Refinement

Lecture 8

Refinement

- Refinement is the process of mapping a specification to a program

- Types
  - Procedural refinement
  - Data refinement

Loops

```
loop
   var              //local var declarations
   change           // list of what variables can change
   keep  inv        // invariant
   until  b
   decrease   //variant
         // loop body
 end;
```

Loops

- Variables in invariants may be primed or unprimed
  - primed = current values at the start of an iteration
  - unprimed = value before the loop started
- Invariant is the only source of information about current values of changing variables
- When loop terminates the state is given by:

$$inv \ \& \ b$$

Loops

Calculate a*b by performing repeated additions

```
var prod: int != 0;
loop
   var j: nat != 0;
   change prod
   keep prod' = a * j'
   until j' = b
   decrease b - j';
       prod! = prod + a,
      j! = j + 1
end;
assert prod = a*b;
value prod
```

Procedural refinement

The previous example might be written to refine a function as follows:

```
function multiply(a, b: int): int
  pre b >= 0
  ^= a * b
  via
      var prod: int != 0;
      loop
        ..
      end;
      assert prod = a*b;
      value prod
  end;
```

Procedural refinement

In the class given below refine each of the given functions to an iterative solution

```
class SeqInt ^=
abstract
  var
    data : seq of int;
  invariant
     #data > 0;
```

Procedural refinement

```
interface
   function allPositive : bool
    ^= (forall x :: data :- x > 0);
   function cardPos : int
    ^= #(for those x :: data :- x > 0 yield x);
   function sum : int
    ^= + over data;
build{}
       post data! = seq of int{1};
end;
```

Procedural refinement

```
function allPositive : bool
 ^= (forall x :: data :- x > 0)
 via
  var x : bool! = true;
  loop
   var n : nat! = 0;
   change x
   keep
       0 <= n' <= #data,
       x' = (forall j :: 0..<n' :- data[j] > 0)
```

Procedural Refinement

```
  until n' = #data
  decrease #data - n';
     n! = n + 1, x! = (x & data[n]>0)
 end;
  value x
end;
```

Procedural refinement

```
function cardPos : int
    ^= #(for those x :: data :- x > 0 yield x)
    via
      var x : int! = 0;
      loop
        var n : nat! = 0;
        change x
        keep
            0 <= n' <= #data,
             x' = #(for those j :: 0..<n' :- data[j] > 0
                  yield data[j])
```

Procedural refinement

```
until n' = #data
    decrease #data - n';
    n! = n + 1,
    x! = ([data[n]>0]:
              x + 1,
           []: x
          )
   end;
    value x
  end;
```

Procedural refinement

```
function sum : int
    ^= + over data
    via
     var s : int! = 0;
     loop
       var n : nat! = 0;
       change s
       keep
            0 <= n' <= #data,
            s' = ( [n' = 0]: 0,
                   []: + over data.take(n')
                 )
```

Procedural refinement

```
until n' = #data
decrease #data - n';
 s! = s + data[n], n! = n + 1
end;
value s
end;
```

Data refinement

Perfect supports data refinement through the use of internal variables and link functions that relate the concrete data structure to the abstract data structure.

Two types can be identified:

- Supplement existing specification to improve performance of the implementation without changing the type of the abstract variables
- Refine abstract variables to a new type with a linking invariant

Data refinement

Given a class that maintains a list of integer values with two methods: add, that allows new numbers to be appended to the list, and sum, that returns the sum of the numbers in the list.

Data refinement

```
class IntList ^=
abstract
   var
    data : seq of int;
interface
   schema !add(x : int)
    post
      data! =data.append(x);
```

Data refinement

```
function sum : int
 ^= ( [#data = 0]: 0,
      []: + over data
    );
build{}
    post data! = seq of int{};
end;
```

Data refinement

Suppose the function sum is invoked frequently and you are asked to refine the specification to improve its overall performance.

Each time the function is invoked the sum is re-calculated. One way to avoid this is to introduce an internal variable whose value is always equal to sum

Data refinement

```
class IntList ^=
abstract
   var
    data : seq of int;
internal
    var
       totl : int;
    invariant
       totl = sumList(data);
```

Data refinement

```
interface
  schema !add(x : int)
   post
     data! = data.append(x)
   via
    data! = data.append(x), totl! = totl + x
   end;
```

Data refinement

```
function sum : int
    ^= sumList(data)
     via
       value totl
     end;
   build{}
        post data! = seq of int{}
       via
        data! = seq of int{}, totl! = 0
       end;
end;
```

Data refinement

Note

To complete the refinement it was necessary to introduce an external function `sumList` as follows:

```
function sumList(s : seq of int) : int
^= ([s.empty]: 0,
     []: + over s
    );
```

This became necessary to allow the invariant `totl = sumList(data)` be written

Data refinement

Given the specification of a queue of integer values refine it so that it is implemented as a circular queue. This avoids shuffling elements in the queue when the front element is removed.

The original specification is given below.

Data refinement

```
class IntQueue ^=
abstract
    var queue: seq of int,
        maxLen: nat > 0;
    invariant #queue <= maxLen;
interface
    function empty: bool
        ^= #queue = 0;
    schema !add(x: int)
        pre ~full
        post queue!= queue.append(x);
```

Data refinement

```
    function full: bool
        ^= #queue = maxLen;
    schema !remove(x!: out int)
        pre ~empty
        post x! = queue.head,
                queue! = queue.tail;
    build{!maxLen: nat}
        pre maxLen ~= 0
        post queue! = seq of int{};
end;
```

Data refinement

```
class CircularIntQueue ^=
abstract
   var queue: seq of int,
       maxLen: nat > 0;
   invariant #queue <= maxLen;
```

Data refinement

```
internal
   var ring: seq of int,
       hd, tl: nat;
   invariant
                  #ring = maxLen + 1,
                  hd < #ring,
                  tl < #ring;
```

Retrieve function

```
function queue ^=
    ([tl >= hd]:
            ring.take(tl).drop(hd),
      [tl < hd]:
            ring.drop(hd) ++ ring.take(tl)
    );
```

Data refinement

```
interface
    ghost operator =(arg);

    function empty: bool
        ^= #queue = 0
        via
                value hd = tl
        end;
```

Data refinement

```
schema !add(x: int)
    pre ~full
    post queue!= queue.append(x)
    via
        ring[tl]! = x,
        tl! = (tl + 1)%(#ring)
    end;
```

Data refinement

```
function full: bool
    ^= #queue = maxLen
    via
            value (>tl)%(#ring) = hd
    end;
```

Data refinement

```
schema !remove(x!: out int)
    pre ~empty
    post x! = queue.head,
            queue! = queue.tail
    via
            x! = ring[hd],
            hd! = (>hd)%(#ring)
    end;
```

Data refinement

```
build{!maxLen: nat}
    pre maxLen ~= 0
    post queue! = seq of int{}
    via
        ring! = seq of int{0}.rep(maxLen + 1),
        hd! = 0,
        tl! = 0
    end;
```

Data refinement

Mapping a set to a sequence

Given a class with abstract variable of type set of int map it to a sequence

```
class SetToSeq ^=
abstract
   var
    dataSet : set of int;
    invariant #dataSet > 0;
```

Data refinement

```
interface
   function dataSet;
   function allPositive : bool
    ^= forall x :: dataSet :- x > 0;
   function sum : int
    ^=
      + over dataSet;
   schema !add(x : int)
    pre x ~in dataSet
    post
      dataSet! = dataSet.append(x);
```

Data refinement

```
schema !del(x : int)
 pre
  x in dataSet
 post
  dataSet! = dataSet.remove(x);
 build{}
     post dataSet! = set of int{1}
end;
```

Data refinement

```
class SetToSeq ^=
abstract
   var
    dataSet : set of int;
    invariant #dataSet > 0;
internal
  var
   dataSeq : seq of int;
```

Data refinement

```
function dataSet ^= dataSeq.ran;
 invariant
   #dataSet = #dataSeq,
   dataSeq.unique,
   #dataSeq > 0,
   forall x :: dataSet :- x in dataSeq,
   forall x :: dataSeq :- x in dataSet;
```

Data refinement

```
operator =(arg);
function dataSet;
function allPositive : bool
 ^= forall x :: dataSet :- x > 0
 via
  value (forall x :: dataSeq :- x > 0)
 end;
```

```
function sum : int
 ^=
   + over dataSet
 via
  value (+ over dataSeq)
 end;
```

```
schema !add(x : int)
 pre x ~in dataSet
 post
   dataSet! = dataSet.append(x)
 via
   dataSeq! = dataSeq.append(x)
end;
```

```
schema !del(x : int)
 pre
    x in dataSet
   post
     dataSet! = dataSet.remove(x)
   via
    (assert x in dataSeq;
     let ind ^= dataSeq.findFirst(x);
     [ind >=0]:
       dataSeq! = dataSeq.take(ind) ++
   dataSeq.drop(ind + 1),
      []: pass
     )
    end;
```

```
build{}
   post dataSet! = set of int{1}
   via
    dataSeq! = seq of int{1}
   end;
```