Further directions in program correctness

Self-checking code

One of the advantages of using specifications is that it allows us to annotate code with assertions that we expect to be true at a given point in time. These assertions can be checked during execution to help ensure that the program is working correctly.

Consider the binary search algorithm above. We can check the precondition to make sure that the algorithm is only used as appropriate. We can check the postcondition, to ensure that the algorithm behaved correctly.

We can also check intermediate assertions (such as loop invariants) to ensure that computations are proceeding as expected.

C/C++ provides an assert “macro” for this purpose. Java provides an assert statement.

```
#include <assert.h>
...
int nroot( const int x ) {
    assert( x >= 0 ) ;
    int y = 0, z = x+1 ;
    while( z-y > 1 ) {
        int m = y + (z-y)/2 ;
        if( m*m <= x ) y = m ;
        else z = m ; }
    assert( y*y <= x && x < (y+1)*(y+1) ) ;
    return y ; }
```

One should consider:

- Which assertions should be included in the compiled versions of the final product?
  - In my opinion all assertions that are not too costly of performance should be left in the final product.
- What should be the product's behaviour when an assertion fails?
  - For example dumping an error report and offering to send it back to the developers.

More interesting algorithms

Complex algorithms can benefit from a formal or semi-formal treatment.

For example, in Dijkstra's shortest path algorithm. The key parts of the invariant are

- For all nodes visited so far, the node is labeled with the length of the shortest path from the source to the node.

The idea of data refining algorithm schemes to obtain algrothms can be extended to algorithmic techniques such as dynamic programming and greedy algorithms.

Correctness of data representations.

Class invariants

In object oriented programming, each object undergoes a 'loop' of use and non use. The body of this loop is a choice between the bodies of the object's public subroutines.

That is, there is an analogy between a class:

class C
    private var *v* : *T*
    public constructor C() (*C0*)
    public method m0() (*M0* ; return *E0*)
    public method m1() (*M1*; return *E1*)
    ...
end class

and a loop

```
var v : T
C0
while true do (
    receive a message
    if message is m0 then
        M0
        reply with E0
    else if message is m1 then
        M1
        reply with E1
    ...
)
```

Thus, it is useful to consider invariants for objects.

The invariant specifies the states the object's fields may be in before and after calls to the public methods.

Class Specification

We can specify classes by providing them with an abstract state and specifications for all public methods

For example. Consider a class representing a set of integers from 0 to 9 inclusive.

class SetSpec
  private var s : set of $\mathbb{Z}$
  invariant $s \subseteq \{0, ..10\}$
  public constructor SetSpec()
    s := $\emptyset$
  public method insert(i : int)
    precondition $0 \leq i < 10$
    $s \ := s \cup \{i\}$
  public method delete(i : int)
    precondition $0 \leq i < 10$
    $s \ := s - \{i\}$
  public method contains(i : int) : $\mathbb{B}$
    precondition $0 \leq i < 10$
    return $i \in s$
end class

We can see that if the preconditions are respected by the callers, then the invariant will always be true between calls.

Class refinement

We can use a data transformation to replace one set of variables by another without affecting the behaviour of the class.

For example we augment variable s above by a variable $a : \{0, ..10\} \to \mathbb{B}$ with the additional invariant

$$\forall i \in \{0, ..10\} \cdot a(i) = (i \in s)$$

This additional invariant is called the **abstraction relation**.

class SetAug **implements** SetSpec
    private var s : set of $\mathbb{Z}$
    private var $a : \{0, ..10\} \to \mathbb{B}$
    invariant
  $s \subseteq \{0, ..10\} \wedge (\forall i \in \{0, ..10\} \cdot a(i) = (i \in s))$
    public constructor SetAug()
        $s := \emptyset$;
        for $i \in \{0, .10\} \cdot a(i) := \mathfrak{false}$
    public method insert(i : int)
        precondition $0 \leq i < 10$
        $s := s \cup \{i\}$; $a(i) := \mathfrak{true}$
    public method delete(i : int)
        precondition $0 \leq i < 10$
        $s := s - \{i\}$; $a(i) := \mathfrak{false}$
    public method contains(i : int) : $\mathbb{B}$
        precondition $0 \leq i < 10$
        return $a(i)$
end class

At this point, we no longer need the variable s except for documentation. Thus we have

class SetImp **implements** SetSpec
  private var $a : \{0, ..10\} \to \mathbb{B}$
  public constructor SetImp()
    for $i \in \{0, .10\} \cdot a(i) := \mathfrak{false}$
  public method insert(i : int)
    precondition $0 \leq i < 10$
    $a(i) := \mathfrak{true}$
  public method delete(i : int)
    precondition $0 \leq i < 10$
    $a(i) := \mathfrak{false}$
  public method contains(i : int) : $\mathbb{B}$
    precondition $0 \leq i < 10$
    return $a(i)$
end class

It is often best to externally document the behaviour of a class in terms of abstract variables (such as s).

Internal documentation should include the abstraction relation.

Class Optimization

Consider a class for computing some function.

It doesn't really matter what the function is, as long as its domain is finite and reasonably small.

I'll use $\binom{n}{r}$ —the number of subsets of size r that a set of size n has— as an example.

Let $A = \{0, ..a\}$ for some $a \in \mathbb{N}$.

Class specification

class ChooseSpec
  public constructor ChooseSpec()
    skip
  public method $c(n : A, r : A) \ : \ \mathbb{N}$
    precondition $0 \leq r \leq n$
    return $\binom{n}{r}$
end class ChooseSpec

An implementation

A slow implementation is given by Pascal's formula

class ChooseSlow **implements** ChooseSpec
    public constructor ChooseSlow() **skip**
    public method $c(n : A, r : A) \ : \ \mathbb{N}$
        precondition $0 \leq r \leq n$
        if $r = 0 \vee r = n$ **then return** 1
        else return $c(n-1, r) + c(n-1, r-1)$
end class ChooseSlow

An optimized implementation

Use a tracking variable in the form of an array m of already computed result.

If $d(n,r)$ then $m(n,r)$ holds the precomputed answer

class ChooseMemo **implements** ChooseSpec
  private var $d : A \times A \to \mathbb{B}$
  private var $m : A \times A \to \mathbb{N}$
  invariant $\forall\, (n,r) \in A \times A \cdot$
    $0 \le r \le n \wedge d(n,r) \Rightarrow m(n,r) = \binom{n}{r}$
  public constructor ChooseMemo()
    for $(n,r) \in A \times A \cdot d(n,r) := \mathfrak{false}$
  public method $c(n : A, r : A) \ : \ \mathbb{N}$
    precondition $0 \le r \le n$
    if $\neg d(n,r)$ **then (**
      if $r = 0 \vee r = n$ **then** $m(n,r) := 1$
      else $m(n,r) := c(n-1,r) + c(n-1,r-1)$
      $d(n,r) := \mathfrak{true}$;**)**
    return $m(n,r)$
end class ChooseMemo

Correctness of parallel programs

Introduce a new operator $S \mid\mid T$ meaning commands S and T are executed in parallel.

We assume that S and T are made up of "atomic parts" that are executed without interruption.

A parallel program such as

$$(z := y \ ; \ w := x) \mid\mid (x := 0 \ ; \ y := 0)$$

can be thought of as an arbitrary interleaving of its parts:

$$\begin{aligned}
&(z := y \ ; \ w := x \ ; \ x := 0 \ ; \ y := 0) \\
\vee\ &(z := y \ ; \ x := 0 \ ; \ w := x \ ; \ y := 0) \\
\vee\ &(z := y \ ; \ x := 0 \ ; \ y := 0 \ ; \ w := x) \\
\vee\ &(x := 0 \ ; \ z := y \ ; \ w := x \ ; \ y := 0) \\
\vee\ &(x := 0 \ ; \ z := y \ ; \ y := 0 \ ; \ w := x) \\
\vee\ &(x := 0 \ ; \ y := 0 \ ; \ z := y \ ; \ w := x)
\end{aligned}$$

[Aside: This definition of the parallel operator is "syntactic" rather than "semantic" in that it relies on the syntactic notion of parts. For example, if we replace $x := 0 \ ; \ y := 0$ with the semantically equivalent $y := 0 \ ; \ x := 0$ we get a different result!]

We can understand a command $S \mid\mid T$ as a loop that nondeterministically interleaves the atomic parts from S and T.

while either S or T is not done
do **if** S is not done **then** do next part from S
□ T is not done **then** do the next part from T

Now the invariant of this loop becomes very important, as it is the only thing that we can depend on between executions of actions.

For example $(z := y \;;\; w := x) \mid\mid (x := 0 \;;\; y := 0)$ can be thought of as

$$
\begin{array}{l}
\textbf{var } c0, c1 := 0, 0 \\
\textbf{while } (c0 < 2) \vee (c1 < 2) \\
\textbf{do } \textbf{if } c0 = 0 \textbf{ then } (z := y; c0 := 1) \\
\qquad \square\ c0 = 1 \textbf{ then } (w := x; c0 := 2) \\
\qquad \square\ c1 = 0 \textbf{ then } (x := 0; c1 := 1) \\
\qquad \square\ c1 = 1 \textbf{ then } (y := 0; c1 := 2)
\end{array}
$$

The theory and practice of concurrent programming will be the subject of ENGI-9869 in the spring term.