Last Time

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

Content

Rough timeline

- Intro & motivation, getting started
- Foundations & Principles
  - Lambda Calculus, natural deduction
  - Higher Order Logic
  - Term rewriting
- Proof & Specification Techniques
  - Isar
  - Inductively defined sets, rule induction
  - Datatypes, recursion, induction
  - Calculational reasoning, mathematics style proofs
  - Hoare logic, proofs about programs

Program Verification

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

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
Plan

Almost all of C can be translated into Simpl.

This is the plan for today.

Slide 9

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
- 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 = i)
- x = f(h) + g(x) — Ok if g and h do not have any side effects
  → Prove all functions in expressions are side-effect free

Alternative: explicitly model nondeterministic order of execution in expressions.

Slide 10

Control flow

\[
\text{do \{ c \} while (condition)};
\]

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

\[
c; \text{while (condition)} \{ c\}
\]

Similarly:

\[
\text{for} (\text{init}; \text{condition}; \text{increment}) \{ c \}
\]

becomes

\[
\text{init}; \text{while (condition)} \{ c; \text{increment}; \}
\]

Slide 11

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.

Slide 12
Exceptions

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.

Slide 13

Break/continue

Break/continue example becomes:

```c
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.

Slide 14

Return

```c
if (P) return x;
foo;
return y;
```

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

```c
try {
    if (Q) { return_val = x; exception = 'return'; throw; } foo;
    return_val = x; exception = 'return'; throw;
} catch { SKIP }
```

Slide 15

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:

\[
\{ Q \} throw \{ P \}. \{ Q \} \quad \{ P \} try c_1 catch c_2 \{ Q \}. \{ E \} \quad \{ P \} c_1 \{ R \}. \{ E \} \quad \{ R \} c_2 \{ Q \}. \{ E \} \\
\{ P \} c_1, c_2 \{ Q \}. \{ E \}
\]

(the other rules analogous)

Slide 16
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

\[ \Gamma \vdash (\text{Skip}, \text{Normal } s) ⇒ \text{Normal } s \]
\[ \Gamma \vdash (\text{Throw}, \text{Normal } s) ⇒ \text{Abrupt } s \]
\[ \ldots \]

\[ \| \Gamma p = \text{Some } c; \Gamma \vdash (c, \text{Normal } s) ⇒ s' \| \Rightarrow \Gamma \vdash (\text{Call } p, \text{Normal } s) ⇒ s' \]
\[ \Gamma p = \text{None} \Rightarrow \Gamma \vdash (\text{Call } p, \text{Normal } s) ⇒ \text{Stuck} \]

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:

\text{call init body restore result = DynCom (λs. init; body; DynCom (λt. restore s t; result t))}

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

\[ y = \text{CALL f(7)} \equiv \text{call (x = 7) (r = x + 2) (λs t. s (| globals := globals t |)) (λt. y = t)} \]
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

We have seen today ...
- C control flow
- Exceptions with Hoare logic rules
- C functions and procedures with Hoare logic rules

DEMO: PROCEDURES