Synthesis, Analysis, and Verification

Lecture 12

Verifying Programs that have Data Structures

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

What we have seen so far

- Programs that manipulate **integers**
- Verification-condition generation for them
- Proving such verification conditions using quantifier elimination

user gives invariants

more predictable

- Using abstract interpretation to infer invariants
- Predicate abstraction as abstract domain, and the idea of discovering new predicates

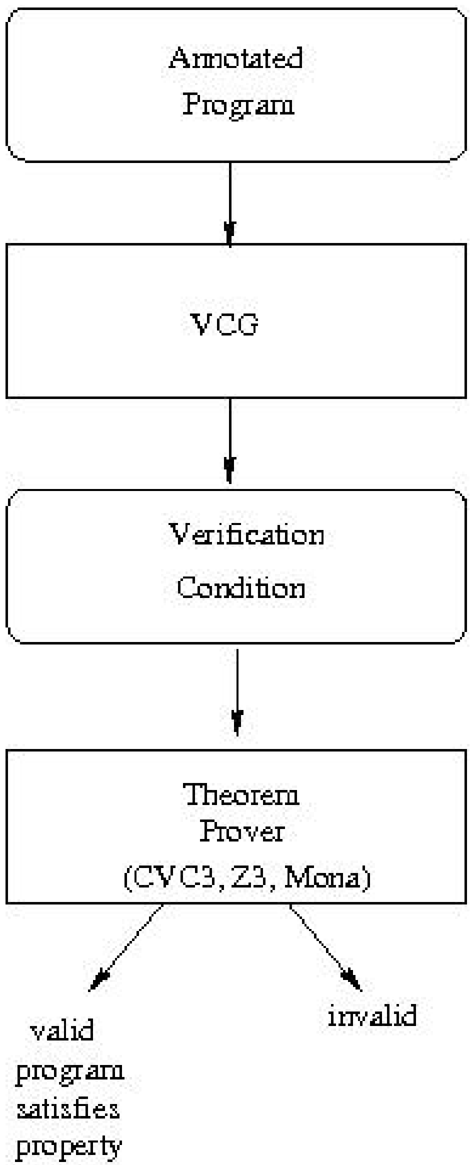
user gives only properties

more automated

QUESTION

What do we need to add to handle more general programs?

Verification-Condition Generation



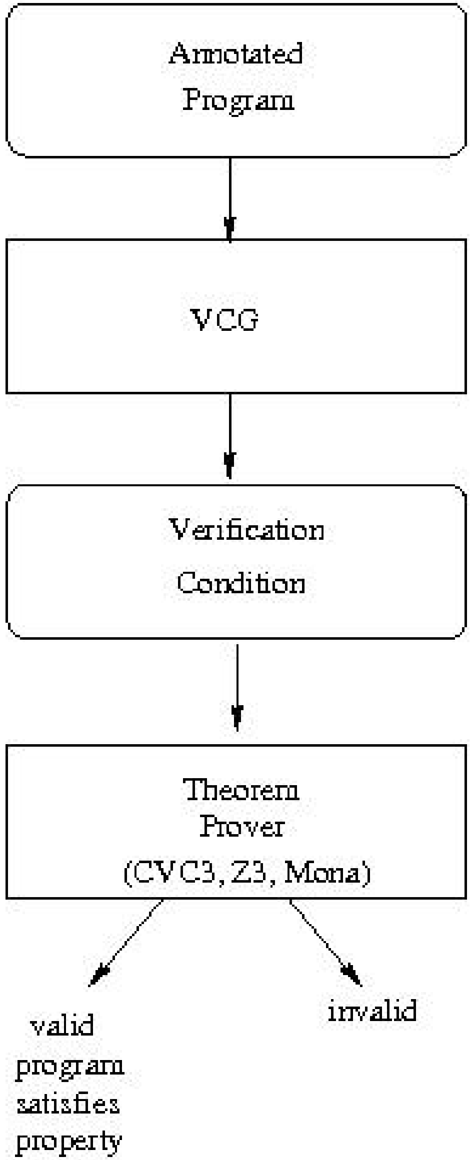
Steps in Verification

- generate a **formulas whose validity implies correctness** of the program
- attempt to prove all formulas
  - if formulas all valid, program is correct
  - if a formula has a counterexample, it indicates one of these:
    - error in the program
    - error in the property
    - error in auxiliary statements (e.g. loop invariants)

Terminology

- generated formulas:
  verification conditions
- generation process:
  verification-condition generation
- program that generates formulas:
  verification-condition generator (VCG)

VCG for Real Languages



Programs that Manipulate Integers,
Arrays, Maps, Linked Data Structures

Compute Formulas from Programs
have more operations in expressions for x:=E

Formulas with Integer Variables and Operations,
as well as variables and operations on functions

Prover for integer linear arithmetic
+ provers for function symbols,
mathematical arrays,
term algebras, ...

Formulas for Loop-Free Code

$x = t$: $x' = t \wedge y' = y$

$assume(F)$: $F \wedge x' = x \wedge y' = y$

$havoc(x)$: $y' = y$

FIND INTEGER OPERATIONS IN THIS PICTURE

F_1, F_2 : $F_1 \vee F_2$

F_1 ; F_2 :

$$\exists x_1, y_1.\ F_1[x' := x_1, y' := y_1] \wedge F_2[x := x_1, y := y_1]$$

x_1, y_1 - fresh

RULES REMAIN SAME

Some Immutable String Operations

Domain is the set of all strings over some a finite set of characters Char, and the empty string, ""

Operations include:

Concatenation: "abc" **++** "def" == "abcdef"
Head: **head**("abcd") == "a"
Tail: **tail**("abcd") == "bcd"

A Program with Immutable Strings

```
var first, second, given : String
var done : Boolean
first = ""
second = given
done = false
while (!done) {
  assume(second != "")
  if (head(second) =="/") {
     second = tail(second)
     done = true
  } else {
      first = first ++ head(second)
      second = tail(second)
   }
}
assert (first ++ "/" ++ second == given)
```

Find a loop invariant.

State verification conditions.

Prove verification conditions.

$(!done \wedge first ++ second = given) \vee$
$(done \wedge first ++ "/" ++ second = given)$

if (done) ... else ...

Some Verification Conditions

!done /\ first ++ second == given /\
  second != "" /\ head(second) != "/" /\
  first' = first + head(second) /\
  second' = tail(second) /\
  done' = done -->
!done' /\ first' ++ second' == given

first ++ head(second) ++ ~~second~~ tail(second) = given

done /\ first ++ second == given /\
  second != "" /\ head(second) == "/" /\
  second' = tail(second) /\
  first' = first /\
  done' = true -->
done' /\ first' ++ "/" ++ second' == given

Remark: Theory of Strings with ++

Given quantifier-free formula in theory of strings, check whether there are values for which formula is true (satisfiability).

NP-hard problem, not known to be in NP, only in PSPACE.

Wojciech Plandowski: Satisfiability of word equations with constants is in PSPACE. J. ACM 51(3): 483-496 (2004)

In the sequel

- We will
  - not look at strings so much
  - use more general notion, Map
  - avoid operations such as concatenation
- Theories of maps (array)
  - using them to represent program data structures
  - reasoning about them

Subtlety of Array Assignment

Rule for wp of assignment of expression E to variable x, for postcondition P:

wp(x=E , P) = $P[x := E]$

$(a(i := y+1))(i) > 5 \wedge$

$(a(i := y+1))(j) > 3$

Example:

wp(x=y+1, x > 5) = $y+1>5$

$x = y+1$

assert$(x>5)$

wp of assignment to a pre-allocated array cell:

wp(a[i]=y+1, a[i]>5) = $y+1>5$

wp(a[i]=y+1, a[i]>5 ⋀ a[j]>3) =

$wp(a = a(i := y+1), a(i) > 5 \wedge a[j] > 3) =$

$=$

$(i = j \wedge y+1 > 5 \wedge y+1 > 3) \vee$

$(i \neq j \wedge y+1 > 5 \wedge a[j] > 3)$

MAPS

Map[A,B] - immutable (function) A -> B

type	*is like...*	*this map*
String		**Map[Int,Char]**
List[B]		**Map[Int,B]**
class A { **var** f: B}		**var f : Map[A,B]**
x.f==y		f(x)==y

for now ignore this:

a1,a2: Array[B]

ga: Map[Object,**Map[Int,B]**]

ga(a1) : Map[Int,B]

ga(a2) : Map[Int,B]

Key Operation on Maps

Map lookup: **f(x)**

Map update: **f(x:=v) == g** meaning f(x->v)==g

1. g(x)=v
2. g(y)=f(y) for y != x.

Represent assignments:

x = a[i] → x = a(i)

a[i]=v → a = a(i:=v)

Pre-Allocated Arrays

- These are static arrays identified by name, to which we can only refer through this name
- Many reasonable languages had such arrays, for example as global array variables in Pascal
- They can be approximated by:
  - static initialized Java arrays, e.g.

    static int[] a = new int[100];

    if we never do array assignments of form **foo=a;**
  - static arrays in C, if we never create extra pointers to them nor to their elements

Modeling Pre-Allocated Arrays

We always update entire map

Copy semantics!

original program

```
b[0]=100;
assert(b(0)==100);
```

guarded commands:

```
b= b(0:=100);
assert(b(0)==100);
```

using Scala immutable maps

```
b= b + (0 -> 100)
assert(b(0)==100)
```

Modeling using Immutable Maps

We always update entire arrays

Copy semantics!

guarded commands:

```
b= b(0:=100);
assert(b(0)==100); ok
a= b; // copy
a= a(0:=200);
assert(b(0)==100); ok
```

corresponds to Scala maps

```
var a = Map[Int,Int]()
var b = Map[Int,Int]()
b= b + (0 -> 100)
assert(b(0)==100)    ok
a= b //  share, immutable
a= a + (0 -> 200)
assert(b(0)==100)    ok
```

Weakest Preconditions for Pre-Allocated Arrays

wp(a[i]=E, P) = $wp\left(a = a(i := E) ,\ P \right)$

$= P[\, a := a(i := E) \,]$

$a[i] = E \quad \longrightarrow \quad a = a(i := E) \longrightarrow \left(\begin{array}{l} a' = a(i := E) \\ \wedge\ b' = b \\ \wedge \ldots \end{array} \right)$

Example

```
if (a[i] > 0) {
  b[k]= b[k] + a[i];
  i= i + 1;
  k = k + 1;
} else {
  b[k] = b[k] + a[j];
  j= j + 1;
  k = k – 1;
}
```

b= b(k:= b(k)+a(i))

Formula for this Example

guarded commands:

```
(assume(a(i) > 0);
  b= b(k:= b(k)+ a(i));
  i= i + 1;
  k = k + 1;)
[] (assume(a(i)<=0);
  b= b(k:= b(k)+ a(j));
  j= j + 1;
  k = k – 1;
)
```

formula:

$$
\begin{aligned}
&(a(i)>0 \wedge \\
&\quad b' = b(k := b(k) + a(i)) \wedge \\
&\quad i' = i+1 \wedge \\
&\quad k' = k+1) \\
\vee\ &(\\
&)
\end{aligned}
$$

Array Bounds Checks: Index >= 0

```
if (a[i] > 0) {
  b[k]= b[k] + a[i];
  i= i + 1;
  k = k + 1;
} else {
  b[k] = b[k] + a[j];
  j= j + 1;
  k = k – 1;
}
```

```
assert(i >= 0)
(assume(a(i) > 0);
  assert k≥0
  assert i≥0
  assert k≥0
  b= b(k:= b(k)+ a(i));
  i= i + 1;
  k = k + 1;)
[] (assume(a(i)<=0);
  assert
  assert
  assert
  b= b(k:= b(k)+ a(j));
  j= j + 1;
  k = k – 1;
)
```

How to model “index not too large”

```
const M = 100
const N = 2*M
int a[N], b[N];
...
if (a[i] > 0) {
  b[k]= b[k] + a[i];
  i= i + 1;
  k = k + 1;
}
```

```
assert
(assume(a(i) > 0);
  assert   k < 200
  assert   i < 200
  assert   k < 200
  b= b(k:= b(k)+ a(i));
  i= i + 1;
  k = k + 1;)
[] (assume(a(i)<=0))
```

Translation of Array Manipulations with Bounds Checks when Size is Known

x= a[i] → assert(0 <= i);

assert(i < a_size);

x = a(i);

a[i] = y → assert(0 <= i);

...assert(i < a_size);

a = a(i:=y)

Example for Checking Assertions

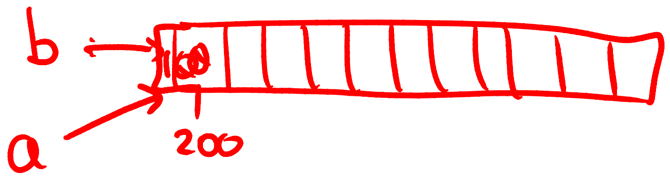
```
const M = 100;
const N = 2*M;
int a[N], b[N];
i = -1;  I ; k=-1;
while (i < N) {        i<N-1
  i= i + 1;
  if (a[i] > 0) {
    k = k + 1;
    b[k]= b[k] + a[i];
  }
}
```

```
i=-1;
( assume(i<N);
  i=i+1;
  assert(i>=0)
  assert(i<N);
  b=(a(i)>0);
  ( assume(b);
    k=k+1;
    assert(k>=0)
    assert(k<N)
    v1=b(k)
    v2=a(i)
    assert(0<=k)
    assert(k<N)
    b=b(k:=v1+v2)
  ) []
  assume(!b)
)*
```

I: $i \geq -1 \wedge k \geq -1 \wedge \boxed{k \leq i}$

1. Translate to guarded commands
2. Find a loop invariant and prove it inductive
3. Show that the invariant implies assertions

Mutable Arrays are by Reference



Java (also Scala arrays and mutable maps):

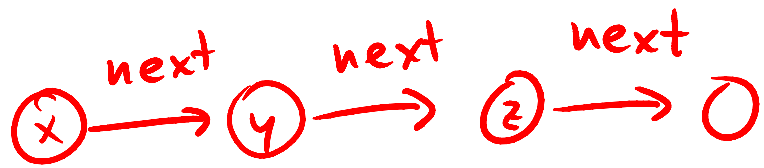
```
b[0]= 100;
assert(b[0]==100);
a= b; // make references point to same array
a[0]= 200;
assert(b[0]==100);  // fails, b[0]==a[0]==200
```

To model Java Arrays, we first examine how to model objects in general

Reference Fields

var next : Map[Node, Node]

class Node { Node next; }



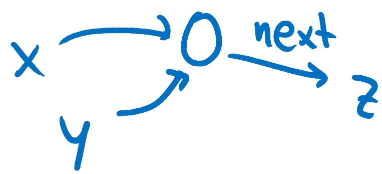
How to model 'next' field?

$\underbrace{next}(x) = y$

x = y;

x.next = z

assert(y.next == z)



y = x.next; → $y = next(x)$

x.next = y; → $next = next(x := y)$

Each Field Becomes Function
Each Field assignment becomes Function Update

```
class Circle {
  int radius;
  Point center;
  void grow() {
    radius = radius * 2;
  }
}
```

radius : Circle -> int

center : Circle -> Point

➔

this.radius = this.radius * 2

➔

radius= radius(this:= radius(this)*2)

Field Manipulations with Checks

x=y.f → assert (y != null)

x= f(y)

y.f = x → assert (y != null)

f= f(y := x)

x.f.f= z.f + y.f.f.f ; →

y1 = y.f

y2 = y1.f

y3 = y2.f

z1 = z.f

x1 = x.f

x1.f = z1 + y3

f = f (x1 := z1 + y3)

All Arrays of Given Result Become One Class
Array Assignment Updates Given Array at Given Index

all possible integer-valued arrays

```
class Array {
  int length;
  data : int[]
}
a[i] = x
```

length : Array -> int

data : Array -> (Int -> Int)

or simply: Array x Int -> Int

➔ a.data[i] = x

➔ data= data((a,i):= x)

Assignments to Java arrays: Now including All Assertions (safety ensured, or your models back)

```
class Array {
  int length;
  data : int[]
}
```

length : Array -> int

data : Array -> (Int -> Int)

or simply: Array x Int -> Int

a[i] = x $\rightarrow$

assert(a != null)

assert($0 \le i \land i <$ length(a))

data= data((a,i):= x)

y = a[i] $\rightarrow$

assert (a != null)

assert($0 \le i \land i <$ length (a))

y = data(a,i)

Variables in C and Assembly

Can this assertion fail in C++ (or Pascal)?

```
void funny(int& x, int& y) {
  x= 4;
  y= 5;
  assert(x==4);
}
int z;
funny(z, z);
```

Memory Model in C and Assembly

Just one global array of locations:

mem : int → int // one big array (or int32 -> int32)

each variable x has address in memory, xAddr, which is &x

We map operations to operations on this array:

```
int x;
int y;
int* p;
y= x        → mem[yAddr]= mem[xAddr]
x=y+z       → mem[xAddr]= mem[yAddr] + mem[zAddr]
y = *p      → mem[yAddr]= mem[mem[pAddr]]
p = &x      → mem[pAddr] = xAddr
*p = x      → mem[mem[pAddr]]= mem[xAddr]
```

Variables in C and Assembly

Can this assertion fail in C++ (or Pascal)?

```
void funny(int& x, int& y) {
  x= 4;
  y= 5;
  assert(x==4);
}
int z;
funny(&z, &z);
```

```
void funny(xAddr, yAddr) {
  mem[xAddr]= 4;
  mem[yAddr]= 5;
  assert(mem[xAddr]==4);
}
zAddr = someNiceLocation
funny(zAddr, zAddr);
```

Exact Preconditions in C,Assembly

Let x be a local integer variable.

In Java:

$wp(x{=}E,\ y > 0) =$

In C:

$wp(x{=}E,\ y > 0) =$

Disadvantage of Global Array

In Java:

$wp(x{=}E,\ y > 0) = y > 0$

In C:

$wp(x{=}E,\ y > 0) =$

$wp(mem[xAddr]{=}E',\ \ mem[yAddr]{>}0) =$

$wp(mem{=}\ mem(xAddr{:=}E'),\ mem(yAddr){>}0) =$

$(mem(yAddr){>}0)[\ mem{:=}mem(xAddr{:=}E')\ ] =$

$(mem(xAddr{:=}E'))(yAddr) > 0$

Each assignment can interfere with each value!

This is absence of interference makes low-level languages unsafe and difficult to prove partial properties.

To prove even simple property, we must know something about everything.

How to do array bounds checks in C?

See e.g. the Ccured project:

http://ostatic.com/ccured

CCured: type-safe retrofitting of legacy software
Necula et al.
ACM Transactions on Programming Languages and Systems (TOPLAS)

Volume 27 Issue 3, May 2005

Back to Memory Safety

Memory Allocation in Java

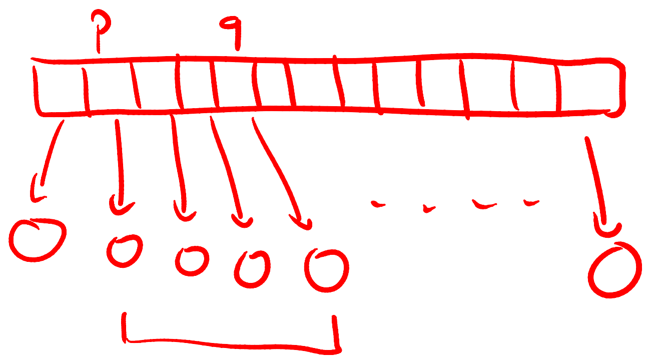
```
x = new C();
y = new C();
assert(x != y);   // fresh object references-distinct
```

Why should this assertion hold?

How to give meaning to ‘new’ so we can prove it?

How to represent fresh objects?

```
assume(N > 0 /\ p > 0 /\ q > 0 /\ p != q);
a = new Object[N];
i = 0;
while (i < N) {
  a[i] = new Object();
  i = i + 1;
}
assert(a[p] != a[q]);
```



A View of the World

Everything exists, and will always exist.
(It is just waiting for its time to become allocated.)

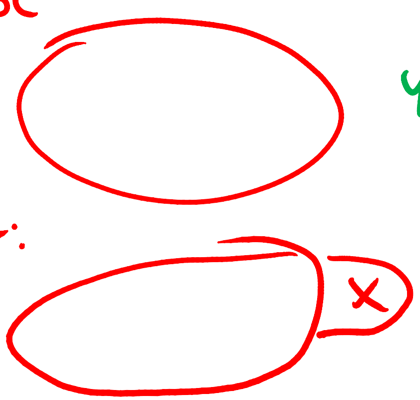
It will never die (but may become unreachable).

alloc : Obj → Boolean i.e. alloc : Set[Obj]

x = new C(); → havoc(x);
assume($x \notin alloc$);
$alloc_1 = alloc \cup \{x\}$; $x \in alloc_1$

^defult constructor

before:
alloc



after:

y = new C()
assert($x \neq y$)

havoc(y);
assume($y \notin alloc_1$); $y \notin alloc_1$
$alloc_2 = alloc_1 \cup \{y\}$