A concrete memory model for CompCert

Frédéric Besson   Sandrine Blazy   **Pierre Wilke**

Rennes, France

IRISA UNIVERSITÉ DE RENNES 1 Inria INVENTEURS DU MONDE NUMÉRIQUE

CompCert

- real-world C to ASM compiler used in industry (commercialised by AbsInt)
- proven correct in Coq: it does not introduce bugs!

CompCert

- real-world C to ASM compiler used in industry (commercialised by AbsInt)
- proven correct in Coq: it does not introduce bugs!

CompCert

- real-world C to ASM compiler used in industry (commercialised by AbsInt)
- proven correct in Coq: it does not introduce bugs!

Each language has a Formal Semantics

i.e. a mathematical meaning for programs

CompCert

- real-world C to ASM compiler used in industry (commercialised by AbsInt)
- proven correct in Coq: it does not introduce bugs!

Each language has a Formal Semantics

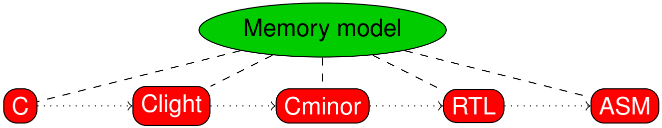
i.e. a mathematical meaning for programs

Proof of semantic preservation

For every source program S that has a defined semantics,
If the compiler succeeds to generate a target program T,
Then T has the same behavior as S.

CompCert

- real-world C to ASM compiler used in industry (commercialised by AbsInt)
- proven correct in Coq: it does not introduce bugs!



Each language has a Formal Semantics

i.e. a mathematical meaning for programs

Proof of semantic preservation

For every source program S that has a defined semantics,
If the compiler succeeds to generate a target program T,
Then T has the same behavior as S.

Goal: Make the semantics of C more defined

Why did C leave some behaviors undefined?

- Portability
- Performance

Why do we want to make it more defined?

- real-life programs use features that are undefined, according to C
- the compilation theorem will be more useful

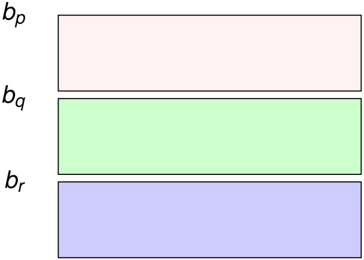
What kind of undefined behaviors do we aim at?

- undefined pointer arithmetic, i.e. bitwise operators
- use of uninitialised memory

Our starting point: CompCert

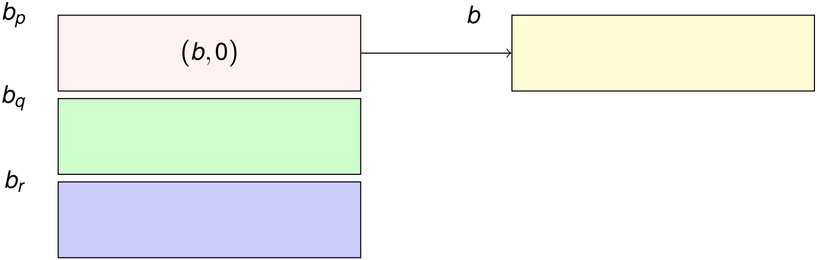
An example of low-level C program in CompCert

```c
int main(){
  int * p = (int *) malloc (sizeof (int));
  *p = 42;
  int * q = p | 5;
  int * r = (q >> 3) << 3;
  return *r;
}
```



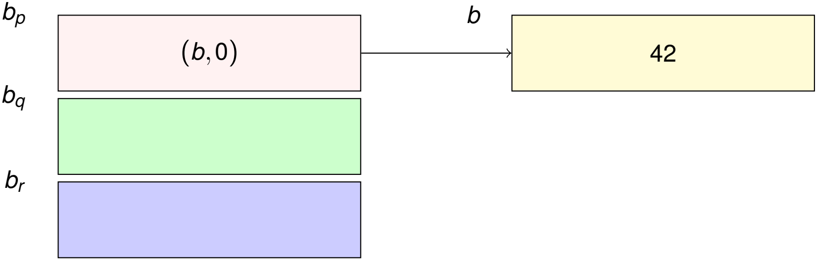
An example of low-level C program in CompCert

```
int main(){
  int * p = (int *) malloc (sizeof (int));
  *p = 42;
  int * q = p | 5;
  int * r = (q » 3) « 3;
  return *r;
}
```



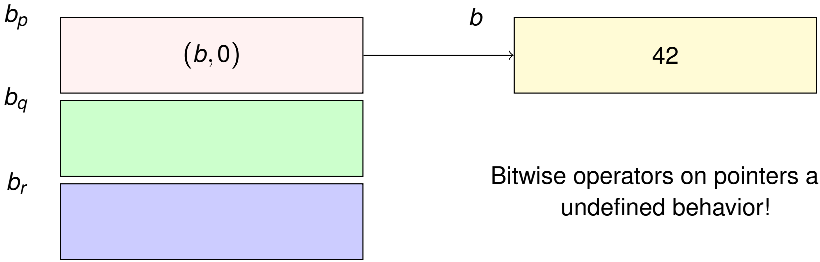
An example of low-level C program in CompCert

```
int main(){
  int * p = (int *) malloc (sizeof (int));
  *p = 42;
  int * q = p | 5;
  int * r = (q » 3) « 3;
  return *r;
}
```



An example of low-level C program in CompCert

```
int main(){
  int * p = (int *) malloc (sizeof (int));
  *p = 42;
  int * q = p | 5;
  int * r = (q » 3) « 3;
  return *r;
}
```



Bitwise operators on pointers are undefined behavior!

CompCert [JAR'09], KCC [POPL'12], Krebbers [POPL'14], Norrish [PhD'98]: undefined behavior

Kang *et al.* [PLDI'15]: don't model bitwise operators

Contributions

- Previous work [APLAS'14]:
  A memory model for low-level programs

- This work:
  - integration of the memory model inside CompCert
  - correctness proofs of the memory model
  - correctness proofs of the transformations of the frontend (up to Cminor)

Outline

1. CompCert's memory model
2. New features of the memory model
3. Consistency of the memory models
4. CompCert proof: Overview
5. Conclusion

Outline

New features of the memory model

Symbolic expressions

$val ::= i \mid (b, o)$ not expressive enough

We change the semantic domain to:

$$expr ::= val \mid op_1\ expr \mid expr\ op_2\ expr$$

New features of the memory model

Symbolic expressions

$val ::= i \mid (b, o)$ not expressive enough
We change the semantic domain to:

$$expr ::= val \mid op_1\ expr \mid expr\ op_2\ expr$$

Alignment constraints

We need information about some bits of the concrete address of a pointer
The `alloc` primitive takes an extra parameter `mask`, such that:

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

Interaction with the memory model

What is the semantics of reading from memory: `*p` ?

In CompCert, `p` is evaluated into a pointer (b, i), then we can use $load(M, b, i)$

In our model, `p` is a symbolic expression. It needs to be transformed into a pointer so that we can use *load*.

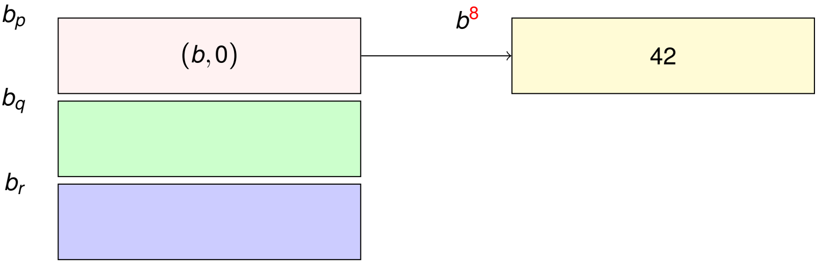
$$normalise : mem \to \texttt{expr} \to \lfloor val \rfloor$$

We need to modify the semantics to include calls to `normalise`

- memory accesses (load and store)
- conditionnal branches

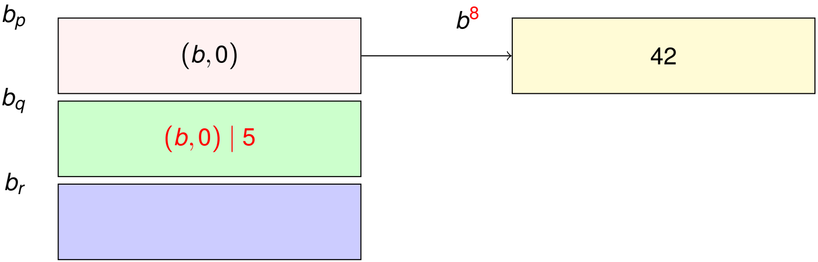
Back to the example

```
int main(){
  int * p = (int *) malloc (sizeof (int));
  *p = 42;
  int * q = p | 5;
  int * r = (q » 3) « 3;
  return *r;
}
```



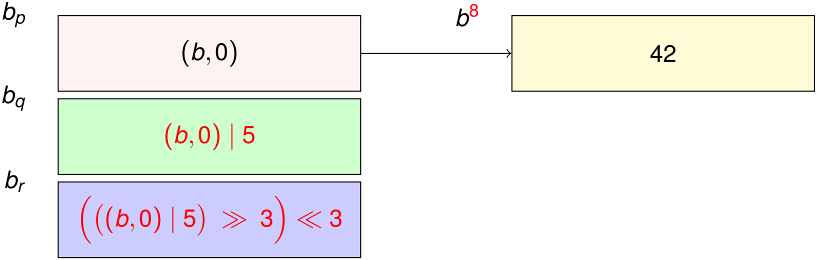
Back to the example

```
int main(){
  int * p = (int *) malloc (sizeof (int));
  *p = 42;
  int * q = p | 5;
  int * r = (q » 3) « 3;
  return *r;
}
```



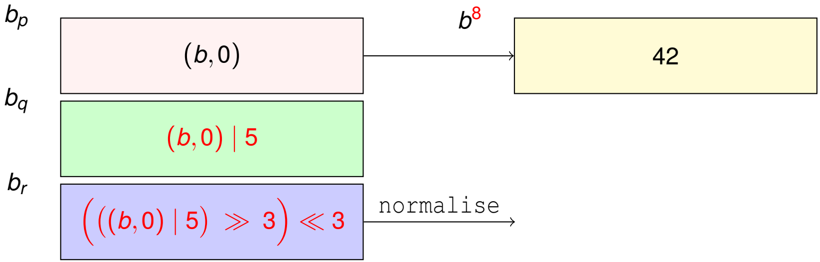
Back to the example

```
int main(){
  int * p = (int *) malloc (sizeof (int));
  *p = 42;
  int * q = p | 5;
  int * r = (q » 3) « 3;
  return *r;
}
```



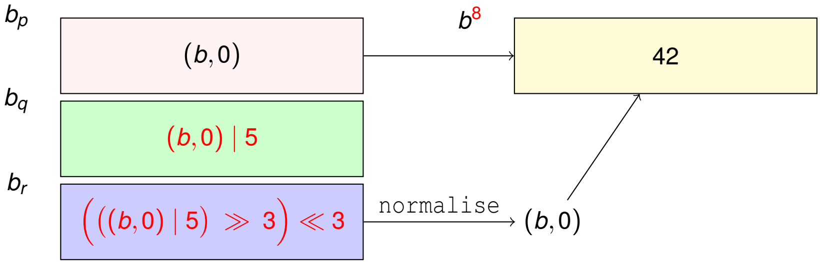
Back to the example

```
int main(){
  int * p = (int *) malloc (sizeof (int));
  *p = 42;
  int * q = p | 5;
  int * r = (q » 3) « 3;
  return *r;
}
```

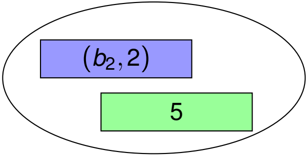


Back to the example

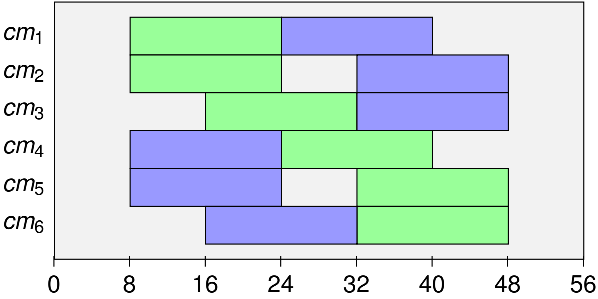
```c
int main(){
  int * p = (int *) malloc (sizeof (int));
  *p = 42;
  int * q = p | 5;
  int * r = (q » 3) « 3;
  return *r;
}
```



Normalisation specification: concrete memories



Abstract memory m



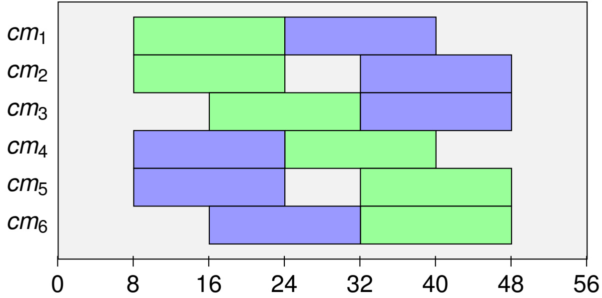
Concrete memories of m

$$cm_i \vdash m$$

- range : $]0;55[$
- no overlap
- alignment

Normalisation: example 1

$$e = (((\,b\,, 0) \mid 5) \gg 3) \ll 3$$



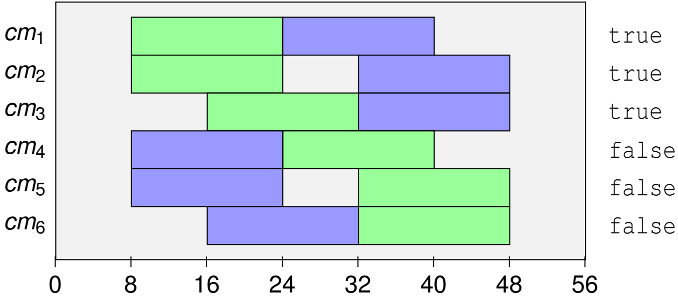
$$8 = [\![(b, o)]\!]_{cm_1}$$
$$8 = [\![(b, o)]\!]_{cm_2}$$
$$16 = [\![(b, o)]\!]_{cm_3}$$
$$24 = [\![(b, o)]\!]_{cm_4}$$
$$32 = [\![(b, o)]\!]_{cm_5}$$
$$32 = [\![(b, o)]\!]_{cm_6}$$

$$\begin{aligned}
[\![e]\!]_{cm_1} &= (((cm_1(b) + 0) \mid 5) \gg 3) = ((8 \mid 5) \gg 3) \\
&= ((\texttt{0b1000} \mid 5) \gg 3) \ll 3 = (\texttt{0b1101} \gg 3) \ll 3 \\
&= \texttt{0b0001} \ll 3 = \texttt{0b1000} = 8 = cm_1(b)
\end{aligned}$$

$\forall i, [\![e]\!]_{cm_i} = cm_i(b)$, hence e normalises into $(b, 0)$

Normalisation: example 2

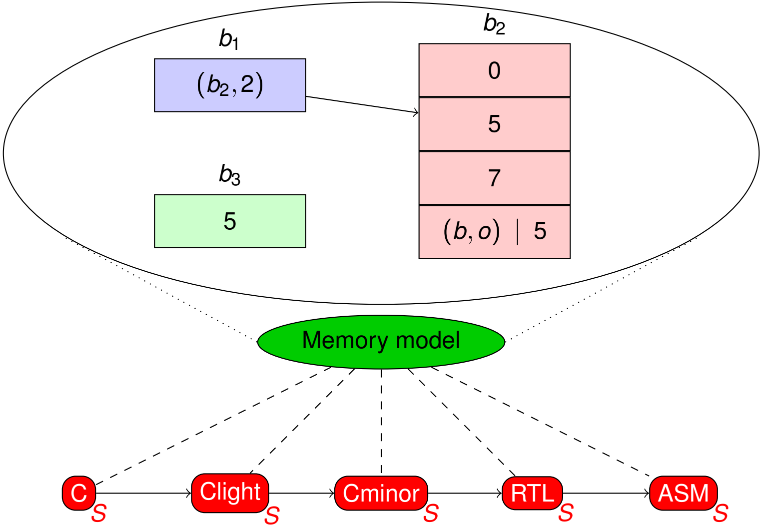
$$e = (\,b\,, 0) > (\,b'\,, 0)$$



There is no v such that $\forall i, [\![e]\!]_{cm_i} = [\![v]\!]_{cm_i}$, hence e doesn't normalise

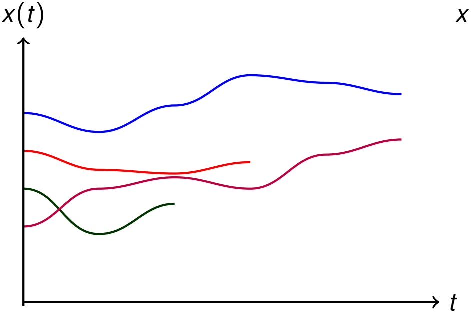
CompCert with symbolic expressions

$$expr ::= val \mid op_1\ expr \mid expr\ op_2\ expr$$

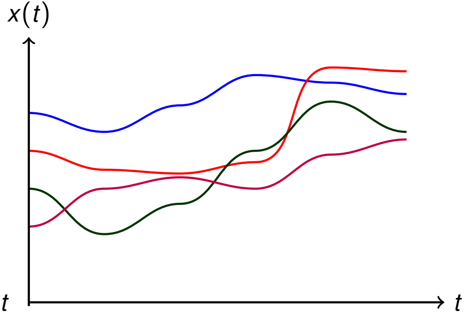


Outline

How does our model compare to CompCert?



Behaviors in CompCert



Behaviors with symbolic expressions

We are an *extension* of CompCert

How does our model compare to CompCert?

Formally,

```
Lemma expr_add_ok: ∀ v₁ v₂ m v,
                    sem_add v₁ v₂ m = ⌊v⌋ →
                    ∃ e, sem_add_expr v₁ v₂ m = ⌊e⌋ ∧
                         normalise m e = v.
```

If the addition of v_1 and v_2 succeeds in CompCert,
Then it should succeed in our model as well,
And the expression we compute should normalise into the same value.

Discovery of bugs

2 cases where our model disagrees with CompCert

- Bug in CompCert 2.4: Pointer comparison to `NULL`
  (fixed in CompCert 2.5)

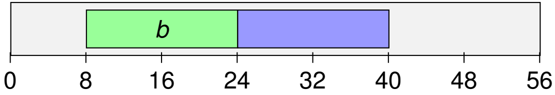
- Bug in our model: incorrect handling of pointers one past the end

Incorrect pointer comparison to NULL

In CompCert:

- pointers are pairs (b, o)
- the NULL pointer is represented as the integer 0

`p == 0` was incorrectly defined to always evaluate to `false` when `p` is a pointer.



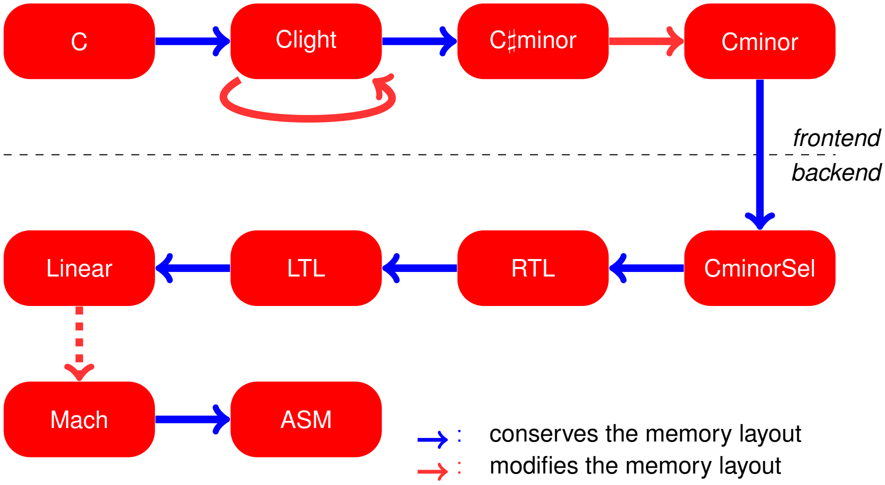
But we need to check that o is a valid offset of b

- $[\![(b, o)]\!]_{cm} = cm(b) + o$ is not zero only in that case
- otherwise $(b, -8)$ evaluates to zero

Outline

1. CompCert's memory model
2. New features of the memory model
3. Consistency of the memory models
4. CompCert proof: Overview
5. Conclusion

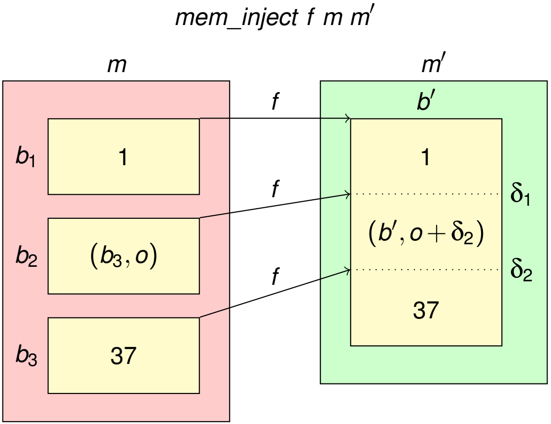
Overview of CompCert architecture



Memory injections: a generic memory transformation

In CompCert C, each local variable has its own block.

During the compilation these variables are merged into a stack frame.



Adapting to symbolic expressions:

- generalization of the injection over values
- lots of proofs to adapt (relation with normalisation)

Memory injections - Central theorem

```
Theorem norm_inject: ∀ f m m' e e'
  (Minj: inject f m m') (Einj: expr_inject f e e'),
 val_inject f (normalise m e) (normalise m' e').
```

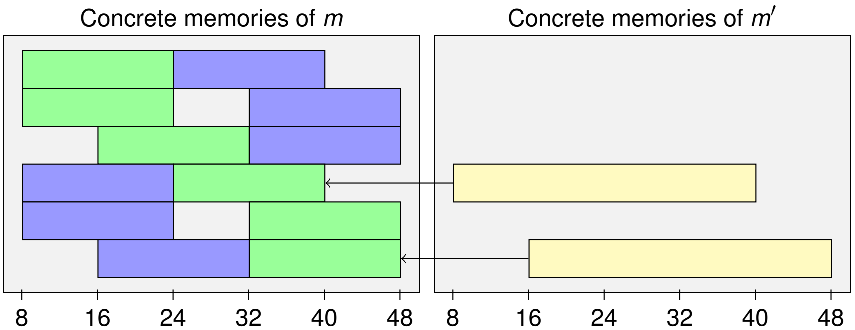
- We can show that: $\exists v, val\_inject\ f\ (normalise\ m\ e)\ v$
- Let's now prove that: $normalise\ m'\ e' = v$
- $\forall cm' \vdash m', [\![e']\!]_{cm'} = [\![v]\!]_{cm'}$
- From the specification of the normalisation of *e* in *m* we know:

$$\forall cm \vdash m, [\![e]\!]_{cm} = [\![normalise\ m\ e]\!]_{cm}$$

- We need a theorem relating evaluations in *cm* and *cm*′!

Memory injections - Evaluation

`mem_inject f m m′`



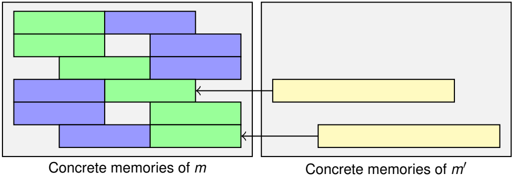
`pre_cm(f,cm′)`: recovers a concrete memory as it was before injection

```
Definition pre_cm f cm' := fun (b: block) ⇒
  let (b', delta) := f b in  cm' b'  + delta.
```

```
Theorem expr_inject_eval: ∀ f cm' e e'
      (Einj: expr_inject f e e'),
      ⟦e'⟧ cm′ = ⟦e⟧ pre_cm(f,cm′).
```

Memory injections - Central theorem

```
Theorem norm_inject: ∀ f m m' e e'
  (Minj: inject f m m') (Einj: expr_inject f e e'),
 val_inject f (normalise m e) (normalise m' e').
```



`expr_inject_eval` :
$expr\_inject\ f\ e\ e' \Rightarrow$
$[\![e']\!]_{cm'} = [\![e]\!]_{pre\_cm(f,cm')}$

- We are left to prove:

$$\forall cm' \vdash m', [\![e']\!]_{cm'} = [\![v]\!]_{cm'}$$

- We rewrite both sides using `expr_inject_eval`, the goal becomes:

$$\forall cm' \vdash m', [\![e]\!]_{pre\_cm(f,cm')} = [\![normalise\ m\ e]\!]_{pre\_cm(f,cm')}$$

- From the specification of the normalisation of e in m we know:

$$\forall cm \vdash m, [\![e]\!]_{cm} = [\![normalise\ m\ e]\!]_{cm}$$

which solves our goal.

□

Outline

Conclusion

A semantics for C

- more precise than CompCert's
- compatible with CompCert
- *nearly* as proven correct as CompCert

Future directions

- finish the proof by adapting the last remaining unproven pass
- add a more concrete assembly language to the certified compilation chain
- plug back in optimizations at RTL level (precision improvement?, still sound?)

Questions?