COMP 4161
NICTA Advanced Course

Advanced Topics in Software Verification

Gerwin Klein, June Andronick, Toby Murray, Rafal Kolanski

C

Slide 1

Last Time

- Verifying C by translating into Simpl
- Expressions
- C control flow
- Exceptions with Hoare logic rules
- C functions and procedures with Hoare logic rules

Slide 2

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

Slide 3

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
- **prevent undefined execution**
- **concrete C data types**
- **C memory model and C pointers**

Slide 4

Undefined Execution

In C, we're not allowed to:

- ➜ divide by zero
- ➜ shift more than <architecture defined> bits
- ➜ dereference a Null pointer
- ➜ access outside array bounds
- ➜ access unallocated memory
- ➜ free unallocated memory
- ➜ ...

Their absence should become proof obligations.

Slide 5

Simpl Guards

Syntax:

```
Guard 'f "'s bexp" "('s,'p,'f) com"
```

Semantics:

$$[\mid s \in g;\ \Gamma \vdash (c, \text{Normal } s) \Rightarrow t \mid] \Longrightarrow \Gamma \vdash (\text{Guard } f\ g\ c, \text{Normal } s) \Rightarrow t$$

$$s \notin g \Longrightarrow \Gamma \vdash (\text{Guard } f\ g\ c, \text{Normal } s) \Rightarrow \text{Fault } f$$

Hoare rules:

$$\frac{\Gamma \vdash_F \{g \wedge P\}\ c\ \{Q\}}{\Gamma \vdash_F \{g \wedge P\}\ \text{Guard } f\ g\ c\ \{Q\}} \qquad \frac{f \in F \quad \Gamma \vdash_F \{g \wedge P\}\ c\ \{Q\}}{\Gamma \vdash_F \{P\}\ \text{Guard } f\ g\ c\ \{Q\}}$$

Slide 6

Simpl Guards: Why two Hoare rules?

Why two Hoare rules?

So we can separate out verification of guards.

F controls which guards are currently assumed and which are proved.

Example:

Do automated verification of array guards separately
⇒ get to assume array guards "for free" in the rest.

Slide 7

Simpl Guards: Why two Hoare rules?

Use Guards for:
Every time an expression or statement does something potentially undefined, add a guard in the translation.

Example:

`x = a / b` ⇒ Guard DivByZero (b ≠ 0) (x :== a / b)

Slide 8

DEMO: GUARDS

Slide 9

C data types

Next problem: C data types

C has the following types:

- ➜ basic: int (long/short, signed/unsigned), char, void, float, double, long double
- ➜ enum types
- ➜ pointers: type*
- ➜ array types: type[n], type[n][m], type[]
- ➜ struct types: like records, but can use recursion for pointers
- ➜ unions: multiple interpretations of same memory content
- ➜ function pointers

Size of basic types is architecture dependent.
Encoding in memory partially compiler dependent.

Slide 10

Basic types

- ➜ float/double ⇒ IEEE floating point numbers. Isabelle formalisation available in AFP.
- ➜ void ⇒ unit type in Isabelle
- ➜ integer types ⇒ finite machine words (x mod 2^{32} etc)

Why bother with finite words? Why not nat/real?

Want to model overflow precisely.

Depending on application, could work with nat and guards instead.

Slide 11

Binary Search (`java.util.Arrays`)


```
1:     public static int binarySearch(int[] a, int key) {
2:         int low = 0;
3:         int high = a.length - 1;
4:
5:         while (low <= high) {
6:             int mid = (low + high) / 2;
7:             int midVal = a[mid];
8:
9:             if (midVal < key)
10:                low = mid + 1
11:            else if (midVal > key)
12:                high = mid - 1;
13:            else
14:                return mid; // key found
15:        }
16:        return -(low + 1);  // key not found.
17:    }
```

```
6:                 int mid = (low + high) / 2;
```

http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html

Slide 12

Machine Words

Goal: want to write things like

$$x \text{ AND } y = 0 \Longrightarrow x + y = x \text{ OR } y$$

$$(x << n) \;!!\; m = x \;!!\; (n + m)$$

$$x << 2 = 4 * x \qquad \text{ucast } (y + 0xFF21) = (x - 0b01001011)$$

$$\text{unat } x + \text{unat } y < 2^{\wedge} \text{ word\_size} \Longrightarrow \text{unat } (x + y) = \text{unat } x + \text{unat } y$$

$$x :: 32 \text{ word} \qquad y :: 8 \text{ word} \qquad z :: n \text{ word}$$

AND bitwise and, OR bitwise or, !! test bit at position n, $<<$ shift left,
"ucast" cast between word sizes, "unat" convert words to nat

Slide 13

Formalisation Idea

Goal:
Create an Isabelle type that captures machine words of length n

Problem:
The parameter n is not a type, but a value.
This is called a **dependent type**.
Isabelle does not support dependent types.

Solutions: make a type 'a word, encode length in type 'a

- ➜ either implicitly as number of elements in 'a,
- ➜ or explicitly via type class function

Slide 14

Representation (no taxation)

Now can encode length. How do we represent words?

Options:

- ➜ nat mod 2^ n
- ➜ int mod 2^ n
- ➜ bool lists of length n
- ➜ test-bit functions nat ⇒ bool

All of these are equivalent. Actual definition in Isabelle is int mod 2^ n.

All others are provided as well as simulated type defs.

Slide 15

Operators

Rest is standard (see HOL/Word/Word.thy + HOL/Word/Examples/):

- ➜ define standard arithmetic and bit-wise operators with syntax
- ➜ prove lemmas connecting to known type representations
- ➜ determine abstract structure:
  commutative ring with 1, partial order, boolean algebra for bitwise ops, etc
- ➜ prove library with characteristic properties
- ➜ provide some automation: smt connection, auto cast to nat
- ➜ . . .
- ➜ profit

Slide 16

DEMO: WORD

Slide 17

C Data types

Can now represent all C types.
(Making explicit architecture assumptions on size etc.)

- ➜ integer types (incl enum): word
- ➜ pointers: **datatype 'a ptr = 32 word**
- ➜ arrays: pointers or array types in Isabelle
- ➜ structs: records or data types
- ➜ unions: separate struct types with conversions
- ➜ function pointers: word

Missing: modelling C memory

Slide 18

C Memory Model

Heap models so far:

- ➜ addr ⇒ obj option
- ➜ separate heaps by type
- ➜ separate heaps by record field

C is more ugly:

- ➜ pointer arithmetic and casting breaks type safety
- ➜ objects could overlap
- ➜ objects can be access under different types (union)
- ➜ systems programmers might rely on data layout (device access)
- ➜ could have pointers into stack (reference to local var)

Our model solves all but the last one.

(Can also solve that one, but it gets even more ugly.)

Slide 19

C Memory Model

The Memory Model:
Heap = function "32 word ⇒ 8 word"

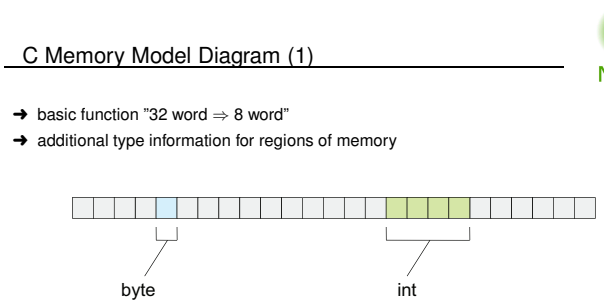
That's it.
Ok, not quite: It's the basis. We build a whole machinery on top.

Basic idea:

- ➜ 32 word ⇒ 8 word is the information that C runtime has
- ➜ we store additional type information for proofs (ghost state)
- ➜ use that type information to automatically get abstract Isabelle objects from heap
- ➜ if we stay in type-safe fragment of C, can reason like in separate heaps.

Slide 20

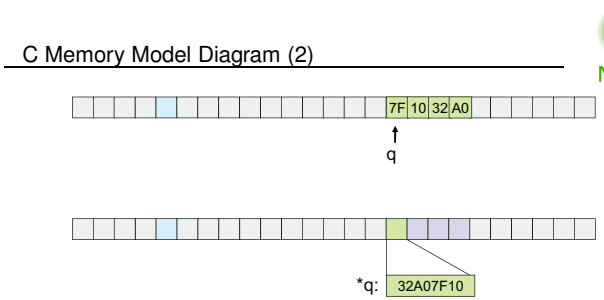
C Memory Model Diagram (1)

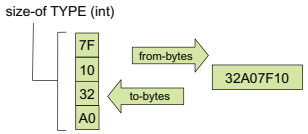


- basic function ”32 word ⇒ 8 word”
- additional type information for regions of memory

Slide 21

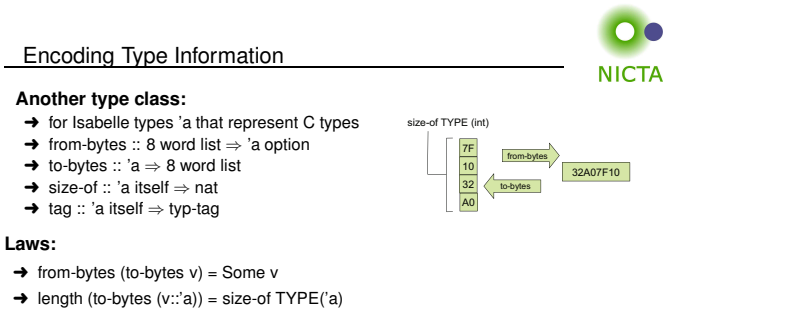
C Memory Model Diagram (2)





Slide 22

Encoding Type Information



Another type class:

- for Isabelle types ’a that represent C types
- from-bytes :: 8 word list ⇒ ’a option
- to-bytes :: ’a ⇒ 8 word list
- size-of :: ’a itself ⇒ nat
- tag :: ’a itself ⇒ typ-tag

Laws:

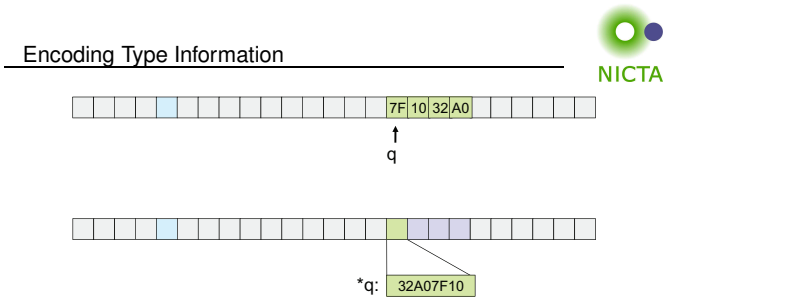
- from-bytes (to-bytes v) = Some v
- length (to-bytes (v::’a)) = size-of TYPE(’a)

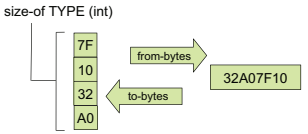
Example picture unsigned int = 32 word (depending on architecture):

- from-bytes/to-bytes = big/little endian encoding (depending on architecture)
- size-of = 4
- tag = ”32 word”

Slide 23

Encoding Type Information



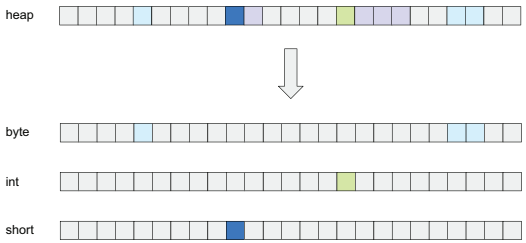


Can now define heap access/update generically for ’a!

Slide 24

C Memory Model Diagram (3)

Goal:



Slide 25

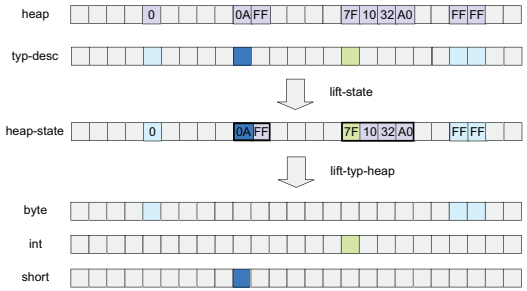
Separate Heaps

Plan:

- ➜ combine type info and real heap into one object typed-hp
- ➜ write 'view' function lift :: typed-hp ⇒ ('a ptr ⇒ 'a option)
- ➜ models type-safe heap access
- ➜ returns None if request type 'a does not match type in memory

Slide 26

C Memory Model Diagram (4)



Slide 27

Separate Heaps Properties

Lemmas about lift and heap-update:

If lift hp (p :: 'a ptr) $\neq$ None, then

- ➜ $\text{lift}_{'a}$ (heap-update p v hp) = ($\text{lift}_{'a}$ hp) (p $\mapsto$ v)
- ➜ TYPE('a) $\perp$ TYPE('b) $\Longrightarrow$ $\text{lift}_{'b}$ (heap-update p v hp) = $\text{lift}_{'b}$

where TYPE('a) $\perp$ TYPE('b) = the two types are disjoint.

This means 'lift' works like a separate heap for each type!

Slide 28

DEMO: POINTERS

Slide 29

DEMO: C PROGRAM TRANSLATION

Slide 30

We have seen today ...

- ➜ preventing undefined execution
- ➜ finite machine words
- ➜ concrete C data types
- ➜ C memory model and pointers

Slide 31