

COMP 4161

NICTA Advanced Course

Advanced Topics in Software Verification

Gerwin Klein, June Andronick, Toby Murray, Rafal Kolanski

C

Last Time



- → Weakest preconditions
- → Verification conditions
- → Arrays, pointers
- → Hard part: finding invariants

Content



| → Intro & motivation, getting started | [1] |
|--|-----------------------|
| → Foundations & Principles | |
| Lambda Calculus, natural deduction | [1,2] |
| Higher Order Logic | [3] |
| Term rewriting | $[4^a]$ |
| → Proof & Specification Techniques | |
| Inductively defined sets, rule induction | [5] |
| Datatypes, recursion, induction | [6, 7] |
| Automated proof and disproof | [7] |
| Hoare logic, proofs about programs, refinement | $[8^b, 9^c, 10]$ |
| Isar, locales | [11 ^d ,12] |

 $^{^{}a}$ a1 due; b a2 due; c session break; d a3 due

Program Verification



So far:

- → have verified functional programs written in HOL
- → learned about verifying imperative programs with Hoare Logic

Next few lectures:

→ real C 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



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



Commands in Simpl

```
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 \Rightarrow ('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



Almost all of C can be translated into Simpl.

This is the plan for today.



Expressions with side effects

$$a = a * b;$$
 $x = f(h);$ $i = ++i - i++;$ $x = f(h) + g(x);$

- → a = a * b Fine: easy to translate into Isabelle
- \rightarrow x = f(h) Fine: may have side effects, but can be translated sanely.
- → i = ++i i++ Seriously? What does that even mean?

 Make this an error, force programmer to write instead:

 i0 = i; i++; i = i i0; (or just i = 1)
- \Rightarrow x = f(h) + g(x) Ok if g and h do not have any side effects \Rightarrow Prove all functions in expressions are side-effect free

Alternative: explicitly model nondeterministic order of execution in expressions.

Control flow



```
do { c } while (condition);
```

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

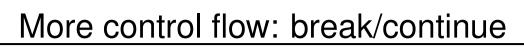
```
c; while (condition) { c }
```

Similarly:

for (init; condition; increment) { c }

becomes

init; while (condition) { c; increment; }





```
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.





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

Exceptions can be modelled with two kinds 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:

```
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; }
```

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

Need to be careful that only the translation has access to exception state.

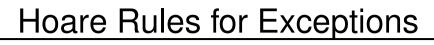
Return



```
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 = y; exception = 'return'; throw;
} catch {
    SKIP
}
```





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

$$\{P\}\ f\ \{Q\}, \{E\}$$

If P holds initially, and

- \rightarrow f terminates in state Normal s, then Q s;
- \rightarrow f terminates in state Abrupt s, then E s

Hoare Rules:

$$\frac{\{P\} \ c_1 \ \{Q\}, \{R\} \ \ \{R\} \ c_2 \ \{Q\}, \{E\}\}}{\{P\} \ \text{try } c_1 \ \text{catch } c_2 \ \{Q\}, \{E\}}$$

$$\frac{\{P\} \ c_1 \ \{R\}, \{E\} \ \ \{R\} \ c_2 \ \{Q\}, \{E\}}{\{P\} \ c_1; c_2 \ \{Q\}, \{E\}}$$

(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.





(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 \Rightarrow com \Rightarrow xstate \Rightarrow xstate \Rightarrow bool

 $\Gamma \vdash (\mathsf{Skip}, \mathsf{Normal}\ s) \Rightarrow \mathsf{Normal}\ s$

 $\Gamma \vdash (\mathsf{Throw}, \mathsf{Normal}\ s) \Rightarrow \mathsf{Abrupt}\ s$

. . .

 $[\mid \Gamma \; p = \mathsf{Some} \; c; \; \Gamma \vdash (c, \mathsf{Normal} \; s) \Rightarrow s' \mid] \Longrightarrow \Gamma \vdash (\mathsf{Call} \; p, \mathsf{Normal} \; s) \Rightarrow s' \mid \Gamma \; p = \mathsf{None} \Longrightarrow \Gamma \vdash (\mathsf{Call} \; p, \mathsf{Normal} \; s) \Rightarrow \mathsf{Stuck}$



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 (\lambdas. init; body; DynCom (\lambdat. restore s t; result t))
```

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

$$y = CALL f(7) \equiv call (x = 7) (r = x + 2) (\lambda s t. s (| globals := globals t |)) (\lambda t. y = r t)$$

Verifying Procedures



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