Transition Systems of Scala Programs

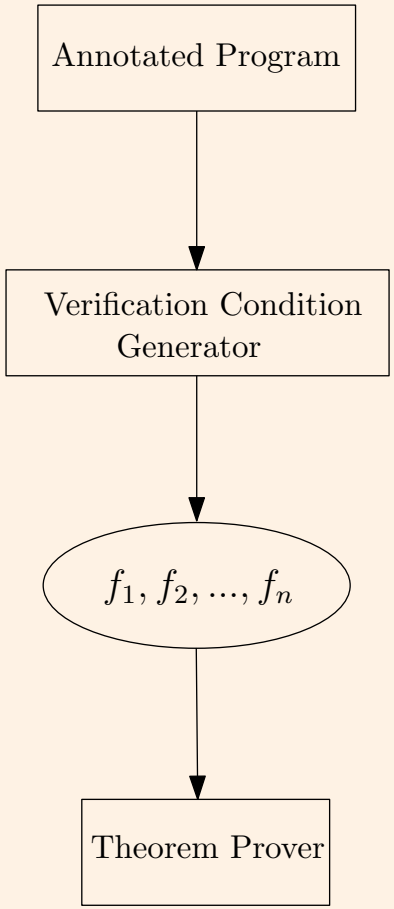
Hossein Hojjat

EPFL

May 28, 2009

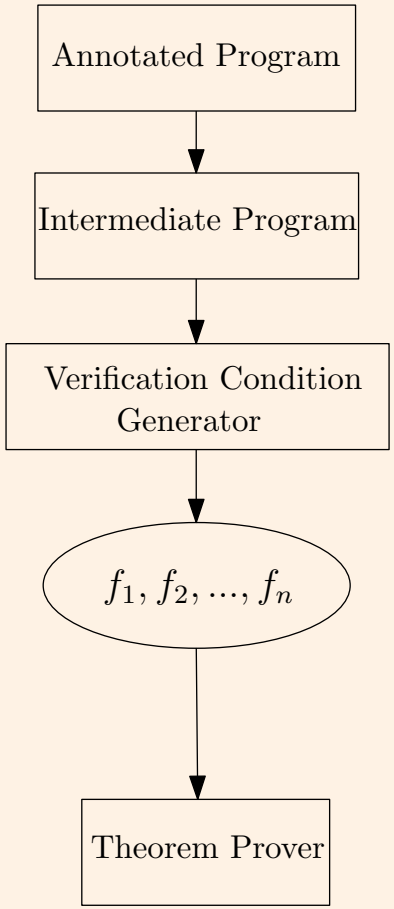
Outline

Big Picture of VCG



- Increase in the complexity of programming languages
- Memory allocation, exception handling, different data structures, ...
- Difficulty of verification condition generation

Big Picture of VCG



- Increase in the complexity of programming languages
- Memory allocation, exception handling, different data structures, ...
- Difficulty of verification condition generation
- Split VC generation into 2 phases
  1. Source and specification are compiled to an intermediate verification language
  2. Generation of VCs from the intermediate language

Different Intermediate Languages

- Intermediate language in many modern program verifiers

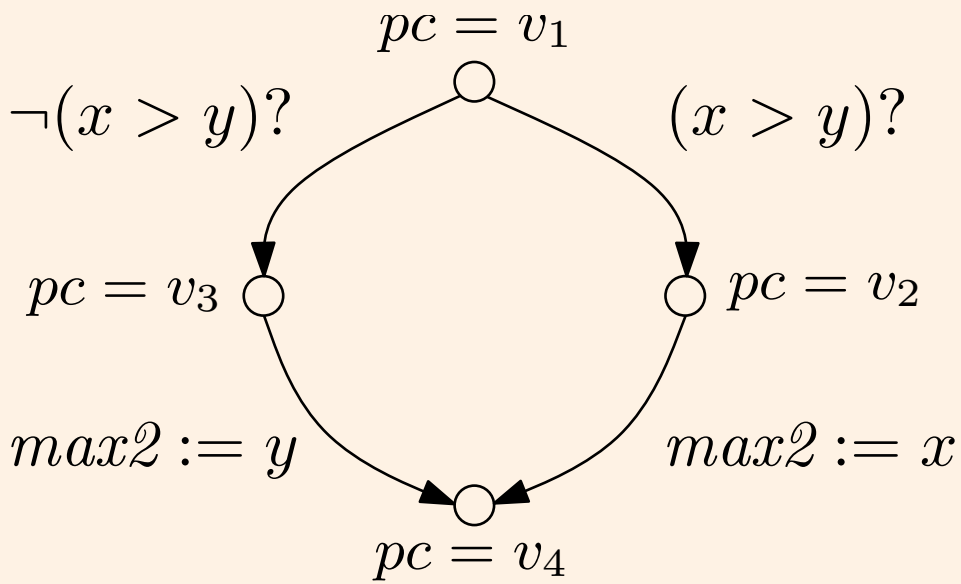
Jahob:	(Extended/Simple) Guarded commands
Spec#:	BoogiePL
ESC/Java:	Guarded Commands
Krakatoa:	Why

- Design an intermediate language for *Scala*
- Based on Control Flow Graphs (CFG)

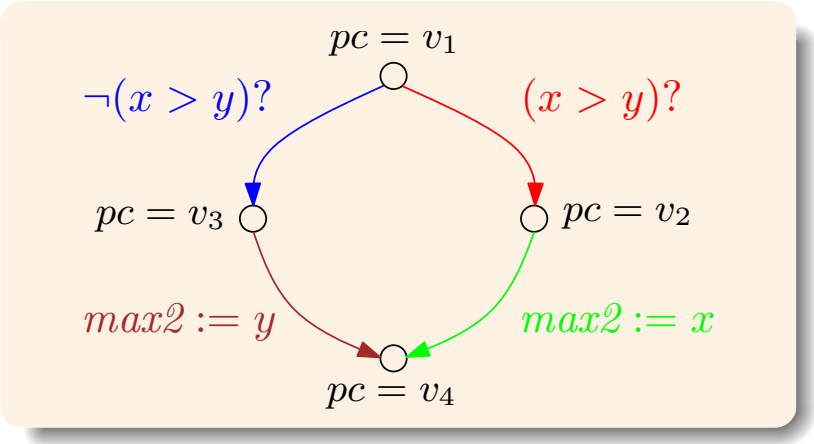
Simple Program

Maximum

```
def max2(x:Int,y:Int):Int= {
  if ( x > y) x
     else y
  }
```



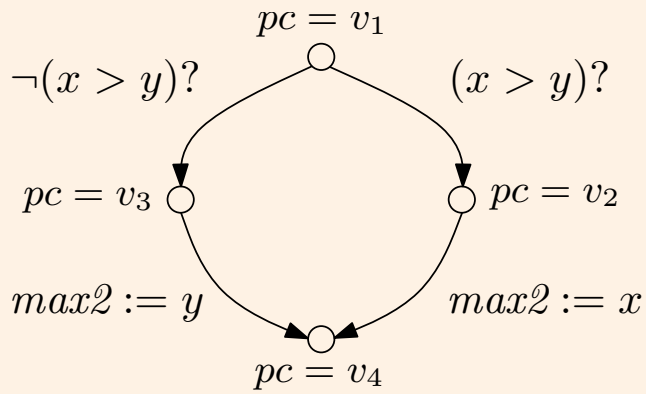
CFG



$$r = \{((pc, x, y, max2), (pc', x', y', max2')) \mid$$
$$((pc = v_1) \wedge (pc' = v_2) \wedge (x > y) \wedge (x' = x) \wedge (y' = y) \wedge (max2' = max2))$$
$$\bigvee((pc = v_1) \wedge (pc' = v_3) \wedge \neg(x > y) \wedge (x' = x) \wedge (y' = y) \wedge (max2' = max2))$$
$$\bigvee((pc = v_2) \wedge (pc' = v_4) \wedge (x' = x) \wedge (y' = y) \wedge (max2' = x))$$
$$\bigvee((pc = v_3) \wedge (pc' = v_4) \wedge (x' = x) \wedge (y' = y) \wedge (max2' = y))\}$$

CFG in Isabelle/HOL

```
datatype label = v1 | v2 | v3 | v4
record state =
  pc   :: label
  x    :: int
  y    :: int
  max2 :: int
definition program :: "(state × state) set"
  where "program ≡{(s,s').
  (s'=s(|pc:=v2|) ∧(x s>y s) ∧ (pc s=v1))∨
  (s'=s(|pc:=v3|) ∧ ¬(x s>y s) ∧ (pc s=v1))∨
  (s'=s(|pc:=v4,max2:=x s|) ∧ (pc s=v2))∨
  (s'=s(|pc:=v4,max2:=y s|) ∧ (pc s=v3))}
```



Correctness

- r relation representing program, s relation representing specification
- Program meets specification iff $r* \subseteq s$
- Unroll the transitive closure for a finite number of times

$$r^n \subseteq r* \subseteq s \ , \ n \in \mathbb{N}$$

Correctness

definition program :: "(state × state) **set**"
where "program ≡{(s,s'). (s'=s⟦pc:=v2⟧ ∧(x s>y s) ∧ (pc s=v1))∨ (s'=s⟦pc:=v3⟧ ∧ ¬(x s>y s) ∧ (pc s=v1))∨ (s'=s⟦pc:=v4,max2:=x s⟧ ∧ (pc s=v2))∨ (s'=s⟦pc:=v4,max2:=y s⟧ ∧ (pc s=v3))}

lemma maximum [simp]:
"(program ∘ program ∘ program) ⊆
{(s,s'). (pc s = v1) ⟶ (max2 s' = max (x s) (y s))}"
apply (unfold program_def)
apply auto
done

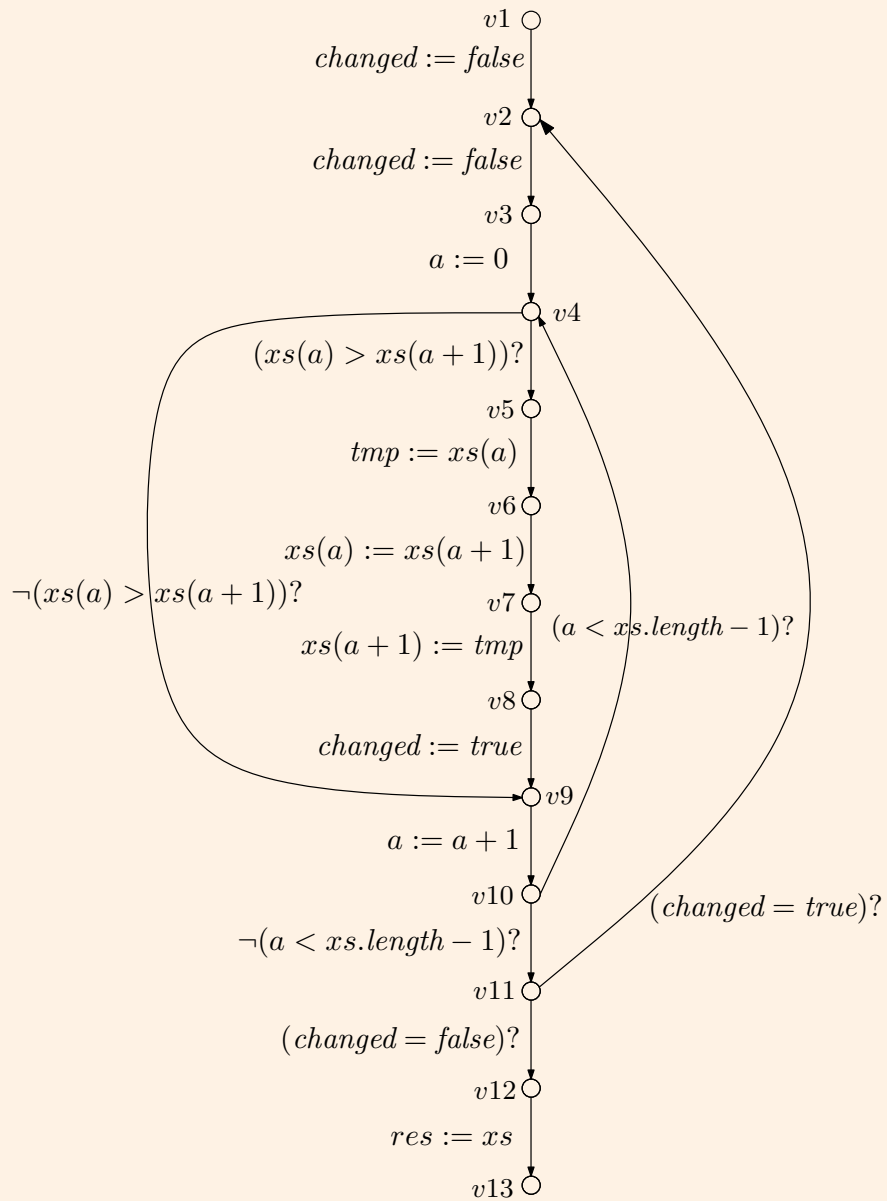
Challenges

1. Find CFG for different programs
   - Data structures: Arrays, maps, ...
   - Recursive functions
   - Concurrent processes
2. Describe CFGs in Isabelle/HOL
3. Come up with a correctness proof

Outline

Bubble Sort

```
def bubblesort(xs:Array[Int])
              :Array[Int]={
  var changed=false
  do{
   changed=false
   for(a←0 until (xs.length - 1))
   if(xs(a)>xs(a+1)){
     var tmp=xs(a)
     xs(a)=xs(a+1)
     xs(a+1)=tmp
     changed=true
     }
   }while(changed)
  xs
  }
```



Arrays

Scala	Isabelle/HOL
xs : Array[Int]	xs :: "int list"
xs(5)	xs ! 5
xs.length	length xs
xs(1) = 4	list_update xs 1 4
xs map (x ⇒ x + 1)	map (λx. x + 1) xs

- Possible problems in the mapping?

Pattern Matching

```
sealed abstract class Expr
  case class Add(v1:Int,v2:Int) extends Expr
  case class Multiply(v1:Int,v2:Int) extends Expr
  def calculate(e:Expr):Int = e match {
    case Add(o1,o2) ⇒ o1 + o2
    case Multiply(o1,o2) ⇒ o1 * o2
  }
```

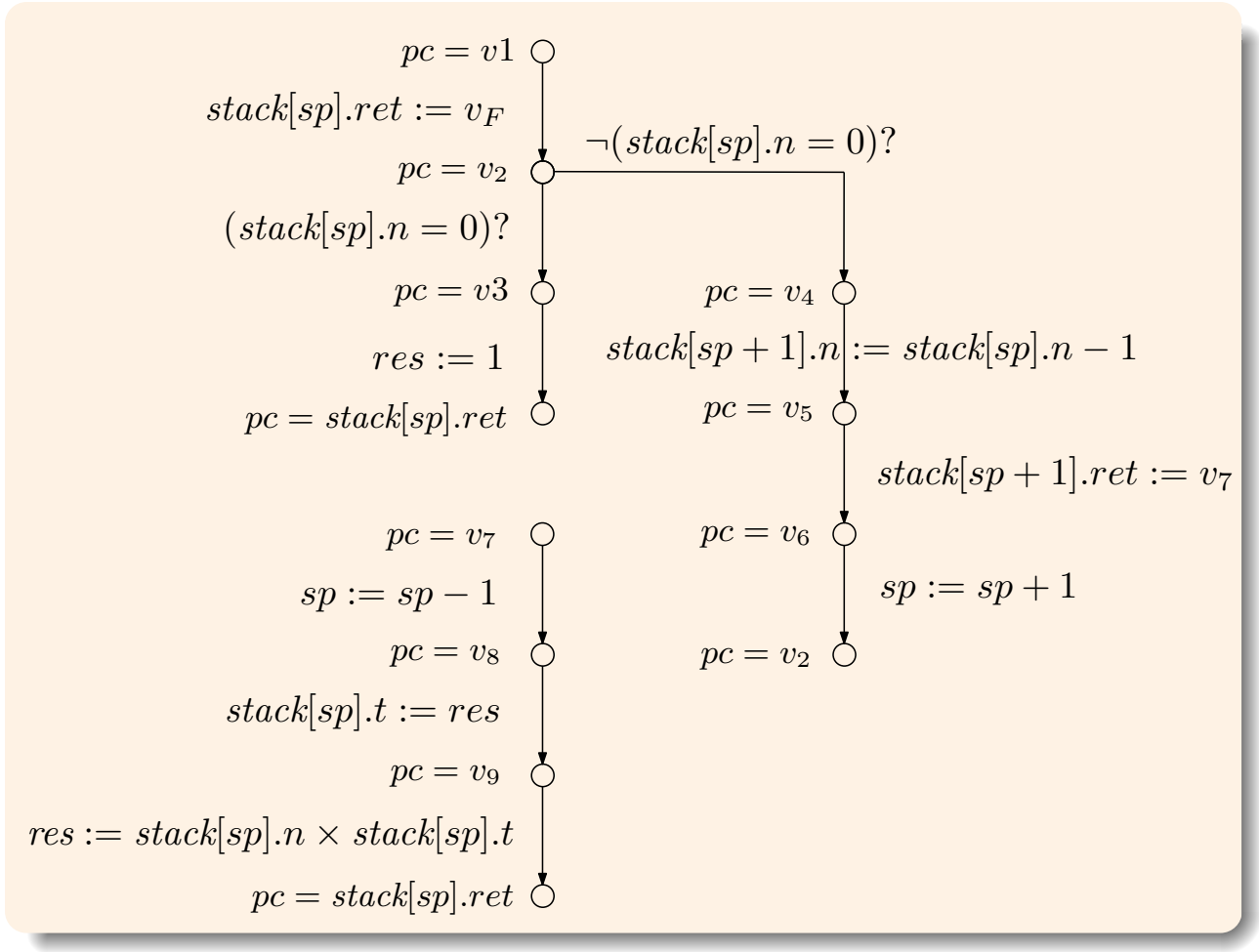
Partial Patterns in Isabelle/HOL

```
datatype label = v1 | v2 | v3
datatype ('o1,'o2) Expr = Add "'o1" "'o2" | Multiply "'o1" "'o2"
record state =
  pc    ::   label
  res   ::   int
  e     ::   "(int,int) Expr"
definition program :: "(state × state) set" where
  "program ≡ {(s,s').
   ( case (e s) of Add o1 o2 ⇒ s' = s(| pc := v2, res := o1 + o2|) ∧
     ( pc s = v1) ) ∨
   ( case (e s) of Multiply o1 o2 ⇒ s' = s(| pc := v3, res := o1 * o2|) ∧
     ( pc s = v1) )}"
```

Outline

Factorial

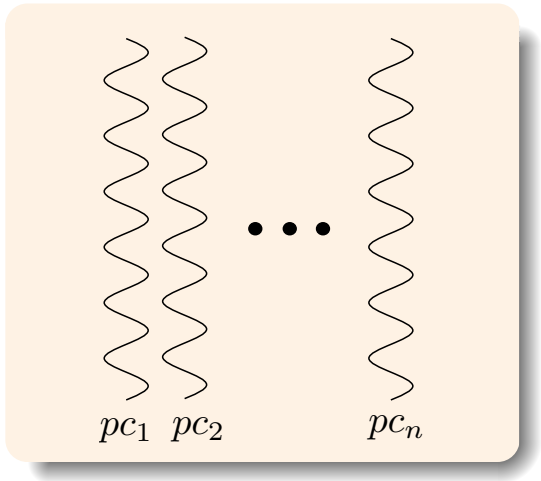
```
def fact(n:Int):Int={
  if(n==0) 1
  else {
   val t=fact(n-1)
   n*t   }
}
```



Outline

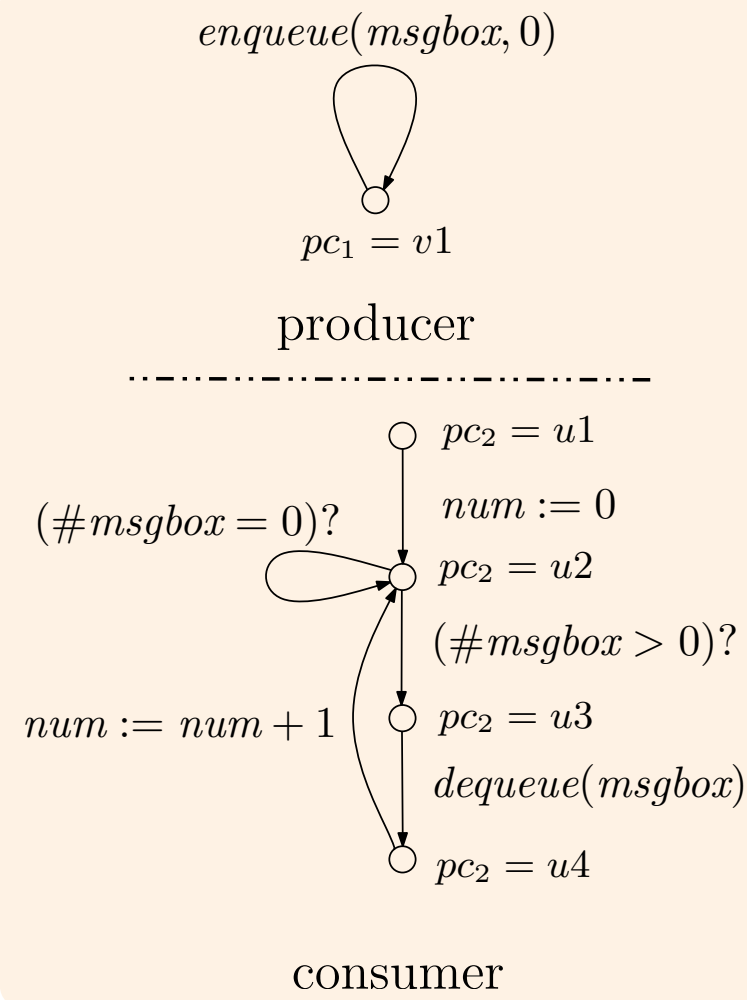
Idea

- Sequential running of all considered cases
- *pc* controls the transitions in a CFG
- Use a separate *pc* for each concurrent process
- Existentially quantify when the number of concurrent processes are not determined

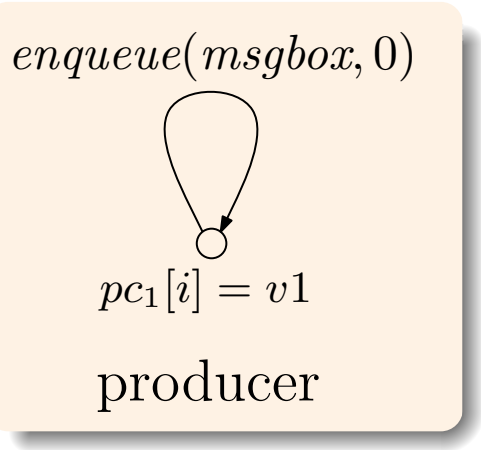


Producer Consumer

```
class Producer(c:Consumer) extends
Actor{
  def act(){
    while(true) c!0
}}
class Consumer extends Actor {
  var num = 0
  def act() {
  react{
  case msg ⇒ {
    num = num + 1
  act}
}}}
```



Unknown number of processes



datatype label = v1
record state =
  pc1 :: "label list"
  msgbox :: "int list"
definition producer :: "(state × state) set"
where "producer ≡ {(s,s').∃i.
  (s' = s⦇ pc1 := list_update (pc1 s) i v1, msgbox := Cons 0 (msgbox s)⦈
  ∧ ((pc1 s)!i = v1))}

Future Work

- Implement the mapping for a subset of Scala
- Adapt to the upcoming theorem prover of LARA