Logic for Program Call chapter 7

Dealing with program calls

- Consider a program with a known specification:

  {* P *} Pr(x) {* Q *}

- Suppose another program S that calls Pr. To reason about S' correctness we will have to include reasoning about Pr as well. Options:
  - Inline the code of Pr → does not work if we do not have Pr's code, or if Pr is recursive.
  - We "plug-in" the specification of Pr. How? The following slides will focus on this approach.

Program call in uPL

- For simplicity, uPL only allows a program to be called is as a "statement" in either of this variant :

  pname(*actual-params*)

  var := pname(*actual-params*)

  Calling a program in expression is not allowed, e.g:

  y := ifoo(x) +1 ;

Instantiating a specification

- Inevitably, when handing a program call we will need to instantiate the callee's specification with the used actual parameters.
- Example, consider this specification:

{* base>0 *} **R**:=res; **incr**(base, **OUT** res) {* res>**R**+base *}

Consider a call **incr**(b+1,b).
Simply replacing the formal with actual parameters can give a wrong specification on a program that can do side effect:

{* b+1>0 *} **B**:=b; **incr**(b+1,b) {* b>**B**+b+1 *}

But this is not a valid specification (take b=0 initially). The issue is caused by confused reference to two different b's. The red b replaces "base", which is a pass-by-value parameter, so in the post-condition it refers to its **initial value**. The purple b replaces "res" which is an OUT parameter, so it refers to its **final value**.

Instantiating a specification

To make it safe, we will require that a program-level specification **can only be instantiated by renaming its parameters and auxiliary variables**, and furthermore they all must be **distinct**.

- Rule 7.2.1 (for two parameters, with one is an OUT param):

$$\frac{\{*\ P\ *\}\quad X,Y:=x,y\ ;\ \mathbf{Pr}(x,\ \mathrm{OUT}\ y)\quad \{*\ Q\ *\}}{\{*\ P[x',y'/x,y]\ *\}\quad X',Y':=x',y'\ ;\ \mathbf{Pr}(x',y')\quad \{*\ Q[x',y',X',Y'/x,y,X,Y]\ *\}}$$

- We will handle complex actual parameters via program transformation (later).

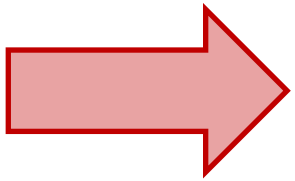
Transforming the call first

Consider the following program, containing a single call to another program:

```
{* b>0 *}

x:=b ; incr(b+1,b)

{* b ≠ x *}
```



```
{* b>0 *}

x:=b ;
@b,@r := b+1,b
incr(@b,@r)
b := @r

{* b ≠ x *}
```

To safely instantiate the specification of incr, we first transform the program to an equivalent one (right). Fresh and distinct variables @b and @r are introduced as proxies of the actual parameters.

Handling the resulting transformation

```
{* b>0 *}

x:=b ;
@b,@r := b+1,b
incr(@b,@r)
b := @r

{* b ≠ x *}
```

How to proceed now to prove the correctness of this program?

- Through wp we can calculate the post-condition for **incr**(@b,@r), namely @r ≠ x. But then, how to reason over the program call itself?
- **Idea**: given the specification of incr, what would be the best pre-condition that guarantees incr to end up with @r ≠ x ?

“Black box” reduction

- (1) The given specification of incr:

{* base>0 *} **R**:=res; **incr**(base, **OUT** res) {* res>**R**+base *}

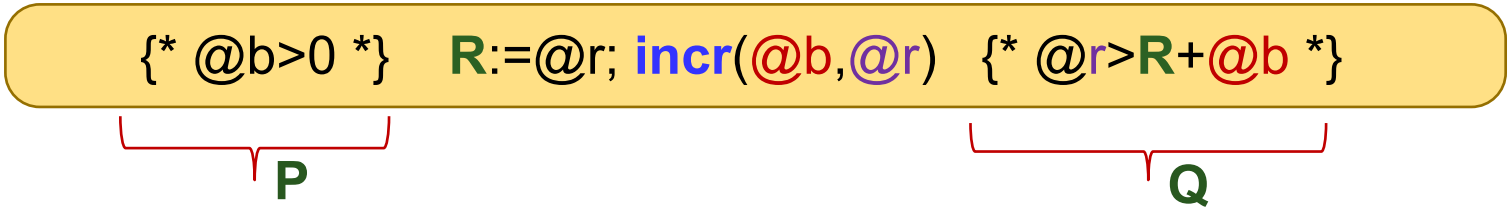
- (2) After instantiation with the actual parameters:

{* @b>0 *} **R**:=@r; **incr**(@b,@r) {* @r>**R**+@b *}

- Consider now a call **incr**(@b,@r) with some arbitrary post condition Q’ to establish. **Question**: given (2) come up with the “best” pre-condition for the call so that it will establish Q’.

“Black box” reduction

- So, given this (the formula 2 from prev. slide) :



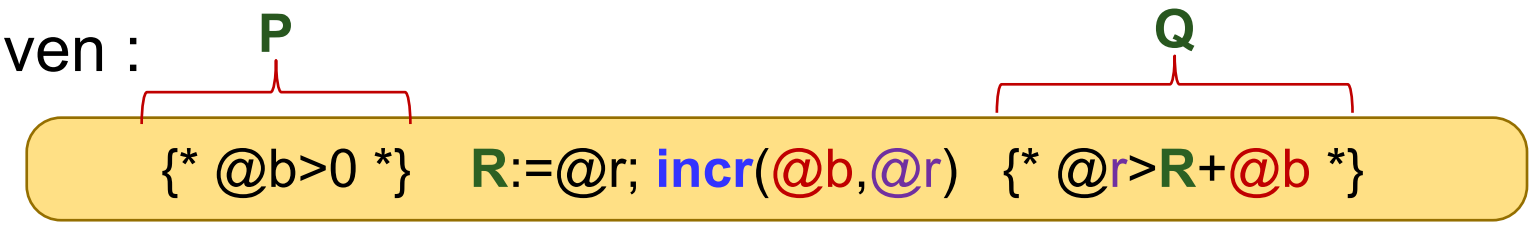
- What is the best P’ such that {* P’ *} **incr**(@b,@r) {* Q’ *} ?

  Idea:

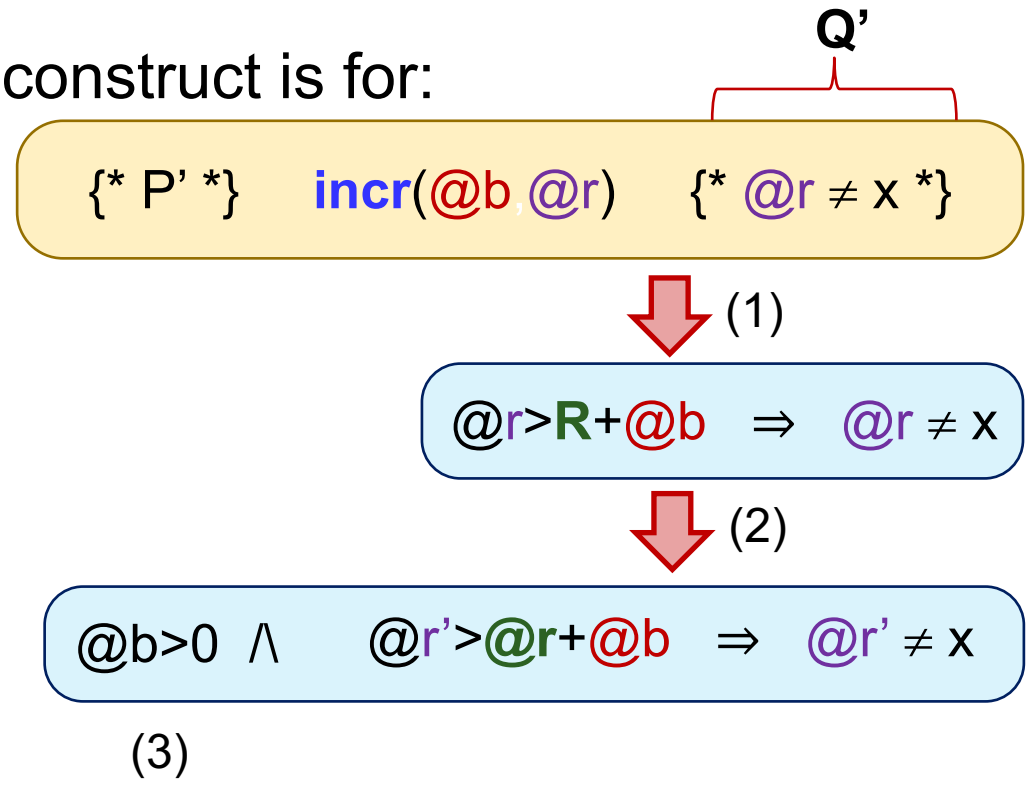
  - Require Q ⇒ Q’ , but to interpret this as a pre-condition requires us to “detach” references to final values of variables that receive side effect of **incr**.
  - We also need to include P, to make sure that **incr** will end up in Q at all.

"Black box" reduction

- Given :



- In our example, the P' to construct is for:



Constructing P' :

1. We need Q⇒Q'
2. Replace occurrences of **out** parameters with fresh variables to detach them.
   Replace occurrences of auxiliary variables representing initial values with the variables they represent.
3. We also need to pre-condition P.

Back to the problem

1

{* b>0 *}

3

{* b+1>0 /\ @r'>b+b+1 ⇒ @r' ≠ b *}

(by calculating wp)

x:=b ;

{* b+1>0 /\ @r'>b+b+1 ⇒ @r' ≠ x *}

(by calculating wp)

@b,@r := b+1,b

{* @b>0 /\ @r'>@r+@b ⇒ @r' ≠ x *}

(by black box reduction we did in the previous slide)

incr(@b,@r)

{* @r ≠ x *}

(by calculating wp)

b := @r

{* b ≠ x *}

2

The calculation produces (3) as a sufficient pre-condition for (2). The correctness of the program can be proven by proving the implication (1) ⇒ (3).

Black Box Rule

- Formally Rule 7.2.2 (for two parameters, with y being an OUT param):

$$\frac{\{*\ P\ *\}\quad X,Y := x,y\ ;\ \mathbf{Pr}(x,\mathbf{OUT}\ y)\quad \{*\ Q\ *\}}{\{*\ P \wedge (Q \Rightarrow \mathbf{Q'})\ [y'/y]\ [x,y/X,Y]\ *\}\quad \mathbf{Pr}(x,y)\quad \{*\ \mathbf{Q'}\ *\}}$$

  where y' must be a fresh variable.

- Notice:
  - Occurrences of OUT param/var in Q is replaced by fresh variables.
  - Occurrences of X and Y (initial values of the params) are replaced by x and y (the vars they represent).

Black Box Rule

- Let $\underline{x}$ be a vector of distinct parameter names, and similarly $\underline{X}$ be a vector of distinct (auxiliary) variables.
- Rule 7.2.2, general form:

$$\frac{\{* P *\} \quad \underline{X} := \underline{x} \,;\, \mathbf{Pr}(\underline{x}) \quad \{* Q *\}}{\{* P \;\wedge\; (Q \Rightarrow \mathbf{Q'})[\underline{x}'//\underline{x}][\underline{x}/\underline{X}] \;*\} \quad \mathbf{Pr}(\underline{x}) \quad \{* \mathbf{Q'} *\}}$$

where $\underline{x}'//\underline{x}$ is a substitution that replaces out-params with fresh variables.

"Functional" box

- Calls to a program that behaves **functionally** can be handled more easily via Rules 7.3.2. Let x be a vector of (distinct) formal parameters and d be a vector of actual parameters (which can be complex):

$$\frac{\{*\ P\ *\}\quad \mathbf{Pr}(\underline{x})\quad \{*\ \text{return} = e\ *\}}{\{*\ P[\underline{d}/\underline{x}] \wedge \mathbf{R}[\mathbf{e'}/y]\ *\}\quad y := \mathbf{Pr}(\underline{d})\quad \{*\ \mathbf{R}\ *\}}$$

where **e'** = e[d/x].

y := e'

See this as **wp** of the assignment y := e'

- all parameters are assumed to be passed-by-val.

Recursion

- Let **Pr**(int n,x) be a program that is recursive over n. Rule 7.4.1: a (simplified) induction rule to handle recursion:

$P \Rightarrow n \geq 0$ \\n is bounded below

{* P /\ n=0 *} **Pr**(n,x) {* Q *} \\base case

{* P /\ n<K *} **Pr**(n,x) {* Q *}
implies
{* P /\ n=K /\ K>0 *} **Pr**(n,x) {* Q *} \\induct. Case

{* P *} **Pr**(n,x) {* Q *}

- Some details omitted; see LN.
- Rule 7.4.2 provides a simplified form to handle recursive programs that behave functionally.

Example

- As a simple example, consider the following recursive program that simply returns 0 if given a integer $n \geq 0$. Let's try to prove its specification:

  {* $n \geq 0$ *}

  P(int n) { **if** n = 0 **then** return n **else** return **P**(n−1) }

  {* return = 0 *}

- The base case is not difficult. Let's see the induction case.

The induction case

- The induction case boils down to proving the following, after reducing the spec to **P**'s body:

{* n ≥ 0 /\ n=K /\ K>0 *}
if n=0 **then** return:=n **else** return := **P**(n−1)
{* return = 0 *}

By applying the functional Black Box rule 7.3.2, and using the induction hypothesis we can calculate the needed pre-condition for this call, namely:

n-1≥0 /\ n-1< K /\ 0=0

Then we can calculate the wp of the if-then-else, and finish the proof.