Contont	0.
Content	NICTA
	Rough timeline
→ Intro & motivation, getting started	[1]
→ Foundations & Principles	
 Lambda Calculus, natural deduction 	[2,3,4 ^a]
Higher Order Logic	[5,6 ^b ,7]
Term rewriting	[8,9,10 ^c]
➔ Proof & Specification Techniques	
• Isar	[11,12 ^d]
 Inductively defined sets, rule induction 	[13 ^e ,15]
 Datatypes, recursion, induction 	[16,17 ^f ,18,19]
 Calculational reasoning, mathematics style proofs 	[20]
Hoare logic, proofs about programs	[21 ^g ,22,23]

^aa1 out; ^ba1 due; ^ca2 out; ^da2 due; ^esession break; ^fa3 out; ^ga3 due

Slide 3



Automation?

Last time: Hoare rule application is nicer than using operational semantic.

BUT:

- → it's still kind of tedious
- → it seems boring & mechanical

Automation?



1

COMP 4161 NICTA Advanced Course **NICTA**

NICTA

Advanced Topics in Software Verification

Gerwin Klein, June Andronick, Toby Murray

 $\{\mathsf{P}\}\ldots\{\mathsf{Q}\}$

Slide 1

Last Time

- → Syntax of a simple imperative language
- → Operational semantics
- → Program proof on operational semantics
- ➔ Hoare logic rules
- → Soundness of Hoare logic

Slide 4

Invariant

NICTA

NICTA

Problem: While – need creativity to find right (invariant) P

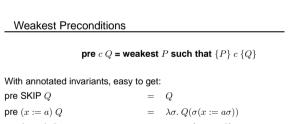
Solution:

- → annotate program with invariants
- → then, Hoare rules can be applied automatically

Example:

 $\{M = 0 \land N = 0\}$ WHILE $M \neq a$ INV $\{N = M * b\}$ DO N := N + b; M := M + 1 OD $\{N = a * b\}$

Slide 5





 $\mathsf{pre}\;(\mathsf{WHILE}\;b\;\mathsf{INV}\;I\;\mathsf{DO}\;c\;\mathsf{OD})\;Q\quad=\quad I$

Verification Conditions



{pre c Q} $c \{Q\}$ only true under certain conditions

These are called **verification conditions** vc c Q:

vc SKIP Q	=	True
$\mathrm{VC}\;(x:=a)\;Q$	=	True
$\operatorname{vc}\left(c_{1};c_{2} ight)Q$	=	$vc \ c_2 \ Q \land (vc \ c_1 \ (pre \ c_2 \ Q))$
$vc \; (IF \; b \; THEN \; c_1 \; ELSE \; c_2) \; Q$	=	$vc \; c_1 \; Q \wedge vc \; c_2 \; Q$
vc (WHILE $b \; {\rm INV} \; I \; {\rm DO} \; c \; {\rm OD}) \; Q$	=	$(\forall \sigma. \ I\sigma \wedge b\sigma \longrightarrow pre \ c \ I \ \sigma) \wedge$
		$(\forall \sigma. \ I\sigma \land \neg b\sigma \longrightarrow Q \ \sigma) \land$
		$VC \ c \ I$

 $\mathsf{vc}\; c\; Q \wedge (P \Longrightarrow \mathsf{pre}\; c\; Q) \Longrightarrow \{P\}\; c\; \{Q\}$

Slide 7

Syntax Tricks



- → $x := \lambda \sigma$. 1 instead of x := 1 sucks
- → $\{\lambda\sigma. \sigma x = n\}$ instead of $\{x = n\}$ sucks as well

Problem: program variables are functions, not values

Solution: distinguish program variables syntactically

Choices:

- → declare program variables with each Hoare triple
 - nice, usual syntax
 - · works well if you state full program and only use vcg
- → separate program variables from Hoare triple (use extensible records), indicate usage as function syntactically
 - more syntactic overhead
 - program pieces compose nicely

Slide 6

Slide 8

Records in Isabelle

NICTA

Records are a tuples with named components

Example:

record A = a :: nat b :: int

- → Selectors: a :: A \Rightarrow nat, b :: A \Rightarrow int, a r = Suc 0
- → Constructors: (| a = Suc 0, b = -1 |)
- → Update: r(| a := Suc 0 |)

Records are extensible:

record B = A + c :: nat list

$$(a = Suc 0, b = -1, c = [0, 0])$$

Slide 9

Arrays				
	Arrays			

Depending on language, model arrays as functions:

- → Array access = function application:
 - a[i] = ai
- → Array update = function update: a[i] :== v = a :== a(i:= v)

Use lists to express length:

- → Array access = nth:
- a[i] = a!i
- → Array update = list update: a[i] :== v = a :== a[i:= v]
- → Array length = list length:

a.length = length a

Slide 10

Pointers



NICTA

Choice	1
--------	---

- datatype ref = Ref int | Null heap = int \Rightarrow val types datatype val = Int int | Bool bool | Struct_x int int bool | ... → hp :: heap, p :: ref → Pointer access: *p = the_Int (hp (the_addr p))
- → Pointer update: *p :== v = hp :== hp ((the_addr p) := v)
- → a bit klunky
- → gets even worse with structs
- → lots of value extraction (the_Int) in spec and program

Slide 11

Pointers



struct with next pointer and element

datatype	ref	= Ref int Null
----------	-----	------------------

types	next_hp	= int \Rightarrow ref
types	next_np	

elem_hp = int \Rightarrow int types

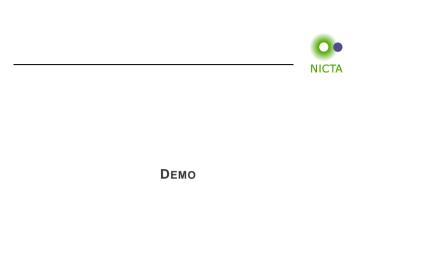
- → next :: next_hp, elem :: elem_hp, p :: ref
- → Pointer access: $p \rightarrow next$ = next (the_addr p)
- → Pointer update: p→next :== v = next :== next ((the_addr p) := v)
- → a separate heap for each struct field
- → buys you p→next \neq p→elem automatically (aliasing)
- → still assumes type safe language

Slide 12



NICTA





Slide 13

We have seen today ... NICTA

- ➔ Weakest precondition
- ➔ Verification conditions
- → Example program proofs
- → Arrays, pointers



Slide 14