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; ^b2 due; ^c3 due
apply \((wp \ extra\_wp\_rules)\)

Tactic for automatic application of \textit{weakest precondition rules}
wp

apply (wp 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: **wpc** for Hoare reasoning over case statements
apply (wp 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: wpc for Hoare reasoning over case statements

When used with AutoCorres, allows automated reasoning about C programs.
apply (wp 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: wpc for Hoare reasoning over case statements

When used with 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
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.
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: 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

```
datatype ('s, 'p, 'f) com =
  Skip
| Basic ''s ⇒ 's''
| Spec `('s * '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 ⇒ ('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

\[ a = a \times b; \quad x = f(h); \quad i = ++i - i++; \quad x = f(h) + g(x); \]
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:
  \[ i0 = i; \quad i++; \quad i = i - i0; \]
  (or just \( i = 1 \))

- \( x = f(h) + g(x) \) — Ok if \( g \) and \( h \) do not have any side effects
  \[ \Rightarrow \text{Prove all functions in expressions are side-effect free} \]
  Alternative: Explicitly model nondeterministic order of execution in expressions.
Expressions with side effects

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

- \( a = a \times b \) — 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:
  \[
  i_0 = i; \quad i++; \quad i = i - i_0;
  \]
  (or just \( i = 1 \))
- \( x = f(h) + g(x) \) — Ok if \( g \) and \( h \) do not have any side effects
  \[
  \Rightarrow \text{Prove all functions in expressions are side-effect free}
  \]
  Alternative: Explicitly model nondeterministic order of execution in expressions.
Expressions with side effects

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

\[\rightarrow a = a \times b\quad \text{— Fine: easy to translate into Isabelle}\]
\[\rightarrow x = f(h)\quad \text{— Fine: may have side effects, but can be translated sanely.}\]
\[\rightarrow i = ++i - i++\]
Expressions with side effects

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

\[ \rightarrow a = a \times b \quad \text{— Fine: easy to translate into Isabelle} \]
\[ \rightarrow x = f(h) \quad \text{— Fine: may have side effects, but can be translated sanely.} \]
\[ \rightarrow i = ++i - i++ \quad \text{— Seriously? What does that even mean?} \]
Expressions with side effects

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

- \[ a = a \times b \] — 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

\[ \Rightarrow \] Prove all functions in expressions are side-effect free

Alternative:

Explicitly model nondeterministic order of execution in expressions.
Expressions with side effects

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

- \( a = a \times b \) — 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 \( i = 1 \))
- \( x = f(h) + g(x) \)
Expressions with side effects

\[
\begin{align*}
  a &= a \times b; & x &= f(h); & i &= ++i - i++; & x &= f(h) + g(x); \\
\end{align*}
\]

→ \( a = a \times b \) — 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
  \( \Rightarrow \) Prove all functions in expressions are side-effect free
Expressions with side effects

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

- \( a = a \times b \) — 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
  \[ \Rightarrow \] Prove all functions in expressions are side-effect free

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

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

automatically translates into:

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

Similarly:

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

becomes

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

```java
while (condition) {
    foo;
    if (Q) continue;
    bar;
    if (P) break;
}
```
More control flow: break/continue

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

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

```java
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:
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.
More control flow: break/continue

\[ \text{while (condition) \{} \]
\[ \quad \text{foo;} \]
\[ \quad \text{if (Q) continue;} \]
\[ \quad \text{bar;} \]
\[ \quad \text{if (P) break;} \]
\[ \} \]

Non-local control flow: \textbf{continue} goes to condition, \textbf{break} goes to end. Can be modelled with exceptions:

\( \rightarrow \) throw exception \textit{'continue'}, catch at end of body.

\( \rightarrow \) throw exception \textit{'break'}, catch after loop.
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; }
}
```

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

Need to be careful that only the translation has access to exception state.
Break/continue example becomes:

```java
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!
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.
if (P) return x;
foo;
return y;

Similar non-local control flow.
**Return**

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

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

```c
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
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
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
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)

\[
\begin{align*}
l1_{\text{some\_func}} & \equiv L1_{\text{seq}}(L1_{\text{init}} \text{ ret\_unsigned\_}'\_update) \\
& \quad (L1_{\text{seq}}(L1_{\text{modify}}(p\_'\_update(\lambda_. \text{ ptr\_coerce}(\text{Ptr (scast 0))]))) \\
& \quad \quad (L1_{\text{seq}}(L1_{\text{condition}}(\lambda s. 0xA < c\_' s) \\
& \quad \quad \quad (L1_{\text{modify}}(\lambda s. s[p\_' := a\_' s]))) \\
& \quad \quad \quad \quad (L1_{\text{modify}}(\lambda s. s[p\_' := b\_' s]))) \\
& \quad \quad \quad \quad \quad (L1_{\text{seq}}(L1_{\text{guard}}(\lambda s. c\_guard(p\_' s)))) \\
& \quad \quad \quad \quad \quad \quad (L1_{\text{seq}}(L1_{\text{modify}}(\lambda s. s[ret\_unsigned\_]' := \\
& \quad \quad \quad \quad \quad \quad \quad \text{h\_val( hrs\_mem(t\_hrs\_' (globals s))(p\_' s))}) \\
& \quad \quad \quad \quad \quad \quad \quad \quad (L1_{\text{modify}}(\text{global\_exn\_var\_'\_update}(\lambda_. \text{ Return})))))))
\end{align*}
\]
Example: L1
(monadic shallow embedding)

l1_some_func ≡ L1_seq (L1_init ret__unsigned_’_update)
(L1_seq (L1_modify (p_’_update (λ_. ptr_coerce (Ptr (scast 0)))))
(L1_seq (L1_condition (λs. 0xA < c_’ s)
  (L1_modify (λs. s(p_’ := a_’ s))))
  (L1_modify (λs. s(p_’ := b_’ s))))
(L1_seq (L1_guard (λs. c_guard (p_’ s)))
  (L1_seq (L1_modify (λs. s(ret__unsigned_’ :=
    h_val (hrs_mem (t_hrs_’ (globals s))) (p_’ s)))))
  (L1_modify (global_exn_var_’_update (λ_. Return)))))

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)

\[
\begin{align*}
l_2\_\text{some}\_\text{func} \ a \ b \ c & \equiv \\
L2\_\text{seq} \ (L2\_\text{condition} \ (\lambda s. \ 0xA < c)) \\
& \hspace{1em} (L2\_\text{gets} \ (\lambda s. a) \ [''p'']) \\
& \hspace{1em} (L2\_\text{gets} \ (\lambda s. b) \ [''p'']) \\
& \hspace{1em} (\lambda p. \ L2\_\text{seq} \ (L2\_\text{guard} \ (\lambda s. c\_\text{guard} \ p)) \\
& \hspace{2em} (\lambda _. \ L2\_\text{gets} \ (\lambda s. h\_\text{val} \ (hrs\_\text{mem} \ (t\_hrs\_''s)) \ p) \ [''ret''])
\end{align*}
\]
Example: L2
(local variables lifted)

l2_some_func a b c ≡
L2_seq (L2_condition (λs. 0xA < c)
  (L2_gets (λs. a) [''p''])
  (L2_gets (λs. b) [''p'']))
  (λp. L2_seq (L2_guard (λs. c_guard p))
    (λ_. L2_gets (λs. h_val (hrs_mem (t_hrs_'s)) p) [''ret''])))

State is a record with just the globals field

- function now takes its parameters as arguments
- local variable p now passed via λ-binding
- L2_gets annotated with local variable names
- This ensures preservation by later AutoCorres phases
Example: HL
(heap abstracted into typed heaps)

\[
\text{hl\_some\_func}\ a\ b\ c \equiv \\
\text{L2\_seq}\ (\text{L2\_condition}\ (\lambda s.\ 0xA < c) \\
(\text{L2\_gets}\ (\lambda s.\ a)\ [\text{''p''}]) \\
(\text{L2\_gets}\ (\lambda s.\ b)\ [\text{''p''}]))) \\
(\lambda r.\ \text{L2\_seq}\ (\text{L2\_guard}\ (\lambda s.\ \text{is\_valid\\_w32}\ s\ r)) \\
(\lambda_.\ \text{L2\_gets}\ (\lambda s.\ \text{heap\\_w32}\ s\ r)\ [\text{''ret''}]))
\]
Example: HL
(heap abstracted into typed heaps)

```
hl_some_func a b c ≡
L2_seq (L2_condition (λs. 0xA < c)
           (L2_gets (λs. a) [''p''])
           (L2_gets (λs. b) [''p'']))
          (λr. L2_seq (L2_guard (λs. is_valid_w32 s r))
                       (λ_. L2_gets (λs. heap_w32 s r) [''ret''])))
```

State is a record with a set of is_valid_ and heap_ fields:
- These store pointer validity and heap contents respectively, per type
- above example has only 32-bit word pointers
Heap Abstraction

C Memory Model      AutoCorres Typed Heaps

heap values

44
w8
f2ff
w16
f300
47
w8
f301
e2
w16
f302
9d
w8
f303
a4
w16
f304
48
w8
f305
59
w16
f306
21

type tags

44
e247

misaligned

word16 heap

word8 heap

misaligned
C Memory Model: by Harvey Tuch

→ **Heap** is a mapping from 32-bit addresses to bytes: 32 word ⇒ 8 word

→ **Heap Type Description** stores type information for each heap location
Example: WA
(words abstracted to ints and nats)

\[ wa_{\text{some\_func}} \ a \ b \ c \equiv \]
\[ \text{L2\_seq} (\text{L2\_condition} (\lambda s. 10 < c)) \]
\[ \quad (\text{L2\_gets} (\lambda s. a) [''p'']) \]
\[ \quad (\text{L2\_gets} (\lambda s. b) [''p''])] \]
\[ (\lambda r. \text{L2\_seq} (\text{L2\_guard} (\lambda s. \text{is\_valid\_w32} s r))) \]
\[ (\lambda _. \text{L2\_gets} (\lambda s. \text{unat} (\text{heap\_w32} s r)) [''\text{ret}''])] \]
Example: WA
(words abstracted to ints and nats)

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

Word abstraction: C int → Isabelle int, C unsigned → Isabelle nat

- Guards inserted to ensure absence of unsigned underflow and overflow
- Signed under/overflow already has guards (it has undefined behaviour)
Example: WA
(words abstracted to ints and nats)

wa_some_func a b c ≡
L2_seq (L2_condition (λs. 10 < c)
        (L2_gets (λs. a) [''p''])
        (L2_gets (λs. b) [''p''])))
        (λr. L2_seq (L2_guard (λs. is_valid_w32 s r))
                   (λ_. L2_gets (λs. unat (heap_w32 s r)) [''ret''])))

Word abstraction:  C int → Isabelle int, C unsigned → Isabelle nat
  → Guards inserted to ensure absence of unsigned underflow and overflow
  → Signed under/overflow already has guards (it has undefined behaviour)

In the example, the unsigned argument c is now of type nat
  → The function also returns a nat result
  → The heap is not abstracted, hence the call to unat
Example: Output
(type strengthening and polish)

\[
\text{some_func'} \ a \ b \ c \equiv \begin{align*}
\text{DO} \ p & \leftarrow \text{oreturn} \ (\text{if} \ 10 < c \ \text{then} \ a \ \text{else} \ b); \\
\text{oguard} & \ (\lambda s. \text{is_valid_w32} \ s \ p); \\
\text{ogets} & \ (\lambda s. \text{unat} \ (\text{heap_w32} \ s \ p))
\end{align*}
\text{OD}
\]
Example: Output
(type strengthening and polish)

\[ \text{some_func'}(a \ b \ c) \equiv \]
\[ \text{DO } \] \( p \leftarrow \text{oreturn} \ (\text{if } 10 < c \ \text{then} \ a \ \text{else} \ b); \)
\[ \quad \text{oguard} (\lambda s. \text{is_valid_w32} \ s \ p); \]
\[ \quad \text{ogets} (\lambda s. \text{unat} (\text{heap_w32} \ s \ p)) \]
\[ \text{OD} \]

Type Strengthening:

- Tries to convert output to a more restricted monad
- The above is in the \textbf{option} monad because it doesn’t modify the state, but might fail
- The \textbf{type} of the option monad implies it cannot modify state
Example: Output
(type strengthening and polish)

some_func’ a b c ≡
DO p ← oreturn (if 10 < c then a else b);
oguard (λs. is_valid_w32 s p);
ogets (λs. unat (heap_w32 s p))
OD

Type Strengthening:
→ Tries to convert output to a more restricted monad
→ The above is in the option monad because it doesn’t modify the state, but might fail
→ The type of the option monad implies it cannot modify state

Polish:
→ Simplify output as much as possible
→ The condition has been rewritten to a return because the condition 10 < c doesn’t depend on the state
Type Strengthening

Example:

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

Example:

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

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

**Effect information** now encoded in function **types**.
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>Read-only, non-failing</td>
<td>'a</td>
<td>0</td>
</tr>
<tr>
<td>gets</td>
<td>Read-only function</td>
<td></td>
<td>'s ⇐ 'a</td>
<td>λs. 0</td>
</tr>
<tr>
<td>option</td>
<td></td>
<td></td>
<td>'s ⇐ 'a option</td>
<td>oreturn 0</td>
</tr>
</tbody>
</table>

**Effect information** now encoded in function **types**

Later proofs get this information for free!
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>Read-only, non-failing</td>
<td>'a</td>
<td>0</td>
</tr>
<tr>
<td>gets</td>
<td>option</td>
<td>Read-only function</td>
<td>'s =&gt; 'a</td>
<td>(\lambda s \cdot 0)</td>
</tr>
<tr>
<td>option</td>
<td></td>
<td></td>
<td>'s =&gt; 'a option</td>
<td>oreturn 0</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
(Reader) Option Monad

Another standard monad, familiar from e.g. Haskell

Return:

\[ \text{oreturn } x \equiv \lambda s. \text{Some } x \]
(Reader) Option Monad

Another standard monad, familiar from e.g. Haskell

Return:

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

Bind:

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

- Infix notation:  |>>|
- Do notation:  DO ... OD
(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: \[\|\]
- Do notation: \[\text{DO } \ldots \text{ OD}\]

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

Another standard monad, familiar from e.g. Haskell

Return:

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

Bind:

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

→ Infix notation:  \( \triangleright\triangleright \)
→ Do notation:  \( \text{DO ... OD} \)

Hoare Logic:

\[ \text{ovalid } P \ f \ Q \equiv \forall s \ r. P \ s \wedge f \ s = \text{Some } r \rightarrow Q \ r \ s \]

\[ \text{ovalid } (P \ x) \ (\text{oreturn } x) \ P \]
(Reader) Option Monad

Another standard monad, familiar from e.g. Haskell

Return:

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

Bind:

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

- Infix notation: \( \triangleright \rangle \)
- Do notation: DO ... OD

Hoare Logic:

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

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

Exceptions used to model early return, break and continue.
Exception Monad

Exceptions used to model early return, break and continue.

Exception Monad: \( 's \Rightarrow (('e + 'a) \times 's) \text{ set} \times \text{ bool} \)

\( \Rightarrow \) Instance of the nondeterministic state monad: return-value type is sum type \( 'e + 'a \)

\( \Rightarrow \) Sum Type Constructors: \( \text{Inl} :: 'e \Rightarrow 'e + 'a \quad \text{Inr} :: 'a \Rightarrow 'e + 'a \)

\( \Rightarrow \) Convention: Inl used for exceptions, Inr used for ordinary return-values
Exception Monad

Exceptions used to model early return, break and continue.

Exception Monad: \( 's \Rightarrow (('e + 'a) \times 's) \set \times \text{bool} \)

\( \Rightarrow \) Instance of the nondeterministic state monad: return-value type is sum type \( 'e + 'a \)

\( \Rightarrow \) Sum Type Constructors: \( \text{Inl} :: 'e \Rightarrow 'e + 'a \quad \text{Inr} :: 'a \Rightarrow 'e + 'a \)

\( \Rightarrow \) Convention: Inl used for exceptions, Inr used for ordinary return-values

Basic Monadic Operations
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: \texttt{Inl} :: 'e ⇒ 'e + 'a \quad \texttt{Inr} :: 'a ⇒ 'e + 'a

- Convention: Inl used for exceptions, Inr used for ordinary return-values

Basic Monadic Operations

\texttt{returnOk} x ≡ \texttt{return (Inr x)}
Exception Monad

Exceptions used to model early return, break and continue.

Exception Monad: \( \text{'}s \Rightarrow ((\text{'}e + \text{'}a) \times \text{'}s) \text{ set} \times \text{bool} \)

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

Basic Monadic Operations

\[
\text{returnOk } x \equiv \text{return } (\text{Inr } x) \quad \text{throwError } e \equiv \text{return } (\text{Inl } e)
\]
Exceptions used to model early return, break and continue.

Exception Monad: \( 's \Rightarrow (('e + 'a) \times 's) \text{ set } \times \text{ bool} \)

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

Basic Monadic Operations

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

Exceptions used to model early return, break and continue.

Exception Monad: \( s \Rightarrow ((e + a) \times s) \setimes \text{ bool} \)

→ Instance of the nondeterministic state monad: return-value type is sum type \( e + a \)

→ Sum Type Constructors: \( \text{ Inl } :: e \Rightarrow e + a \quad \text{ Inr } :: a \Rightarrow e + a \)

→ Convention: Inl used for exceptions, Inr used for ordinary return-values

Basic Monadic Operations

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

\[
\text{bindE: } a \ggg E b \equiv a \ggg (\text{lift } b)
\]
Exception Monad

Exceptions used to model early return, break and continue.

Exception Monad: \( s \Rightarrow ((e + a) \times s) \) set \( \times \) bool

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

Basic Monadic Operations

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

\[
\text{bindE: } a \ggg=E b \equiv a \ggg=E (\text{lift } b)
\]

Do notation: \( \text{doE ... odE} \)
Hoare Rules for Exceptions

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

\[
\{ P \} \; f \; \{ Q \}, \; \{ E \}
\]
Hoare Rules for Exceptions

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

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

Weakest Precondition Rules:

\[ \{ P \ x \ \text{returnOk} \ x \} \equiv \{ P \} \]
\[ \{ E \ e \ \text{throwError} \ e \} \equiv \{ P \} \]
\[ \{ E \} \ \bigwedge \ x. \ \{ R \ x \ \text{b} \ x \} \equiv \{ Q \} \]
\[ \{ E \ a \ \gg \ E \ b \} \equiv \{ Q \} \]

(other rules analogous)
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:

\[
\{ P x \} \ \text{returnOk} \ x \ \{ P \}, \{ E \} \\
\{ E e \} \ \text{throwError} \ e \ \{ P \}, \{ E \} \\
\]
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:

\[
\{ P \ x \} \ \text{returnOk} \ x \ \{ P \}, \{ E \} \\
\{ E \ e \} \ \text{throwError} \ e \ \{ P \}, \{ E \} \\
\wedge x. \ \{ R \ x \} \ b \ x \ \{ Q \}, \{ E \} \\
\{ P \} \ a \ \{ R \}, \{ E \} \\
\{ P \} \ a \ >> \{ E \ b \ \{ Q \}, \{ E \} \\
\]

(other rules analogous)
Today we have seen

- The automated proof method \texttt{wp}
- The C Parser and translating C into Simpl
- AutoCorres and translating Simpl into monadic form
- The option and exception monads