COMP4161: Advanced Topics in Software Verification

Gerwin Klein, Johannes Åman Pohjola, Christine Rizkallah, Miki Tanaka
T3/2020
Last Time

- Deep and shallow embeddings
- Isabelle records
- Nondeterministic State Monad with Failure
- Monadic Weakest Precondition Rules
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, datatype induction, primitive recursion [4,5]
   • General recursive functions, termination proofs [7]b
   • Proof automation, Hoare logic, proofs about programs, invariants [8]
   • C verification [9,10]
   • Practice, questions, examp prep [10]c

a1 due; a2 due; a3 due
apply \((wp \text{ extra\_wp\_rules})\)

Tactic for automatic application of **weakest precondition rules**

- Originally developed by Thomas Sewell, NICTA/Data61, for the seL4 proofs
- Knows about a huge collection of existing wp rules for monads
- Works best when precondition is a schematic variable
- Related tool: \texttt{wpc} for Hoare reasoning over \texttt{case} statements

When used with \texttt{AutoCorres}, allows automated reasoning about C programs.

**Today we will learn about AutoCorres and C verification.**
Demo

Introduction to AutoCorres and wp
A Brief Overview of C and Simpl
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.

AutoCorres will help.
C Parser: translates C into Simpl

Simpl: deeply embedded imperative language in Isabelle.

- generic imperative language by Norbert Schirmer, TU Munich
- state space and basic expressions/statements can be instantiated
- has operational semantics
- has its own Hoare logic with soundness and completeness proof, plus automated vcg

C Parser: parses C, produces Simpl definitions in Isabelle

- written by Michael Norrish, NICTA/Data61 and ANU
- Handles a non-trivial subset of C
- Originally written to verify seL4’s C implementation
- AutoCorres is built on top of the C Parser
Commands in Simpl

data type (s, p, f) com =
  Skip
  | Basic "s \rightarrow s"
  | Spec "(s \times s) set"
  | Seq "(s, p, f) com" "(s, p, f) com"
  | Cond "s set" "(s, p, f) com" "(s, p, f) com"
  | While "s set" "(s, p, f) com"
  | Call 'p
  | DynCom "s \Rightarrow (s, p, f) com"
  | Guard 'f "s set" "(s, p, f) com"
  | Throw
  | Catch "(s, p, f) com" "(s, p, f) com"

s = state, p = procedure names, f = faults
Expressions with side effects

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

- \( a = a * b \) — 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 \( i = 1 \))
- \( x = f(h) + g(x) \) — Ok if \( g \) and \( h \) do not have any side effects
  \( \implies \) Prove all functions in expressions are side-effect free

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

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

automatically translates into:

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

Similarly:

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

becomes

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

```c
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.
Break/continue

Break/continue example becomes:

```c
try {
    while (condition) {
        try {
            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
}
AutoCorres
AutoCorres

**AutoCorres:** reduces the pain in reasoning about C code

- Written by David Greenaway, NICTA and UNSW
- Converts C/Simpl into (monadic) shallow embedding in Isabelle
- Shallow embedding easier to reason about than Simpl

**Is self-certifying:** produces Isabelle theorems proving its own correctness

For each Simpl definition $C$ and its generated shallow embedding $A$:

- AutoCorres proves an Isabelle theorem stating that $C$ **refines** $A$
- Every behaviour of $C$ has a corresponding behaviour of $A$
- Refinement guarantees that properties proved about $A$ will also hold for $C$.
- (Provided that $A$ never fails. c.f. Total Correctness)
AutoCorres Process

**L1**: initial monadic shallow embedding

**L2**: local variables introduced by λ-bindings

**HL**: heap state abstracted into a set of **typed heaps**

**WA**: machine words abstracted to idealised integers or nats

**Output**: human-readable output with **type strengthening**, polish

**On-the-fly proof:**
**Simpl** refines **L1** refines **L2** refines **HL** refines **WA** refines **Output**
Example: C99

We will use the following example program to illustrate each of the phases.

```c
unsigned some_func(unsigned *a, unsigned *b, unsigned c) {
    unsigned *p = NULL;
    if (c > 10u) {
        p = a;
    } else {
        p = b;
    }
    return *p;
}
```
Example: Simpl

some_func_body ≡
TRY
  `p := ptr_coerce (Ptr (scast 0));;
  IF 0xA < `c THEN
    `p := `a
  ELSE
    `p := `b
  FI;;
Guard C_Guard {c_guard `p}
  (creturn global_exn_var_'_update ret_unsigned_'_update
   (λs. h_val (hrs_mem (t_hrs_' (globals s))) (p_' s)));
Guard DontReach {} SKIP
CATCH SKIP END
Example: L1
(monadic shallow embedding)

\[
\begin{align*}
l1\_some\_func & \equiv L1\_seq (L1\_init \ ret\_unsigned\_\_update) \\
& (L1\_seq (L1\_modify (p\_\_update (\lambda_. \ ptr\_coerce (Ptr (scast 0))))) \\
& (L1\_seq (L1\_condition (\lambda s. 0xA < c\_\_s)) \\
& \quad (L1\_modify (\lambda s. s(p\_ := a\_\_s))) \\
& \quad (L1\_modify (\lambda s. s(p\_ := b\_\_s)))) \\
& (L1\_seq (L1\_guard (\lambda s. c\_guard (p\_\_s))) \\
& \quad (L1\_seq (L1\_modify (\lambda s. s(ret\_unsigned\_\_update := \\
& \quad \quad h\_val (hrs\_mem (t\_hrs\_\_ (globals s)) (p\_\_s))))) \\
& \quad (L1\_modify (global\_exn\_var\_\_update (\lambda_. Return))))))))
\end{align*}
\]

State type is the same as Simpl, namely a record with fields:

- **globals**: heap and type information
- **a\_\_, b\_\_, c\_\_, p\_\_** (parameters and local variables)
- **ret\_unsigned\_\_, global\_exn\_var\_\_** (return value, exception type)
Example: L2
(local variables lifted)

\[
\begin{align*}
&\text{l2\_some\_func\ a\ b\ c \equiv} \\
&\text{L2\_seq (L2\_condition (\lambda s. 0xA < c)} \\
&\quad \text{(L2\_gets (\lambda s. a) [''p''])} \\
&\quad \text{(L2\_gets (\lambda s. b) [''p''])}) \\
&\quad \text{(\lambda p. L2\_seq (L2\_guard (\lambda s. c\_guard p))} \\
&\quad \quad \text{(\lambda_. L2\_gets (\lambda s. h\_val (hrs\_mem (t\_hrs\_''s)) p) [''ret}}
\end{align*}
\]

State is a record with just the \textbf{globals} field

→ function now takes its parameters as arguments
→ local variable \textbf{p} now passed via \lambda-binding
→ \textbf{L2\_gets} annotated with local variable names
→ This ensures preservation by later AutoCorres phases
Example: HL
(heap abstracted into typed heaps)

\[
\text{hl\_some\_func } a \ b \ c \equiv \nn \text{L2\_seq} \ (\text{L2\_condition} \ (\lambda s. \ 0xA < c) \nn \quad \ (\text{L2\_gets} \ (\lambda s. a) [''p'']) \nn \quad \ (\text{L2\_gets} \ (\lambda s. b) [''p'']) ) \nn \quad \ (\lambda r. \text{L2\_seq} \ (\text{L2\_guard} \ (\lambda s. \text{is\_valid\_w32} \ s \ r)) \nn \quad \ (\lambda_. \text{L2\_gets} \ (\lambda s. \text{heap\_w32} \ s \ r) [''ret'']))
\]

State is a record with a set of \text{is\_valid} and \text{heap} fields:

- These store \text{pointer validity} and \text{heap contents} respectively, per type
- above example has only 32-bit word pointers
Heap Abstraction

C Memory Model  AutoCorres Typed Heaps

Heap Type Description stores type information for each heap location

C Memory Model: by Harvey Tuch

- Heap is a mapping from 32-bit addresses to bytes: 32 word ⇒ 8 word
Example: WA
(words abstracted to ints and nats)

\[\text{wa\_some\_func \ a \ b \ c \equiv}
\]
\[L2\_seq \ (L2\_condition \ (\lambda s. \ 10 < c)
\]
\[\ (L2\_gets \ (\lambda s. \ a) [''p''])
\]
\[\ (L2\_gets \ (\lambda s. \ b) [''p''])
\]
\[\ (\lambda r. \ L2\_seq \ (L2\_guard \ (\lambda s. \ is\_valid\_w32 \ s \ r))
\]
\[\ (\lambda _. \ L2\_gets \ (\lambda s. \ unat \ (heap\_w32 \ s \ r)) [''ret''])\]

Word abstraction: \ C \ int \rightarrow \ Isabelle \ int, \ C \ unsigned \rightarrow \ Isabelle \ nat

\rightarrow \ Guar ds \ inserted \ to \ ensure \ absence \ of \ unsigned \ underflow \ and \ overflow

\rightarrow \ Signed \ under/overflow \ already \ has \ guards \ (it \ has \ undefined \ behaviour)

In the example, the unsigned argument \ c \ is \ now \ of \ type \ nat

\rightarrow \ The \ function \ also \ returns \ a \ nat \ result
Example: Output
(type strengthening and polish)

\[
\text{some\_func'} \ a \ b \ c \equiv
\]

\[
\text{DO } \ p \leftarrow \text{oreturn} \ (\text{if } 10 < c \ \text{then} \ a \ \text{else} \ b); \\
\text{oguard} \ (\lambda s. \text{is\_valid\_w32} \ s \ p); \\
\text{ogets} \ (\lambda s. \text{unat} \ (\text{heap\_w32} \ s \ p))
\]

OD

Type Strengthening:

\[\rightarrow\] Tries to convert output to a more restricted monad

\[\rightarrow\] The above is in the \text{option} monad because it doesn’t modify the state, but might fail

\[\rightarrow\] The \text{type} of the option monad implies it cannot modify state

Polish:

\[\rightarrow\] Simplify output as much as possible

\[\rightarrow\] The \text{condition} has been rewritten to a \text{return} because the condition \(10 < c\) doesn’t depend on the state
Type Strengthening

Example:

```c
unsigned zero(void){ return 0u; }
```

<table>
<thead>
<tr>
<th>Monad</th>
<th>Type</th>
<th>Kind</th>
<th>Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>pure</td>
<td>Pure function</td>
<td>'a</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>gets</td>
<td>Read-only, non-failing</td>
<td>'s ⇒ 'a</td>
<td>λs. 0</td>
<td></td>
</tr>
<tr>
<td>option</td>
<td>Read-only function</td>
<td>'s ⇒ 'a option</td>
<td>oreturn 0</td>
<td></td>
</tr>
</tbody>
</table>

Effect information now encoded in function types

Later proofs get this information for free!

Can be controlled by the ts_force option of AutoCorres
(Reader) Option Monad

Another standard monad, familiar from e.g. Haskell

Return:
\[ \text{oreturn } x \equiv \lambda s. \text{Some } x \]

Bind:
\[ \text{obind } a \ b \equiv \lambda s. \text{case } a s \text{ of None } \Rightarrow \text{None } | \text{Some } r \Rightarrow b r s \]

→ Infix notation: \( \gg \)
→ Do notation: \( \text{DO } \ldots \text{ OD} \)

Hoare Logic:
\[ \text{ovalid } P \ f \ Q \equiv \forall s \ r. \ P s \land f s = \text{Some } r \rightarrow Q r s \]

\[ \begin{align*}
\text{ovalid } (P \ x) \ (\text{oreturn } x) \ P & \quad \text{\( \bigwedge r. \text{ovalid } (R r) (g r) Q \quad \text{ovalid } P (f \ | \gg \ g) Q \)}
\end{align*} \]
Exception Monad

Exceptions used to model early return, break and continue.

Exception Monad: ‘s ⇒ ((‘e + ‘a) × ‘s) set × bool

→ Instance of the nondeterministic state monad: return-value type is sum type ‘e + ‘a
→ Sum Type Constructors: \( \text{Inl} :: ‘e \Rightarrow ‘e + ‘a \) \( \text{Inr} :: ‘a \Rightarrow ‘e + ‘a \)
→ Convention: Inl used for exceptions, Inr used for ordinary return-values

Basic Monadic Operations

\[
\begin{align*}
\text{returnOk} \ x & \equiv \ \text{return} \ (\text{Inr} \ x) \\
\text{throwError} \ e & \equiv \ \text{return} \ (\text{Inl} \ e) \\
\text{lift} \ b & \equiv \ (\lambda x. \ \text{case} \ x \ \text{of} \ \text{Inl} \ e \ \Rightarrow \ \text{throwError} \ e \ | \ \text{Inr} \ r \ \Rightarrow \ b \ r)
\end{align*}
\]

\text{bindE: } a \ggg \text{=} E \ b \equiv \ a \ggg \ (\text{lift} \ b) \quad \text{Do notation: } \ doE \ldots \ odE
Hoare Rules for Exceptions

New kind of Hoare triples to model normal and exceptional cases:

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

\[\equiv\]

\[
\{ P \} f \{ \lambda x \ s. \ \text{case} \ x \ \text{of} \ \text{Inl} \ e \ \Rightarrow \ E \ e \ s \ | \ \text{Inr} \ r \ \Rightarrow \ Q \ r \ s \}\]

Weakest Precondition Rules:

\[
\{ P \ x \} \ \text{returnOk} \ x \ \{ P \}, \{ E \}
\]

\[
\{ E \ e \} \ \text{throwError} \ e \ \{ P \}, \{ E \}
\]

\[
\bigwedge x. \ \{ R \ x \} \ b \ x \ \{ Q \}, \{ E \} \quad \{ P \} \ a \ \{ R \}, \{ E \}
\]

\[
\{ P \} \ a \ \gg\!\gg=E \ b \ \{ Q \}, \{ E \}
\]

(other rules analogous)
Today we have seen

- The automated proof method $wp$
- The C Parser and translating C into Simpl
- AutoCorres and translating Simpl into monadic form
- The option and exception monads