CompCert guarantees for low-level C programs

Sandrine Blazy

UNIVERSITÉ DE RENNES 1

UMR IRISA

Inria informatics mathematics

joint work with Frédéric Besson and Pierre Wilke

IFIP W.G. 2.11, Bloomington, 2016-08-23

The CompCert C verified compiler

Compiler + proof that the compiler does not introduce bugs

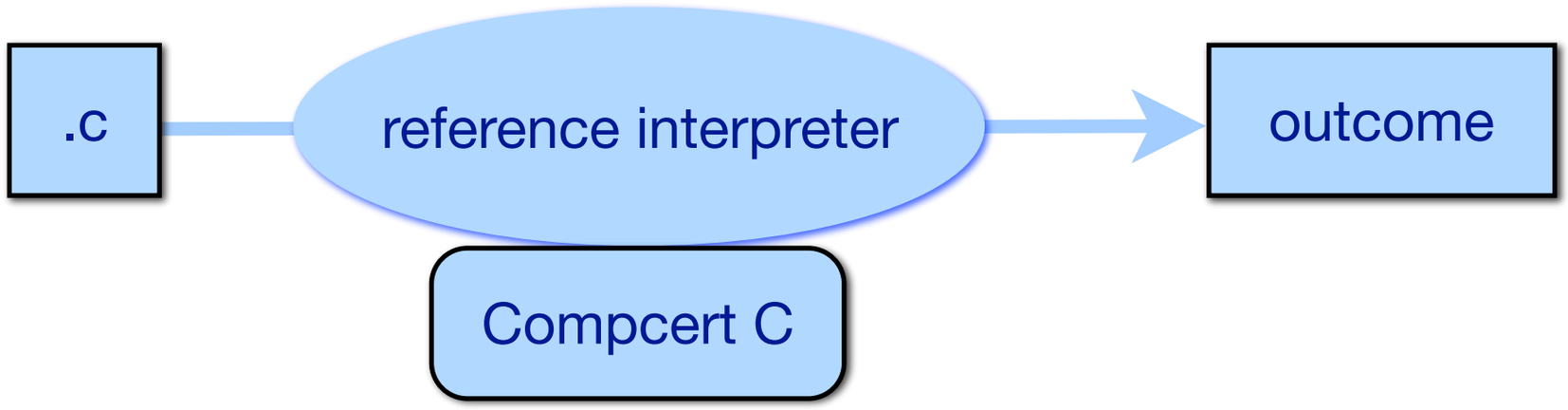
CompCert, a moderately optimising C compiler usable for critical embedded software

- Fly-by-wire software, Airbus A380 and A400M, FCGU (3600 files): mostly control-command code generated from Scade block diagrams + mini. OS

Using the Coq proof assistant, we prove the following semantic preservation property:

> For all source programs S and compiler-generated code C,
> if the compiler generates machine code C from source S,
> without reporting a compilation error,
> if S does not exhibit undefined behaviours,
> then C behaves like S.

The CompCert C reference interpreter



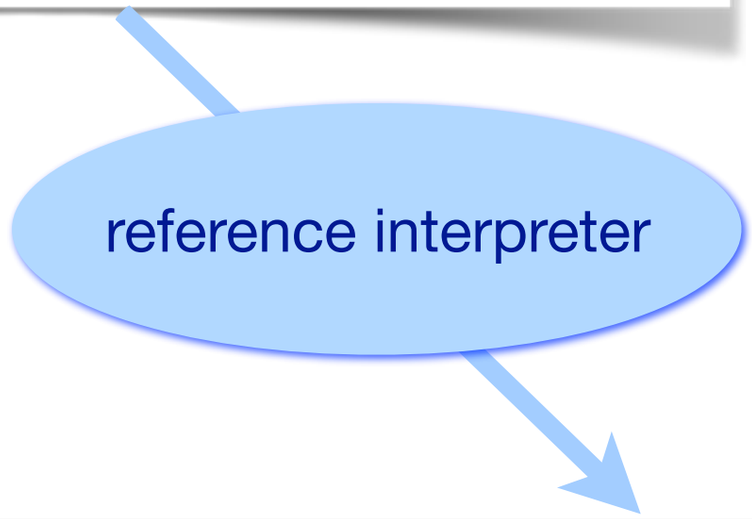
Outcome:

- normal termination or aborting on an undefined behaviour
- observable effects (I/O events)

Faithful to the formal semantics of the CompCert C language; the interpreter displays all the behaviours according to the formal semantics.

Using the reference interpreter
An example

```
int main()
{   int x[2] = { 12, 34 };
   printf("x[2] = %d\n", x[2]);
   return 0;   }
```




```
Stuck state: in function main, expression
   <printf>(<ptr __stringlit_1>, <loc x+8>)
Stuck subexpression: <loc x+8>
ERROR: Undefined behaviour
```

Undefined behaviours

ISO C standard

- signed integer overflow: `MA`
- sequence point violations: `(x`
- access to uninitialised data: `int x; x=x+1;`
- bitwise pointer arithmetic: `int *p = &x; p`
- out-of-bounds access: `int a[4]; a[`
- dereference of a NULL pointer: `int`

defined in CompCert

our work

still undefined

In those cases, a compiler is allowed to produce any code.

Low-level C code
Linux red-black trees /include/linux/rbtree.h

```
struct rb_node {
    uintptr_t rb_parent_color;
    struct rb_node *rb_right;
    struct rb_node *rb_left;   };

#define rb_color(r) (((r)-> rb_parent_color) & 1)
#define rb_parent(r) ((struct rb_node *) ((r)-> rb_parent_color & ~3))
```

Example: r.rb_parent_color = 0b0110 1110 1110 1001

- rb_color(r) ⇝ 1
- rb_parent(r) ⇝ 0b0110 1110 1110 1000

The 2 least significant bits are necessarily zeros.

Low-level C code (cont'd)
Free BSD libc implementation lib/libc/stdlib/rand.c

Random number generator (generation of a random seed)

```
struct timeval tv;
unsigned long junk; // left uninitialised on purpose
gettimeofday(&tv, NULL);
srand((getpid() « 16) ^ tv.tv_sec ^ tv.tv_usec ^ junk);
```

The C standard imposes no requirement about the compiled program.

Anecdote: clang eliminates all computations based on junk, resulting in a constant seed.

Objective of this work
CompCertS

Compile low-level programs faithfully to the programmer's intentions

Pointers are mere 32-bit integers

- They can be treated as such (e.g. bitwise operations).
- They have alignment constraints (e.g. pointers to int are 4-byte aligned).

Access to uninitialised data results in an arbitrary value

- We can operate on such a value.
- It is not a trap representation.

Similar to « friendly C » proposed by J.Regher et al.

Outline

- Defining a semantics for low-level C programs
  - A new memory model for CompCert
  - Experimental evaluation
- Proving the CompCertS compiler

An example of low-level C program

16-byte aligned

```
int main() {
 int * p = (int *) malloc (sizeof (int));

    *p = 42;
    int * q =  p | (hash(p) & 0xF) ;
    int * r = ( q >> 4 ) << 4 ;

    return *r;
}
```

p = 0x681d83a0

q = 0x681d83a5

r = 0x681d83a0 == p

ISO C standard

Undefined behaviour

Error: the first argument of '|' is not an integer type.

«Real life»

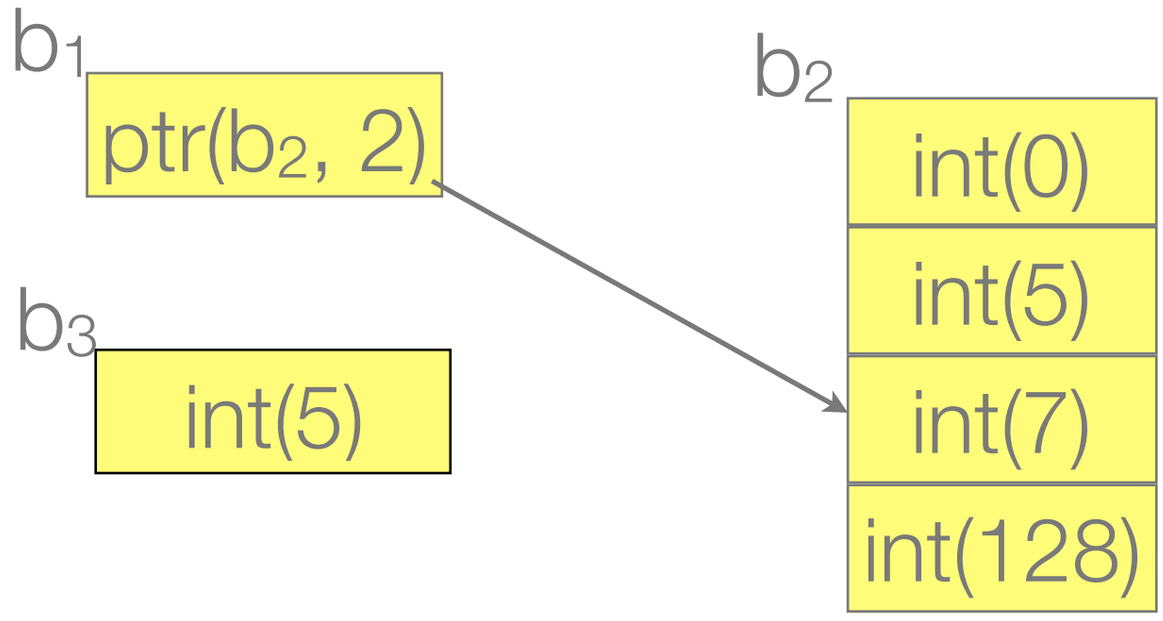
Terminates and returns 42

The CompCert memory model

- The memory state is seen as a collection of separate blocks, where each block is an array of bytes.

- Values

  v:val ::= int(i) | ptr(b,o) | undef (| long(l) | single(s) | float(f))

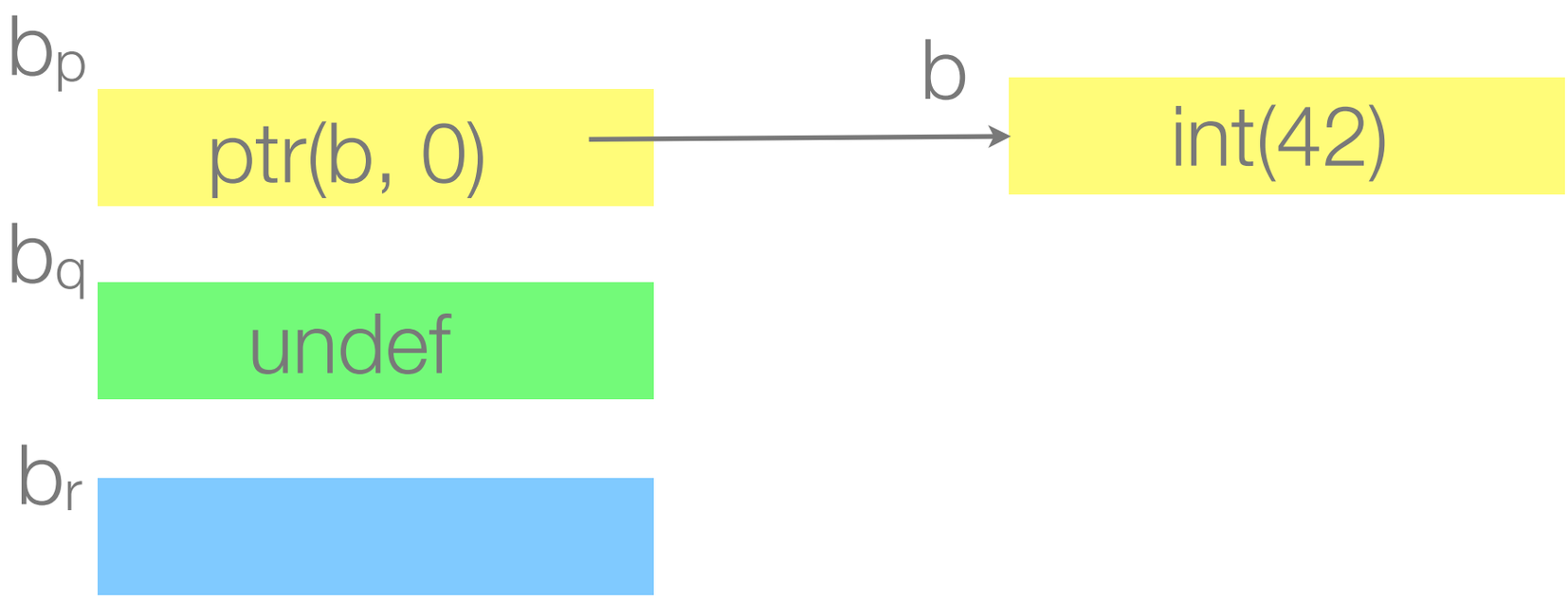


- Memory operations (alloc, free, load, store)

- The integrity of stored values is preserved (good variable properties).

Back to the example

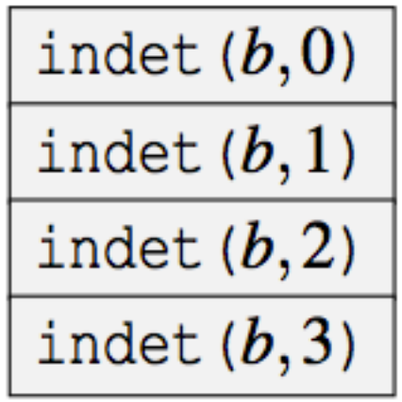
```
int main() {
 int * p = (int *) malloc (sizeof (int));

    *p = 42;

    int * q =  p | 5 ;

    int * r = ( q >> 4 ) << 4 ;

    return *r;
}
```



A new memory model for CompCert

- Symbolic values
  sv:sval ::= v
  | indet (b,i) labelled uninitialised value
  | op1 sv
  | sv1 op2 sv2

  - Example: int x; return (x-x);



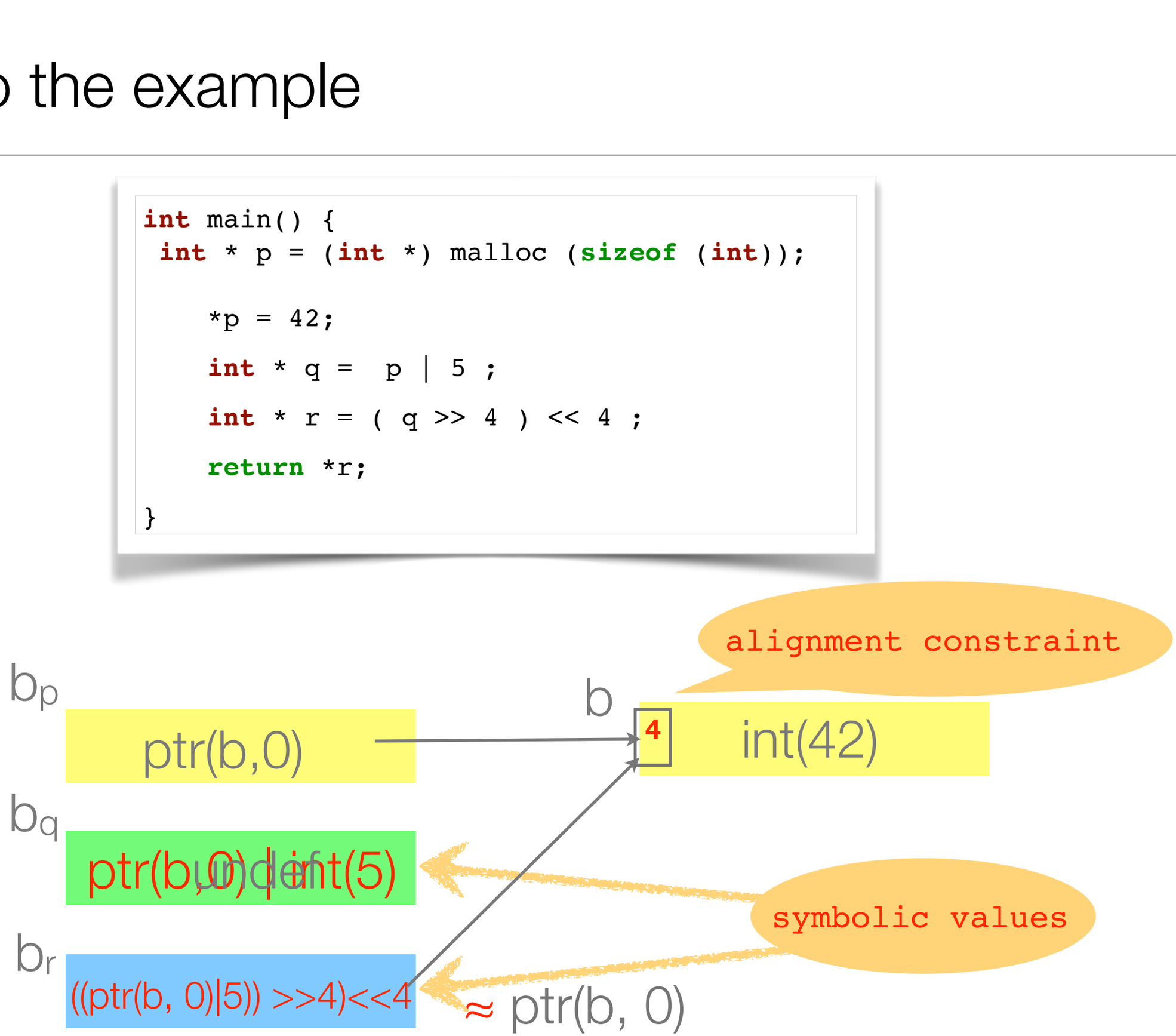
- Memory operations
  load κ m b i = ⌊sv⌋

  store κ m b i sv = ⌊m'⌋

  …

Back to the example

```
int main() {
 int * p = (int *) malloc (sizeof (int));

    *p = 42;

    int * q =  p | 5 ;

    int * r = ( q >> 4 ) << 4 ;

    return *r;
}
```



Updating the CompCert semantics
Introduce normalisation when needed

Normalisation function to transform symbolic values into values

normalise: memory → sval → val

- Memory access

$$\frac{\vdash a, M \rightarrow sv_a \qquad normalise(M, sv_a) = ptr(b,o) \qquad load(M, b, o) = \lfloor sv \rfloor}{\vdash *a, M \leftarrow sv}$$

$$\frac{\vdash a, M \rightarrow sv_a \qquad normalise(M, sv_a) = ptr(b,o) \qquad store(M, b, o, sv) = \lfloor M' \rfloor}{\vdash *a = sv, M \rightarrow skip, M'}$$

- Control flow

$$\frac{\vdash a, M \rightarrow sv_a \qquad normalise(M, sv_a) = int(i) \qquad is\_true(i)}{\vdash \text{if } a \text{ then } s1 \text{ else } s2, M \rightarrow s1, M}$$

Normalisation: intuition

Concrete memory cm : block → int

memory m

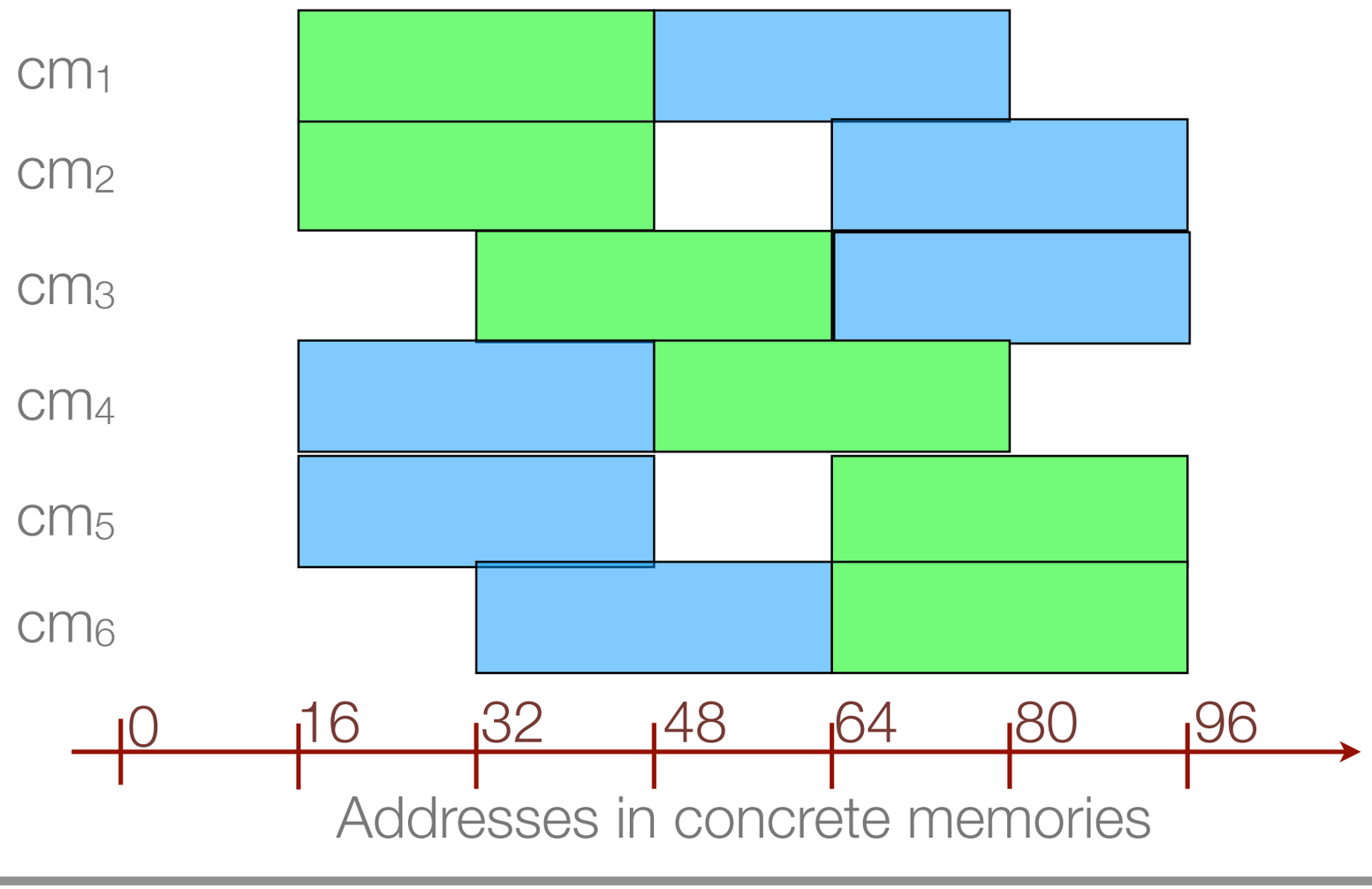
v is a sound normalisation of sv iff

v and sv evaluate the same in any cm valid for m

b_p

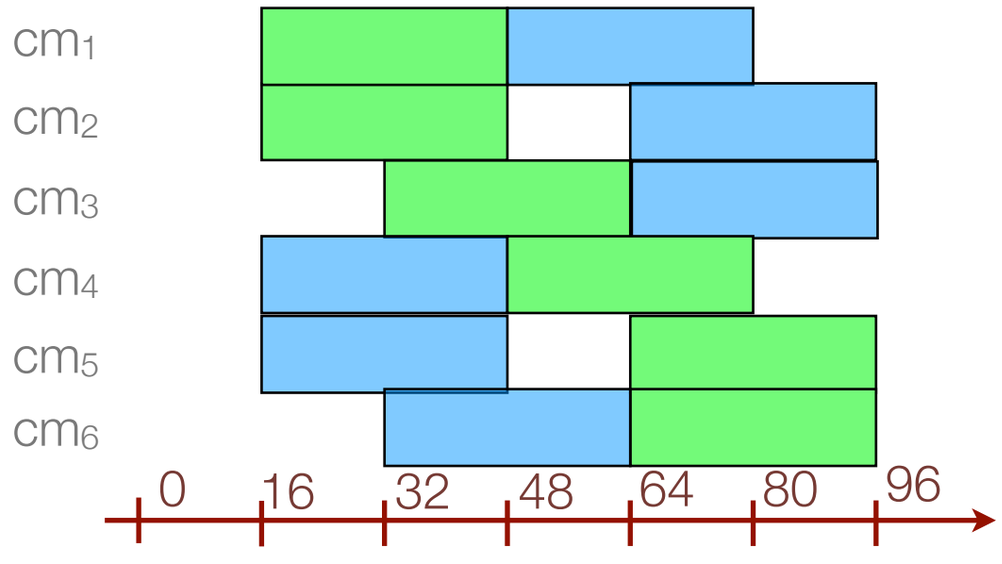
b_q

6 concrete memories of m



Sound normalisation

Validity of concrete memories



A concrete memory cm is valid for a memory m (cm ⊢ m) iff

- valid locations lie strictly between 0 and 2^{32}-1,
- valid locations from distinct blocks do not overlap,
- blocks are mapped to suitably aligned addresses.

Theorem uniqueness_of_sound_normalisation :
for any memory m and symbolic value sv,
there is at most one sound normalisation.

In particular, int(i) and ptr(b,o) cannot be sound normalisations of a same sv.

Properties of the memory model
Good-variable properties

Theorem load_store_same_old :
$\forall\ \kappa$ m b o v m', store κ m b o v = ⌊m'⌋ → load κ m' b o = ⌊v⌋ .

- store κ_{int} m b 0 int(i) = ⌊m'⌋
- load_store_same κ_{int} m' b 0 int(i) = ⌊sv⌋ with sv = ((i >> (8∗3))&0xFF) << (8∗3)
  + …
  + ((i >> (8 ∗ 0))&0xFF) << (8 ∗ 0)
- sv ≠ int(i), but sv ≈ int(i)

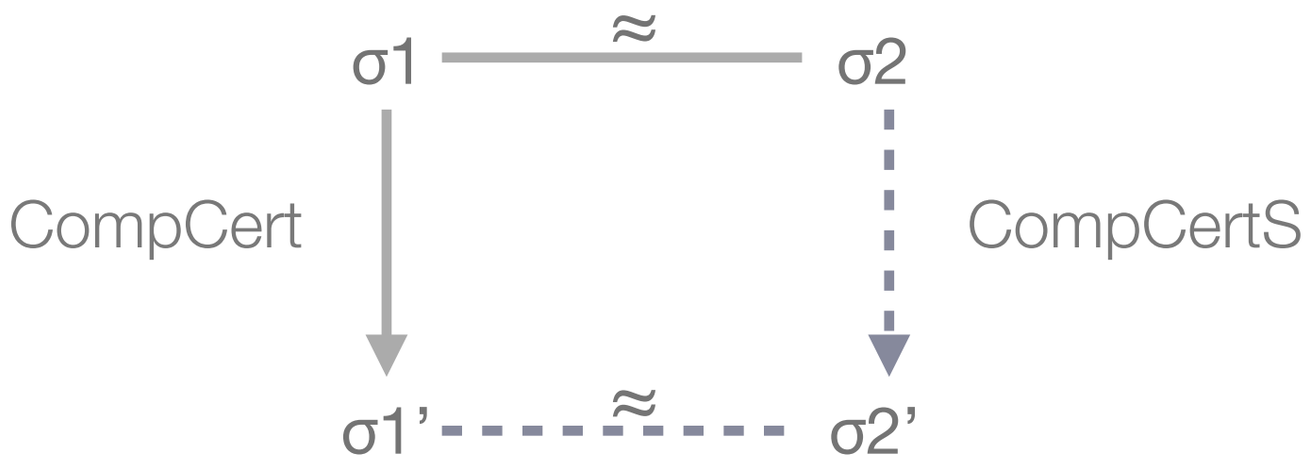
Theorem load_store_same :
$\forall\ \kappa$ m b o v m', store κ m b o v = ⌊m'⌋ →
∃ sv, load κ m' b o = ⌊sv⌋ ∧ sv ≈ v.

Experimental evaluation

- We implemented the normalisation with a SMT solver.

- Executable semantics of C, tested on CompCert benchmark examples, hand-written examples, libraries `dlmalloc` and `pdclib`.

- Test of the executable semantics
  Cross-validation: check that we preserved the CompCert's defined behaviours.



Comparison to NULL pointers

In CompCert 2.4, pointer values ptr(b,o) always compare unequal to NULL.

That snippet of code never terminates according to CompCert 2.4.

```
int main() {
 int x, *p;
 for (p = &x; p != 0; p++) /*skip*/;
   return 0;
}
```

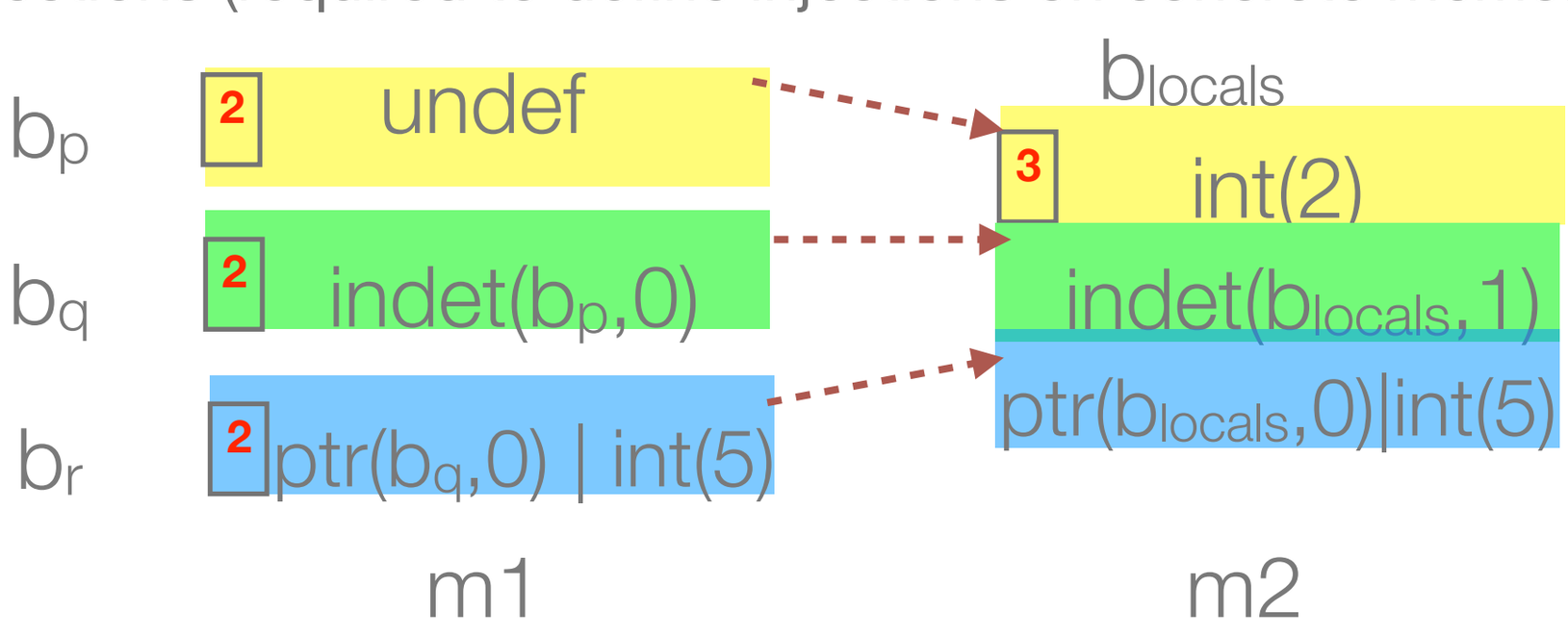
However, when run on a physical machine, it terminates when the representation of p wraps around and becomes 0.

Fixed in CompCert 2.5+: ptr(b,o) ≠ 0 only defined when (valid m b o).

Proof of the compiler passes

The architecture of the proofs from CompCert has been mostly preserved.

Main difficulty: generalizing memory injections, and relating normalisation and memory injections (required to define injections on concrete memories).



Other passes are reproved by generalising the invariants, e.g. using equivalence instead of equality.

Conclusion

A new memory model for arbitrary pointer arithmetic and uninitialised data

- symbolic values
- normalisation (implemented using a SMT solver)
- executable semantics

Finite memory → compilation in decreasing memory

Adapted (most of) the proofs of CompCert

- memory injections generalised
- formal guarantees for more programs

Perspectives

Handle freed blocks better (their size is 0, they can therefore overlap)

Apply our model to security

- Obfuscation, e.g. variable splitting: split x into x1 = x/2 and x2 = x%2
- Software Fault Isolation (Appel & al., Portable SFI, CSF 2014)
  - Mask pointers using bitwise operations
  - Currently modelled as an external call

Questions ?