# COMP4161
# Advanced Topics in Software Verification

$\gg=$

Gerwin Klein, Miki Tanaka, Johannes Åman Pohjola, Rob Sison

T3/2023

# Content

➜ Foundations & Principles

- Intro, Lambda calculus, natural deduction [1,2]
- Higher Order Logic, Isar (part 1) [2,3[a]]
- Term rewriting [3,4]

➜ Proof & Specification Techniques

- Inductively defined sets, rule induction [4,5]
- Datatype induction, primitive recursion [5,7]
- General recursive functions, termination proofs [7]
- Proof automation, Isar (part 2) [8[b]]
- Hoare logic, proofs about programs, invariants [8,9]
- C verification [9,10]
- Practice, questions, exam prep [10[c]]

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

 

# Deep Embeddings

We used a **datatype** *com* to represent the **syntax** of IMP.

- ➜ We then defined semantics over this datatype.

This is called a **deep embedding**:

- ➜ separate representation of language terms and their semantics.

**Advantages:**

- ➜ Prove general theorems about the **language**, not just of programs.
- ➜ e.g. expressiveness, correct compilation, inference completeness ...
- ➜ usually by induction over the syntax or semantics.

**Disadvantages:**

- ➜ Semantically equivalent programs are not obviously equal.
- ➜ e.g. "IF True THEN SKIP ELSE SKIP = SKIP" is not a true theorem.
- ➜ Many concepts already present in the logic must be reinvented.

 

## Shallow Embeddings

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

- ➜ A definition for each language construct, giving its **semantics**.
- ➜ Programs are represented as instances of these definitions.

**Example:** program semantics as functions *state* ⇒ *state*

$$\begin{aligned} \text{SKIP} &\equiv \quad \lambda \text{s. s} \\ \text{IF b THEN c ELSE d} &\equiv \quad \lambda \text{s. if b s then c s else d s} \end{aligned}$$

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

Today: a shallow embedding for (interesting parts of) C semantics

# Records in Isabelle

Records are *n*-tuples with named components

**Example:**

$$\textbf{record}\ \mathsf{A} = \quad \mathsf{a} :: \mathsf{nat} \quad \mathsf{b} :: \mathsf{int}$$

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

**Records are extensible:**

$$\textbf{record}\ \mathsf{B} = \mathsf{A} + \quad \mathsf{c} :: \mathsf{nat\ list}$$

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

# Demo

# Nondeterministic State Monad with Failure

**Shallow embedding** suitable for (a useful fragment of) C.

Can 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 microkernel verification work.

**AutoCorres**: verified translation from deeply embedded C to monadic representation

- ➜ Specifically designed for humans to do proofs over.

# State Monad: Motivation

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

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

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*:

➜ models e.g. exit status and return values

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

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

 

# State Monad: Basic Operations

**get** – returns the entire state without modifying it:

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

**put** – replaces the state and returns the unit value ():

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

**bind** – sequences two computations; 2nd takes the first's result:

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

**gets** – returns a projection of the state; leaves state unchanged:

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

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

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

## Monads, Laws

**Formally:** a monad **M** is a type constructor with two 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 $a$ $b$

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

**Monad Laws:**

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

# State Monad: Example

A fragment of C:

```
void f(int *p) {
    int x = *p;
    if (x < 10)
{
        *p = x+1;
    }
}
```

**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 ()
}

 

## State Monad with Failure

Computations can **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 $\gg=$ ($\lambda$s. assert (P s))

# 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 ($\lambda$s. valid s p);
    x $\leftarrow$ gets ($\lambda$s. hp s p);
    **if** x $<$ 10 **then**
      modify (hp_update ($\lambda$h. (h(p := x + 1))))
    **else**
      return ()
  }

 

# Nondeterministic State Monad with Failure

Computations can 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 2nd computation for all results returned by the first:

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

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

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

**select** – nondeterministic selection from a set:

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

 

# Demo

# 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 parameter
- ➜ **fails** if the loop body ever fails or if the loop never terminates

**Example:** whileLoop $(\lambda p\ s.\ \text{hp}\ s\ p = 0)$ $(\lambda \text{p}.\ \text{return}\ (\text{ptrAdd}\ p\ 1))$ $p$

 

# Defining While Loops Inductively

**Two-part definition:** results and termination

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

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

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

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

 

# Defining While Loops Inductively

**Termination:**

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

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

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

$$\begin{aligned}&\text{whileLoop}\ C\ B \equiv\\ &\quad (\lambda r\ s.\ (\{(r',s').\ (\text{Some}\ (r,\ s),\ \text{Some}\ (r',\ s')) \in \text{while\_results}\ C\ B\},\\ &\qquad (\text{Some}\ (r,\ s),\ \text{None}) \in \text{while\_results}\ \lor\\ &\qquad \neg\text{while\_terminates}\ C\ B\ r\ s))\end{aligned}$$

 

# 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 return-value and 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 $\{\!|Q|\!\}$

 

# More Hoare Logic Rules

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

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

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

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

 

# Demo

# We have seen today

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