

COMP 4161

NICTA Advanced Course

Advanced Topics in Software Verification

Gerwin Klein, June Andronick, Toby Murray, Rafal Kolanski

C

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

Content

- Intro & motivation, getting started [1]

- Foundations & Principles
 - Lambda Calculus, natural deduction [1,2]
 - Higher Order Logic [3^a]
 - Term rewriting [4]

- Proof & Specification Techniques
 - Isar [5]
 - Inductively defined sets, rule induction [6^b]
 - Datatypes, recursion, induction [7^c, 8]
 - Calculational reasoning, code generation [9]
 - Hoare logic, proofs about programs [10^d,11,12]

^a a1 due; ^b a2 due; ^c session break; ^d a3 due

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**

Undefined Execution

In C, we're not allowed to:

- divide by zero
- shift more than $\langle \text{architecture defined} \rangle$ bits
- dereference a Null pointer
- access outside array bounds
- access unallocated memory
- free unallocated memory
- ...

Their absence should become proof obligations.

Simpl Guards

Syntax:

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

Semantics:

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

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

Hoare rules:

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

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.

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:

$$x = a / b \Rightarrow \text{Guard DivByZero } (b \neq 0) (x ::= a / b)$$

DEMO: GUARDS

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.

Basic types

- float/double \Rightarrow IEEE floating point numbers, no Isabelle formalisation yet.
(Any takers?)
- void \Rightarrow unit type in Isabelle
- integer types \Rightarrow finite machine words ($x \bmod 2^{32}$ 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.

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>

Machine Words

Goal: want to write things like

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

$$(x \ll n) !! m = x !! (n + m)$$

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

$$\text{unat } x + \text{unat } y < 2^{\text{word_size}} \implies \text{unat}(x + y) = \text{unat } x + \text{unat } y$$

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

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

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

Formalisation in Isabelle

Type class used in HOL/Word/Word.thy:

- 'a must be class len
- class len has function len_of :: 'a itself \Rightarrow nat
- to implement class len, a type must provide that function

'a itself:

- 'a itself is a type with one element of type 'a
- the one element is written TYPE('a)

Numeric types in Library/Numeral_Type.thy:

- create types written as numbers (type 1, 16, etc)
- have 1, 16, etc elements
- the numbers are syntax for type constructors encoding 0, 1, 2^n , 2^{n+1}

Representation (no taxation)

Now can encode length. How do we represent words?

Options:

- $\text{nat mod } 2^n$
- $\text{int mod } 2^n$
- bool lists of length n
- test-bit functions $\text{nat} \Rightarrow \text{bool}$

All of these are equivalent. Actual definition in Isabelle is $\text{int mod } 2^n$.

All others are provided as well as simulated type defs.

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

DEMO: WORD

C Data types

Can now represent all C types apart from float.

(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

C Memory Model

Heap models so far:

- addr \Rightarrow 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.)

C Memory Model

The Memory Model:

Heap = function "32 word \Rightarrow 8 word"

That's it.

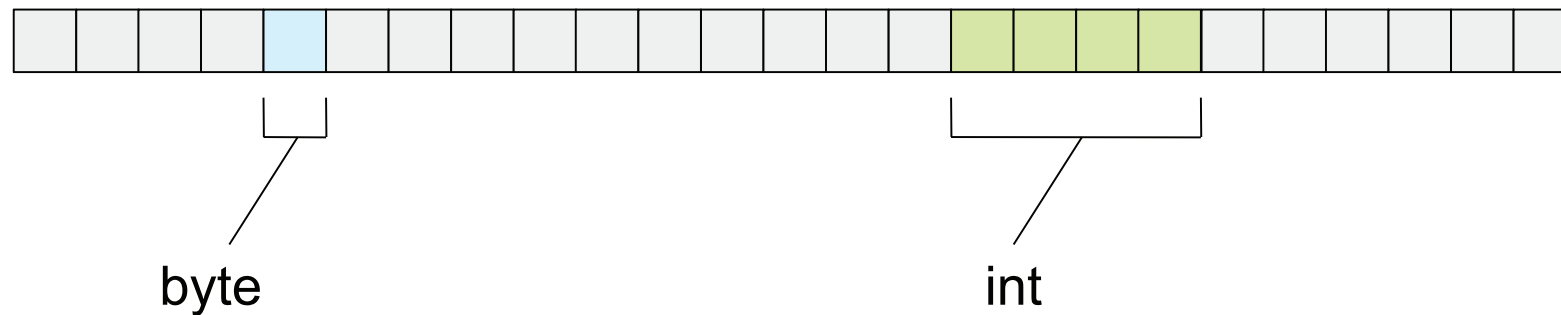
Ok, not quite: It's the basis. We build a whole machinery on top.

Basic idea:

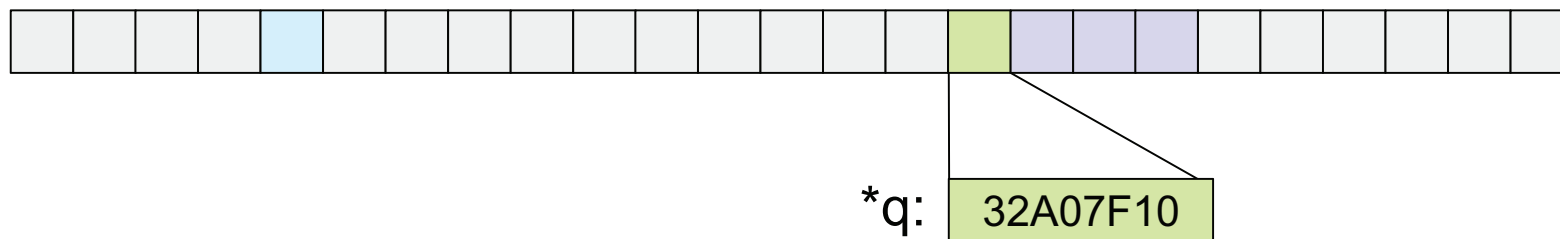
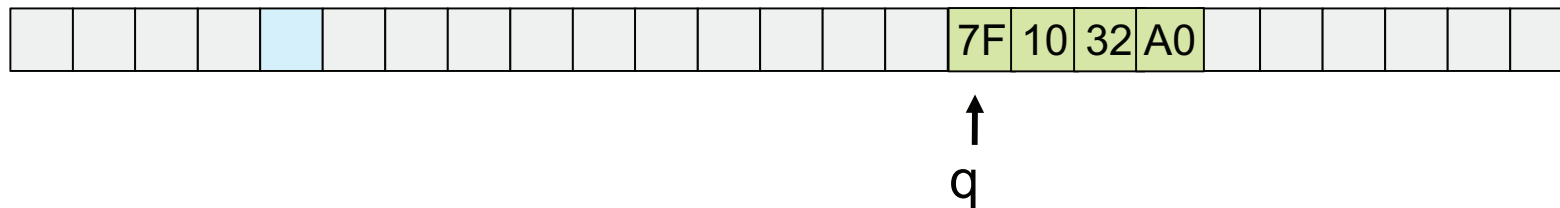
- 32 word \Rightarrow 8 word is the information that C runtime has
- we store additional type information for proofs (ghost state)
- use that type information to automatically get abstract Isabelle objects from heap
- if we stay in type-safe fragment of C, can reason like in separate heaps.

C Memory Model Diagram (1)

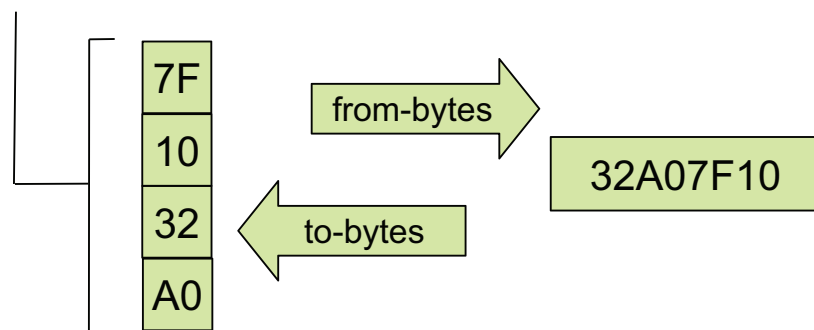
- basic function "32 word \Rightarrow 8 word"
- additional type information for regions of memory



C Memory Model Diagram (2)



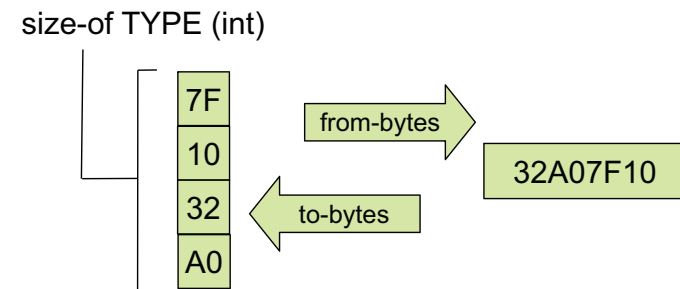
size-of TYPE (int)



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



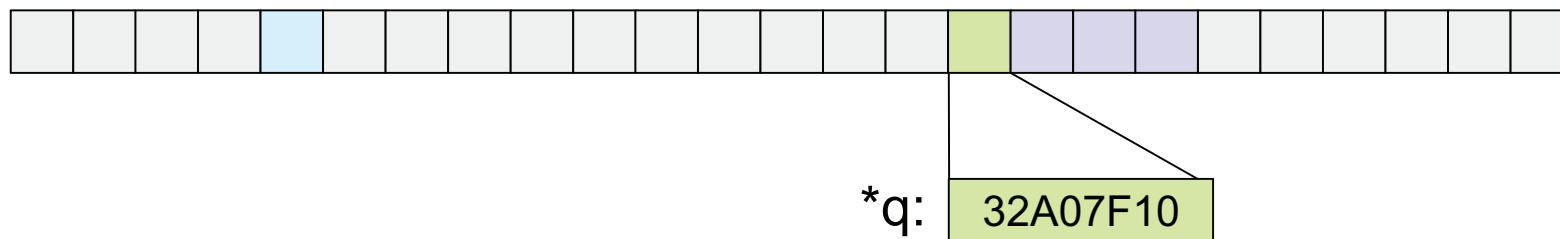
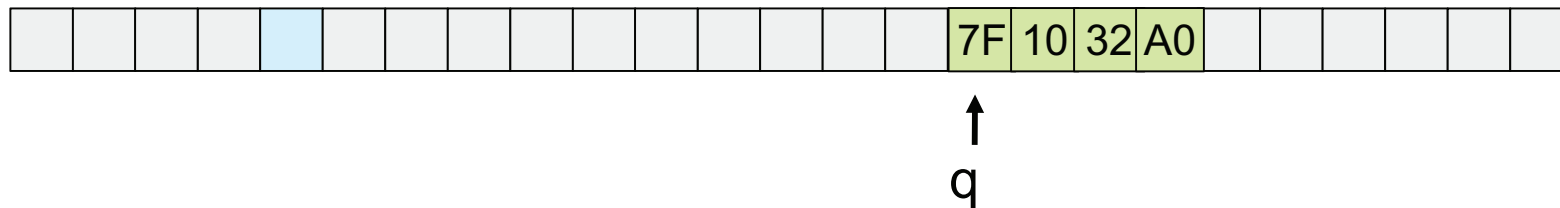
Laws:

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

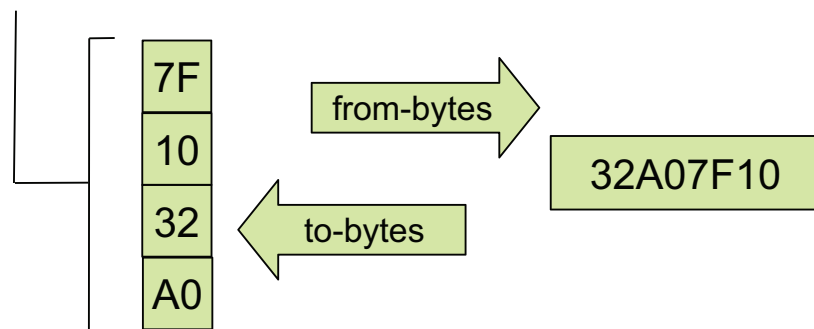
Example picture unsigned int = 32 word (depending on architecture):

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

Encoding Type Information



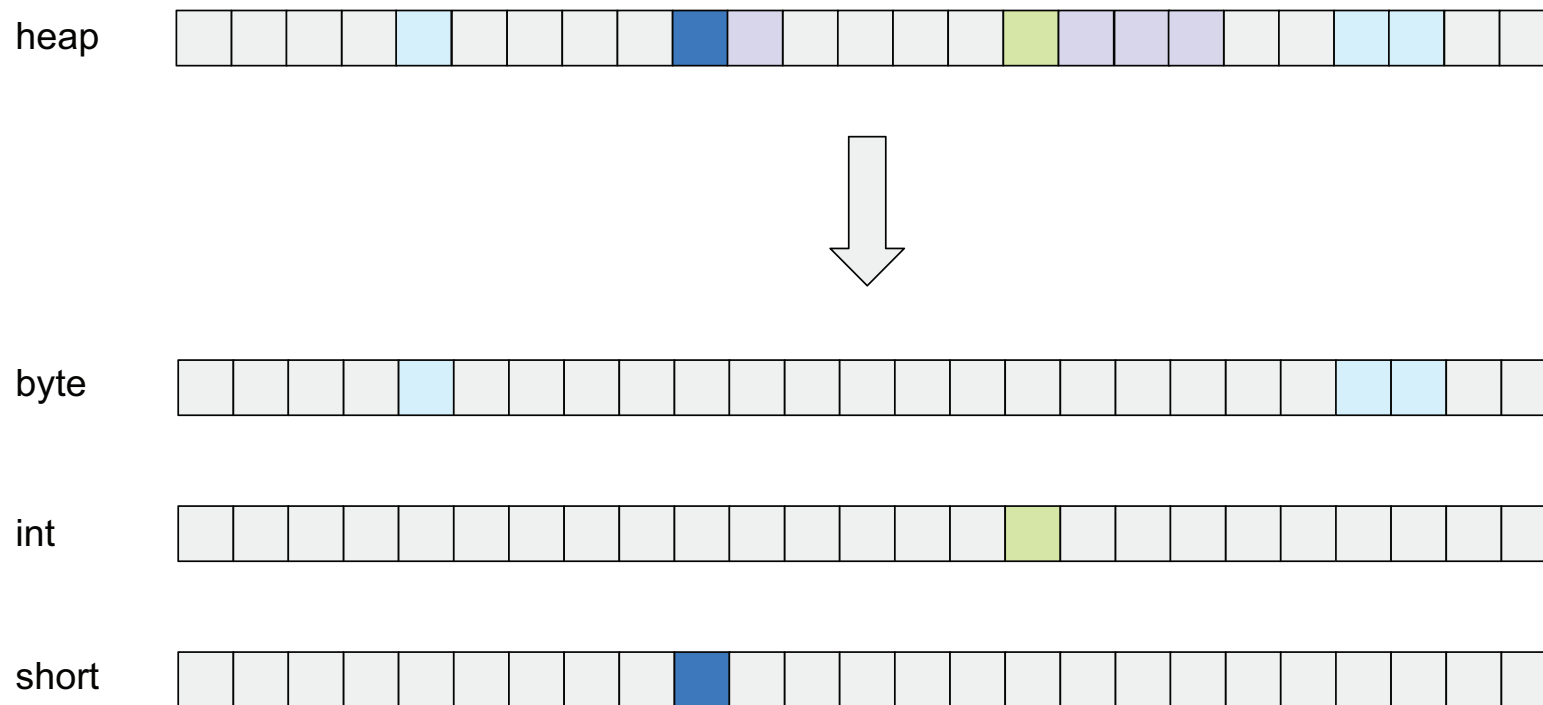
size-of TYPE (int)



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

C Memory Model Diagram (3)

Goal:

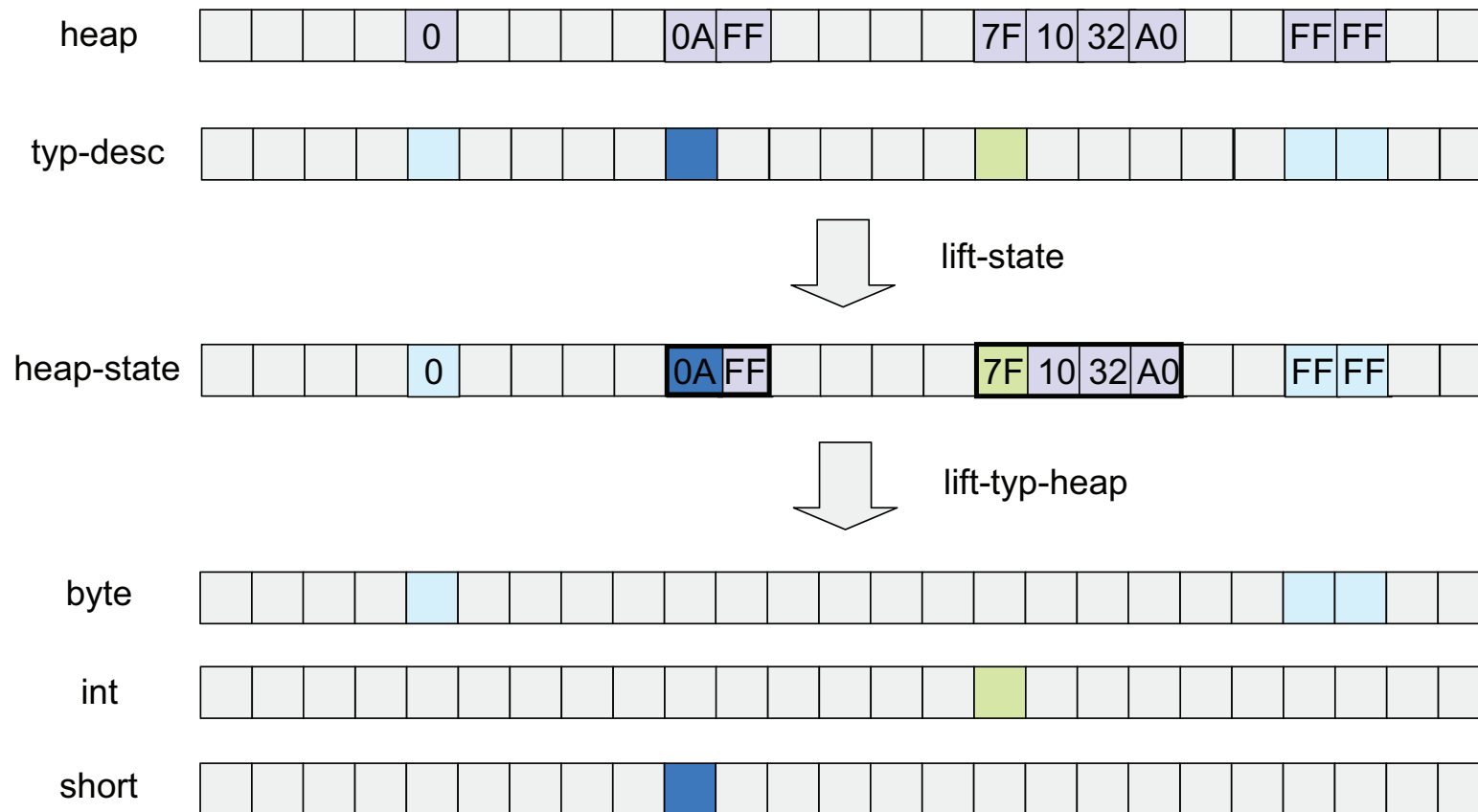


Separate Heaps

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

C Memory Model Diagram (4)



Separate Heaps Properties

Lemmas about lift and heap-update:

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

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

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

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

DEMO: POINTERS

DEMO: C PROGRAM TRANSLATION

We have seen today ...

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