

# Lecture 3

## From (Integer) Programs to Formulas

Viktor Kuncak

# Verification Condition Generation Example

We examine algorithms for going from programs to their verification conditions.

Program and postcondition:

```
def f(x : Int) : Int = {  
  if (x > 0)  
    2*x + 1  
  else 42  
} ensuring (res ==> res > 0)
```

Verification condition saying “program satisfies postcondition”:

$$\left[ \left( (x > 0) \wedge res = 2x + 1 \right) \vee \left( \neg(x > 0) \wedge res = 42 \right) \right] \rightarrow res > 0$$

For above formula, we would check *validity*: all variables are universally quantified

# What Relations Can Represent

Let  $r$  be  $\rho(c)$ , the relation describing the command  $c$ .  
For an initial state  $s$ , we can compute the set of states that the system can end up after executing  $c$ :

$$r[\{s\}] = \{s' \mid (s, s') \in r\}$$

This set of states can be

- ▶ a singleton set  $\{s'\}$ , meaning that precisely one result is possible
- ▶

# Verification Condition Generation (VCG) For Functions

```
def f( $\bar{x}$  : Intn) : Int = {  
  b( $\bar{x}$ )  
} ensuring (res ==> Post( $\bar{x}$ , res))
```

- ▶ Function  $f$  with arguments  $\bar{x}$  and body  $b(\bar{x})$ , built from:
  - ▶ Presburger Arithmetic (PA) expressions, as well as  $x/K$ ,  $x\%K$
  - ▶ **if** statement, and local value definitions (**val** in Scala)
- ▶ Postcondition  $Post(\bar{x}, res)$  written in quantifier-free PA

Claim: there is **polynomial-time** algorithm to construct formula  $V(\bar{x})$  such that

- ▶ the execution of  $f$  on input  $\bar{x}$  meets the Post iff  $V(\bar{x})$   
Hence, it always meets postcondition iff  $\forall \bar{x}. V(\bar{x})$
- ▶  $V(\bar{x})$  is quantifier-free or has only top-level  $\forall$  quantifiers

Idea: perhaps  $V(\bar{x})$  could be  $Post(\bar{x}, b(\bar{x}))$  ? Yes, if it was in PA

## PA with $x/K$ , $x\%K$ , **if**, **val**

Context-Free grammar (syntax) of extended PA formulas

$F, b$  : Boolean,  $t$  : Int

$$\begin{aligned} F & ::= b \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid \neg F \mid \exists x.F \mid \forall x.F \mid t_1 < t_2 \mid t_1 = t_2 \\ & \quad \mid \{\mathbf{val} \mathbf{x} = \mathbf{t}; \mathbf{F}\} \mid \{\mathbf{val} \mathbf{b} = \mathbf{F}_1; \mathbf{F}\} \\ t & ::= x \mid K \mid t_1 + t_2 \mid K \cdot t \\ & \quad \mid \mathbf{t}/\mathbf{K} \mid \mathbf{t} \% \mathbf{K} \mid \mathbf{if}(\mathbf{F}) \mathbf{t}_1 \mathbf{else} \mathbf{t}_2 \mid \{\mathbf{val} \mathbf{x} = \mathbf{t}_1; \mathbf{t}_2\} \end{aligned}$$

We show how to translate  $x/K$ ,  $x\%K$ , **if**, **val** into other constructs

- ▶ without changing the meaning of a formula
- ▶ without adding alternations of quantifiers
- ▶ in time polynomial in input  
(result is thus also in polynomial size)

# Reminder: Free Variables and Substitutions

## Free Variables

$FV(t)$ ,  $FV(F)$  denotes free variables in term  $t$  or formula  $F$   
Normally we just collect all variables:

$$FV(x + y < z) = \{x, y, z\}$$

We do not count quantified occurrences of variables:

$$FV(\exists x. x + y < z) = \{y, z\}$$

If it occurs quantified somewhere it can still be free overall:

$$FV((\exists x. \exists y. x < y + u) \wedge (\exists y. x + y < z + 100)) = \{u, x, z\}$$

Rules for FV are of two kinds: operations  $\odot$  (e.g.,  $\wedge$ ,  $<$ ,  $+$ ) and binders  $Q$  (e.g.,  $\forall$ ,  $\exists$ ,  $\text{val}$ )

$$FV(x) = \{x\}, \text{ if } x \text{ is a variable}$$

$$FV(F_1 \odot F_2) = FV(F_1) \cup FV(F_2)$$

$$FV(Qx.F) = FV(F) \setminus \{x\}$$

## Substitutions

One possible convention: write  $F(x)$  and later  $F(t)$ . Then  $F$  is not a formula but function from terms to formulas

(Or we do not even know what  $F$  is.)

Our notation: write  $F$ , and instead of  $F(t)$  write  $F[x := t]$

- ▶ closer to a typical implementation

Definition of substitution:

$$\begin{aligned}(F_1 \odot F_2)[x := t] &\rightsquigarrow (F_1[x := t]) \odot (F_2[x := t]) \\ (Qy.F)[x := t] &\rightsquigarrow Qy.(F[x := t])\end{aligned}$$

Capture:

The following formula is true in integers for all  $x$ :  $\exists y.x < y$

If we naively substitute  $x$  with  $y + 1$  we obtain:  $\exists y.y + 1 < y$

Problem:  $t$  has  $y$  free. A solution: rename  $y$  to fresh  $y_1$

$$(Qy.F)[x := t] \rightsquigarrow (Qy_1.F[y := y_1])[x := t] \rightsquigarrow Qy_1.(F[y := y_1][x := t])$$



# Summary of Our Translation Goal

Transform logic of this grammar

$F, b$  : Boolean,  $t$  : Int

$$\begin{aligned} F & ::= b \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid \neg F \mid \exists x.F \mid \forall x.F \mid t_1 < t_2 \mid t_1 = t_2 \\ & \quad \mid \{\mathbf{val\ } x = \mathbf{t}; \mathbf{F}\} \mid \{\mathbf{val\ } b = \mathbf{F}_1; \mathbf{F}\} \\ t & ::= x \mid K \mid t_1 + t_2 \mid K \cdot t \\ & \quad \mid \mathbf{t/K} \mid \mathbf{t \% K} \mid \mathbf{if\ (F)\ t_1\ else\ t_2} \mid \{\mathbf{val\ } x = \mathbf{t}_1; \mathbf{t}_2\} \end{aligned}$$

Into a logic for which we did quantifier elimination, which omits the bold symbols:

- ▶ val (let) definitions in formulas and terms
- ▶ conditionals
- ▶ division by a constant
- ▶ computing modulo by a constant as a term

## About val Definitions

$$\{val\ x = t; E\}$$

Equivalent ways of saying:

- ▶ in the rest of the block, introduce read-only variable  $x$  with value equal to  $t$
- ▶ let  $x$  have the value  $t$  in  $E$  (written so in ML, Haskell)
- ▶  $E$ , where  $x$  has the value  $t$  (math, Haskell's where clause)
- ▶ in lambda calculus:  $(\lambda x.E)t$

Slightly different cases depending on whether types of  $t$  and  $E$  (each of which can be Boolean or Int)

Note:  $x$  is bound to  $t$  inside  $E$ , but not inside  $t$  or anywhere else

# Free Variables and Substitution for val

Computing free variable:

$$FV(\{val\ x = t; E\}) = FV(t) \cup (FV(E) \setminus \{x\})$$

Substitution, for  $x \neq y$ ,  $x \notin FV(s)$  (otherwise, rename  $x$ ):

$$(\{val\ x = t; E\})[y := s] = \{val\ x = t[y := s]; (E[y := s])\}$$

Renaming means transforming  $\{val\ x = t; E\}$  into  $\{val\ x_1 = t; E[x := x_1]\}$  where  $x_1$  is different from other relevant variables (clear from the context)

# How to Translate Value Definitions

Construct:  $\{val\ x = t; F\}$  where we additionally require  $x \notin FV(t)$   
(otherwise just rename  $x$ )

Example

$$\{val\ x = y + 1; x < 2x + 5\}$$

Becomes one of these:

$(y + 1) < 2(y + 1) + 5$	substitution
$\exists x. x = y + 1 \wedge x < 2x + 5$	one-point rule
$\forall x. x = y + 1 \rightarrow x < 2x + 5$	dual one-point rule

## Rule to Translate Value Definitions

In general, for  $x \notin FV(t)$

$$\{val\ x = t; F\}$$

Becomes one of these:

$F[x := t]$	substitution
$\exists x. x = t \wedge F$	one-point rule
$\forall x. x = t \rightarrow F$	dual one-point rule

Substitution can square formula size

- ▶ Do it several times  $\rightsquigarrow$  exponential increase

The other rules add quantified variables

- ▶ but we can choose which way they are quantified, to avoid adding quantifier alternations

## Dual of val elimination is flattening: remove nested Terms

Similar to compilation

Example:

$$x + 3y < z$$

flattening  $3y$  and denoting it by  $y_1$  we get

$$\{val\ y_1 = 3y; x + y_1 < z\}$$

and then flattening  $x + y_1$  denoting it by  $y_2$  we get

$$\{val\ y_1 = 3y; \{val\ y_2 = x + y_1; y_2 < z\}\}$$

which we may write as

```
{ val y1=3y
  val y2=x+y1
  y2 < z
}
```

# Flattening Rule

Suppose  $F$  contains  $t_1 \odot t_2$  somewhere and we wish to pull it out.  
For some fresh  $y_1$  then  $F$  becomes

$$\{ \text{val } y_1 = t_1 \odot t_2; F[t_1 \odot t_2 := y_1] \}$$

## We can now handle val for formulas. What about terms?

Lifting val-s outside until they reach formulas

$$\{val\ x = a + 1; 2x\} + 5 < y$$

becomes

$$\{val\ x = a + 1; 2x + 5 < y\}$$



## val given by val rule

$$\{val\ x = \{val\ y = a + 1; y + y\}; x < 2x\}$$

becomes

$$\{val\ y = a + 1; \{val\ x = y + y; x < 2x\}\}$$

which we pretty-print as

$$\{val\ y = a + 1; val\ x = y + y; x < 2x\}$$

Flat form:

- ▶ each operation  $\odot$  is inside a  $\{val\ x = y_1 \odot y_2; F\}$
- ▶ atomic formulas only use variables
- ▶ val applies to formulas only (not terms)

## Translating **if**

$F, b$  : Boolean,  $t$  : Int

$F ::= b \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid \neg F \mid \exists x.F \mid \forall x.F \mid t_1 < t_2 \mid t_1 = t_2$   
 $\mid \{\mathbf{val\ } x = \mathbf{t}; \mathbf{F}\} \mid \{\mathbf{val\ } b = \mathbf{F}_1; \mathbf{F}\}$

$t ::= x \mid K \mid t_1 + t_2 \mid K \cdot t$   
 $\mid \mathbf{t/K} \mid \mathbf{t \% K} \mid \mathbf{if(F) t_1 else t_2} \mid \{\mathbf{val\ } x = \mathbf{t_1}; \mathbf{t_2}\}$

Suppose terms are in flat form. We only need to handle:

$\{\mathit{val\ } x = (\mathit{if}(b_1)\ t_1\ \mathit{else}\ t_2); F\}$

Note that the logical equality

$x = (\mathit{if}(b_1)\ t_1\ \mathit{else}\ t_2) \quad (*)$

is equivalent to

$(b_1 \wedge x = t_1) \vee (\neg b_1 \wedge x = t_2)$

as well as to:

$((b_1 \rightarrow x = t_1) \wedge (\neg b_1 \rightarrow x = t_2))$

## Translating **if**

From two one-point rule translations of `val`, we can thus transform

$$\{val\ x = (if(b_1)\ t_1\ else\ t_2);\ F\}$$

into any of these:

$$\begin{aligned} &\exists x. \left[ ((b_1 \wedge x = t_1) \vee (\neg b_1 \wedge x = t_2)) \wedge F \right] \\ &\exists x. \left[ ((b_1 \rightarrow x = t_1) \wedge (\neg b_1 \rightarrow x = t_2)) \wedge F \right] \\ &\forall x. \left[ ((b_1 \wedge x = t_1) \vee (\neg b_1 \wedge x = t_2)) \rightarrow F \right] \\ &\forall x. \left[ ((b_1 \rightarrow x = t_1) \wedge (\neg b_1 \rightarrow x = t_2)) \rightarrow F \right] \end{aligned}$$

This translates `if-else` without duplicating sub-formulas (thanks to boolean variable  $b_1$ ).

# Integer Division by a Constant

Consider

$$\{\text{val } q = p/K; F\}$$

The corresponding equality  $q = p/K$  is equivalent to

$$Kq \leq p \wedge p < K(q + 1)$$

Which gives corresponding translations:

$$\begin{aligned} &\exists q. [Kq \leq p \wedge p < K(q + 1) \wedge F] \\ &\forall q. [(Kq \leq p \wedge p < K(q + 1)) \rightarrow F] \end{aligned}$$

# Remainder Modulo a Constant

$$\{val\ r = p \% K; F\}$$

One way:

$$\{val\ r = p - K(p/K); F\}$$

## Quantifier-Free Polynomial-Sized VC

```
def f( $\bar{x} : \text{Int}^n$ ) : Int = {  
  b( $\bar{x}$ )  
} ensuring (res ==> Post( $\bar{x}$ , res))
```

VC in quantifier-free PA extended with val, if, /, % :

$$res = b(\bar{x}) \rightarrow Post(res, \bar{x})$$

Eliminate extensions, choosing always existential quantifiers for new variables  $\bar{z}$ . Moreover, such existentials can be pulled to top-level, because we only introduced  $\vee, \wedge$  and never  $\neg$  for sub-formulas. We obtain:

$$(\exists \bar{z}. F(res, \bar{x}, \bar{z})) \rightarrow Post(res, \bar{x})$$

which is equivalent to

$$\forall \bar{z}. [F(res, \bar{x}, \bar{z}) \rightarrow Post(res, \bar{x})]$$

So, all variables are universally quantified.

## Explaining $(\exists F) \rightarrow G$

Indeed, from first-order logic we have these equivalent formulas:

$$\begin{aligned} & (\exists \bar{z}. F(res, \bar{x}, \bar{z})) \rightarrow Post(res, \bar{x}) \\ & \neg(\exists \bar{z}. F(res, \bar{x}, \bar{z})) \vee Post(res, \bar{x}) \\ & (\forall \bar{z}. \neg F(res, \bar{x}, \bar{z})) \vee Post(res, \bar{x}) \\ & \forall \bar{z}. [\neg F(res, \bar{x}, \bar{z}) \vee Post(res, \bar{x})] \\ & \forall \bar{z}. [F(res, \bar{x}, \bar{z}) \rightarrow Post(res, \bar{x})] \end{aligned}$$

Checking validity is same as showing that

$$F(res, \bar{x}, \bar{z}) \rightarrow Post(res, \bar{x})$$

is true for all values of variables, or that

$$F(res, \bar{x}, \bar{z}) \wedge \neg Post(res, \bar{x})$$

has no satisfying assignments.

# Adding State and Non-Determinism



# VC Generation for Imperative Non-Deterministic Programs

Program can be represented by a formula relating initial and final state. Consider program with variables  $x, y, z$

program:  $x = x + 2; y = x + 10$

relation:  $\{(x, y, z, x', y', z') \mid x' = x + 2 \wedge y' = x + 12 \wedge z' = z\}$

formula:  $x' = x + 2 \wedge y' = x + 12 \wedge z' = z$

Specification:  $z = \text{old}(z) \wedge (\text{old}(x) > 0 \rightarrow (x > 0 \wedge y > 0))$

Adhering to specification is relation subset:

$$\begin{aligned} & \{(x, y, z, x', y', z') \mid x' = x + 2 \wedge y' = x + 12 \wedge z' = z\} \\ \subseteq & \{(x, y, z, x', y', z') \mid z' = z \wedge (x > 0 \rightarrow (x' > 0 \wedge y' > 0))\} \end{aligned}$$

or validity of the following implication:

$$\begin{aligned} & x' = x + 2 \wedge y' = x + 12 \wedge z' = z \\ \rightarrow & z' = z \wedge (x > 0 \rightarrow (x' > 0 \wedge y' > 0)) \end{aligned}$$

# Imperative Presburger Arithmetic Programs

$F$  - formulas,  $t$  - terms - as in functional programs so far

Fixed number of mutable integer variables  $V = \{x_1, \dots, x_n\}$

Imperative statements:

- ▶  **$x = t$** : change  $x \in V$  to have value given by  $t$ ; leave vars in  $V \setminus \{x\}$  unchanged
- ▶ **if( $F$ ) $c_1$  else  $c_2$** : if  $F$  holds, execute  $c_1$  else execute  $c_2$
- ▶  **$c_1; c_2$** : first execute  $c_1$ , then execute  $c_2$

Statements for introducing and restricting non-determinism:

- ▶ **havoc( $x$ )**: non-deterministically change  $x \in V$  to have an arbitrary value; leave vars in  $V \setminus \{x\}$  unchanged
- ▶ **if( $*$ )  $c_1$  else  $c_2$** : arbitrarily choose to run  $c_1$  or  $c_2$
- ▶ **assume( $F$ )**: block all executions where  $F$  does not hold

Given such loop-free program  $c$  with conditionals, compute a polynomial-sized formula  $R(c)$  of form:  $\exists \bar{z}. F(\bar{x}, \bar{z}, \bar{x}')$  describing relation between initial values of variables  $x_1, \dots, x_n$  and final values of variables  $x'_1, \dots, x'_n$

# Construction Formula that Describe Relations

$c$  - imperative command

$R(c)$  - formula describing relation between initial and final states of execution of  $c$

If  $\rho(c)$  describes the relation, then  $R(c)$  is formula such that

$$\rho(c) = \{(\bar{v}, \bar{v}') \mid R(c)\}$$

$R(c)$  is a formula between unprimed variables  $\bar{v}$  and primed variables  $\bar{v}'$

## Formula for Assignment

$$x = t$$

$R(x = t)$ :

$$x' = t \wedge \bigwedge_{v \in V \setminus \{x\}} v' = v$$

Note that the formula must explicitly state which variables remain the same (here: all except  $x$ ). Otherwise, those variables would not be constrained by the relation, so they could take arbitrary value in the state after the command.

## Formula for if-else

After flattening,

*if*(*b*) *c*<sub>1</sub> *else* *c*<sub>2</sub>

*R*(*if*(*b*) *c*<sub>1</sub> *else* *c*<sub>2</sub>):

$$(b \wedge R(c_1)) \vee (\neg b \wedge R(c_2))$$

## Command semicolon

$c_1; c_2$

Reminder about relation composition and its definition:

$$r_1 \circ r_2 = \{(a, c) \mid \exists b. (a, b) \in r_1 \wedge (b, c) \in r_2\}$$

What are  $R(c_1)$  and  $R(c_2)$  and in terms of which variables they are expressed?

$R(c_1; c_2) \equiv$

$$\exists \bar{z}. R(c_1)[\bar{x}' := \bar{z}] \wedge R(c_2)[\bar{x} := \bar{z}]$$

where  $\bar{z}$  are freshly picked names of intermediate states.

- ▶ a useful convention:  $\bar{z}$  refer to position in program source code

# havoc

## Definition of HAVOC

1. wide and general destruction: devastation
2. great confusion and disorder

Example of use:

```
y = 12; havoc(x); assume(x + x = y)
```

Translation,  $R(\text{havoc}(x))$ :

$$\bigwedge_{v \in V \setminus \{x\}} v' = v$$

This again illustrates “politically correct” approach to describing the destruction of values of variables: just do not mention them.

## Non-deterministic choice

*if*(\*)  $c_1$  *else*  $c_2$

$R(\textit{if}(*))$   $c_1$  *else*  $c_2$ ):

$R(c_1) \vee R(c_2)$

- ▶ translation is simply a disjunction – this is why construct is interesting
- ▶ corresponds to branching in control-flow graphs



## assume

*assume*( $F$ )

$R(\text{assume}(F))$ :

$$F \wedge \bigwedge_{v \in V} v' = v$$

- ▶ This command does not change any state.
- ▶ If  $F$  does not hold, it stops with “instantaneous success”.

## Example of Translation

0  
(if (b) x = x + 1 else y = x + 2);  
1  
x = x + 5;  
2  
(if (\*) y = y + 1 else x = y)  
3

becomes

$$\begin{aligned} \exists x_1, y_1, x_2, y_2. & ((b \wedge \mathbf{x}_1 = \mathbf{x} + \mathbf{1} \wedge y_1 = y) \vee (\neg b \wedge x_1 = x \wedge \mathbf{y}_1 = \mathbf{x} + \mathbf{2})) \\ & \wedge (\mathbf{x}_2 = \mathbf{x}_1 + \mathbf{5} \wedge y_2 = y_1) \\ & \wedge ((x' = x_2 \wedge \mathbf{y}' = \mathbf{y}_2 + \mathbf{1}) \vee (x' = \mathbf{y}_2 \wedge y' = y_2)) \end{aligned}$$

Think of execution trace  $(x_0, y_0), (x_1, y_1), (x_2, y_2), (x_3, y_3)$  where

- ▶  $(x_0, y_0)$  is denoted by  $(x, y)$
- ▶  $(x_3, y_3)$  is denoted by  $(x', y')$

# Imperative Presburger Arithmetic Programs

$F$  - formulas,  $t$  - terms - as in functional programs so far

Fixed number of mutable integer variables  $V = \{x_1, \dots, x_n\}$

Imperative statements:

- ▶  **$x = t$** : change  $x \in V$  to have value given by  $t$ ; leave vars in  $V \setminus \{x\}$  unchanged
- ▶ **if( $F$ ) $c_1$  else  $c_2$** : if  $F$  holds, execute  $c_1$  else execute  $c_2$
- ▶  **$c_1; c_2$** : first execute  $c_1$ , then execute  $c_2$

Statements for introducing and restricting non-determinism:

- ▶ **havoc( $x$ )**: non-deterministically change  $x \in V$  to have an arbitrary value; leave vars in  $V \setminus \{x\}$  unchanged
- ▶ **if( $*$ )  $c_1$  else  $c_2$** : arbitrarily choose to run  $c_1$  or  $c_2$
- ▶ **assume( $F$ )**: block all executions where  $F$  does not hold

Given such loop-free program  $c$  with conditionals, compute a polynomial-sized formula  $R(c)$  of form:  $\exists \bar{z}. F(\bar{x}, \bar{z}, \bar{x}')$  describing relation between initial values of variables  $x_1, \dots, x_n$  and final values of variables  $x'_1, \dots, x'_n$

# Justifying the name for `assume(F)`

Compute and simplify as much as possible each of the following expressions:

1.  $R(\text{assume}(F); c) = F \wedge R(c)$

2.  $R(c; \text{assume}(F)) = R(c) \wedge F[\bar{x} := \bar{x}']$

where  $F[\bar{x} := \bar{x}']$  denotes  $F$  with all variables replaced with primed versions

# Expressing **if** through non-deterministic choice and assume

```
if (b) c1 else c2
```

|||

```
if (*) {  
  assume(b);  
  c1  
} else {  
  assume(!b);  
  c2  
}
```

Indeed, apply translation to both sides and observe that generated formulas are equivalent.

# Expressing assignment through havoc and assume

$x = e$

|||

havoc(x);  
**assume**(x == e)

Under what conditions this holds?

$x \notin FV(e)$

Illustration of the problem: *havoc*(x); *assume*(x == x + 1)

Luckily, we can rewrite it into  $x_{fresh} = x + 1; x = x_{fresh}$