MP4161
Advanced Topics in Software Verification

$\gg=$

Gerwin Klein, June Andronick, Miki Tanaka, Johannes Åman Pohjola

T3/2022

Content

- ➜ Foundations & Principles
  - Intro, Lamb[illegible], natural deduction [1,2]
  - Higher Orde[illegible]r (part 1) [2,3[a]]
  - Term rewriti[illegible] [3,4]
- ➜ Proof & Specification Techniques
  - Inductively defined sets, rule induction [4,5]
  - Datatype induction, primitive recursion [5,7]
  - General recursive functions, termination proofs [7[b]]
  - Proof automation, Isar (part 2) [8]
  - 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 s[illegible] over this datatype.

This is called a **deep** [illegible]**ing**:

➜ 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 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 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* $\Rightarrow$ *state*

$$\text{SKIP} \equiv \quad \lambda s.\ s$$

$$\text{IF b THEN c ELSE d} \equiv \quad \lambda s.\ \text{if b s then c s else d s}$$

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

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

Records in Isabelle

Records are n-tuples with named components

Example:

$$A = \quad a :: \text{nat}$$
$$\qquad b :: \text{int}$$

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

Records are extensible:

record $B = A +$
$\qquad c :: \text{nat list}$

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

Demo

Nondeterministic State Monad with Failure

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

Can express lots of C[illegible]

- ➜ Access to `volati`[illegible]es, 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

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

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 =
  p :: int ptr ⇒ int

t 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

Computatio[illegible]il: $'s \Rightarrow (('a \times \ 's) \underline{\times \text{ bool}})$

bind – fails when eit[illegible]utation fails

bind a $b \equiv$ **let** $((r,s'),$ [illegible] $((r'',s''),f') = \text{b } r\ s'$ **in** $((r'',s''),\ f \vee f')$

fail – the computation that always fails:

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

assert – fails when given condition is False:

$$\text{assert P} \equiv \textbf{if } \text{P } \textbf{then } \text{return () } \textbf{else } \text{fail}$$

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

$$\text{guard P} \equiv \text{get} \gg= (\lambda\text{s. assert (P s)})$$

Guards

Used to assert t[illegible]e of **undefined behaviour** in C

➜ pointer validity, a[illegible]ivide by zero, signed overflow, etc.

```
f p ≡
  do {
    y ← guard (λs. valid s p);
    x ← gets (λs. hp s p);
    if x < 10 then
      modify (hp_update (λh. (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:

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

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

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

➜ Others are similar.

select – nondeterministic selection from a set:

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

Demo

While Loops

Monadic while loop, defined **inductively**.

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

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

Defining While Loops Inductively

Two-part definition: results and termination

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

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

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

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

Defining While Loops Inductively

Termination:

while_terminates :: $\Rightarrow$ bool) $\Rightarrow$
$\Rightarrow$ ('a $\times$'s) set $\times$ bool)) $\Rightarrow$
$\Rightarrow$ bool

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

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

whileLoop C B $\equiv$

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

Hoare Logic over Nondeterministic State Monads

Partial correctness:

$\{\!|P|\!\}\ m\ \{\!|Q|\!\} \equiv \forall s.$ [illegible] $\forall (r,s') \in \text{fst}\ (m\ s).\ \ Q\ r\ s'$

➜ Post-condition Q [illegible]ate of return-value and result state.

Weakest Precondition Rules

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

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

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

More Hoare Logic Rules

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

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

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

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

Demo

We have seen today

- Deep and shallow [illegible]gs
- Isabelle records
- Nondeterministic [illegible]ad with Failure
- Monadic Weakest Precondition Rules