Software Verification

Matthew Parkinson
Lent 2010 (16 Lectures)

Overview

Part 1: Basics
- Introduction
- Control Flow
- Functions and local variables
- Arrays

Part 2: Heap

Part 3: Concurrency

Part I: Basics

1. Introduction

The first lecture covers the basic of Hoare logic. To supplement this material, I would recommend you read:

Mike Gordon's, Specification and Verification I, Course Notes, Chapters 1 and 2; and

Glynn Winskel's, Formal Semantics of Programming Languages, Chapter 6 and 7.

For background on operational semantics see Winskel's book as well.

Examples

Double free

Buffer overrun

Memory leaks

Termination failure


```c
#include <SLAyer_malloc.h>

int main() {
  int *x = (int*)malloc(sizeof(int));
  free(x);
  free(x);
  x = (int*)malloc(sizeof(int));
}
```

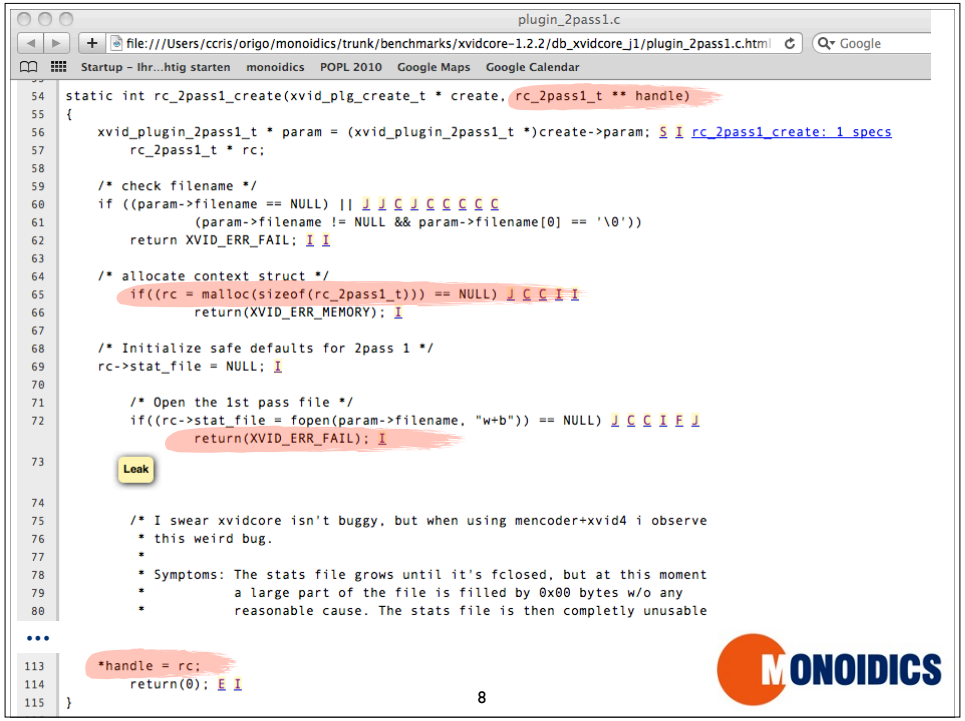

Here is a trivial example of an incorrect C program. It allocates a block of memory using malloc and then frees it twice.

[Thanks to Samin Istiaq and the rest of the SLAyer team for this example.]

```c
/**

   reverse_list_unsafediv2.
   Like reverse_list, but is memory unsafe *and*
   nondeterministically diverges.

 **/

//#include <stdio.h>
#include "sll.h"


/*
  Reverse the list pointed to by l.
  Implemented by poping off each elt of *l into lr.

  This reverse diverges non-deterministically.
*/
void reverse(PSLL_ENTRY *l)
{
  PSLL_ENTRY x = *l, lr = NULL;
  while(x != NULL) {
    PSLL_ENTRY t;
    t = x;
    x = x->Flink;
    t->Flink = lr;
    lr = t;
    if (nondet() /*&& x!=NULL*/) {
      t = x;
      x = x->Flink;
      t->Flink = lr;
      lr = t;
    }
  }
  *l = lr;
}
```

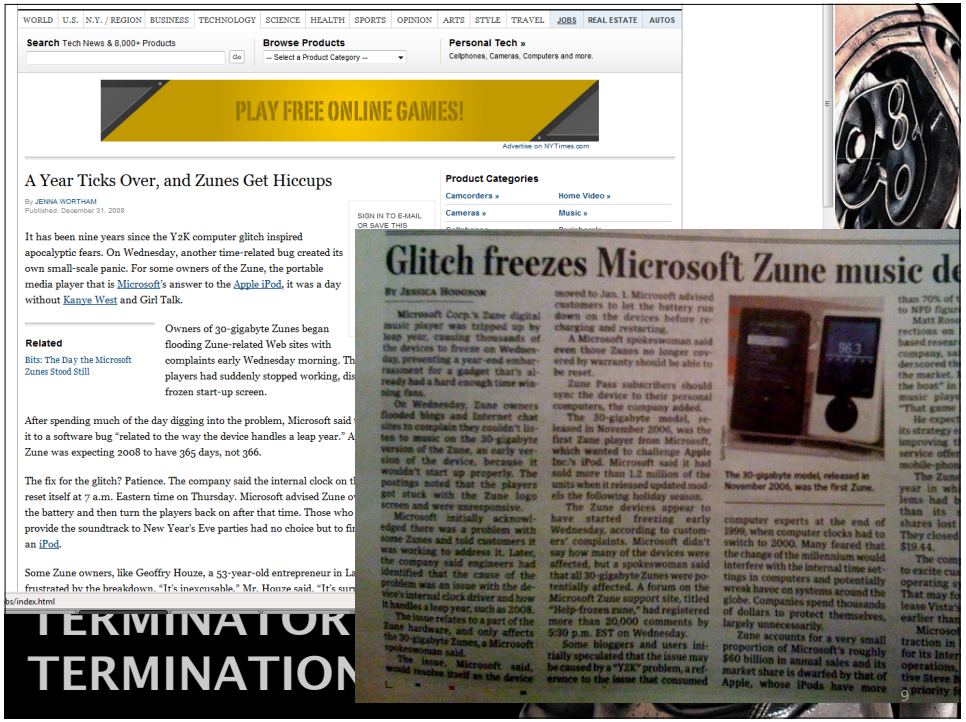

The commented x!=NULL should not be commented. This means the part that non-deterministically skips an element can go passed the end of the list. This bug is based on a bug found in a device driver. They have a common pattern of applying an operation to some elements of a list.

[Thanks to Samin Istiaq and the rest of the SLAyer team for this example.]



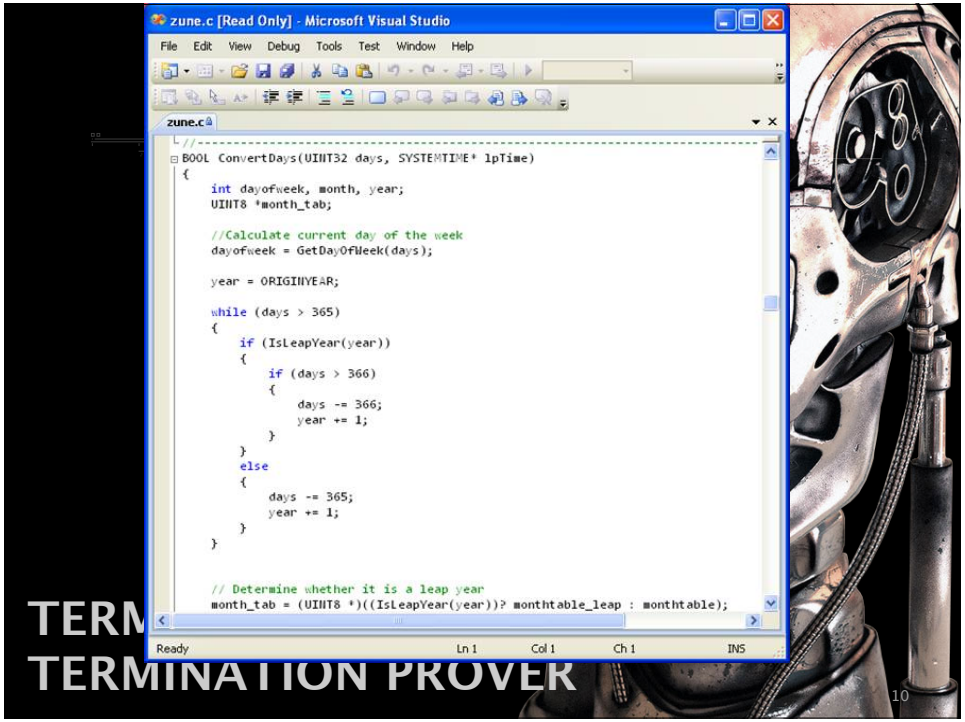
In this example the procedure takes a pointer to a pointer to a handle, on completion this should point to an appropriate C structure. This structure is allocated, but in the case of failing to open the file, an error code is returned and the handle is not updated. Hence, we have a memory leak.

[Thanks to Cristiano Calcagno and the rest of the SpaceInvader/ Monoidics team]



You may have heard that on the first of January 2009 the Microsoft music device, Zune, stopped working for one day. Why was this?

[Thanks to Samin Istiaq and the rest of the Terminator team for this example.]



It is because of this loop not terminating. Why doesn't it terminate?

More details in
Wednesday seminar 14:15, 17th February by Byron Cook
and
Guest lecture, Monday 22nd February at 2pm, by Byron Cook.

[Thanks to Samin Istiaq and the rest of the Terminator team for this example.]

Programming language

Boolean expressions:

B ::= true | false | E = E | E < E | ¬B | B ∧ B | …

Integer expressions:

E ::= n | x | E + E | E - E | ...

Commands:

C ::= x := E | if B then C else C | C;C | skip | …

Through this course we will consider a single simple programming language. The language has integer and boolean expressions, both of which are side-effect free, (they cannot mutate the state of the machine.) These expressions are used in commands. Commands can update the state of the machine. We will extend this language through the course to add new programming concepts and associated reasoning principles.

Boolean expressions are boolean constants, true and false; equalities between integer expressions, E=E; inequalities between integer expressions, E<E; and boolean operations such as negation, ¬B, and conjunction, B∧B. Integer expressions are integer constants, n; program variables, x; and integer operations such as addition, E+E, and subtraction, E-E. Commands are assignments, x:=E; and control flow operations such as conditional branches, if B then C else C, sequencing, C;C, and the empty command, skip.

The semantics of the boolean and integer expressions is simply to evaluate the expression in the current state. The assignment command, x:=E, updates the program variable x to be the value of evaluating the expression E in the current state. The conditional evaluates the boolean expression and executes the first branch if it is true, and the second branch if it evaluates to false. Sequencing executes the first command, and upon successful completion executes the second. The empty command, skip, does nothing.

Assertions

Classical logic interpreted over program states

P ::= B | P ∧ P | P ∨ P | ¬P | P ⇒ P | ∃x. P | ∀x. P | …

We use standard assertions of classical logic. We have primitive assertions of boolean program expressions. We interpret assertion over the state of the machine, σ. A machine state is a mapping from program variables to their values. We use $\sigma \models P$ to mean the state σ satisfies P. It is defined as:

$\sigma \models B$	⇔	$[\![B]\!]s = true$
$\sigma \models P_1 \wedge P_2$	⇔	$\sigma \models P_1$ and $\sigma \models P_2$
$\sigma \models P_1 \vee P_2$	⇔	$\sigma \models P_1$ or $\sigma \models P_2$
$\sigma \models \neg P_1$	⇔	not $\sigma \models P_1$
$\sigma \models P_1 \Rightarrow P_2$	⇔	If $\sigma \models P_1$, then $\sigma \models P_2$
$\sigma \models \exists x.\ P_1$	⇔	If there exists v such that $\sigma[x:=v] \models P_1$
$\sigma \models \forall x.\ P_1$	⇔	For all v, $s[x:=v] \models P_1$

Floyd/Hoare logic

Specification (Hoare Triple):

{ P } C { Q }

Semantics

If executing C in an initial state satisfying P terminates, then the final state satisfies Q.

The basic building block of Hoare logic is the triple: {P}C{Q}. Here P is called the pre-condition; Q is called the post-condition; and C is the program, or command. P and Q are logical assertions about the state of the machine.

The idea is that before using/calling C the programmer must establish P is true, after calling/using C the programmer can assume that Q will hold. We do not need to consider what C is. We can reason solely about its specification. Note that the specification does not say anything about the final state if the pre-condition is not satisfied on entry. Also, this does not specify whether C will terminate or not, this is called "partial correctness". There is an alternative formulation of Hoare logic called "total correctness", written [P]C[Q]. Semantically, [P]C[Q] means if C is executed in a initial state satisfying P, then it will terminate in a final state satisfying Q.

You can see this as a formal way to view an API, or other library documentation. An API typically describes what the program must establish before calling a method or function. The API also describes what it will establish upon completing the method.

Example triples

{ x=5 } x := 3 { x =3 }

{ x=4 } x := x+1 { x=5 }

{ y=3 ∧ x=3 } y := x+y { y=6 ∧ x=3 }

{ odd(x) } y := x + x + x { odd(y) ∧ odd(x) }

The first triple says

if we start in a state in which the value of the variable x is 5, then if we terminate then it will end in a state in which x has value 3.

The second triple says

if we start in a state in which the value of the variable x is 4, then if we terminate then it will end in a state in which x has value 5.

The two programs are quite different, the first does not care about the initial value of x to achieve its post-condition. Its specification's pre-condition is two strong for its post-condition. Alternatively, we can see this as abstracting details, both x:=3 and x:=x-2 satisfy this specification.

The fourth specification does not demand a precise value of x in the pre-condition just that it be odd. It asserts that the value of y will also be odd afterwards.

Example triples (cont)

{ i < j } quicksort(a, i, j) { sorted(a,i,j) }

Example - Java LinkedList

remove

`public boolean` **`remove`**`(Object o)`

Removes the first occurrence of the specified element in this list. If the list does not contain the element, it is unchanged. More formally, removes the element with the lowest index `i` such that `(o==null ? get(i)==null : o.equals(get(i)))` (if such an element exists).

Specified by:
`remove` in interface `List`
Overrides:
`remove` in class `AbstractCollection`

Parameters:
`o` - element to be removed from this list, if present.
Returns:
`true` if the list contained the specified element.

Now consider a more realistic specification. Often code will initially check its arguments, this is called defensive programming. If you read the source of a lot of the standard Java libraries, a lot of the code is defensive. Here we can precisely state the requirements, and the obligations of the library. This has been found useful for testing as well as verification: Spec# and JML add runtime tests based on the specifications.

Note that, this specification is not going to be good enough, as we will see in later lectures. It does not specify that the array is a permutation of the original data.

toArray

`public Object[] toArray(Object[] a)`

Returns an array containing all of the elements in this list in the correct order; the runtime type of the returned array is that of the specified array. If the list fits in the specified array, it is returned therein. Otherwise, a new array is allocated with the runtime type of the specified array and the size of this list.

If the list fits in the specified array with room to spare (i.e., the array has more elements than the list), the element in the array immediately following the end of the collection is set to null. This is useful in determining the length of the list *only* if the caller knows that the list does not contain any null elements.

Specified by:
`toArray` in interface `List`
Overrides:
`toArray` in class `AbstractCollection`

Parameters:
`a` - the array into which the elements of the list are to be stored, if it is big enough; otherwise, a new array of the same runtime type is allocated for this purpose.
Returns:
an array containing the elements of the list.
Throws:
`ArrayStoreException` - if the runtime type of a is not a supertype of the runtime type of every element in this list.
`NullPointerException` - if the specified array is null.

Is the NullPointerException that can be thrown a client bug?

If it returns a new array, can you still use the array you pass as an argument?

Skip axiom

{ P } skip { P }

The skip is the empty command: it does nothing. The rule says whatever is true before executing skip is also true afterwards. It does nothing after all.

If rule

$$\frac{\{ B \wedge P \}\ C_1\ \{ Q \} \qquad \{ \neg B \wedge P \}\ C_2\ \{ Q \}}{\{ P \}\ \text{if } B \text{ then } C_1 \text{ else } C_2\ \{ Q \}}$$

19

This rule deals with a conditional test. If the condition, B, is evaluated to true then the C_1 branch will be followed. And if the condition is evaluated to false then the C_2 branch will be executed.

The rule can be read as,

If executing C_1 in a state satisfying both B and P terminates, then the resulting state will satisfy Q; and

if executing C_2 in a state satisfying both ¬B and P terminates, then the resulting state will satisfy Q;

then if executing "if B then C_1 else C_2" in a state satisfying P terminates, then the resulting state will satisfy Q.

For example, consider trying to find the max of two numbers

{true} if x>y then z:=x else z:=y { z = max(x,y) }

Then we must just prove that

{ true ∧ x>y } z:= x { z=max(x,y) }

and

{ true ∧ x ≤ y } z:=y { z=max(x,y) }

We will get to proving them later.

Sequencing rule

$$\frac{\{ P \}\ C_1\ \{ R \} \qquad \{ R \}\ C_2\ \{ Q \}}{\{ P \}\ C_1; C_2\ \{ Q \}}$$

20

To verify a sequential composition, we must provide an intermediate assertion, R, for the state between the two commands.

Here we can see part of the abstraction that specifications provide. The assertion R that is true between the execution of C_1 and C_2, does not appear in the overall specification. Hence, the specification does not capture all the details of the computation, just the pertinent details of the start and the end of the computation. For example consider:

x := 2;
y := 3

and

y := 3;
x := 2

These two programs can be given the same specification

{ true } _ { x=2 ∧ y=3 }

The intermediate assertion, will however be different in each proof.

Assignment axiom

$\{ P [x:=E] \}\ x := E\ \{ P \}$

This is perhaps one of the hardest axioms of Hoare logic to understand.

We define P[x:=E], E'[x:=E] and B[x:=E], to mean replace all occurrences of x in the formula P, E' and B with the expression E:

$x\,[x := E] \equiv E$

$y\,[x := E] \equiv y$ where $y \neq x$

$n\,[x := E] \equiv n$

$E_1 + E_2\,[x:=E] \equiv E_1[x:=E] + E_2[x:=E]$

…

$E_1 = E_2\ [x:=E] \equiv E_1[x:=E] = E_2[x:=E]$

$P_1 \wedge P_2\ [x:=E] \equiv P_1[x:=E] \wedge P_2[x:=E]$

…

$\exists y.\ P\ [x:=E] \equiv \exists y.\ P[x:=E]$ where $y \neq x$ and $y \notin FV(E)$

Note that we can always alter the variable bound by a existential or universal quantifier such that the replacement can occur.

We can read the axiom as "Whatever is true of x after the assignment must also have been true of E before the assignment." For example,

$\{ odd(a+b) \}\ x := a+b;\ \{ odd(x) \}$

If x is odd after the assignment of a+b to x, then a+b must have been odd before.

Incorrect assignment axioms

Provide counter examples for the following rules

- $\{ P \}\ x := E\ \{ P \wedge x=E \}$
- $\{ P \}\ x := E\ \{ P[x:=E] \}$

It is possible to fix the first. How?

The backwards nature of the previous axiom can seem unnatural. However, it is quite a lot harder to give a forwards assignment axiom.

The first can be fixed by using existential quantification. We need to consider the old value of x as well as the new value of x. Exercise, try and do this.

Consequence rule

$P \Rightarrow P'$

$Q' \Rightarrow Q$

$$\frac{\{ P' \}\ C\ \{ Q' \}}{\{ P \}\ Q\ \{ Q \}}$$

In our proofs we will need to use logical inferences from first-order logic. This rule allows us to use logical inferences to manipulate the pre and post-condition of a command. For example if we have proved

{ P } C { x= 3*(2n+1) }

we can then derive the weaker property

{ P } C { odd(x) }

as we can prove

x= 3*(2n+1) ⇒ odd(x)

Similarly, if we have proved

{ x > 0 } C { Q }

then we can weaken this to

{ x=3 } C { Q }

because

x=3 ⇒ x>0

The rule of consequence allows us to strengthen the pre-condition, make it more specific, and weaken the post-condition, make it less specific.

Example proof (i)

By assignment axiom:

{ x=3 [x:=3] } x:= 3 { x=3 }

By rule of consequence

x = 5 ⇒ x=3 [x:=3]

x=3 ⇒ x=3

$$\frac{\{ x=3\ [x:=3] \}\ \ x:=3\ \{ x=3 \}}{\{ x=5 \}\ x:=3\ \{ x=3 \}}$$

Now let us give the precise proof of the first example triple

{ x=5 } x:=3 { x=3 }

This uses the rule of consequence at the outside with the implication

x=5 ⇒ x=3[x:=3] ⇔ 3=3 ⇔ true

Exercise: do the other example triples from the

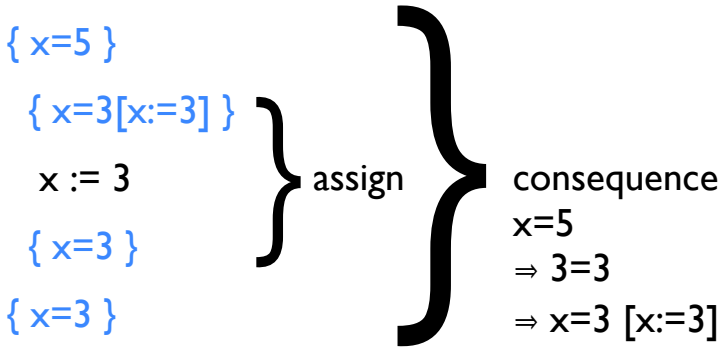
{ x=4 } x := x+1 { x=5 }

{ y=3 ∧ x=3 } y := x+y { y=6 ∧ x=3 }

{ odd(x) } y := x + x + x { odd(y) ∧ odd(x) }

Give full details of the rules used.

Example proof (i)'



It is common to present proofs as outlines. This is where intermediate assertions used in rules are given. This stops the need to duplicate the program many times. We will typically use this presentation in the course.

Exercise: Redo the proofs from the previous slide in this presentation.

Logical variables

Some specifications are not strong enough for clients to use

{ odd(x) } y := x + x { even(y) ∧ odd(x) }

Clients may want to know x is unchanged.

{ odd(x) ∧ x=X } y := x+x { even(y) ∧ odd(x) ∧x=X}

Here X is used to represent the value of the variable x. This allows the specification to say that the value of x is unchanged by this procedure.

We can see this is intuitively meaning

∀X. { odd(x) ∧ x=X } y := x+x { even(y) ∧ odd(x) ∧x=X}

That is, a proof that is independent of the actual value of X.

Logical variable elimination rule

$$\frac{\{ P \}\ C\ \{ Q \}}{\{ \exists X.\ P \}\ C\ \{ \exists X.\ Q \}}$$

Invariance rule

$$\frac{\{ P \}\ C\ \{ Q \}}{\{ P \wedge R \}\ C\ \{ Q \wedge R \}}$$

Provided free variables of R are not in mod(C).

We introduce a second set of variable names, X, Y, Z for logical variables (typically capital letters). We assume this set is distinct from the program variables, x,y,z . We extend the state to interpret logical variables in the same was as program variables.

Caveat: Other approaches will conflate logical and program variables into a single set and then provide side-conditions on what can be considered logical at a particular point in the program. We separate the two uses to simplify the presentation.

Exercise: We can also define a rule

$$\frac{\{ P \}\ C\ \{ Q \}}{\{ \exists X.\ P \}\ C\ \{ Q \}}$$

Provided X not mentioned Q.

Prove this is equi-expressive as the other rule. That is a proof with one rule can be encoded as a proof with the other rule.

Hint: You need to use the rule of consequence and properties of existential quantifiers.

Here we use a modifies set which specifies an over-approximation of the variables a program modifies. However, there are still cases where logical variables are required even with modified sets. We define the modified variables of a command as

mod(x := E) = {x}
mod(if B then C_1 else C_2) = mod(C_1) ∪ mod(C_2)
mod(C_1; C_2) = mod(C_1) ∪ mod(C_2)
mod(skip) = {}

We can preserve any property about variables not modified in the command.

There are still cases where we require the logical variables to capture the old values of a command. Consider specifying increment:
{ x=X} x:=x+1 {x=X+1}

Example proof (ii)

{ odd(x) ∧ x = 5 }
{ ∃X. odd(x) ∧ x = X ∧ X=5}
{ odd(x) ∧ x=X ∧ X=5}
{ odd(x) ∧ x=X }
y := x+x
{ even(y) ∧ odd(x) ∧x=X}
{ even(y) ∧ odd(x) ∧ x=X ∧ X=5}
{ ∃X. even(y) ∧ odd(x) ∧ x=X ∧ X=5}
{ even(y) ∧ odd(x) ∧ x=5}

In this example, we want to instantiate the logical variable X, and use the invariance rule to preserve the fact that X=5 across the assignment.

Formal Semantics

Programming language:

$C, \sigma \rightarrow C', \sigma'$

Logic semantics:

$\sigma \models P$

Triple semantics, {P}C{Q},

$\forall \sigma.\ \sigma \models P \ \Rightarrow\ \forall \sigma'.\ C,\sigma \rightarrow^* \text{skip},\sigma' \ \Rightarrow\ \sigma' \models Q$

We define the semantics of programs using evaluation contexts:
ε ::= * | ε ; C
We evaluate the outermost expression, or to the left of an assignment.
We define ε[C] as
*[C] = C
and
ε ; C' [C] = ε[C] ; C'

Programming language
ε[x:=E], σ → ε[skip], σ[x:=v] where ⟦ E ⟧σ = v (E evaluates to v in σ)
ε[skip;C],σ → ε[C],σ
ε[if B then C_1 else C_2],σ → ε[C_1],σ where ⟦ B ⟧σ = true (B evaluates to true in σ)
ε[if B then C_1 else C_2],σ → ε[C_2],σ where ⟦ B ⟧σ = false
We define →* as the transitive and reflexive closure of →.

Exercises

Write a rule for a one armed conditional.
[Hint: consider: if B then C else skip]

Prove

{x=N}
if x=5 then y:=1
{ (N=5 ⇒ y=1) ∧ x=N }

Prove the soundness of the skip rule.

Prove the soundness of the assignment axiom.

For the assignment axiom proof you may assume:
Lemma: $\sigma \models P[x:=E] \Leftrightarrow \sigma[x:=v] \models P$ where E evaluates to v in state σ.

Exercise, prove this lemma.

Aside: Two state post-conditions

Specification:

{ P } C { R }

Semantics

If executing C in an initial state satisfying P terminates, then the initial and final state are related by R.

An alternative presentation is to use the post-condition as a relation between the start and end state.

2. Control flow

For a different exposition see
R. D. Tennent, Semantics of Programming Lanuages.
Chapter 7 covers Hoare Doubles.

Programming language

Commands:

C ::= … | while B do C
| break | continue | return E | …

In this lecture we will add more interesting control flow to our language.

The semantics of this language is a little harder to define. We use a configuration of a command, C, and a stack of active loops and their continuations (what to do after the loop), and a state, σ.
C, Ws, σ

We define the semantics as

ε[while B do C], Ws, σ
→ if B then C;continue else break, ε[while B do C]::Ws, σ
ε[break], ε'[while B do C]::Ws, σ → ε'[skip], Ws, σ
ε[continue], C::Ws, σ → C, Ws, σ
ε[return E], Ws, σ → return E, [], σ

The rest of the commands are the same as before just preserving the Ws component of the configuration.

We assume the program starts with an empty active loop stack, Ws=[].

While rule

$$\frac{P \wedge \neg B \Rightarrow Q \qquad \{ B \wedge P \}\ C\ \{ P \}}{\{ P \}\ \text{while } B \text{ do } C\ \{ Q \}}$$

Example: Euclid

```
{ a>0 ∧ b>0 ∧ a=A ∧ b=B}
while a != b do
    if a > b then a := a % b;
    if a < b then b := b % a
{ a = gcd(A,B) }
```


The rule for a while loop is quite different to the rules we have seen so far. The pre and post-conditions are very similar, as they are both based on P. Initially it may look like the loop cannot do much as we are constrained to specify the body preserves P. This is the key to reasoning about a loop finding what remains true on each iteration: the loop invariant.

Note that, we could have equally made the post-condition simply $P \wedge \neg B$. We introduced the Q as it simplifies presentation later.

Earlier we discussed total correctness. This rule does not deal with termination of the loop. The invariant only deals with the property the loop tried to establish. For total correctness, we also require a metric that decreases.

$$\frac{B \wedge P \Rightarrow E > 0 \qquad P \wedge \neg B \Rightarrow Q \qquad \{ B \wedge P \wedge E = X \}\ C\ \{ P \wedge E < X \}}{\{ P \}\ \text{while } B \text{ do } C\ \{ Q \}}$$

Here E is a rank that must decrease on every iteration.

Here a % b is the modulus. That is the remainder when dividing a by b.

We use the following facts about gcd:

$\gcd(a,b) = \gcd(b,a)$

$\gcd(a,a) = a$

$a>b \Rightarrow \gcd(a,b) = \gcd(a\%b, b)$

The loop invariant for this example is

$\{ a > 0 \wedge b>0 \wedge \gcd(a,b) = \gcd(A,B) \}$

[This example is taken from Mike Gordon's notes exercise 36.]

Second Non-example

{ z = Z ∧ y = Y}
x := 1;
while z > 0 do
    if z & 1 = 1 then x := y*x;
    z := z >> 2;
    y := y * y;
{ x = Y^Z }

Control structures

Often we write loops with more interesting contol flow, such as

- Break (abruptly terminates the loop)
- Continue (jumps back to the start of the loop)
- Return (completes execution)

How can we deal with this in Hoare logic

Proposed loop invariant: Y^Z = x * y^z

We can give a proof outline of the body of the loop as:

```
  { Y^Z = x * y^z}
  if z & 1 = 1 then
    { Y^Z = x * y^z ∧ ∃n. 2n+1 = z }
     x := y*x;
    { ∃n. Y^Z = x * y^(2n) ∧ ∃n. 2n+1 = z }
  else
    { Y^Z = x * y^z ∧ ∃n. 2n = z }
     skip
    { ∃n. Y^Z = x * y^(2n) ∧ 2n = z }
  { ∃n. Y^Z = x * y^(2n) ∧ (2n = z ∨ 2n+1=z) }
  { ∃n. Y^Z = x * y^(2n) ∧ (n = z >> 2 ∨ n=z >> 2) }
  z := z >> 2;
  { ∃n. Y^Z = x * y^(2z) }
  y := y * y;
 { ∃n. Y^Z = x * y^z }
```

Exercise: Prove the prelude establishes the loop invariant.
Exercise: Show that the loop invariant is not strong enough to establish the post-condition.
Exercise: Find a new loop invariant, and pre-condition for the code, to establish the post-condition.

Context

We extend triples to contain a context of specifications for the different continuations:

$\Gamma \vdash \{ P \}\ C\ \{ Q \}$

where

$\Gamma ::= \{ P \}\text{break} \mid \{ P \}\text{continue} \mid \{ P \}\text{return} \mid \Gamma,\Gamma \mid \ldots$

We will write Γ for a list of these pre-condition and "label" pairs. We will implicitly assume that the left of the ⊢ can be treated as a function from label to pre-condition, that is, we do not define a label twice. We allow weakening of this context:

$$\frac{\Gamma \vdash \{P\}\ C\ \{Q\}}{\Gamma', \Gamma \vdash \{P\}\ C\ \{Q\}}$$

Break/Continue

$\{ Q \}\ \text{break} \vdash \{ Q \}\ \text{break}\ \{ \text{false} \}$

$\{ P \}\ \text{continue} \vdash \{ P \}\ \text{continue}\ \{ \text{false} \}$

$\{ R \}\ \text{return} \vdash \{ R\ [\text{return}:=E] \}\ \text{return}\ E\ \{ \text{false} \}$

In the return rule, there is a special variable "return" for returning the value given to return. We assume return is a variable not used in the program except through the return command (and later in function calls).

Why is the post-condition false?

While

$P \wedge \neg B \Rightarrow Q$

$\underline{\Gamma, \{P\}\text{continue}, \{Q\}\text{break} \vdash \{ B \wedge P \}\ C\ \{ P \}.}$

$\Gamma \vdash \{ P \}\ \text{while } B \text{ do } C\ \{ Q \}$

Here Γ is used to carry any labels not overwritten by the while loop, such as return labels (and later function specifications).

Implicitly we assume that Γ does not contain break or continue.

Example: isPrime

{return = prime(x)}return ⊢

```
{ x > 1 }
i := 1;
while true do
   i := i + 1;
   if i = x then break;
   if x mod i = 0 then return false;
return true
{false}
```


Here the loop invariant is $\forall j.\ 2 \le j \le i \Rightarrow x \bmod j \neq 0 \ \wedge\ 1 \le i < x$

The break assertion is prime(x), where we define prime as $\forall j.\ 2 \le j < x \Rightarrow x \bmod j \neq 0$

We verify the body of the loop as

{r. r = prime(x)}return, {true=prime(x)} break ⊢

```
  { ∀j. 2 ≤ j ≤ i ⇒ x mod j ≠ 0 ∧ i < x}
  i := i + 1;
  { ∀j. 2 ≤ j ≤ i-1 ⇒ x mod j ≠ 0 ∧ 1 ≤ i-1 < x}
  if i = x then
     { ∀j. 2 ≤ j ≤ i-1 ⇒ x mod j ≠ 0 ∧ 1 ≤ i-1 < x ∧ i=x}
     { ∀j. 2 ≤ j ≤ x-1 ⇒ x mod j ≠ 0 ∧ 1 ≤ x-1 < x }
     { ∀j. 2 ≤ j < x ⇒ x mod j ≠ 0 }
     { true=prime(x) }
     break;
     {false}
  else
     { ∀j. 2 ≤ j ≤ i-1 ⇒ x mod j ≠ 0 ∧ 1 ≤ i-1 < x ∧ i ≠ x}
     { ∀j. 2 ≤ j ≤ i-1 ⇒ x mod j ≠ 0 ∧ 2 ≤ i < x }
     skip
     { ∀j. 2 ≤ j ≤ i-1 ⇒ x mod j ≠ 0 ∧ 2 ≤ i < x}
  if x mod i = 0 then
     { ∀j. 2 ≤ j ≤ i-1 ⇒ x mod j ≠ 0 ∧ x mod i = 0 ∧ 2 ≤ i < x}
     { ∃j. 2 ≤ j < x ∧ x mod j = 0}
     { false=prime(x) }
     return false
  { ∀j. 2 ≤ j ≤ i ⇒ x mod j ≠ 0 ∧ 2 ≤ i < x}
  { ∀j. 2 ≤ j ≤ i ⇒ x mod j ≠ 0 ∧ 1 ≤ i < x}
```

Lemma : $i-1 < x \wedge i \neq x \Rightarrow i < x$

Exercise: Try to prove the program without the $1 \le i < x$ conjunct in the loop invariant.

Prime Palindromes

Are there any prime palindromes between 20 and 100?

```
i := 19
while true do
  i := i + 1;
  if i > 100 then break
  if ¬prime(i) then continue
  if palindrome(i) then break
```


The loop invariant, P, for this example is

$\forall\, j.\quad 20 \le j \le i \Rightarrow \neg prime(j) \vee \neg palindrome(j)$

The break assertion, Q, is

$i > 100 \Rightarrow \forall\, j.\quad 10 \le j \le 100 \Rightarrow \neg prime(j) \vee \neg palindrome(j)$

$\wedge\quad 20 \le i \le 100 \Rightarrow prime(i) \wedge palindrome(i)$

Exercise: Prove this meets is specification.

Note a palindrome is a number or word that is the same when read forwards or backwards. Here again we see abstraction, we do not need to know what prime and palindrome mean to verify this program.

Exercises

Give a rule for reasoning about

do C while B

This command repeatedly executes C until B no longer holds.

Extend the break and continue to deal with nested loops, where you supply a level to specify which loop you break/continue.

Hints.

For the first question consider the program

while true do (C; if B then break)

or

C; while ¬B do C

For the second question, consider adding more structure to Γ to account for the nesting.

Semantics

We define

$\{P\}\text{break}, \{Q\}\text{continue} \models \{R\}\ C\ \{S\}$

as

$\sigma \models R \Rightarrow$

- $C, [], \sigma \rightarrow^* \text{skip}, [], \sigma' \Rightarrow \sigma' \models S$
- $C, [], \sigma \rightarrow^* \varepsilon[\text{break}], [], \sigma' \Rightarrow \sigma' \models P$
- $C, [], \sigma \rightarrow^* \varepsilon[\text{continue}], [], \sigma' \Rightarrow \sigma' \models Q$

Exercise: Prove while rule is sound. You may assume the following lemma.

Lemma: $C,[],\sigma \rightarrow^* C',[],\sigma' \Rightarrow C,Ws,\sigma \rightarrow^* C',Ws,\sigma'$

Exercuse: Extend the semantics to deal with return as well.

3. Functions and local variables

Programming language

C ::= … | local x in C
| let $f(x_1,\ldots,x_n)$ = C in C } Call-by-value
| $f(E_1,\ldots,E_n)$ | …
| x := $f(E_1,\ldots,E_n)$
| let $f(z_1,\ldots,z_m;x_1,\ldots,x_n)$ = C in C } Call-by-reference;value
| $f(z_1,\ldots,z_m;E_1,\ldots,E_n)$ | …

Context

We extend the context of specifications with function specifications:

$\Gamma \vdash \{ P \}\ C\ \{ Q \}$

where

Γ ::= … | $\{ P \}\ f(x_1,\ldots,x_n)\ \{ Q \}$
| $\{ P \}\ f(z_1,\ldots,z_m;x_1,\ldots,x_n)\ \{ Q \}$ | …

In this section, we introduce functions and local variables. We begin by presenting functions with just call-by-value parameters. We then extend this to deal with functions that return values, and finally we consider functions with call-by-reference parameters as well as value parameters. We place the reference parameters before the value, and separate them with a semi-colon.

We make the implicit assumptions that functions only access their parameters, that is, for

let $f(x_1,\ldots,x_n) = C_f$ in C

to be well-formed requires the free variables of C_f to be contained in $x_1,\ldots,x_n$. And similarly for function with reference parameters as well. This prevents us dealing with global variables. Note that these can be encoded with reference parameters.

We can only pass variables for reference arguments, and each parameter must be syntactically distinct.

$f(z_1,\ldots,z_m; E_1,\ldots,E_n)$

that is if z_i is the same variable as z_j then i=j. This prevents us having to deal with variable aliasing.

Note that we treat the parameters, e.g. $x_1,\ldots x_n$, as binders for P and Q, that is

$\{ P \}\ f(x)\ \{ Q \} = \{ P[x:=y] \}\ f(y)\ \{ Q[x:=y] \}$

where y is not free in P or Q.

Local variables

$$\frac{\Gamma \vdash \{ P \}\ C\ \{ Q \}}{\Gamma \vdash \{ P \}\ \text{local } x \text{ in } C\ \{ Q \}}$$

Provided x not free in P and Q

Note that the modified set is updated by this command:
mod(local x in C) = mod(C) \ {x}
We require this to create local state for a function to use and modify.

Function call

(call-by-value)

$$\Gamma, \{ P \}\ f(x_1,\ldots,x_n)\ \{ Q \} \vdash \{ P\ [x_1:=E_1,\ldots,x_n:=E_n] \}\ f(E_1,\ldots,E_n)\ \{ Q\ [x_1:=E_1,\ldots,x_n:=E_n] \}$$

This rule looks up the specification of the function in the context. The specification is given with respect to parameter list

The key thing about this rule is that substitution is used for arguments to the function

All the parameters are dealt with in call-by-value, that is, we copy the values. We don't pass references to them.

As this function doesn't return anything, and we don't allow global variables it is currently useless. Later, when we deal with the heap in separation logic this rule will be useful. At the moment we simply present it as part of the development.

Function call

(call-by-value)

$$\Gamma, \{\ P\ \}\ f(X_1,\ldots,X_n)\ \{\ Q\ \} \vdash$$
$$\{\ P \wedge X_1 = E_1 \wedge \ldots \wedge X_n = E_n\ \}$$
$$f(E_1,\ldots,E_n)$$
$$\{\ Q\ \}$$

Function definition

$$\Gamma \vdash \{P_f\}\ C_f\ \{Q_f\}$$
$$\underline{\Gamma, \{P_f\} f(x_1,\ldots,x_n) \{Q_f\} \vdash \{\ P\ \}\ C\ \{\ Q\ \}\ .}$$
$$\Gamma \vdash \{\ P\ \}\ \text{let } f(x_1,\ldots,x_n) = C_f \text{ in } C\ \{Q\}$$

Provided $x_1,\ldots,x_n$ not in $\text{mod}(C_f)$

We can rewrite the previous axiom to use logical variables rather than substitution. We are making use of the binding nature of the variables in the specification. We are simply renaming the parameters in the specification to logical variables, which we can then use in the specification.

This rule may seem more complex, but as we add the features such as return and reference parameters this presentation will become simpler.

To verify the function definition, we must provide a specification for the function. We verify the body meets this specification, then in the client code, C, we can assume the specification of the function.

We have restricted such that we cannot modify our parameters, this is not essential but simplifies the presentation. If we remove the restriction then the rule is unsound. Consider

```
let f(x) = x:=1 in local y in f(y); assert y=1
```

We could verify this

(1) { true } x:=1 { x=1 }

(2) { true } f(x) { x=1 } ⊢ { true } local y in f(y); assert y=1 { true }

The first follows trivially from the rules, the second

```
{ true }
  f(y)
{ y =1 }
  assert y=1
{ y=1 }
{ true }
```

The problem is that the interpretation of y in Q_f is with respect to the modified value of the parameters in the first (1), but in the second should be with respect to the original value of parameters. This tie needs breaking. Preventing modification does this trivially as the old and new values are the same. By introducing new local variables this tie can also be broken.

Exercise: Give a rule for function definition that deals with modifying parameters. [Hint: Consider creating new local variables for each parameter.]

Recursive Function definition

$\Gamma, \{P_f\}f(x_1,\ldots,x_n)\{Q_f\} \vdash \{P_f\}\ C_f\ \{Q_f\}$

$$\frac{\Gamma, \{P_f\}f(x_1,\ldots,x_n)\{Q_f\} \vdash \{\ P\ \}\ C\ \{\ Q\ \}}{\Gamma \vdash \{\ P\ \}\ \text{let } f(x_1,\ldots,x_n) = C_f \text{ in } C\ \{Q\}}$$

Provided $x_1,\ldots,x_n$ not in $\text{mod}(C_f)$

We can extend the function definition rule to deal with recursively defined functions. We simply add the assumption that the function meets its specification to the verification of the body.

Question: Does this work if we consider total correctness?

Returning values

We allow our specifications to use the variable return in the post-condition.

$\Gamma ::= \ldots \mid \{P\}\ f(xs)\ \{Q\}$

A specification is well-formed

iff $FPV(P) \subseteq xs$ and $FPV(Q) \subseteq xs \cup \{return\}$.

We define the modified variables of a function call as

$\text{mod}(x := f(Es)) = \{x\}$

Function call with return

$\Gamma, \{ P \}\ f(x_1,\ldots,x_n)\ \{ Q \} \vdash$
$\{ P\ [x_1:=E_1,\ldots,x_n:=E_n] \}$
$y := f(E_1,\ldots,E_n)$
$\{ (\exists y.\ Q\ [x_1:=E_1,\ldots,x_n:=E_n])\ [return := y] \}$

Too weak for some uses!

This rule uses substitution for the parameters into the specification.

However it is not strong enough, for example, consider
{ true } sum(x,y) { return = x+y }

Now if we use this in
{x = 3}
x = sum(x,5)
{x = 8}

However, let's try and do this proof. Using the function call rule we get
{ true }
x = sum(x,5)
{ (∃ x. return = x + 5) [return := x] }
{ ∃ z. x = z + 5 }

we cannot use this specification to prove our original goal.

Function call with return

$\Gamma, \{ P \}\ f(x_1,\ldots,x_n)\ \{ Q \} \vdash$
$\{ P\ [x_1:=E_1,\ldots,x_n:=E_n] \wedge y=Y \}$
$y := f(E_1,\ldots,E_n)$
$\{ (\exists y.\ Q\ [x_1:=E_1,\ldots,x_n:=E_n] \wedge y=Y)\ [return := y] \}$

The weakness of the previous rule can be addressed by preserving the old value of y with the logical variable Y. This specification can now prove the goal we want.

Again assume the specification:
{ true } sum(x,y) { return = x+y }
Now using the rule we get

{ true ∧ x = X}
x := sum(x,5)
{ (∃ x. return = x + 5 ∧ x=X) [return := x] }
{ x = X + 5 }

This is now strong enough to prove our goal.

{ x = 3 }
{ ∃ X. true ∧ x=X ∧ X=3 }
{ true ∧ x=X ∧ X=3 }
{ true ∧ x=X }
x:=sum(x,5)
{ x = X+5 }
{ x=X+5 ∧ X=3 }
{∃X. x=X+5 ∧ X=3 }
{ x=8 }

Note that the function call rule is equivalent to
$\{ P\ [x_1:=E_1,\ldots,x_n:=E_n] \wedge y=Y \}$
$y := f(E_1,\ldots,E_n)$
$\{ Q\ [x_1:=E_1,\ldots,x_n:=E_n][y:=Y][return := y] \}$

Alternative Function call

$\Gamma, \{ P \}\ f(X_1,\ldots,X_n)\ \{ Q \} \vdash$
$\{ P \wedge X_1 = E_1 \wedge \ldots \wedge X_n := E_n \}$
$y := f(E_1,\ldots,E_n)$
$\{ Q\ [return := y] \}$

Function definition

$\Gamma, \{Q_f\}return \vdash \{P_f\}\ C_f\ \{ false \}$
$\Gamma, \{P_f\}f(x_1,\ldots,x_n)\{Q_f\} \vdash \{ P \}\ C\ \{ Q \}$.
$\Gamma \vdash \{ P \}$ let $f(x_1,\ldots,x_n) = C_f$ in $C\ \{Q\}$

Provided $x_1,\ldots,x_n$ not in $mod(C_f)$

In the same way as we did before, we can simply use logical variables rather than substitution for the function call rule. This has the advantage that we do not need to have as many substitutions

Consider our previous example
{ true } sum(x,y) { return = x+y }
Now by renaming the bound parameters we get
{ true } sum(X,Y) { return = X+Y }

So for the call we have
{ true ∧ X=x ∧ Y=5 }
x := sum(x,5)
{ (return = X + Y)[return := x] }
{ x = X+Y }

Now we can use this as
{ x = 3 }
{ ∃X,Y. true ∧ X=x ∧ Y=5 ∧ Y=5 ∧ X=3 }
{ true ∧ X=x ∧ Y=5 ∧ Y=5 ∧X=3 }
{ true ∧ X=x ∧ Y=5 }
x := sum(x,5);
{ x = X+Y }
{ x = X + Y ∧ X=3 ∧ Y= 5}
{ ∃X,Y. x = X + Y ∧ X=3 ∧ Y= 5}
{ x = 8 }

Now rather than specifying the post-condition, we add that the specification of return must be the post-condition of the function. We make the post-condition for verification false, so that we must call return to exit the function.

We can now consider verifying the implementation of the sum function

let sum(x,y) = return x+y in
local x;
x := sum(x,5);
assert x = 8

This requires us to prove
{ return = x+y } return ⊢ { true } return x+y { false}
which follows from the rule of consequence, and the return rule.

{ return = x+y } return ⊢ { x+y=x+y } return x+y { false }
{ return = x+y } return ⊢ { true } return x+y { false }

Example - Fibonacci

{ x≥0 }
let f (x) =
    local y,z;
    if x=1 then return 1;
    if x=0 then return 0;
    y := f(x-1);
    z := f(x-2);
    return y+z;
{ return = fib(x) }

Assume the function symbol fib satisfies the following equalities
    fib(0) = 0
    fib(1) = 1
    fib(n+2) = fib(n) + fib(n+1)

We can prove the program meets its specification. We omit the first part of the proof.
{ x ≥ 2 }
{ fib(x) = fib(x-1) + fib(x-2) ∧ x≥2}
{ ∃X. fib(x) = fib(x-1) + fib(x-2) ∧ x≥2 ∧ X ≥ 0 ∧ X=x-1 ∧ X=x-1}
  { fib(x) = fib(x-1) + fib(x-2) ∧ x≥2 ∧ X ≥ 0 ∧ X=x-1 ∧ X=x-1}
    { X ≥ 0 ∧ X=x-1 }
    y := f(x-1);
    { y = fib(X) }
  { fib(x) = fib(x-1) + fib(x-2) ∧ x≥2 ∧ y=fib(X) ∧ X=x-1}
{ ∃X. fib(x) = fib(x-1) + fib(x-2) ∧ x≥2 ∧ y=fib(X) ∧ X=x-1}
{ fib(x) = y + fib(x-2) ∧ x-2≥0 }

The final part of the proof proceeds in a similar way, we present only a skeletal outline:
{ fib(x) = y + fib(x-2) ∧ x-2≥0 }
    z := f(x-2)
{ fib(x) = y + z }
   return y+z
{false}

Call by reference

Some languages allow parameters to be passed by reference.

C ::= … | let f(z_1,…,z_m;x_1,…,x_n) = C in C
        | f(z_1,…,z_m;E_1,…,E_n) | …

We now assume two lists of parameters to a function. The first are the call-by-reference parameters, and the second the call-by-value parameters.

At call sites we insist that z_1,…,z_m are distinct program variables. If we did not, then we would have to deal with aliasing of variables.

Here are some examples to help understand the difference.
  let f(x;) = x := x+1 in local y in y := 0; f(y;); assert(y=1)
and
  let f(;x) = x := x+1 in local y in y := 0; f(;y); assert(y=0)
both of these assertion will succeed. A variable passed by reference will be updated if the function modifies it, where as a value parameter will not update the original variable.

Note that we cannot pass expressions by reference, only variables.

We assume the call-by-reference parameters are all modified
  mod(f(z_1,…,z_m;E_1,…,E_n)) = {z_1,…,z_m}

Note, we will not use call-by-reference much. In the next lecture when we cover arrays, they are typically dealt with in call-by-reference rather than value. So this enables us to model them correctly before we introduce pointers.

Function call

$$\Gamma, \{ P \}\ f(z_1,\ldots,z_m; X_1,\ldots,X_n)\ \{ Q \} \vdash$$
$$\{ P \wedge E_1,\ldots,E_n = X_1,\ldots,X_n \}$$
$$f(z_1,\ldots z_m; E_1,\ldots,E_n)$$
$$\{ Q \}$$

Here we see the axiom for dealing with function call with both reference and value parameters.

{ true } sum(z;x,y) { z = x+y }

{ true $\wedge$ x=X } sum(x;y) { x = X+y }

Here we define the first as
sum(z;x,y) = z := x+y
and the second as
sum(x;y) = x := x+y

Prove both bodies meet their specification.

Exercise: derive a rule that uses substitution rather than logical variables. Prove your rule can derive the original rule, or exhibit a counter example.

Function definition

$$\Gamma \vdash \{P_f\}\ C_f\ \{Q_f\}$$
$$\underline{\Gamma, \{P_f\} f(z_1,\ldots,z_n; x_1,\ldots,x_n) \{Q_f\} \vdash \{ P \}\ C\ \{ Q \}}$$
$$\Gamma \vdash \{ P \}\ \text{let } f(x_1,\ldots,x_n) = C_f \text{ in } C\ \{Q\}$$

Provided $x_1,\ldots,x_n$ not in mod(C_f)

The function definition with reference parameters is almost identical to the previous rule.

Exercise

If the call-by-value parameters are distinct from the values assigned to, then the rule can be simplified. Give this simplification, and use it to perform the Fibonacci verification.

Write and verify two recursive versions of Factorial. Use both a function that returns a value, and a function that uses a reference parameter.

4. Arrays

Programming language

E ::= … | a[E]

C ::= … | a[E] := E

In function calls, we will only pass arrays by reference as in Java. Call-by-value arrays are strange and do not correspond naturally to normal programming languages. Note, this is the main reason for introducing call-by-reference in the previous section. If we used the call-by-value rule on an array in this logic it would simply be like copying the whole array.

Theory

Syntax (array expression):

$A ::= a \mid A\{E_1 \leftarrow E_2\}$

Axioms:

(ax1) $A\{E_1 \leftarrow E_2\}[E_1] = E_2$

(ax2) $E_1 \neq E_2 \Rightarrow A\{E_1 \leftarrow E_2\}[E_3] = A[E_3]$

Array expressions are either, variables of type array, a; or updates of array expressions, $A\{E_1 \leftarrow E_2\}$ where the value at index E_1 is replaced with E_2.

We give two axioms for dealing with array expressions.

Axiom

$\{P\ [a := a\{E_1 \leftarrow E_2\}]\ \}\ a[E_1] := E_2\ \{\ P\ \}$

Example

Prove

{true}
a[3] := 5;
a[4] := 4;
x = a[3] + a[4]
{x = 9}

{true}
{ a{3 ← 5}[3] = 5 } (by ax1)
  a[3] := 5;
{ a[3] = 5 }
{ a{4 ← 4}[3] = 5 } (by ax2)
{ a{4 ← 4}[3] = 5 ∧ a{4 ← 4}[4] = 4} (by ax1)
  a[4] := 4;
{a[3] = 5 ∧ a[4] = 4}
  x = a[3] + a[4]
{x = a[3] + a[4] ∧ a[3] = 5 ∧ a[4] = 4}
{x = 9}

Example: Swap

$\{ i \neq j \wedge a[i] = X \wedge a[j] = Y \}$
local t in
t := a[i];
a[i] := a[j];
a[j] := t
$\{ i \neq j \wedge a[i] = Y \wedge a[j] = X \}$

This code modifies the array a. But only some of the elements. If we want to use this method, then we would have to specify that the rest of the array is unchanged.

Example: Swap

$\{ i \neq j \wedge a[i] = X \wedge a[j] = Y \wedge a = A \}$
local t in
t := a[i];
a[i] := a[j];
a[j] := t
$\{ i \neq j \wedge a[i] = Y \wedge a[j] = X$
$\quad \wedge \forall k.\ k \neq i \wedge k \neq j \Rightarrow a[k] = A[k] \}$

This function only modifies the array a, but we must explicitly say which entries it does not modify. Here we use a logical array variable A. We use this to save the old values of the array, so that we can describe that the values other than i and j are unmodified by the body.
Array equality is defined as

$a=A \Leftrightarrow \ \forall k.\ a[k]=A[k]$

Example: Swap

{ i ≠ j ∧ a = A }
local t in
t := a[i];
a[i] := a[j];
a[j] := t
{ A{i←A[j]}{j←A[i]} = a }

A simpler specification is perhaps to give in terms of the operations on the array.

Exercise: Consider using XOR to swap the elements of the array without additional storage:
  a[i] := a[i] XOR a[j];
  a[j] := a[j] XOR a[i];
  a[i] := a[j] XOR a[i]
Verify this body also meets the same specification. Hint: XOR is commutative, associative, and satisfies
  (x XOR a) XOR a = x
for all x and a.

Example - Insertion Sort

i := 1;
while(i<n)
  j := 0;
  while(j<i)
    if a[i] < a[j] then swap(a;i,j);
    j := j + 1;
  i := i + 1;
{ sorted(a,n) }

The loop invariant for the outer loop is
  sorted(a,i) ∧ i ≤ n

The loop invariant for the inner loop is
  sorted(a,j-1)
  ∧ ∀ x. x < j ⇒ a[x] ≤ a[i]
  ∧ j ≤ i < n
where
  sorted(a,n) = ∀ i,j. 0 ≤ j < i < n ⇒ a[j] ≤ a[i]

Let us verify the call to swap loop:

{ sorted(a,j-1) ∧ ∀ x. x < j ⇒ a[x] ≤ a[i] ∧ j < i < n ∧ a[i] < a[j]}
{ ∃A. A=a ∧ sorted(A,j-1) ∧ (∀ x. x < j ⇒ A[x] ≤ A[i]) ∧ j < i < n ∧ A[i] < A[j] ∧ i≠j}
  { A=a ∧ i≠j}
  swap(a;i,j);
  { A{i←A[j]}{j←A[i]}=a }
{ ∃A. A {i←A[j]}{j←A[i]}=a ∧ sorted(A,j-1) ∧ (∀ x. x < j ⇒ A[x] ≤ A[i]) ∧ j < i < n ∧ A[i] < A[j]}

Now we can prove sorted(A,j-1) ⇒ sorted(a,j-1) as the updates to i and j are out of the range considered by sorted.

Similarly, we can show (∀ x. x < j ⇒ A[x] ≤ A[i]) ⇒ (∀ x. x < j ⇒ a[x] ≤ a[i])

And finally, we can show A[i] < A[j] ⇒ a[i]>a[j], so we can get

{ sorted(a,j-1) ∧ (∀ x. x < j ⇒ a[x] ≤ a[i]) ∧ j < i < n ∧ a[i] > a[j]}
{ sorted(a,j-1) ∧ (∀ x. x ≤ j ⇒ a[x] ≤ a[i]) ∧ j < i < n}

The rest of the proof is fairly straightforward.

Bubble sort

```
swapped := true;
while swapped do
  i := 0;
  while i < n-2 do
    if a[i] > a[i+1] then
      swap(a;i,i+1);
      swapped := true;
```


What are the loop invariants for the inner and outer loops?

Encoding heap and fields

In languages like C a common source of problems is pointers in the heap.

C ::= … | [E] := E | x := [E] | …

Here we use integer expressions inside square brackets access the heap location at that integer location.

Initially, we will ignore allocation of memory in the heap and assume all location exist and are accessible.

We will assume all function have an implicit heap parameter, e.g.
  let f(zs;xs) = C in C‘
is rewritten to
  let f(@heap,zs;xs) = C in C‘
and
  f(zs;Es)
to
  f(@heap,zs;Es)

Rules

$\{ P\ [@heap := \{E \leftarrow E'\}\ @heap] \}\ [E] := E'\ \{ P \}$

$\{ P\ [x := @heap[E]] \}\ x := [E]\ \{ P \}$

We use a special array variable @heap that is not used in the rest of the program to stand for the heap, and then use the array axioms to reason about accessing this memory.

Examples

{ true }

```
[10] := 11;
[11] := 10;
x := [10];
while x≠0 do
   x := [x];
```

{ false }

Exercise: Prove the program meets this specification.

Loop invariant

$(x = 10 \vee x = 11) \wedge @heap[10] = 11 \wedge @heap[11] = 10$

Heap and functions

All function will take @heap as a reference parameter.

How does framing work in this world?