



COMP4161: Advanced Topics in Software Verification

C

Gerwin Klein, June Andronick, Christine Rizkallah, Miki Tanaka  
S2/2018

[data61.csiro.au](http://data61.csiro.au)



# Last Time



# Last Time



→ Deep and shallow embeddings

# Last Time



- Deep and shallow embeddings
- Isabelle records

# Last Time



- Deep and shallow embeddings
- Isabelle records
- Nondeterministic State Monad with Failure

# Last Time



- Deep and shallow embeddings
- Isabelle records
- Nondeterministic State Monad with Failure
- Monadic Weakest Predondition Rules

# Content



- Intro & motivation, getting started [1]
  
- Foundations & Principles
  - Lambda Calculus, natural deduction [1,2]
  - Higher Order Logic [3<sup>a</sup>]
  - Term rewriting [4]
  
- Proof & Specification Techniques
  - Inductively defined sets, rule induction [5]
  - Datatypes, recursion, induction [6, 7]
  - Hoare logic, proofs about programs, invariants [8<sup>b</sup>,9]
  - (mid-semester break)
  - C verification [10]
  - CakeML, Isar [11<sup>c</sup>]
  - Concurrency [12]

---

<sup>a</sup>a1 due; <sup>b</sup>a2 due; <sup>c</sup>a3 due

# wp



**apply** (wp *extra\_wp\_rules*)

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



**apply** (`wp extra_wp_rules`)

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

- Originally developed by Thomas Sewell, NICTA, 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, 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, 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.**



DATA  
61



# Demo

Introduction to AutoCorres and wp

# A Brief Overview of C and Simpl

# C



**Main new problems in verifying C programs:**

# C



## 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



## 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.**



# C



## 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.**

# C



## 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, later.**

# 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 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 * b;    x = f(h);    i = ++i - i++;    x = f(h) + g(x);
```

# 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



# 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.

# 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++`

# 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?

# 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 = 1`)

# 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 = 1`)
- `x = f(h) + g(x)`

# 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  $i = 1$ )
- $x = f(h) + g(x)$  — Ok if  $g$  and  $h$  do not have any side effects  
⇒ Prove all functions in expressions are side-effect free

# 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  $i = 1$ )
- $x = f(h) + g(x)$  — 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.

# 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; }
```



# More control flow: break/continue



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

# 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.

# 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:

# 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.

# 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.

# 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; }
```

# 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!**

# 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.

# 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
}
```

DATA  
61



# 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

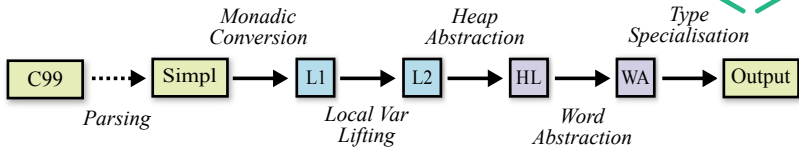
- Written by David Greenaway, NICTA and UNSW
- Converts C/Simpl into (monadic) shallow embedding in Isabelle
- Shallow embedding easier to reason about than Simpl

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

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

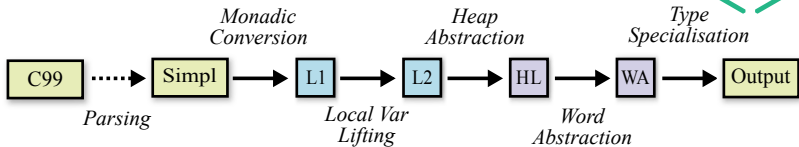
- AutoCorres proves an Isabelle theorem stating that  $C$  **refines**  $A$
- Every behaviour of  $C$  has a corresponding behaviour of  $A$
- Refinement guarantees that properties proved about  $A$  will also hold for  $C$ .
- (Provided that  $A$  never fails. c.f. Total Correctness)

# AutoCorres Process



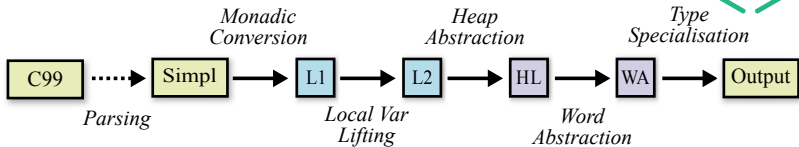


# AutoCorres Process



**L1:** initial monadic shallow embedding

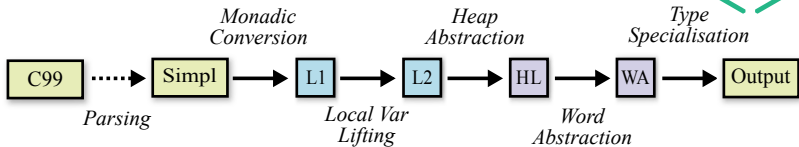
# AutoCorres Process



**L1:** initial monadic shallow embedding

**L2:** local variables introduced by  $\lambda$ -bindings

# AutoCorres Process

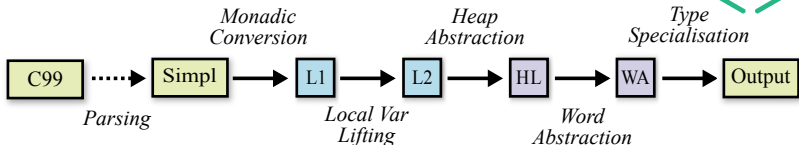


**L1:** initial monadic shallow embedding

**L2:** local variables introduced by  $\lambda$ -bindings

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

# AutoCorres Process



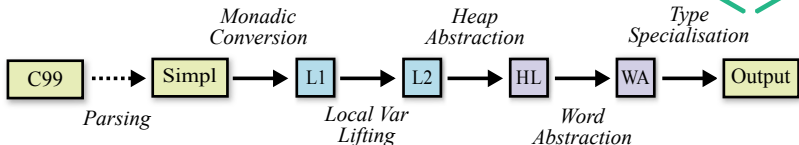
**L1:** initial monadic shallow embedding

**L2:** local variables introduced by  $\lambda$ -bindings

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

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

# AutoCorres Process



**L1:** initial monadic shallow embedding

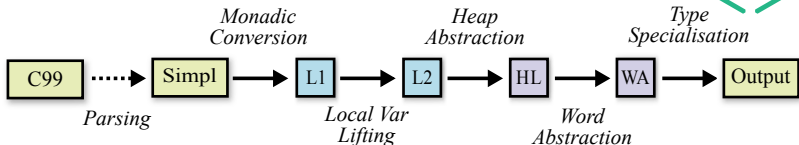
**L2:** local variables introduced by  $\lambda$ -bindings

**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

# AutoCorres Process



**L1:** initial monadic shallow embedding

**L2:** local variables introduced by  $\lambda$ -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.

```
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)



```
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)))))))))
```

# 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)



```
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'])))
```

# 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  $\lambda$ -binding
- **L2\_gets** annotated with local variable names
- This ensures preservation by later AutoCorres phases

# 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''])))
```

# Example: HL

(heap abstracted into typed heaps)



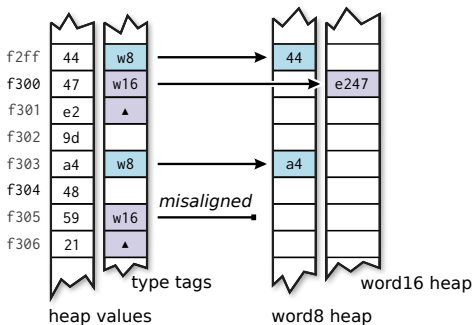
```
hl_some_func a b c ≡
L2_seq (L2_condition (λs. 0xA < c)
        (L2_gets (λs. a) ['p']))
        (L2_gets (λs. b) ['p']))
(λr. L2_seq (L2_guard (λs. is_valid_w32 s r)
                 (λ_. L2_gets (λs. heap_w32 s r) ['ret'])))
```

State is a record with a set of **is\_valid\_** and **heap\_** fields:

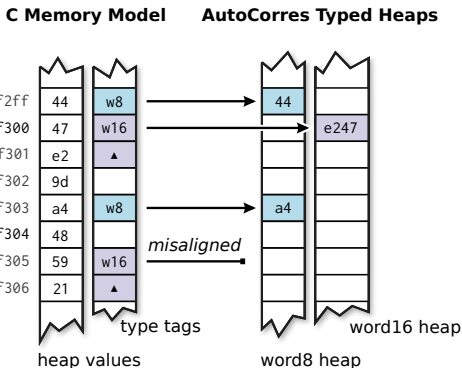
- These store **pointer validity** and **heap contents** respectively, per type
- above example has only 32-bit word pointers

# Heap Abstraction

## C Memory Model      AutoCorres Typed Heaps



# Heap Abstraction



## C Memory Model: by Harvey Tuch

- **Heap** is a mapping from 32-bit addresses to bytes: 32 word  $\Rightarrow$  8 word
- **Heap Type Description** stores type information for each heap location



# 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''])))
```

# 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, because is 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, because is 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)



```
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
```

# 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

# 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:

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



# Type Strengthening



Example:

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

Monad Type	Kind	Type	Example
pure	Pure function	'a	0
gets	Read-only, non-failing	's $\Rightarrow$ 'a	$\lambda s. 0$
option	Read-only function	's $\Rightarrow$ 'a option	oreturn 0



# Type Strengthening



Example:

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

Monad Type	Kind	Type	Example
pure	Pure function	'a	0
gets	Read-only, non-failing	's $\Rightarrow$ 'a	$\lambda s. 0$
option	Read-only function	's $\Rightarrow$ 'a option	oreturn 0

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

# Type Strengthening



Example:

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

Monad Type	Kind	Type	Example
pure	Pure function	'a	0
gets	Read-only, non-failing	's $\Rightarrow$ 'a	$\lambda s. 0$
option	Read-only function	's $\Rightarrow$ 'a option	oreturn 0

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

**Later proofs get this information for free!**

# Type Strengthening



Example:

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

Monad Type	Kind	Type	Example
pure	Pure function	'a	0
gets	Read-only, non-failing	's $\Rightarrow$ 'a	$\lambda s. 0$
option	Read-only function	's $\Rightarrow$ 'a option	oreturn 0

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

**Later proofs get this information for free!**

Can be controlled by the **ts\_force** option of AutoCorres

# Option Monad



Another standard monad, familiar from e.g. Haskell

# Option Monad



Another standard monad, familiar from e.g. Haskell

**Return:**

$\text{oreturn } x \equiv \lambda s. \text{Some } x$

# Option Monad



Another standard monad, familiar from e.g. Haskell

**Return:**

$\text{return } x \equiv \lambda s. \text{Some } x$

**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

# Option Monad



Another standard monad, familiar from e.g. Haskell

**Return:**

$\text{return } x \equiv \lambda s. \text{Some } x$

**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

**Hoare Logic:**

$\text{valid } P \ f \ Q \equiv \forall s \ r. P \ s \wedge f \ s = \text{Some } r \longrightarrow Q \ r \ s$

# 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

**Hoare Logic:**

$\text{ovoid } P \ f \ Q \equiv \forall s \ r. P \ s \wedge f \ s = \text{Some } r \longrightarrow Q \ r \ s$

$\text{ovoid } (P \ x) \ (\text{oreturn } x) \ P$



# 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

**Hoare Logic:**

$\text{ovalid } P \ f \ Q \equiv \forall s \ r. P \ s \wedge f \ s = \text{Some } r \longrightarrow Q \ r \ s$

$$\text{ovalid } (P \ x) \ (\text{oreturn } x) \ P \quad \frac{\bigwedge r. \text{ovalid } (R \ r) \ (g \ r) \ Q \quad \text{ovalid } P \ f \ R}{\text{ovalid } P \ (f \ |>> \ g) \ Q}$$

# Exception Monad



**Exceptions** used to model early return, break and continue.

# Exception Monad



**Exceptions** used to model early return, break and continue.

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

- Instance of the nondeterministic state monad: return-value type is **sum type**  $'e + 'a$
- Sum Type Constructors: **Inl** ::  $'e \Rightarrow 'e + 'a$       **Inr** ::  $'a \Rightarrow 'e + 'a$
- **Convention:** Inl used for exceptions, Inr used for ordinary return-values

# Exception Monad



**Exceptions** used to model early return, break and continue.

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

- Instance of the nondeterministic state monad: return-value type is **sum type**  $'e + 'a$
- Sum Type Constructors: **Inl** ::  $'e \Rightarrow 'e + 'a$       **Inr** ::  $'a \Rightarrow 'e + 'a$
- **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:**  $'s \Rightarrow ((\underline{'e + 'a}) \times 's) \text{ set} \times \text{bool}$

- Instance of the nondeterministic state monad: return-value type is **sum type**  $'e + 'a$
- Sum Type Constructors: **Inl** ::  $'e \Rightarrow 'e + 'a$       **Inr** ::  $'a \Rightarrow 'e + 'a$
- **Convention:** Inl used for exceptions, Inr used for ordinary return-values

## Basic Monadic Operations

$\text{returnOk } x \equiv \text{return (Inr } x)$

# Exception Monad



**Exceptions** used to model early return, break and continue.

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

- Instance of the nondeterministic state monad: return-value type is **sum type**  $'e + 'a$
- Sum Type Constructors: **Inl** ::  $'e \Rightarrow 'e + 'a$       **Inr** ::  $'a \Rightarrow 'e + 'a$
- **Convention:** Inl used for exceptions, Inr used for ordinary return-values

## Basic Monadic Operations

`returnOk x`  $\equiv$  `return (Inr x)`

`throwError e`  $\equiv$  `return (Inl e)`

# Exception Monad



**Exceptions** used to model early return, break and continue.

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

- Instance of the nondeterministic state monad: return-value type is **sum type**  $'e + 'a$
- Sum Type Constructors: **Inl** ::  $'e \Rightarrow 'e + 'a$       **Inr** ::  $'a \Rightarrow 'e + 'a$
- **Convention:** Inl used for exceptions, Inr used for ordinary return-values

## Basic Monadic Operations

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

# Exception Monad



**Exceptions** used to model early return, break and continue.

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

- Instance of the nondeterministic state monad: return-value type is **sum type**  $'e + 'a$
- Sum Type Constructors: **Inl** ::  $'e \Rightarrow 'e + 'a$       **Inr** ::  $'a \Rightarrow 'e + 'a$
- **Convention:** Inl used for exceptions, Inr used for ordinary return-values

## Basic Monadic Operations

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

**bindE:**  $a \gg=E b \equiv a \gg= (\text{lift } b)$



# Exception Monad



**Exceptions** used to model early return, break and continue.

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

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

## Basic Monadic Operations

$\text{returnOk } x \equiv \text{return (Inr } x)$        $\text{throwError } e \equiv \text{return (Inl } e)$   
 $\text{lift } b \equiv (\lambda x. \text{case } x \text{ of Inl } e \Rightarrow \text{throwError } e \mid \text{Inr } r \Rightarrow b r)$

**bindE:**  $a \gg=E b \equiv a \gg= (\text{lift } b)$       **Do notation:**  $\text{doE } \dots \text{odE}$

# 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:

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

# Hoare Rules for Exceptions



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

$$\begin{aligned} & \{P\} f \{Q\}, \{E\} \\ & \equiv \\ & \{P\} f \{\lambda x s. \text{case } x \text{ of } \text{Inl } e \Rightarrow E e s \mid \text{Inr } r \Rightarrow Q r s\} \end{aligned}$$

**Weakest Precondition Rules:**

$$\overline{\{P x\} \text{returnOk } x \{P\}, \{E\}}$$

$$\overline{\{E e\} \text{throwError } e \{P\}, \{E\}}$$

# Hoare Rules for Exceptions



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

$$\{P\} f \{Q\}, \{E\} \\ \equiv \\ \{P\} f \{\lambda x s. \text{case } x \text{ of } \text{Inl } e \Rightarrow E e s \mid \text{Inr } r \Rightarrow Q r s\}$$

## Weakest Precondition Rules:

$$\frac{}{\{P x\} \text{returnOk } x \{P\}, \{E\}} \qquad \frac{}{\{E e\} \text{throwError } e \{P\}, \{E\}}$$
$$\frac{\bigwedge x. \{R x\} b x \{Q\}, \{E\} \quad \{P\} a \{R\}, \{E\}}{\{P\} a \gg=E b \{Q\}, \{E\}}$$

# Hoare Rules for Exceptions



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

$$\begin{aligned} & \{P\} f \{Q\}, \{E\} \\ & \equiv \\ & \{P\} f \{\lambda x s. \text{case } x \text{ of } \text{Inl } e \Rightarrow E e s \mid \text{Inr } r \Rightarrow Q r s\} \end{aligned}$$

**Weakest Precondition Rules:**

$$\frac{}{\{P x\} \text{returnOk } x \{P\}, \{E\}} \qquad \frac{}{\{E e\} \text{throwError } e \{P\}, \{E\}}$$

$$\frac{\bigwedge x. \{R x\} b x \{Q\}, \{E\} \quad \{P\} a \{R\}, \{E\}}{\{P\} a \gg=E b \{Q\}, \{E\}}$$

(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