Reasoning about Programs

Jeremy Bradley, Francesca Toni and Xiang Feng

`Room 372.  Office hour - Tuesdays at noon.  Email: jb@doc.ic.ac.uk`

Department of Computing, Imperial College London

Produced with prosper and LaTeX

Haskell v Java

- Cannot change values of variables in Haskell
  - Not allowed: `a := a + 1;`
- In Java:
  - Allowed: `a := a + 1;`
- In Java: try not to let functions change values of variables outside of scope of function

KISS Principle

- Reasoning will be easy if parts of program are simple:

  "There are two ways of constructing a first rate program: one is to make it so simple that there are obviously no deficiencies; the other is to make it so complicated that there are no obvious deficiencies." *Tony Hoare*

Pre/Post/Mid Conditions

- **Pre-condition** must be true before a method or function is entered, if code is to operate correctly
- **Post-condition** will be true after code has executed (as long as Pre-condition was met)
- **Mid-condition** is true at a specific *checkpoint* in the code while it is running

Sequential reasoning

```
void swapInts (int x, int y) {
  // pre: none
  // post: (x == y_0 && y == x_0)
  int z = x;
  x = y;
  y = z;
}
```

- In pre/post: var_0 refers to an input variable's initial value, var is intermediate/final value
- Allows reasoning about variables whose value alters over the course of the function
- Variables not mentioned in pre/mid/post are assumed unchanged i.e. $var = var_0$

Conditional reasoning

```
int intMin(int x, int y) {
    // pre: none
    // post: (res == x_0 || res == y_0)
    //          && (res <= x_0 && res <= y_0))

    int res;
    if (x <= y)
        res = x;
    else
        res = y;
    return res;
}
```

- where `res` is notation for return variable

intMin with mid-conditions

```
int intMin(int x, int y) {
    // pre: none
    // post: (res == x_0 || res == y_0)
    //          && (res <= x_0 && res <= y_0))

    int res;
    if (x <= y)
        res = x;
    // mid case x <= y: (res == x_0 && res <= y_0 )
    else
        res = y;
    // mid case x > y: (res == y_0 && res <= x_0 )
    return res;
}
```

Reasoning with mid-conditions

- From intMin program:
  - Need to reason from pre-condition to mid-condition:

$$\begin{aligned} \mathsf{tt} \;\vdash\; & (res = x_0 \land res \leq y_0) \\ & \lor (res = y_0 \land res \leq x_0) \end{aligned}$$

  - Need to reason from mid-condition to post-condition:

$$\begin{aligned} & (res = x_0 \land res \leq y_0) \\ \vdash\; & (res = x_0 \lor res = y_0) \\ & \land (res \leq x_0 \land res \leq y_0)) \end{aligned}$$

Swapping variable values

```
class Swap1 {
    public static void swap (int i, int j) {
        int t=i;
        i = j;
        j = t;
        return;
    }
    public static void main ( String args[] ) {
        int a = 1;
        int b = 2;
        swap(a,b);
    }
}
```

Swapping variable values

- The method `swap1.swap` does not swap the values of i and j
- Why? – *call-by-reference* versus *call-by-value*
  - i.e. no side-effects
- In Java, all user classes are passed *by reference*
  - i.e. side-effects can happen

Call-by-reference in Java

- For the following coordinate class:

```
class Point {
    int xc;
    int yc;

    Point (int i, int j) {
        xc = i;
        yc = j;
    }
}
```

Call-by-reference in Java

```
class Swap { \\ Swaps coordinates of point Q
    public static void swap (Point Q) {
        int t = Q.xc;
        Q.xc=Q.yc;
        Q.yc=t;
        return;
    }
    public static void main ( String args[] ) {
        Point P = new Point (10,25);
        swap (P);
    }
}
```

- Correct (but complicated) swap method

Simplified swap method

```
public void swap () {
    // Pre: none
    // Post: xc == yc_0 && yc == xc_0
    int t;
    t = xc;
    xc = yc;
    yc = t;
    return;
}
```

- Simpler class-related swap implementation

Simplified swap method

```
      public void swap () {
         // Pre: none
         // Post: xc == yc_0 && yc == xc_0
         int t;
[1]      t = xc;  // a. t == xc_0 && yc == yc_0
[2]      xc = yc; // b. t == xc_0 && xc == yc_0
[3]      yc = t;  // c. xc == yc_0 && yc == xc_0
         return;
      }
```

- Here we have 2 mid-conditions (a) and (b), and the post-condition (c)
- Important lines of code are numbered [n]

Using natural deduction

- From pre-condition to mid-condition (a):
  - $\vdash t = xc_0 \wedge yc = yc_0$

1.	$xc = xc_0$	var$\mathcal{I}$
2.	$yc = yc_0$	var$\mathcal{I}$
3.	$t = xc$	code[1]$\mathcal{I}$
4.	$t = xc_0$	$=$trans$(1, 3)$
5.	$t = xc_0 \wedge yc = yc_0$	$\wedge\mathcal{I}(2, 4)$

New reasoning tools

- var$\mathcal{I}$
  - used to introduce implicit pre-condition assumptions
  - not needed if pre-condition is stated in full
- code[n]$\mathcal{I}$
  - used to introduce line n from the program
- trans
  - transitivity property, e.g.
    - if $a = b$ and $b = c$ then $a = c$
    - if $x \leq y$ and $y \leq z$ then $x \leq z$

New reasoning tools

Also require:

- def
  - when using a definition e.g.

$$a \leq b \equiv a = b \vee a < b$$

- $=$subs
  - using an equality to replace a variable e.g.
    1. $x = z + 1$
    2. $\vdots$
    3. $z = y_0$
    4. $x = y_0 + 1$ $\qquad$ $=$subs$(1, 3)$

Back to intMin

```
int intMin(int x, int y) {
    // pre: none
    // post: (res == x_0 || res == y_0)
    //           && (res <= x_0 && res <= y_0))
    int res;
    if (x <= y)
[1]     res = x;
    // mid case x <= y: (res == x_0 && res <= y_0 )
    else
[2]     res = y;
    // mid case x > y: (res == y_0 && res <= x_0 )
    return res;
}
```

Pre-condition to mid-condition

- Require to show:

$$\vdash (res = x_0 \wedge res \leq y_0) \vee (res = y_0 \wedge res \leq x_0)$$

1.	$x = x_0$	var$\mathcal{I}$
2.	$y = y_0$	var$\mathcal{I}$
3.	$x \leq y \vee x > y$	lem

4.	$x \leq y$	ass	10.	$x > y$	ass
5.	$res = x$	code[1]$\mathcal{I}$	11.	$res = y$	code[2]$\mathcal{I}$
6.	$res = x_0$	$=$trans$(1, 5)$	12.	$res = y_0$	$=$trans$(2, 11)$
7.	$res \leq y$	$=$subs$(4, 5)$	13.	$res < x$	$=$subs$(10, 11)$
8.	$res \leq y_0$	$=$subs$(2, 7)$	14.	$res < x_0$	$=$subs$(1, 13)$
9.	$res = x_0 \wedge res \leq y_0$	$\wedge\mathcal{I}(6, 8)$	15.	$res < x_0 \vee res = x_0$	$\vee\mathcal{I}(14)$
			16.	$res \leq x_0$	$\leq$def(15)
			17.	$res = y_0 \wedge res \leq x_0$	$\wedge\mathcal{I}(12, 16)$

18.	$(res = x_0 \wedge res \leq y_0) \vee (res = y_0 \wedge res \leq x_0)$	$\vee\mathcal{E}(3, 4, 9, 10, 17)$

How to cope with `x = x + 1`

- How do we deal with statements that modify an input variable x based on the old value of x. e.g.
  - `x = x + 1`
  - `x = 2 * x`
  - `x = 3 * z % x`
- Answer: need to introduce a sequence of x variables as well as x_0: i.e. x_1, x_2,x_3,...
- Extra variables keep track of all the intermediary values of x before the final version is calculated

Example: extra variables

```
public int intInc (int x) {
        // Pre: none
        // Post: x == 2*x_0 + 2
[1]     x = x + 1;
[2]     x = 2 * x;

    return x;
}
```

- Extra variables needed as x has 3 values during method execution
- We will see that we also need to modify the behaviour of VAR and CODE keywords...

Example: extra variables

- Reasoning for `intInc` method:
  1. $x_1 = x_0$ — var $\mathcal{I}$
  2. $x_2 = x_1 + 1$ — code[1] $\mathcal{I}$
  3. $x_3 = 2 * x_2$ — code[2] $\mathcal{I}$
  4. $x = x_3$ — var $\mathcal{I}$
  5. $x_2 = x_0 + 1$ — =subs$(1, 2)$
  6. $x_3 = 2 * (x_0 + 1)$ — =subs$(3, 5)$
  7. $x_3 = 2 * x_0 + 2$ — distributivity def(6)
  8. $x = 2 * x_0 + 2$ — =subs$(7, 4)$

Modifications to var

- var
  - is used to introduce the first extra variable in terms of the initial value: $x_1 = x_0$
  - is used to set the final value, x, to the last in the sequence of extra x-variables, in this case: $x = x_3$

Modifications to code

- code[n]
  - is used to introduce code from line n
  - if a variable undergoes a change of value during reasoning e.g. $x = f(x)$, then extra variables must be used, i.e.

$$x_{i+1} = f(x_i)$$

    where i is the index of the last extra variable used

Modifications to code

- code[n]
  - code[n] statements must be introduced in program order so that correct variable names can be set
  - code[n] statements in while/if clauses can only be introduced if associated branch/loop tests are true

Summary: Extra variables

- Note that the final result value is still x and is equal to the last supplementary variable
- We should not need many extra variables if we create sufficient mid-conditions
  - mid-conditions help to break up the reasoning into smaller easier chunks
- The result value x might be the value in a mid-condition **or** a post-condition depending on which we are trying to derive

More mid-conditions...

- Need to augment `Point` class with `up` and `right` methods:

```
...
public void up (int n) {
    // Pre: none
    // Post: xc == xc_0 && yc == yc_0 + n
    yc = yc + n;
}
public void right (int n) {
    // Pre: none
    // Post: xc == xc_0 + n && yc == yc_0
    xc = xc + n;
}
...
```

More mid-conditions...

- Can reason about evolution of coordinates from method call to method call

```
    ...
    public static square (Point P, int n) {
        // Pre: none
        // Post: xc == xc_0 && yc == yc_0

[1]     P.right(n);  // xc == xc_0+n && yc == yc_0
[2]     P.up(n);     // xc == xc_0+n && yc == yc_0+n
[3]     P.right(-n); // xc == xc_0   && yc == yc_0+n
[4]     P.up(-n);    // xc == xc_0   && yc == yc_0
    }
    ...
```

Using lower level post-conditions

- We are going to assume that `Point.left` and `Point.right` have been proved correct
- We now have to prove that `square` meets its post-condition
- i.e. $\vdash xc = xc_0 \wedge yc = yc_0$

1. $xc_1 = xc_0$ var $\mathcal{I}$
2. $yc_1 = yc_0$ var $\mathcal{I}$
3. $xc_2 = xc_1 + n \wedge yc_2 = yc_1$ pc[1] $\mathcal{I}$
4. $xc_3 = xc_2 \wedge yc = yc_2 + n$ pc[2] $\mathcal{I}$
5. $\vdots$

Some more extra notation

- pc[n]
  - Introduces the post condition of the method at line n
- Same behaviour as code[n] when creating intermediate variables between the initial value xc_0 and final value xc
  - hence introduction xc_1 between start of `square` and beginning of `P.right(n)`
  - might optionally need xc_2, $xc_3, \ldots$ depending on how many post-conditions we are using

Important rules

- For pc/code statements:
  - introduce lines into reasoning in program order
  - only introduce pc/code statements from if/while clauses if branch/loop tests met
- If variable changes value during reasoning then will require extra variables
  - applies to local and global method variables

Class invariants

- Reasoning specific to an OO paradigm
- Class invariant
  - is a logical property that is true of a class and its data at all times
  - needs to be true for after each constructor method
  - needs to be shown that invariant is *reestablished* after each (non-constructor) method call

Class invariant example

```
class Total {
    // Class invariant: i >= 0
   int i;

    Total () {
        i = 0;
    }
    void addto(int x) {
        // Pre: x_0 > 0
        // Post: i == (i_0 + x_0)
        i += x;
    }
}
```

Class invariant

- After `Total ()`: $i = 0 \geq 0 \surd$
- Invariant re-established after `addup(x)`:
  - Show: $i_0 \geq 0 \wedge x_0 > 0 \wedge (x_0 > 0 \rightarrow (i = (i_0 + x_0))) \vdash i \geq 0$
  - In general:

    variant before$\wedge$pre$\wedge$(pre $\rightarrow$ post) $\vdash$ variant after