COMP 4161
NICTA Advanced Course

Advanced Topics in Software Verification

Gerwin Klein, June Andronick, Toby Murray
Last Time

- Weakest preconditions
- Verification conditions
- Example program proofs
- Arrays, pointers
- Hard part: finding invariants
Content

➜ Intro & motivation, getting started

➜ Foundations & Principles
  • Lambda Calculus, natural deduction [2,3,4]
  • Higher Order Logic [5,6,7]
  • Term rewriting [8,9,10]

➜ Proof & Specification Techniques
  • Isar [11,12]
  • Inductively defined sets, rule induction [13,15]
  • Datatypes, recursion, induction [16,17,18,19]
  • Calculational reasoning, mathematics style proofs [20]
  • Hoare logic, proofs about programs [21,22,23]

\[a\] a1 out; \[b\] a1 due; \[c\] a2 out; \[d\] a2 due; \[e\] session break; \[f\] a3 out; \[g\] a3 due
So far:
- have verified functional programs written in HOL
- generated ML/Haskell/OCaml code for them
- learned about verifying imperative programs with Hoare Logic

Next few lectures:
- real C programs
- real Haskell programs
Main new problems in verifying C programs:

- expressions with side effects
- more control flow (do/while, for, break, continue, return)
- local variables and blocks
- functions & procedures
- concrete C data types
- C memory model and C pointers

C is not a nice language for reasoning.

Things are going to get ugly.
Approach for verifying C programs:
Translate into existing, clean imperative language in Isabelle.

Simpl:
- generic imperative language by Norbert Schirmer, TU Munich
- state space and basic expressions/statements can be instantiated
- has operational semantics
- Hoare logic with soundness and completeness proof
- automated vcg
- available from the Archive of Formal Proofs http://afp.sf.net
type_synonym 's bexp = "'s set"

datatype ('s, 'p, 'f) com =
  Skip
  | Basic "'s => 's"
  | Spec "('s * 's) set"
  | Seq "('s ,'p, 'f) com" "('s,'p,'f) com"
  | Cond "'s bexp" "('s,'p,'f) com" "('s,'p,'f) com"
  | While "'s bexp" "('s,'p,'f) com"
  | Call 'p
  | DynCom "'s => ('s,'p,'f) com"
  | Guard 'f "'s bexp" "('s,'p,'f) com"
  | Throw
  | Catch "('s,'p,'f) com" "('s,'p,'f) com"

's = state, 'p = procedure names, 'f = faults
DEMO: SIMPL
Plan

Almost all of C can be translated into Simpl.

This is the plan for today.
Expressions with side effects

\[ a = a \times b; \quad x = f(h); \quad i = ++i - i++; \quad x = f(h) + g(x); \]

\[ \rightarrow a = a \times b \quad \text{— Fine: easy to translate into Isabelle} \]

\[ \rightarrow x = f(h) \quad \text{— Fine: may have side effects, but can be translated sanely.} \]

\[ \rightarrow i = ++i - i++ \quad \text{— Seriously? What does that even mean? Make this an error, force programmer to write instead:} \]
\[ \quad i0 = i; \ i++; \ i = i - i0; \ (or \ just \ i = 1) \]

\[ \rightarrow x = f(h) + g(x) \quad \text{— Ok if } g \text{ and } h \text{ do not have any side effects} \]
\[ \quad \Rightarrow \text{Prove all functions in expressions are side-effect free} \]

**Alternative:** explicitly model nondeterministic order of execution in expressions.
Control flow

double \{ c \} while (condition);

Already can treat normal while-loops! Automatically translate into:

c; while (condition) \{ c \}

Similarly:

for (init; condition; increment) \{ c \}

becomes

init; while (condition) \{ c; increment; \}
More control flow: break/continue

while (condition) {
    foo;
    if (Q) continue;
    bar;
    if (P) break;
}

Non-local control flow: continue goes to condition, break goes to end.

Can be modelled with exceptions:
- throw exception continue, catch at end of body.
- throw exception break, catch after loop.
Exceptions

Do not exist in C, but can be used to model C constructs.

Exceptions can be modelled with two kinds of state:

- **normal** states as before
- **abrupt** states — an exception was raised, normal commands are skipped.

**Simpl commands:**

- **throw**: switch to abrupt state
- **try** `{ c1 } catch { c2 }`:
  - if `c1` terminates abruptly, execute `c2`, otherwise execute only `c1`.

Use state to store which exception was thrown.
Break/continue

Break/continue example becomes:

```plaintext
try {
    while (condition) {
        try {
            foo;
            if (Q) { exception = 'continue'; throw; }
            bar;
            if (P) { exception = 'break'; throw; }
        } catch { if (exception == 'continue') SKIP else throw; }
    } catch { if (exception == 'break') SKIP else throw; }
} catch { if (exception == 'break') SKIP else throw; }
```

This is not C any more. But it models C behaviour!

Need to be careful that only the translation has access to exception state.
if (P) return x;
foo;
return y;

Similar non-local control flow. **Similar solution:** use throw/try/catch

```
try {
  if (P) { return_val = x; exception = 'return'; throw; }
  foo;
  return_val = x; exception = 'return'; throw;
} catch {
  SKIP
}
```
Hoare Rules for Exceptions

Need new kind of Hoare triples to model normal and abrupt state:

\[ \{P\} f \{Q\}, \{E\} \]

If \(P\) holds initially, and
- \(f\) terminates in state Normal \(s\), then \(Q\) \(s\);
- \(f\) terminates in state Abrupt \(s\), then \(E\) \(s\)

**Hoare Rules:**

\[
\begin{align*}
\{Q\} & \text{ throw } \{P\}, \{Q\} & \{P\} c_1 \{Q\}, \{R\} & \{R\} c_2 \{Q\}, \{E\} \\
\{P\} & \text{ try } c_1 \text{ catch } c_2 \{Q\}, \{E\} \\
\{P\} c_1 \{R\}, \{E\} & \{R\} c_2 \{Q\}, \{E\} \\
\{P\} c_1; c_2 \{Q\}, \{E\} \\
\end{align*}
\]

(the other rules analogous)
DEMO: CONTROL FLOW
Procedures in Simpl

Simpl com datatype

- has Call command
- but no procedure declaration
- and no local variables or parameters!

They can be simulated.
Operational Semantics of Simpl

(types s, p, f as before, Semantic.thy)

datatype xstate = Normal s | Abrupt s | Fault f | Stuck

type_synonym procs = p ⇒ com option

inductive exec :: procs ⇒ com ⇒ xstate ⇒ xstate ⇒ bool

Γ ⊢ (Skip, Normal s) ⇒ Normal s
Γ ⊢ (Throw, Normal s) ⇒ Abrupt s

⋯

[ | Γ p = Some c; Γ ⊢ (c, Normal s) ⇒ s' | ] ⟷ Γ ⊢ (Call p, Normal s) ⇒ s'
Γ p = None ⟷ Γ ⊢ (Call p, Normal s) ⇒ Stuck
Formal procedure parameters and local variables

Simpl only has one global state space.

**Basic idea:**
- separate all locals and all globals
- keep both in one state space record
- on procedure entry, set formal parameters to actual values
- on procedure exit, restore previous values of all locals

Implemented using DynCom:

```
call init body restore result =
  DynCom (λs. init; body; DynCom (λt. restore s t; result t))
```

**Example:** for procedure \( f(x) = \{ r = x + 2 \} \)

\( y = \text{CALL } f(7) \equiv \text{call } (x = 7) \ (r = x + 2) \ (λs \ t. \ s (| \text{globals} := \text{globals} \ t |)) \ (λt. \ y = r \ t) \)
Simple idea: replace/inline body. Does not work for recursion.

Instead:

- introduce assumed specifications for procedures
- outside call: no specification known, user provided
- but: can assume current specification for recursive call
- works like induction
- is proved by induction on the recursive call depth
DEMO: PROCEDURES
We have seen today ...

- C control flow
- Exceptions with Hoare logic rules
- C functions and procedures with Hoare logic rules