A precise and abstract memory model for C using symbolic values

Frédéric Besson Sandrine Blazy **Pierre Wilke**

Rennes, France

What does this program do?

```
int main(){
  int * p = (int *) malloc (sizeof (int));
  //
  *p = 42;
  int * q =  p | (checksum(p) & 0xF);
  //
  assert( checksum( (q » 4) « 4 ) == (q & 0xF) );
  int * r = ( q » 4 ) « 4;
  //
  return *r;
}
```

What does this program do?

```
int main(){
  int * p = (int *) malloc (sizeof (int));
  // p = 0xTUVWXYZ0
  *p = 42;
  int * q =  p | (checksum(p) & 0xF);
  //
  assert( checksum( (q » 4) « 4 ) == (q & 0xF) );
  int * r = ( q » 4 ) « 4;
  //
  return *r;
}
```

What does this program do?

```
int main(){
  int * p = (int *) malloc (sizeof (int));
  // p = 0xTUVWXYZ0
  *p = 42;
  int * q =  p | (checksum(p) & 0xF);
  // q = 0xTUVWXYZ5
  assert( checksum( (q » 4) « 4 ) == (q & 0xF) );
  int * r = ( q » 4 ) « 4;
  //
  return *r;
}
```

What does this program do?

```
int main(){
  int * p = (int *) malloc (sizeof (int));
  // p = 0xTUVWXYZ0
  *p = 42;
  int * q =  p | (checksum(p) & 0xF);
  // q = 0xTUVWXYZ5
  assert( checksum( (q » 4) « 4 ) == (q & 0xF) );
  int * r = ( q » 4 ) « 4;
  // r = 0xTUVWXYZ0 == p
  return *r;
}
```

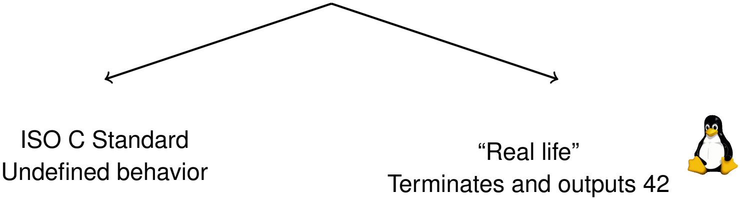
What does this program do?

```
int main(){
  int * p = (int *) malloc (sizeof (int));
  // p = 0xTUVWXYZ0
  *p = 42;
  int * q =  p | (checksum(p) & 0xF);
  // q = 0xTUVWXYZ5
  assert( checksum( (q » 4) « 4 ) == (q & 0xF) );
  int * r = ( q » 4 ) « 4;
  // r = 0xTUVWXYZ0 == p
  return *r;
}
```

"Real life"
Terminates and outputs 42

What does this program do?

```
int main(){
  int * p = (int *) malloc (sizeof (int));
  // p = 0xTUVWXYZ0
  *p = 42;
  int * q =  p | (checksum(p) & 0xF);
  // q = 0xTUVWXYZ5
  assert( checksum( (q » 4) « 4 ) == (q & 0xF) );
  int * r = ( q » 4 ) « 4;
  // r = 0xTUVWXYZ0 == p
  return *r;
}
```

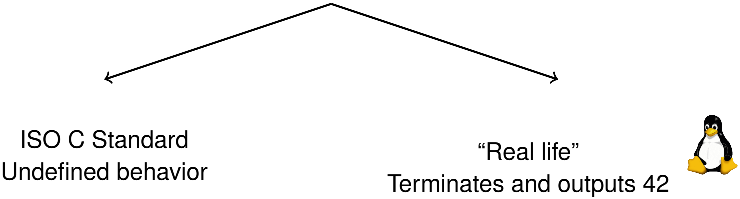


ISO C Standard
Undefined behavior

“Real life”
Terminates and outputs 42

What does this program do?

```
int main(){
  int * p = (int *) malloc (sizeof (int));
  // p = 0xTUVWXYZ0
  *p = 42;
  int * q =  p | (checksum(p) & 0xF);
  // q = 0xTUVWXYZ5
  assert( checksum( (q » 4) « 4 ) == (q & 0xF) );
  int * r = ( q » 4 ) « 4;
  // r = 0xTUVWXYZ0 == p
  return *r;
}
```



ISO C Standard
Undefined behavior

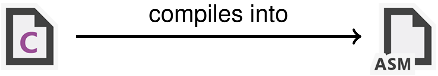
“Real life”
Terminates and outputs 42

This work: an executable semantics for this kind of programs

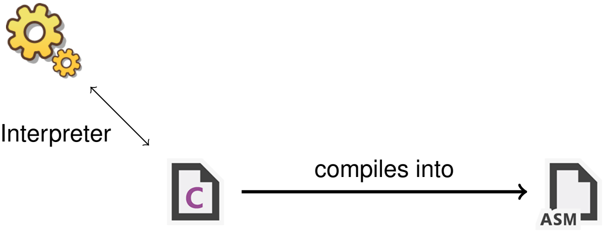
State of the art: formal semantics for C

- Cholera [ESOP'99]: first formal semantics of C
- CompCert [JAR'09]: large subset of C, executable, abstract memory model
- KCC [POPL'12]: large subset of C, executable, abstract memory model
- Krebbers [POPL'14]: closer to the ISO C standard
- Our work: CompCert + more defined semantics + **low level** memory model

CompCert



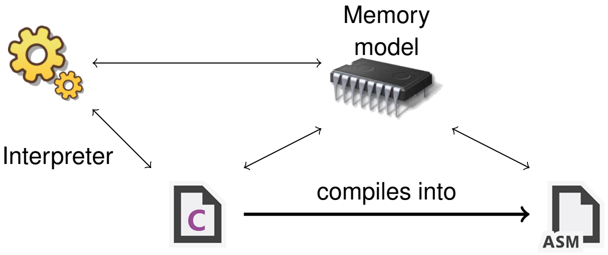
CompCert



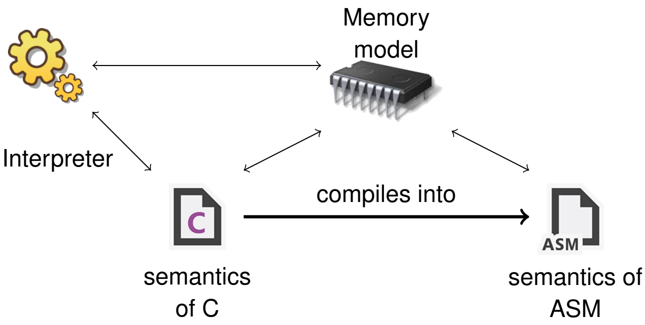
Interpreter

compiles into

CompCert



CompCert



Memory model

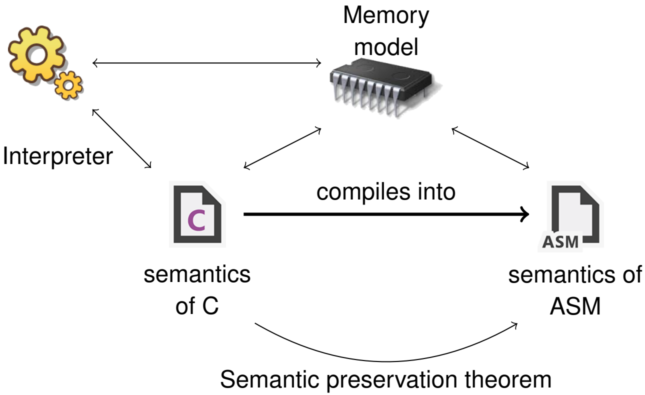
Interpreter

compiles into

semantics of C

semantics of ASM

CompCert



Memory model

Interpreter

compiles into

semantics of C

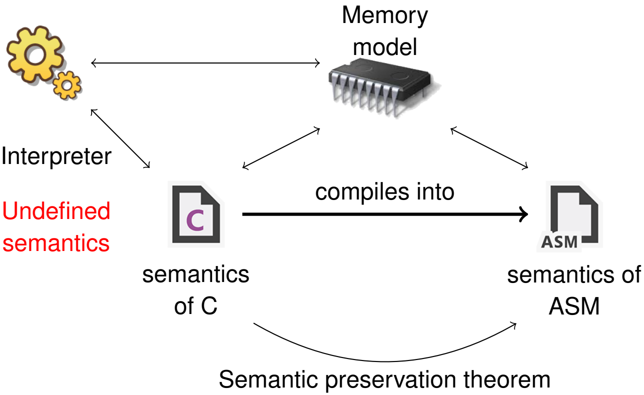
semantics of ASM

Semantic preservation theorem

Coq

***If** the program has well-defined semantics*
***Then** the compiler does not introduce bugs*

CompCert



Memory model

Interpreter

Undefined semantics

compiles into

semantics of C

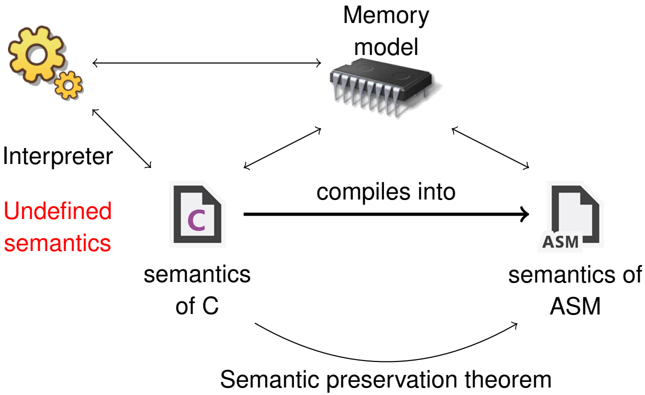
semantics of ASM

Semantic preservation theorem

Coq

***If** the program has well-defined semantics*
***Then** the compiler does not introduce bugs*

CompCert



Memory model

Interpreter

Undefined semantics

compiles into

semantics of C

semantics of ASM

Semantic preservation theorem

Coq

***If** the program has well-defined semantics*

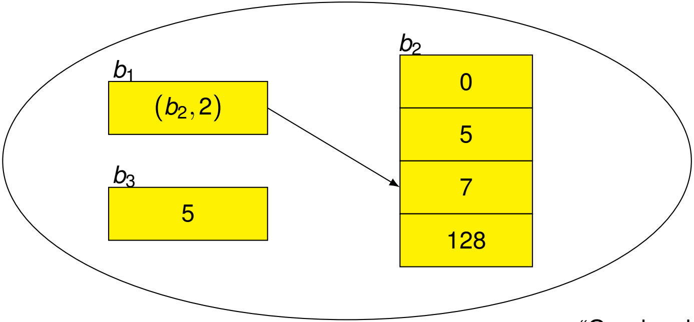
***Then** the compiler does not introduce bugs*

CompCert's memory model

- set of disjoint blocks
- Values:

$$val ::= i \mid (b, o) \mid \texttt{Vundef}$$

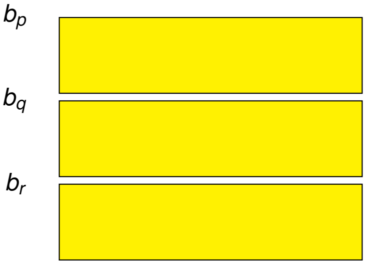
- $load(M, b, o) = \lfloor v \rfloor$
- $store(M, b, o, v) = \lfloor M' \rfloor$
- $alloc(M, lo, hi) = (M', b)$
- $free(M, b) = \lfloor M' \rfloor$



"Good variable" properties:
$load(store(M, b, o, v), b, o) = v$

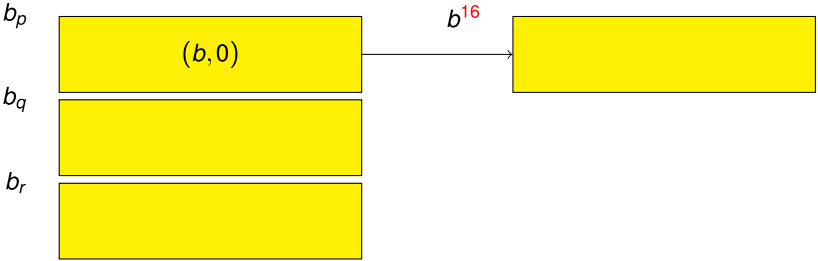
Back to the example

```
int main(){
  int * p = (int *) malloc (sizeof (int));
  *p = 42;
  int * q = p | 5;
  int * r = (q » 4) « 4;
  return *r;
}
```



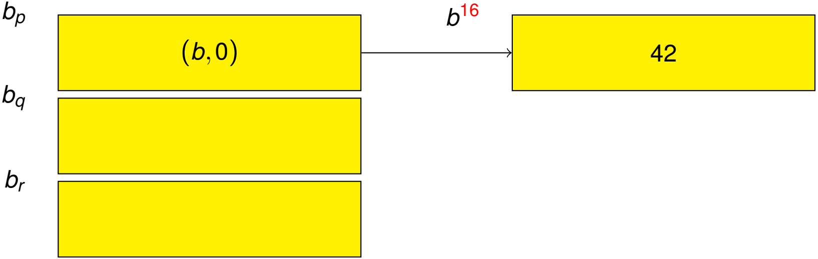
Back to the example

```
int main(){
  int * p = (int *) malloc (sizeof (int));
  *p = 42;
  int * q = p | 5;
  int * r = (q » 4) « 4;
  return *r;
}
```



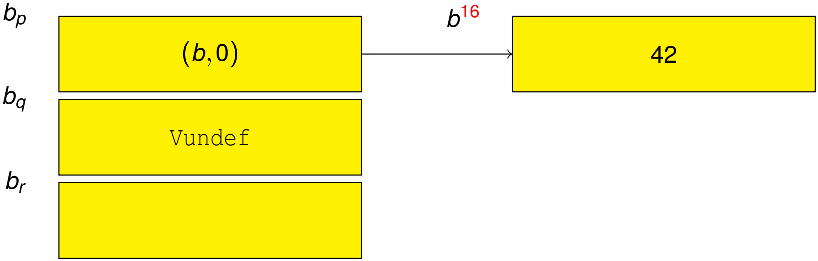
Back to the example

```
int main(){
  int * p = (int *) malloc (sizeof (int));
  *p = 42;
  int * q = p | 5;
  int * r = (q » 4) « 4;
  return *r;
}
```



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;
}
```



b_p

$(b, 0)$

b^{16}

42

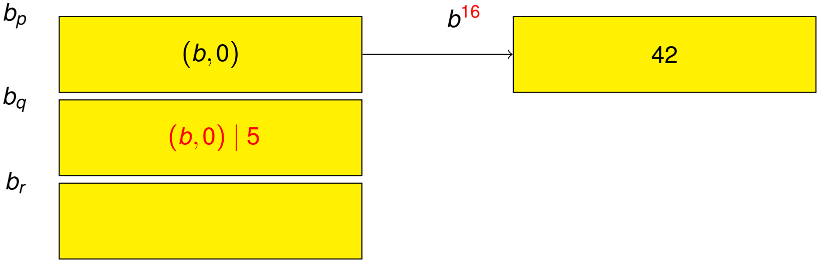
b_q

Vundef

b_r

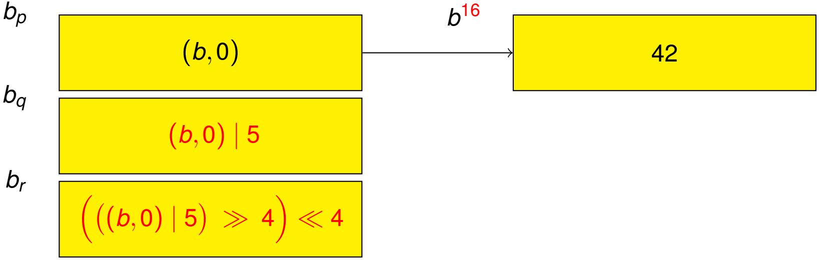
Back to the example

```
int main(){
  int * p = (int *) malloc (sizeof (int));
  *p = 42;
  int * q = p | 5;
  int * r = (q » 4) « 4;
  return *r;
}
```



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;
}
```



b_p $(b, 0)$

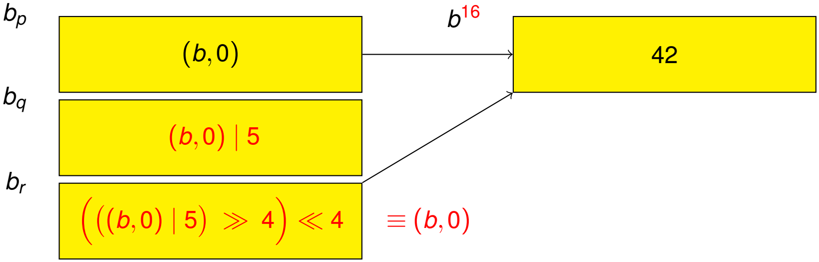
b^{16} 42

b_q $(b, 0) \mid 5$

b_r $\left(\left((b, 0) \mid 5\right) \gg 4\right) \ll 4$

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;
}
```



b_p $(b, 0)$

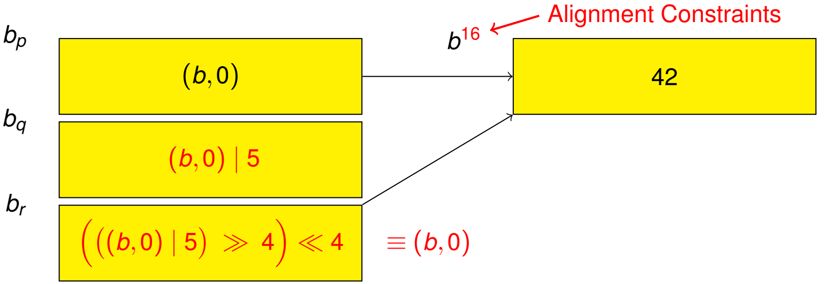
b^{16} 42

b_q $(b, 0) \mid 5$

b_r $\Big(\big((b, 0) \mid 5\big) \gg 4\Big) \ll 4$ $\equiv (b, 0)$

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;
}
```



Alignment Constraints

b^{16}

b_p $(b, 0)$

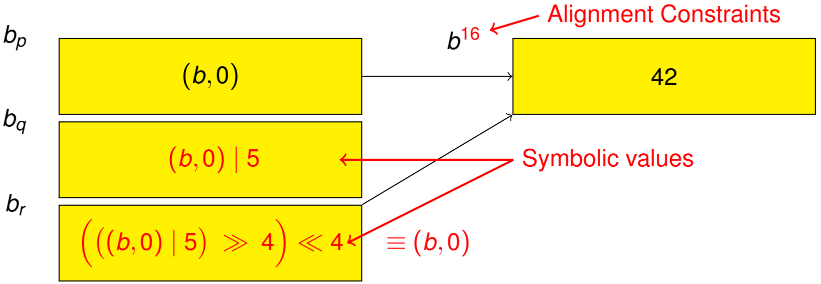
b_q $(b, 0) \mid 5$

b_r $\Big(\big(((b, 0) \mid 5) \gg 4\big) \ll 4$ $\equiv (b, 0)$

42

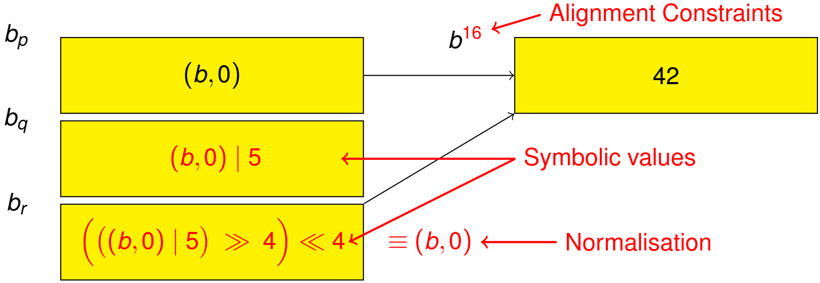
Back to the example

```
int main(){
  int * p = (int *) malloc (sizeof (int));
  *p = 42;
  int * q = p | 5;
  int * r = (q » 4) « 4;
  return *r;
}
```



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;
}
```



Outline

1. A C semantics with symbolic values

2. Normalisation: specification and implementation

3. Experimental evaluation

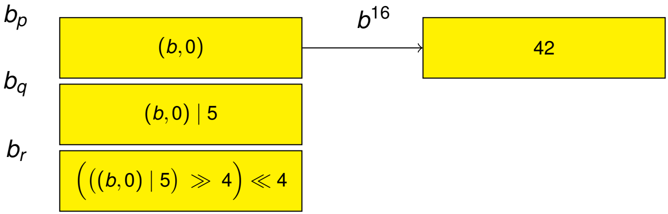
Outline

1. A C semantics with symbolic values

2. Normalisation: specification and implementation

3. Experimental evaluation

CompCert's memory model with symbolic values

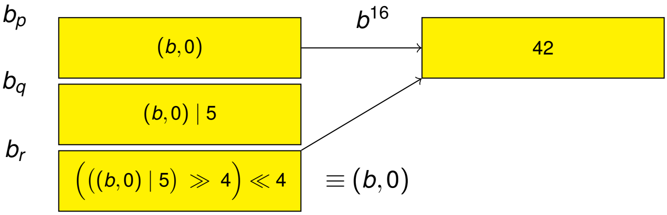


- Alignment
  - $alloc(M, lo, hi, \mathbf{mask}) = (M', b)$

$$A(b) \;\&\; mask = A(b)$$

- Symbolic values
  - $sv ::= val \;\mid\; op_1\; sv \;\mid\; sv\; op_2\; sv$
  - $load(M, b, o) = \lfloor \mathbf{sv} \rfloor$
  - $store(M, b, o, \mathbf{sv}) = \lfloor M' \rfloor$

CompCert's memory model with symbolic values



`normalise` : $\textit{mem} \rightarrow \textit{sv} \rightarrow \textit{option val}$

When do we need to normalise symbolic values?

- Memory accesses:
  - `return *r;`
  - `*p = 42;`
- Control flow:
  - `if (c) { ... } else { ... }`

Adapting the CompCert semantics

Semantic rules

$$\frac{\vdash a, M \rightarrow (b, o) \quad \texttt{load}(M, b, o) = \lfloor v \rfloor}{\vdash *a, M \rightarrow v}$$

$$\frac{\vdash a, M \rightarrow (b, o) \quad \texttt{store}(M, b, o, v) = \lfloor M' \rfloor}{\vdash *a = v, M \rightarrow M'}$$

$$\frac{\vdash \texttt{is\_true}(a)}{\vdash \texttt{if}\ a\ \texttt{then}\ s_1\ \texttt{else}\ s_2, M \rightarrow s_1, M}$$

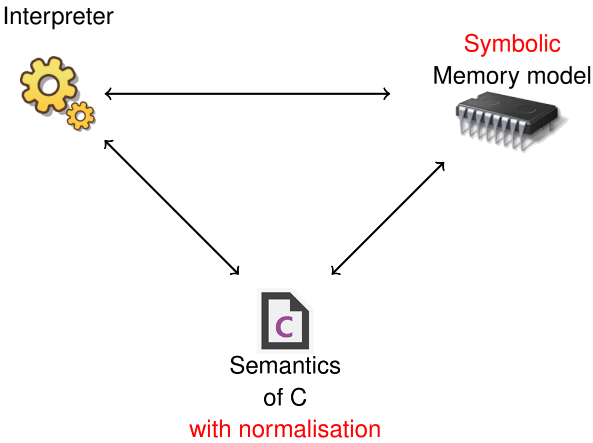
Adapting the CompCert semantics

Semantic rules

$$\frac{\vdash a, M \to sv_a \quad \texttt{normalise}(M, sv_a) = \lfloor(b, o)\rfloor \quad \texttt{load}(M, b, o) = \lfloor sv \rfloor}{\vdash *a, M \to sv}$$

$$\frac{\vdash a, M \to sv_a \quad \texttt{normalise}(M, sv_a) = \lfloor(b, o)\rfloor \quad \texttt{store}(M, b, o, sv) = \lfloor M' \rfloor}{\vdash *a = sv, M \to M'}$$

$$\frac{\vdash \texttt{normalise}(M, a) = \lfloor i \rfloor \quad \texttt{is\_true}(i)}{\vdash \texttt{if } a \texttt{ then } s_1 \texttt{ else } s_2, M \to s_1, M}$$



Interpreter

Symbolic
Memory model

Semantics
of C
with normalisation

Outline

1. A C semantics with symbolic values

2. Normalisation: specification and implementation

3. Experimental evaluation

Evaluation of symbolic values

- Input:
  - *a memory mapping $A : block \to int_{32}$*
  - *a symbolic value sv*
- Output: the set of machine integers that *sv* evaluates to.

$$[\![\cdot]\!]_A : sv \to \mathcal{P}(int_{32})$$

$$\frac{}{i \in [\![i]\!]_A} \qquad \frac{}{A(b) + o \in [\![(b,o)]\!]_A} \qquad \frac{}{n \in [\![\texttt{Vundef}]\!]_A}$$

$$\frac{v_1 \in [\![e_1]\!]_A \quad \texttt{eval\_unop}(op_1, v_1) = \lfloor v \rfloor}{v \in [\![op_1\ e_1]\!]_A}$$

$$\frac{v_1 \in [\![e_1]\!]_A \quad v_2 \in [\![e_2]\!]_A \quad \texttt{eval\_binop}(op_2, v_1, v_2) = \lfloor v \rfloor}{v \in [\![e_1\, op_2\ e_2]\!]_A}$$

Valid memory mapping : $A \models M$

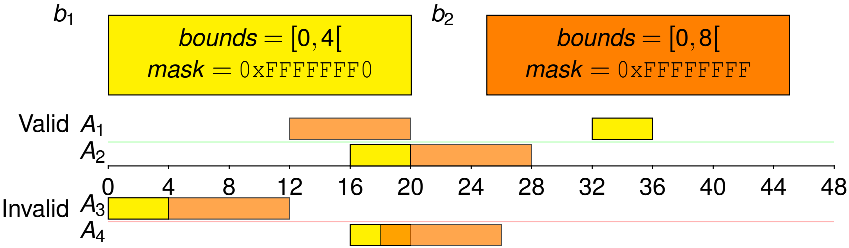
A memory mapping $A : \mathit{block} \to \mathit{int}_{32}$ is valid for memory M iff:

1. addresses from distinct blocks do not overlap,
2. the address of a block satisfies its alignment constraints:

$$A(b)\ \&\ \mathtt{mask}(M, b) = A(b)$$

3. valid addresses are not null.

Example:



b_1: $\mathit{bounds} = [0, 4[$, $\mathit{mask} = \mathtt{0xFFFFFFF0}$

b_2: $\mathit{bounds} = [0, 8[$, $\mathit{mask} = \mathtt{0xFFFFFFFF}$

Normalisation: specification

If $\texttt{normalise}(M, sv) = \lfloor v \rfloor$ then:

- $v \neq \texttt{Vundef}$
- $\forall A \models M, [\![sv]\!]_A = [\![v]\!]_A$

Example: Consider block b with bounds $[0, 4[$ and 16-byte aligned.

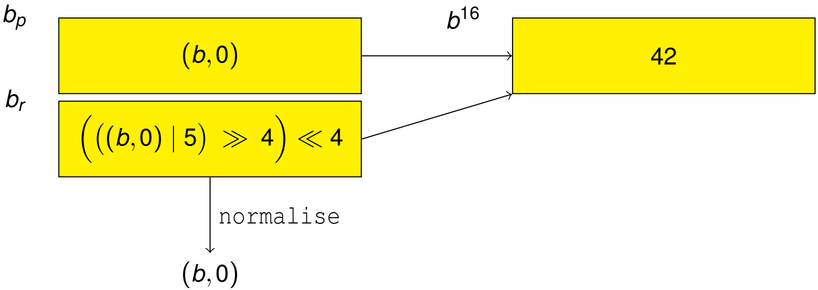
$sv = \Big(\big((b, 0) \mid 5 \big) \gg 4 \Big) \ll 4$

$v = (b, 0)$

A	$[\![\mathbf{sv}]\!]_{\mathbf{A}}$		$[\![\mathbf{v}]\!]_{\mathbf{A}}$
$\{b \mapsto 16\}$	$(((16+0) \mid 5) \gg 4) \ll 4$	$= 16$	16
$\{b \mapsto 32\}$	$(((32+0) \mid 5) \gg 4) \ll 4$	$= 32$	32
$\{b \mapsto 16k\}$	$(((16k+0) \mid 5) \gg 4) \ll 4$	$= 16k$	$16k$

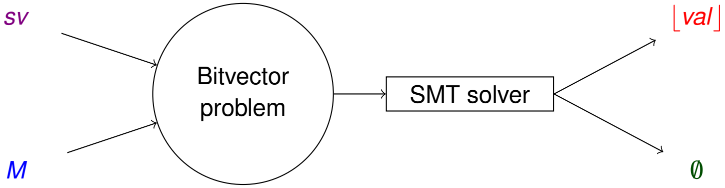
Normalisation: implementation

```
int main(){
  int * p = (int *) malloc (sizeof (int));
  *p = 42;
  int * q = p | 5;
  int * r = (q » 4) « 4;
  return *r;
}
```



How to compute normalisation?

$$\texttt{normalise}(M, sv) = \begin{cases} \lfloor val \rfloor \\ \emptyset \end{cases}$$



Normalisation in our example

Let A be a valid memory layout for memory M: $A \vDash M$

$$sv_r = \Big(\big((b,0) \mid 5 \big) \gg 4 \Big) \ll 4$$

Translation into a bitvector expression

$$bv_r = \Big(\big((A(b) \; + \; 0) \mid 5 \big) \gg 4 \Big) \ll 4$$

Goal: find a unique model (b, i) such that: $bv_r = A(b) + i$

Two steps:

- find a model (b_0, i_0) such that $bv_r = A(b_0) + i_0$
  - $(b_0, i_0) = (b, 0)$
- check that this solution is unique: $\textbf{unsat}(bv_r = A(b) + i \wedge b \neq b_0)$
  - the solution is indeed unique
  - the normalisation returns $\lfloor (b, 0) \rfloor$

Outline

1. A C semantics with symbolic values
2. Normalisation: specification and implementation
3. Experimental evaluation

Experiments

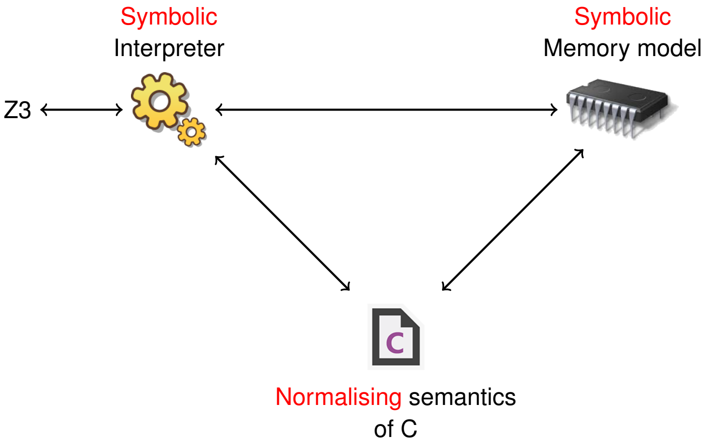
Real life programs

- a `malloc` implementation (`dlmalloc.c`)
- excerpts from a C library : Public Domain C Library
- excerpts from a cryptographic library: Networking and Cryptographic library
- hand-written C programs

What kind of symbolic values do these programs trigger?

- bitwise operations on pointer
- use of undefined values

Putting the pieces together



Symbolic Interpreter

Symbolic Memory model

Z3

Normalising semantics of C

Coq development available at `http://www.irisa.fr/celtique/ext/csem/`

Conclusion

Results:

- non-regression
- more programs have their expected semantics
- limits: some undefined behaviors are not captured by our semantics
    - pointer comparison

Ongoing work:

- Proofs of the memory model
    - *e.g.* $load(store(M, b, o, \mathbf{sv}), b, o) = \mathbf{sv}$
- Proof of the whole CompCert compiler?
    - memory injections redefined

Questions?