pre isElementOf(x) & #data > 1
    post data! = data.remove(x);

schema !removeSet(a : set of int)
    pre #a < #data
    post data! = data -- a;
```

Simple Resource Manager

Specify a resource manager that allocates a pool of resources to users. When a user is allocated a resource it is no longer available until it is freed by the user. When this happens it is put back in the pool of available resources.

A resource might be a hotel room, a car, a printer, etc.

Simple Resource Manager

```
class Resource ^=
 abstract
   var
        id : string;
 interface
        function id;
        build{!id:string};
 end;
```

Simple Resource Manager

```
class ResourceManager ^=
abstract
 var
  resources : set of Resource,
  available : set of Resource,
  allocated : set of Resource;
```

Simple Resource Manager

```
invariant
   available <<= resources,
   allocated <<= resources,
   available ++ allocated = resources,
   available ** allocated = set of Resource{};
```

Simple Resource Manager

```
build{}
    post
        resources! = set of Resource{},
        available! = set of Resource{},
        allocated! = set of Resource{};
```

Simple Resource Manager

```
function resources;

 function allocated;

 function available;

 function checkAvailable(r : Resource):bool
  ^= r in available;
```

Simple Resource Manager

```
schema !add(r : Resource)
   pre r ~in resources
   post
        resources! = resources.append(r),
       available! = available.append(r);
```

Simple Resource Manager

```
schema !allocate(r : Resource)
    pre r in available
    post
        allocated! = allocated. append(r),
        available! = available.remove(r),
        resources! = resources;
```

Simple Resource Manager

```
schema !free(r : Resource)
    pre r in allocated
    post
        allocated! = allocated -- set of Resource{r},
        available! = available ++ set of Resource{r};
```

Simple Resource Manager

Allocate r followed by free r leaves state space unchanged

```
ghost schema !requestReturn(r:Resource)
  pre r in available
  post
    !allocate(r) then !free(r)
  assert
    available' = available,
    allocated' = allocated;
```