COMP 4161
NICTA Advanced Course

Advanced Topics in Software Verification

Toby Murray, June Andronick, Gerwin Klein

$\gg=$

Slide 1

Last Time

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

Slide 2

Content

[a] a1 due; [b] a2 due; [c] a3 due

Slide 3

Deep Embeddings

For the IMP language, we used a **datatype** *com* to represent its **syntax**.

- We then defined its semantics over this datatype.

This is called a **deep embedding**: separate representation of language terms and their semantics.

Advantages:

- Can prove general theorems about the **language**, not just of programs.
- e.g. expresiveness, correct compilation, completeness of inference system ...
- usually by structural induction over the syntax type.

Disadvantages:

- Semantically equivalent programs are not obviously equal.
- e.g. "IF True THEN SKIP ELSE SKIP = SKIP" is not a true theorem.
- Many concepts that we already have in the logic are reinvented in the language.

Slide 4

Shallow Embeddings

NICTA

Shallow Embedding: represent only the semantics, directly in the logic.

- ➜ Write a definition for each language construct, which gives its **semantics**.
- ➜ Programs are represented as instances of these definitions.

Example: model the semantics of programs as functions of type *state* ⇒ *state*

SKIP ≡ λs. s

IF b THEN c ELSE d ≡ λs. if b s then c s else d s

- ➜ "IF True THEN SKIP ELSE SKIP = SKIP" is now a true statement.
- ➜ can use the simplifier to do semantics-preserving program rewriting.

Today we learn about a formalism suitable for shallowly embedding C semantics.

Slide 5

Records in Isabelle

NICTA

Records are a tuples with named components

Example:

record A = a :: nat
b :: int

- ➜ Selectors: a :: A ⇒ nat, b :: A ⇒ int, a $r = \mathrm{Suc}\ 0$
- ➜ Constructors: $(\!|\ \mathrm{a} = \mathrm{Suc}\ 0,\ \mathrm{b} = -1\ |\!)$
- ➜ Update: $r(\!|\ \mathrm{a} := \mathrm{Suc}\ 0\ |\!)$, b_update $(\lambda b.\ b + 1)\ r$

Records are extensible:

record B = A +
c :: nat list

$(\!|\ \mathrm{a} = \mathrm{Suc}\ 0,\ \mathrm{b} = -1,\ \mathrm{c} = [0, 0]\ |\!)$

Slide 6

NICTA

DEMO

Slide 7

Nondeterministic State Monad with Failure

NICTA

Shallow embedding suitable to represent (a useful fragment of) C programs.

Able to express lots of C ideas:

- ➜ Access to `volatile` variables, external APIs: **Nondeterminism**
- ➜ Undefined behaviour: **Failure**
- ➜ Early exit (`return`, `break`, `continue`): **Exceptional control flow**

Relatively straightforward Hoare logic

Used extensively in the seL4 verification work:

- ➜ Formalism for the seL4 abstract, design and *capDL* specifications
- ➜ Refinement calculus for proving **refinment** between them and down to code.

AutoCorres: verified translation of C to monadic representation

- ➜ Specifically designed for humans to do proofs over.

Slide 8

State Monad: Motivation

Model the **semantics** of a (deterministic) computation as a function of type

$$'s \Rightarrow ('a \times 's)$$

The computation operates over a **state** of type *'s*:

- ➜ Includes all global variables, external devices, etc.

The computation also yields a **return value** of type *'a*:

- ➜ e.g. a program's exit status (in POSIX, *'a* would be the type of 8-bit words)
- ➜ e.g. return-value of a C function

return – the computation that leaves the state unchanged and returns its argument:

$$\text{return } x \equiv \lambda s.\ (x,s)$$

Slide 9

State Monad: Basic Operations

get – returns the entire state without modifying it:

$$\text{get} \equiv \lambda s.\ (s,s)$$

put – updates the state with its argument and returns the unit value ():

$$\text{put } s \equiv \lambda\_.\ ((),s)$$

bind – sequences two computations; the second takes the first's return-value:

$$c \gg= d \equiv \lambda s.\ \textbf{let } (r,s') = c\ s \textbf{ in } d\ r\ s'$$

gets – returns a projection of the state; leaves the state unmodified:

$$\text{gets } f \equiv \text{get} \gg= (\lambda s.\ \text{return } (f\ s))$$

modify – applies its argument to modify the state; returns ():

$$\text{modify } f \equiv \text{get} \gg= (\lambda s.\ \text{put } (f\ s))$$

Slide 10

Monads, Laws

Formally: a monad **M** is a type constructor with two associated operations.

$$\text{return} :: \alpha \Rightarrow \mathbf{M}\ \alpha \qquad \text{bind} :: \mathbf{M}\ \alpha \Rightarrow (\alpha \Rightarrow \mathbf{M}\ \beta) \Rightarrow \mathbf{M}\ \beta$$

Infix Notation: $a \gg= b$ is infix notation for bind a b

- ➜ $\gg=$ binds to the left: $(a \gg= b \gg= c) = ((a \gg= b) \gg= c)$

Do-Notation: $a \gg= (\lambda x.\ b\ x)$ is often written as **do** $x \leftarrow a$; $b\ x$ **od**

Monad Laws:

return-absorb-left:	$(\text{return } x \gg= f)$	=	$f\ x$
return-absorb-right:	$(m \gg= \text{return})$	=	m
bind-assoc:	$((a \gg= b) \gg= c)$	=	$(a \gg= (\lambda x.\ b\ x \gg= c))$

Slide 11

State Monad: Example

A fragment of C:

```
void f(int *p) {
    int x = *p;
    if (x < 10) {
        *p = x++;
    }
}
```

record state =
  hp :: int ptr ⇒ int

f :: "int ptr ⇒ (state ⇒ (unit,state))"

f *p* ≡
  do
    x ← gets (λs. hp s p);
    if x < 10 **then**
      modify (hp_update (λh. (h(p := x + 1))))
    else
      return ()
  od

Slide 12

State Monad with Failure

Allows computations to **fail**: $'s \Rightarrow (('a \times 's) \underline{\times \text{bool}})$

bind – fails when either computation fails

bind a $b \equiv$ **let** $((r,s'),f)$ = a s; $((r'',s''),f')$ = b r s' **in** $((r'',s''), f \vee f')$

fail – the computation that always fails:

fail $\equiv \lambda$s. (undefined, True)

assert – fails when given condition is False:

assert P $\equiv$ **if** P **then** return () **else** fail

guard – fails when given condition applied to the state is False:

guard P $\equiv$ get $\gg=$ (λs. assert (P s))

Slide 13

Guards

Used to assert the absence of **undefined behaviour** in C

➜ pointer validity, absence of divide by zero, signed overflow, etc.

f p $\equiv$

do

y $\leftarrow$guard (λs. valid s p);

x $\leftarrow$ gets (λs. hp s p);

if x $<$ 10 **then**

modify (hp_update (λh. (h(p := x + 1))))

else

return ()

od

Slide 14

Nondeterministic State Monad with Failure

Allows computations to be **nondeterministic:** $'s \Rightarrow (('a \times 's)\ \underline{\text{set}} \times \text{bool})$

Nondeterminism: computations return a **set** of possible results.

➜ Allows **underspecification:** e.g. malloc, external devices, etc.

bind – runs the second computation for all results returned by the first:

bind a $b \equiv$ λs. $(\{(r'',s'').\ \exists(r',s') \in \text{fst}\ (a\ s).\ (r'',s'') \in \text{fst}\ (b\ r'\ s')\},$

$\text{snd}\ (a\ s) \vee (\exists(r', s') \in \text{fst}\ (a\ s).\ \text{snd}\ (b\ r'\ s')))$

All non-failing computations so far are **deterministic**:

➜ e.g. return $x \equiv \lambda$s. $(\{(x,s)\},\text{False})$

➜ Others are similar.

select – nondeterministic selection from a set

select $A \equiv \lambda$s. $((A \times \{s\}),\text{False})$

Slide 15

DEMO

Slide 16

While Loops

Monadic while loop, defined **inductively**.

$$\text{whileLoop} :: ('a \Rightarrow 's \Rightarrow \text{bool}) \Rightarrow$$
$$('a \Rightarrow ('s \Rightarrow ('a \times 's)\ \text{set} \times \text{bool})) \Rightarrow$$
$$('a \Rightarrow ('s \Rightarrow ('a \times 's)\ \text{set} \times \text{bool}))$$

whileLoop C B

- ➜ **condition** C: takes **loop parameter** and **state** as arguments, returns **bool**
- ➜ **monadic body** B: takes **loop parameter** as argument, return-value is the **updated** loop paramter
- ➜ **fails** if the loop body ever fails or if the loop never terminates

Example: whileLoop ($\lambda p\ s.$ hp s p = 0) (λp. return (ptrAdd p 1)) p

Slide 17

Defining While Loops Inductively

Two-part definition: results and termination

Results: $\text{while\_results} :: ('a \Rightarrow 's \Rightarrow \text{bool}) \Rightarrow$
$$('a \Rightarrow ('s \Rightarrow ('a \times 's)\ \text{set} \times \text{bool})) \Rightarrow$$
$$((('a \times 's)\ \text{option}) \times (('a \times 's)\ \text{option}))\ \text{set}$$

$$\frac{\neg\, C\ r\ s}{(\text{Some}\ (r,s),\ \text{Some}\ (r,s)) \in \text{while\_results}\ C\ B}\ \text{(terminate)}$$

$$\frac{C\ r\ s \quad \text{snd}\ (B\ r\ s)}{(\text{Some}\ (r,s),\ \text{None}) \in \text{while\_results}\ C\ B}\ \text{(fail)}$$

$$\frac{C\ r\ s \quad (r',s') \in \text{fst}\ (B\ r\ s) \quad (\text{Some}\ (r',\ s'),\ z) \in \text{while\_results}\ C\ B}{(\text{Some}\ (r,s),\ z) \in \text{while\_results}\ C\ B}\ \text{(loop)}$$

Slide 18

Defining While Loops Inductively

Termination: $\text{while\_terminates} :: ('a \Rightarrow 's \Rightarrow \text{bool}) \Rightarrow$
$$('a \Rightarrow ('s \Rightarrow ('a \times 's)\ \text{set} \times \text{bool})) \Rightarrow$$
$$'a \Rightarrow 's \Rightarrow \text{bool}$$

$$\frac{\neg\, C\ r\ s}{\text{while\_terminates}\ C\ B\ r\ s}\ \text{(terminate)}$$

$$\frac{C\ r\ s \quad \forall (r',s') \in \text{fst}\ (B\ r\ s).\ \text{while\_terminates}\ C\ B\ r'\ s'}{\text{while\_terminates}\ C\ B\ r\ s}\ \text{(loop)}$$

$$\text{whileLoop}\ C\ B \equiv$$
$$(\lambda r\ s.\ (\{(r',s').\ (\text{Some}\ (r,\ s),\ \text{Some}\ (r',\ s')) \in \text{while\_results}\ C\ B\},$$
$$(\text{Some}\ (r,\ s),\ \text{None}) \in \text{while\_results} \vee (\neg \text{while\_terminates}\ C\ B\ r\ s)))$$

Slide 19

Hoare Logic over Nondeterministic State Monads

Partial correctness: $\{\!|P|\!\}\ m\ \{\!|Q|\!\} \equiv \forall s.\ P\ s \longrightarrow \forall (r,s') \in \text{fst}\ (m\ s).\ Q\ r\ s'$

- ➜ Post-condition Q is a predicate of the return-value and the result state.

Weakest Precondition Rules

$\{\!|\lambda s.\ P\ x\ s|\!\}$ return x $\{\!|\lambda r\ s.\ P\ r\ s|\!\}$ $\quad$ $\{\!|\lambda s.\ P\ s\ s|\!\}$ get $\{\!|P|\!\}$ $\quad$ $\{\!|\lambda s.\ P\ ()\ x|\!\}$ put x $\{\!|P|\!\}$

$\{\!|\lambda s.\ P\ (f\ s)\ s|\!\}$ gets f $\{\!|P|\!\}$ $\quad$ $\{\!|\lambda s.\ P\ ()\ (f\ s)|\!\}$ modify f $\{\!|P|\!\}$

$\{\!|\lambda s.\ P \longrightarrow Q\ ()\ s|\!\}$ assert P $\{\!|Q|\!\}$ $\quad$ $\{\!|\lambda\_.\ \text{True}|\!\}$ fail $\{\!|Q|\!\}$

Slide 20

More Hoare Logic Rules

$$\frac{P \Longrightarrow \{\!|Q|\!\}\ f\ \{\!|S|\!\} \quad \neg\ P \Longrightarrow \{\!|R|\!\}\ g\ \{\!|S|\!\}}{\{\!|\lambda s.(P \longrightarrow Q\ s) \wedge (\neg P \longrightarrow R\ s)|\!\}\ \mathbf{if}\ P\ \mathbf{then}\ f\ \mathbf{else}\ g\ \{\!|S|\!\}}$$

$$\frac{\bigwedge x.\ \{\!|B\ x|\!\}\ g\ x\ \{\!|C|\!\} \quad \{\!|A|\!\}\ f\ \{\!|B|\!\}}{\{\!|A|\!\}\ \mathbf{do}\ x \leftarrow f;\ g\ x\ \mathbf{od}\ \{\!|C|\!\}}$$

$$\frac{\{\!|R|\!\}\ m\ \{\!|Q|\!\} \quad \bigwedge s.\ P\ s \Longrightarrow R\ s}{\{\!|P|\!\}\ m\ \{\!|Q|\!\}}$$

$$\frac{\bigwedge r.\ \{\!|\lambda s.\ I\ r\ s \wedge C\ r\ s|\!\}\ B\ \{\!|I|\!\} \quad \bigwedge r\ s.\ [\![I\ r\ s;\ \neg\ C\ r\ s]\!] \Longrightarrow Q\ r\ s}{\{\!|I\ r|\!\}\ \text{whileLoop}\ C\ B\ r\ \{\!|Q|\!\}}$$

Slide 21

DEMO

Slide 22

We have seen today

- ➜ Deep and shallow embeddings
- ➜ Isabelle records
- ➜ Nondeterministic State Monad with Failure
- ➜ Monadic Weakest Predondition Rules

Slide 23