Derivation of Algorithms

Lecture 3
Guarded Command Programming Language

COMP H 4018
Lecturer: Stephen Sheridan

Guarded Command Programming Language

In this section the guarded command programming notation is introduced. The semantics of each construct will be described informally and will rely on your knowledge of programming language constructs and your experience of programming. The semantics will be defined formally in chapter 2.

Code block: |[..]|

Constant: const

Data types: int, char, bool, real

Sequence: name[0..N) of type denotes a sequence of N elements indexed from 0 .. N-1

Declaration: var

Assignment: :=

Concurrent Assignment: var-list := dat-list e.g a,b := 3,4

Semi-colon: ;

Do nothing: skip

Guarded command: b → S

if .. fi : case analysis where each case is evaluated and one of those evaluating to true is chosen.

An example is:

```
if b.0  → s.0
[] b.1  →  s.1
[] b.2  →  s.2
:
[] b.n-1 → s.n-1
fi
```

Loop: do b → s od

Assertions: { .. }

In programs assertions describe the states of variables. They will play a very important part in both proving programs correct and in deriving programs to meet specifications. At this stage we will just use them to describe variable states and ignore any semantic role that they might play.

Sample Programs

```
|[ con N : int;
  { N ≥ 0 }
  var f, t : int;
  t, f := 1, 0;
  { f = # j : 0 ≤ i < t : j mod 3 = 0}
  do t < N →
    if t mod 3 = 0 →
        f := f + 1
    [] t mod 3 ≠ 0 →
        skip
     fi;
     t := t + 1
  od
  { f = # j : 0 ≤ j < N : j mod 3 = 0}
]|
```

```
|[ con N : int; { N ≥ 0 }
  var n : int;
  var freq : int;
  var f : array[0..N) of int;
  n : = 0;
  do n < N →
     f.n := rand(20) + 1;
     n := n + 1
  od
  { ∀j : 0 ≤ j < N : f.j > 0 ∧ f.j ≤ 20  }
  n : = 0; freq := 0;
  {freq = #j : 0 ≤ j < n : f.j = 12}
  do n < N →
      if f.n = 12 →
        freq := freq + 1
      [] f.n ≠ 12 →
         skip
      fi;
       n := n + 1
  od
  {freq = #j : 0 ≤ j < N : f.j = 12}
]|
```

Making Specifications

A program is specified by

- its state space
- a pre-condition
- a post-condition

Ex1: Specify a program to compute the quotient and remainder on dividing X by Y.

Specification

```
|[ con x,y : int; { x ≥ 0 ∧ y > 0}
   var q,r : int;
    S
  { x = y*q + r ∧ 0 ≤ r < y }
]|
```

Ex2: Given an integer array f[0..N), N ≥ 0, specify a program to compute the sum of the elements in f.

Specification

```
|[ con N : int; { N ≥ 0 }
  con f : array[0..N) of int;
  var sum : int;
  S
  { sum = +j:0 ≤ j < N : f.j}
]|
```

Ex3: Given an integer array f[0..N), N ≥ 0, initialised to random values in the range 1 .. 1000 specify a program that sorts the elements in f.

Specification

```
|[ con N : int; { N ≥ 0 }
  var f : array[0..N) of int;
  {∀j : 0 ≤ j < N : 1 ≤ f.j ≤ 1000}
  S
  {∀j : 0 < j < N : f.j-1  ≤ f.j}
]|
```

Axiomatic Semantics of Guarded Commands

To develop programs formally we need a formal definition of the behaviour of our programming constructs. To do this we need to develop a formal model of programs, define each programming statement using a formal semantics and give a meaning for the semi-colon i.e. the notion of sequence. In this part of the course we will formally specify the semantics of each of the statements in the guarded command language. We will also introduce the idea of proving programs correct using the formal semantics and the idea of state transformations. See the discussion of the semi-colon below.

State-Space

The variables of a program define the state space. For example, a declaration
x,y :int; gives a state space = Z * Z.

Note Predicates introduce sub-sets of the state space.

Consider the specification given below

$$
\begin{array}{l}
|[\ \text{con } x,y : \underline{\text{int}};\ \{\, x \geq 0 \wedge y > 0 \}\\
\quad \text{var } q,r : \text{int};\\
\quad\ S\\
\quad \{\, x = y*q + r \wedge 0 \leq r < y \,\}\\
]|
\end{array}
$$

The operational interpretation of this specification is:

> program S satisfies the specification if for all integers x, y, execution of S starting in a state satisfying
>
> $\{\, x \geq 0 \wedge y > 0 \}$
>
> **terminates** in a state satisfying
>
> $\{\, x = y * q + r \wedge 0 \leq r < y \,\}$.

In general, let P, Q be predicates and S a program

$$\{P\}\ S\ \{Q\}$$

states that: *execution of S terminates in a state satisfying Q when applied to a state satisfying P.*

Laws

1: Law of Excluded Miracle:

{P} S {false} is equivalent to [P ≡ false], i.e., execution of S satisfying P terminates in a state satisfying false (i.e. **no state**).

2 : {P} S {true} ≡ *S started in a state satisfying P terminates*.

Guarded Command Language

skip

The **skip** instruction brings about no change in the state space. Its semantics are defined as follows:

{P} skip {Q} ≡ [P ⇒ Q]

Example

```
|[ var x,y : int;
    {x ≥ 1}
      skip
    {x ≥ 0}
 ]|
```

follows from truth of

[x ≥ 1 ⇒ x ≥ 0]

Note: Weakest solution for {P} **skip** {Q} is Q because [Q ⇒ Q] for all Q.

Ex Prove the following program block correct.

```
|[ var x,y : int;
    { x > 0 ∧ y > 0}
      skip
    {x > 0}
]|
```

Proof :

{ x > 0 ∧ y > 0} skip {x > 0}
≡ [*def skip*]
(x > 0 ∧ y > 0) ⇒ x > 0
≡ [∧ *simplification*]
x>0⇒ x > 0
≡ [⇒]
true

Assignment

Let x be a program variable and E an expression of x's type x:=E replaces the value of x by that of E.

$$\{P\}\ x:=E\ \{Q\}$$

is equivalent to

$$P \Rightarrow Q\ (\ x:=E\ )$$

Note All changes of state are due to assignments

end note.

Prove

$\{\ x \geq z\ \}\ x:=\ x+1\ \{\ x \geq 0\}$

$$
\begin{aligned}
& (\ x \geq 0\ )\ (\ x:=\ x+1) \\
\equiv\ & [\text{substitution }] \\
& x+1 \geq 0 \\
\equiv\ & [\text{arithmetic }] \\
& x \geq -\ 1 \\
\Leftarrow\ & [\ \text{arithmetic }] \\
& x \geq 3
\end{aligned}
$$

Note 1 Read ' $\Leftarrow$ ' as " follows from "

end note

Note 2 The weakest solution of

$\{P\}\ x:=E\ \{Q\}$

is

$Q\ (\ x:=E\ )$

Note 3 Assignment only valid when E defined

$\therefore$ strictly the definition should be written as

$\{P\}\ x:=E\ \{Q\}$

is equivalent to

$P \Rightarrow def.E \wedge Q\ (x:=E)$

Note 4 Concurrent Assignment

Allow multiple assignments take place concurrently.

Ex $x,y\ :=\ y,x;$ swaps the values of x and y.

Catenation

S ; T
first execute S and then execute T.

{P} S ; T {Q}

is equivalent to a predicate R exists such that

{P} S {R} and {R} T {Q}

Note ' ; ' is a composition operator.

Ex : Given |[{P} x:=y+1; y:=5 {x > y }]| , find P.

(x > y) (y:=5)
≡ [substitution]
(x > 5) - R

(x > 5)(x:=y+1)
≡ [substitution]
y+1 > 5
≡ [arithmetic]
y > 4

P ≡ { y > 4 }

Annotating the program gives

|[{ y > 4 }
x:=y+1;
{x > 5}
y:=5
{x > y}
]|

Example 2
Prove

|[var a,b : Bool;
{ (a ≡ A) ∧ (b ≡ B) }
a:= a ≡ b;
b:= a ≡ b;
a:= a ≡ b
{ (a ≡ B) ∧ (b ≡ A) }
]|

Proof

((a ≡ B) ∧ (b ≡ A)) (a:= a ≡ b)
≡ [substitution]
((a ≡ b ≡ B) ∧ (b ≡ A))

((a ≡ b ≡ B) ∧ (b ≡ A)) (b:= a ≡ b)
≡ [substitution]
((a ≡ a ≡ b ≡ B) ∧ (a ≡ b ≡ A)
≡[≡]
(b ≡ B) ∧ (a ≡ b ≡ A)

((b ≡ B) ∧ (a ≡ b ≡ A))(a:= a ≡ b)
≡ [substitution]
(b ≡ B) ∧ (a ≡ b ≡ b ≡ A)
≡ [≡]
(b ≡ B) ∧ (a ≡ A)

Annotated Program

|[var a,b : bool;
{ (a ≡ A) ∧ (b ≡ B) }
a:= a ≡ b;
{(b ≡ B) ∧ (a ≡ b ≡ A) , Proof 0}
b:= a ≡ b;
{ (a≡ b ≡ B) ∧ (b ≡ A) , Proof 1 }
a:= a ≡ b
{ (a ≡ b) ∧ (b ≡ A) , Proof 2 }
]|

Selection

A guarded command is a statement whose execution is determined by the truth value of its guard. The format is:

$$b \rightarrow S$$

where b is an assertion and S a statement.

If .. fi Statement

Format

```
if b.0  → s.0
[] b.1  →  s.1
[] b.2  →  s.2
:
[] b.n-1 → s.n-1
fi
```

Notes $0 \le i < n : b.i$ - boolean expressions
$0 \le i < n : s.i$ - Statements

Operational Interpretation

Evaluate guards $(b.i)$ and if none evaluates to true **abort** ; otherwise choose one that is true and execute the corresponding $s.i$.

$$\{P\}\ \text{if}\ B0 \rightarrow S0 \ \square\ B1 \rightarrow S1\ \text{fi}\ \{Q\}$$

$\equiv$

(1) $P \Rightarrow B0 \vee B1$

(2) $\{ P \wedge B0 \}\ S0\ \{Q\}$ and
$\{ P \wedge B1 \}\ S1\ \{Q\}$

Annotated Program

Case where only two guarded commands :

```
{P}
if B0 → {P ∧ B0} S0 {Q, Proof 0}
□ B1 → {P ∧ B1} S1 {Q, Proof 1}
fi
{Q, Proof 2}
```

where

Proof 0: prove $\{P \wedge B0\}\ S0\ \{Q\}$
Proof 1: prove $\{P \wedge B1\}\ S1\ \{Q\}$
Proof 2: prove $[P \Rightarrow B0 \vee B1]$

Obviously we can extend this to general case.

Ex : Prove

$\{x=0\}$
if $true \rightarrow \{x=0 \wedge true\}\ x:=x+1\ \{x=1\}$
$[]\ true \rightarrow \{x=0 \wedge true\}\ x:=x+1\ \{x=1\}$
fi
$\{x=1\}$

Proof 0

$(x=1)(x:=x+1)$
$\equiv$ [substitution]
$x+1=1$
$\equiv$ [arithmetic]
$x=0$
$\equiv$ [calculus]
$\{x=0 \wedge true\}$

Proof 1 Similar.

Proof 2

$true \vee true$
$\equiv$ [calculus]
$true$
$\Leftarrow$
$x=0$

Alternative proof

$x=0 \Rightarrow true \vee true$
$\equiv$ [const]
$x=0 \Rightarrow true$
$\equiv$ [$\Rightarrow$]
$true$

Ex : Prove

```
|[ var a,b : bool;
{ true }
if ¬ a ∨ b → a : = ¬ a
□ a ∧ ¬ b → b : = ¬ b
fi
{ a ∨ b }
]|
```

Must Prove

Proof 0 : {true ∧ ¬ a ∨ b} a : = ¬ a {a ∨ b}
Proof 1 : {true ∧ a ∧¬ b} b : = ¬ b {a ∨ b}
Proof 2 : true ⇒ (¬ a ∨ b) ∨ (a ∧¬ b)

Proof 0

(a ∨ b) (a:= ¬ a)
≡ [substitution]
¬ a ∨ b
≡ [true - false]
true ∧ (¬ a ∨ b)

Proof 1

(a ∨ b)(b : = ¬ b)
≡ [substitution]
a ∨ ¬ b

Proof 2

true ⇒ (¬ a ∨ b) ∨ (a ∨ ¬ b)
≡ [assosciativity, commutativity]
true ⇒ (¬ a ∨ a) ∨ (b ∨ ¬b)
≡ [false - true]
true ⇒ true ∨ true
≡ [⇒]
true

Repetition

format

```
do B.0 → S.0
□ B.1 → S.1
:
od
```

Operational meaning

Evaluate guards (B.i) and if all false then skip : otherwise, choose one that is true and execute the corresponding S.i : following which, the repetition is executed again.

```
{P} do B → S od {Q}
```

is equivalent to

```
{P} if ¬ B → skip
    [] B → S
            ; do B → S od
   fi
{Q}
```

Annotating above gives:

```
{P}
if ¬ B → {P ∧ ¬ B} skip {Q}
[]  B → {P ∧ B} S; {P}
        do B → S od {Q}
fi
{Q}
```

Why choose P as intermediate ?

```
{P} do B → S od {Q}
```

should hold.

Note P is known as the **invariant** of the loop

Proof obligations

(1) $[P \wedge \neg B \Rightarrow Q]$
(2) $\{P \wedge B\}\ S\ \{P\}$
(3) $\{P\}$ do $B \rightarrow S$ od $\{Q\}$

where (3) gives rise to (1), (2) and (3) again.

But proof of (1) and (2) implies (3) **if termination can be shown.**

To prove termination you need a bound integer function on the state space that is bounded from below and that **decreases** with each execution of the loop. In your proof you must show that the function t is initially ≥ 0 and that it is a decreasing function.