CompCertS: a Memory-Aware Verified C Compiler using Pointer as Integer Semantics

Frédéric Besson, Sandrine Blazy, **Pierre Wilke**

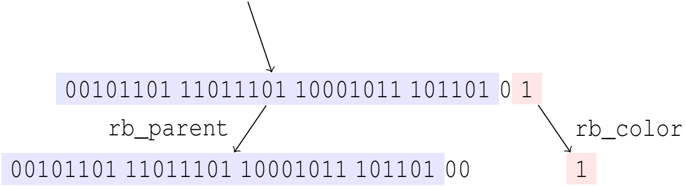
Université Rennes 1

Yale University

ITP 2017 – September 26$^{\mathrm{th}}$, 2017

Linux Red-Black Trees: `include/linux/rbtree.h`

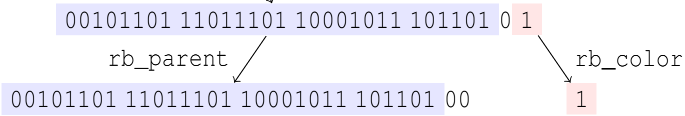
```c
struct rb_node {
  struct rb_node *rb_right;
  struct rb_node *rb_left;
  uintptr_t rb_parent_color;
};
```




```c
rb_node * rb_parent(rb_node * r) {
  return ((rb_node *) (r -> rb_parent_color & ~3));
}
void rb_set_parent_color(rb_node * rb, rb_node * p, int color) {
  rb->rb_parent_color = p | color;
}
```

Linux Red-Black Trees: include/linux/rbtree.h

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




```c
rb_node * rb_parent(rb_node * r) {
  return ((rb_node *) (r -> rb_parent_color & ~3));
}
void rb_set_parent_color(rb_node * rb, rb_node * p, int color) {
  rb->rb_parent_color = p | color;
}
```

Undefined behavior

Linux Red-Black Trees: include/linux/rbtree.h

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

Formal guarantees?

00101101 11011101 10001011 101101 0 1

rb_parent

00101101 11011101 10001011 101101 00

rb_color

1

```
rb_node * rb_parent(rb_node * r) {
  return ((rb_node *) (r -> rb_parent_color & ~3));
}
void rb_set_parent_color(rb_node * rb, rb_node * p, int color) {
  rb->rb_parent_color = p | color;
}
```

Undefined behavior

CompCert: a formally verified C compiler

Leroy, “Formal verification of a realistic compiler”, CACM’2009.

Theorem (CompCert’s theorem)

Let P be a C program.
If P has ***defined semantics*** *,*
if CompCert successfully generates an assembly program P',
then P' ***behaves as*** *P.*

Unfortunately, the red-black tree example does not have **defined semantics**.
Can we achieve a similar result for this program?

Previous work: a relaxed semantics for low-level C programs

Symbolic Memory Model: Frédéric Besson, Sandrine Blazy, and Pierre Wilke. "A Precise and Abstract Memory Model for C Using Symbolic Values." In: *APLAS*. 2014

Front-end of the compiler: Frédéric Besson, Sandrine Blazy, and Pierre Wilke. "A Concrete Memory Model for CompCert". In: *ITP*. 2015

Features:

- defined semantics for bitwise manipulation of pointer values
  - use **symbolic values** to represent undefined computations
- finite memory
  - the allocation of memory fails when full

CompCertS: a formally verified compiler for low-level code

Contributions of this work: whole compiler proof with the symbolic semantics

- proofs of correctness of most compiler passes of CompCert
  - *(no inlining or tailcall optimizations)*
- **preservation of the absence of memory overflows**
- **formal safeguard against over-aggressive optimizations**

Outline

1. CompCert and Previous Work on Symbolic Values

2. Compiler proofs: Finite memory

3. Compiler proofs: Optimizations

CompCert's memory model

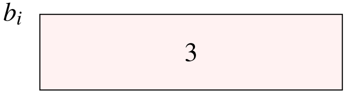
Memory is a collection of **blocks**.

⇒
```
int i = 3;
int * p = &i;
uintptr_t x = p | 1;
int * q = p & ~1;
assert ( p == q );
```

CompCert's memory model

Memory is a collection of **blocks**.

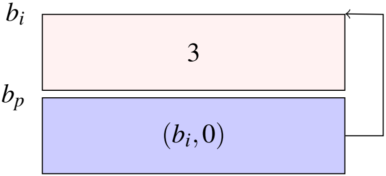
```
  int i = 3;
⇒ int * p = &i;
  uintptr_t x = p | 1;
  int * q = p & ~1;
  assert ( p == q );
```



CompCert's memory model

Memory is a collection of **blocks**.
Locations are pairs (b, o) where b is a block identifier, and o is an offset.

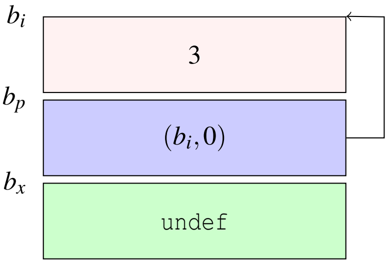
```
  int i = 3;
  int * p = &i;
⇒ uintptr_t x = p | 1;
  int * q = p & ~1;
  assert ( p == q );
```



CompCert's memory model

Memory is a collection of **blocks**.
Locations are pairs (b, o) where b is a block identifier, and o is an offset.

```
  int i = 3;
  int * p = &i;
⇒ uintptr_t x = p | 1;
  int * q = p & ~1;
  assert ( p == q );
```

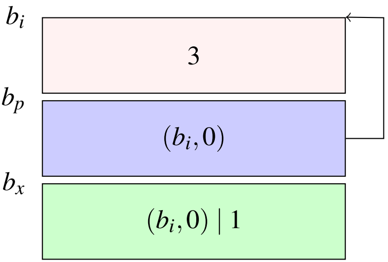


Not captured by most existing formal semantics for C (Cholera, KCC, CH_20).
Captured by Kang et al.'s semantics.

Overcoming CompCert's limitations: symbolic values

Frédéric Besson, Sandrine Blazy, and Pierre Wilke. "A Precise and Abstract Memory Model for C Using Symbolic Values." In: *APLAS*. 2014

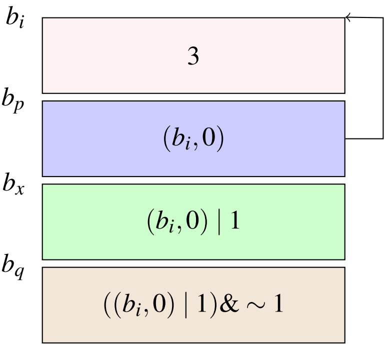
```
  int i = 3;
  int * p = &i;
  uintptr_t x = p | 1;
⇒ int * q = p & ~1;
  assert ( p == q );
```



Overcoming CompCert's limitations: symbolic values

Frédéric Besson, Sandrine Blazy, and Pierre Wilke. "A Precise and Abstract Memory Model for C Using Symbolic Values." In: *APLAS*. 2014

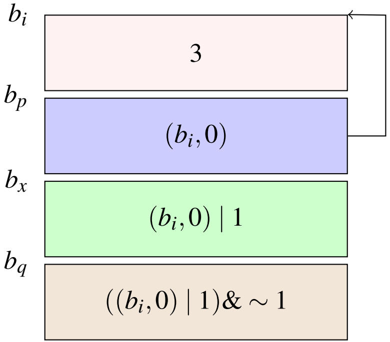
```
  int i = 3;
  int * p = &i;
  uintptr_t x = p | 1;
  int * q = p & ~1;
⇒ assert ( p == q );
```



Overcoming CompCert's limitations: symbolic values

Frédéric Besson, Sandrine Blazy, and Pierre Wilke. "A Precise and Abstract Memory Model for C Using Symbolic Values." In: *APLAS*. 2014

```
   int i = 3;
   int * p = &i;
   uintptr_t x = p | 1;
   int * q = p & ~1;
⇒ assert ( p == q );
```

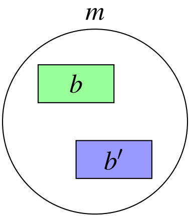


$$(b_i, 0) == ((b_i, 0) \mid 1) \& \sim 1 \xrightarrow{\texttt{normalize}} 1$$

Normalization: semantics

$$\texttt{normalize} : mem \rightarrow sval \rightarrow val$$

$$sv = (b,0) {==} ((b,0) \mid 1) \& \sim 1$$

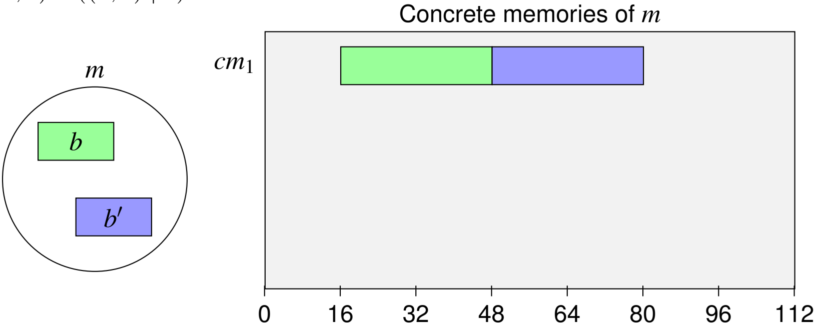


We expect that $\texttt{normalize}\ m\ sv\ =\ 1$

Normalization: semantics

$$\texttt{normalize} : mem \rightarrow sval \rightarrow val$$

$sv = (b,0) == ((b,0) \mid 1) \& \sim 1$



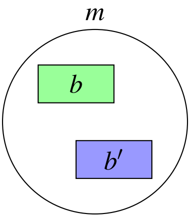
We expect that $\texttt{normalize}\ m\ sv\ =\ 1$

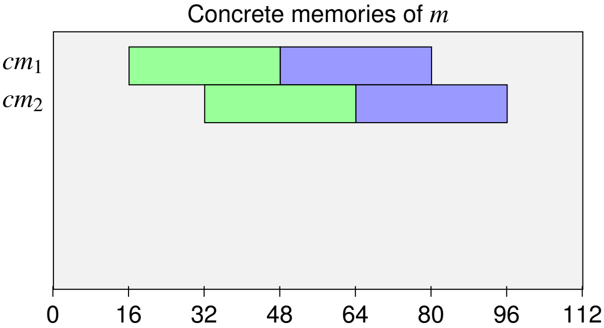
$[\![sv]\!]_{cm_1} \quad = \quad 16 == (16 \mid 1) \& \sim 1 \quad = \quad 16 == 16 \quad = \quad 1$

Normalization: semantics

$$\texttt{normalize} : \mathit{mem} \to \mathit{sval} \to \mathit{val}$$

$sv = (b,0) {==} ((b,0) \mid 1) \& \sim 1$





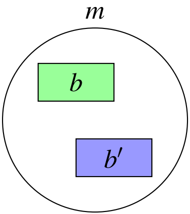
We expect that $\texttt{normalize}\ m\ sv = 1$

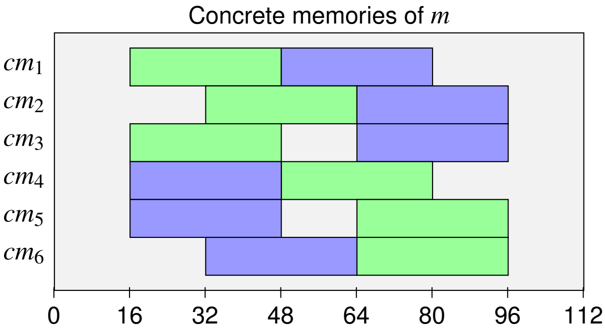
$$\llbracket sv \rrbracket_{cm_1} = 16{==}(16 \mid 1)\& \sim 1 = 16{==}16 = 1$$
$$\llbracket sv \rrbracket_{cm_2} = 32{==}(32 \mid 1)\& \sim 1 = 32{==}32 = 1$$

Normalization: semantics

$$\texttt{normalize} : mem \rightarrow sval \rightarrow val$$

$sv = (b,0) == ((b,0) \mid 1) \& \sim 1$



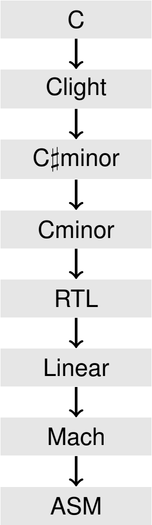


We expect that $\texttt{normalize}\ m\ sv\ =\ 1$

$$[\![sv]\!]_{cm_1} = 16 == (16 \mid 1) \& \sim 1 = 16 == 16 = 1$$
$$[\![sv]\!]_{cm_2} = 32 == (32 \mid 1) \& \sim 1 = 32 == 32 = 1$$

$$\forall cm,\ [\![sv]\!]_{cm} = 1 \quad \Rightarrow \quad \texttt{normalize}\ m\ sv = 1$$

CompCertS overall architecture



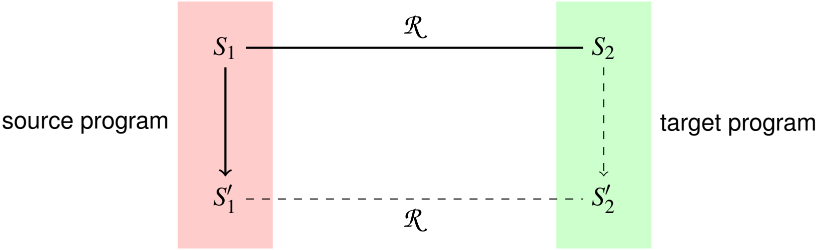
- use symbolic values instead of values
- introduce calls to normalization at:
  - memory accesses
  - conditionals
- adapt the proof of semantic preservation for each pass

Proofs of semantic preservation: simulation relations

Each compiler pass is proved **semantics preserving** using simulation relations.

Theorem (Forward simulation)

Every semantic step in the source program can be ***simulated*** *by a sequence of steps in the target program.*



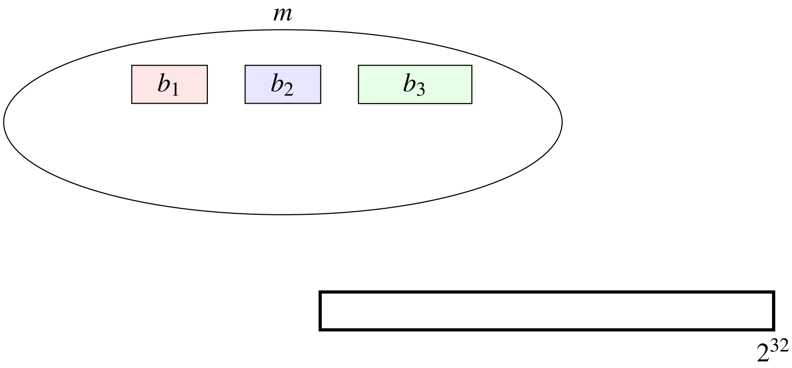
All such preservation theorems are eventually composed into a preservation theorem from C to assembly.

2. Compiler proofs: Finite memory

The need for finite memory

Because we map (an unbounded set of) blocks onto a (finite) address space, we model a **finite memory**.

For every memory m, there must exist a concrete memory.

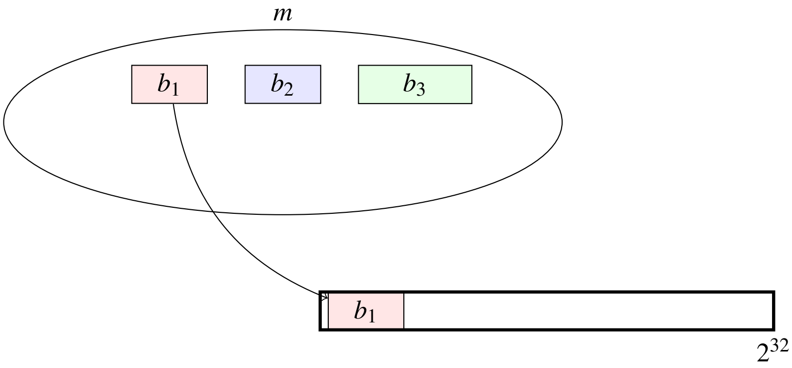


As a result, memory allocation **fails** when the memory is full

The need for finite memory

Because we map (an unbounded set of) blocks onto a (finite) address space, we model a **finite memory**.

For every memory m, there must exist a concrete memory.

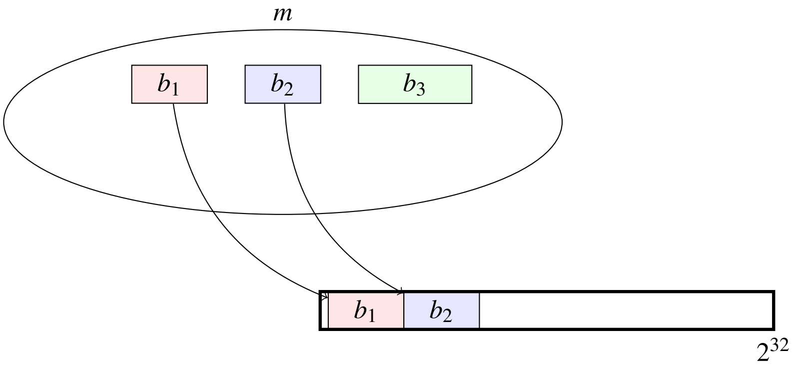


As a result, memory allocation **fails** when the memory is full

The need for finite memory

Because we map (an unbounded set of) blocks onto a (finite) address space, we model a **finite memory**.

For every memory m, there must exist a concrete memory.

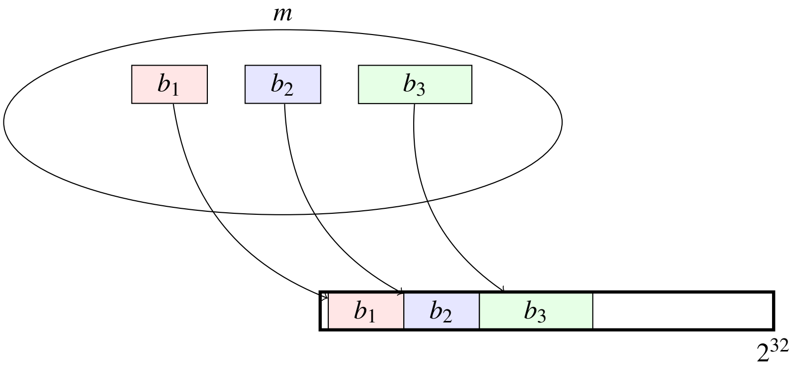


As a result, memory allocation **fails** when the memory is full

The need for finite memory

Because we map (an unbounded set of) blocks onto a (finite) address space, we model a **finite memory**.

For every memory m, there must exist a concrete memory.

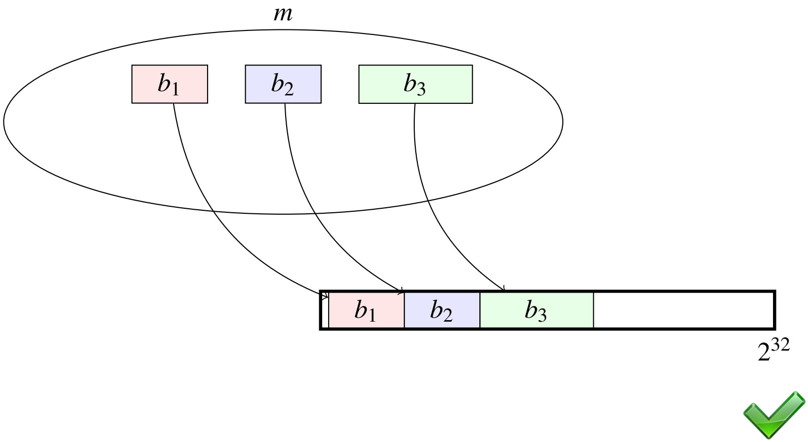


As a result, memory allocation **fails** when the memory is full

The need for finite memory

Because we map (an unbounded set of) blocks onto a (finite) address space, we model a **finite memory**.

For every memory m, there must exist a concrete memory.

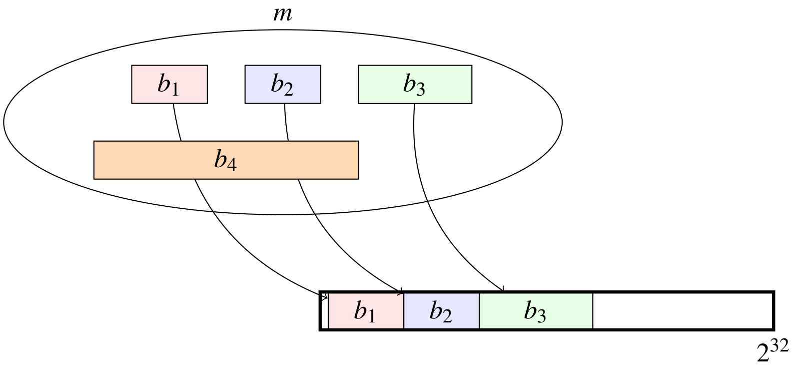


As a result, memory allocation **fails** when the memory is full

The need for finite memory

Because we map (an unbounded set of) blocks onto a (finite) address space, we model a **finite memory**.

For every memory m, there must exist a concrete memory.

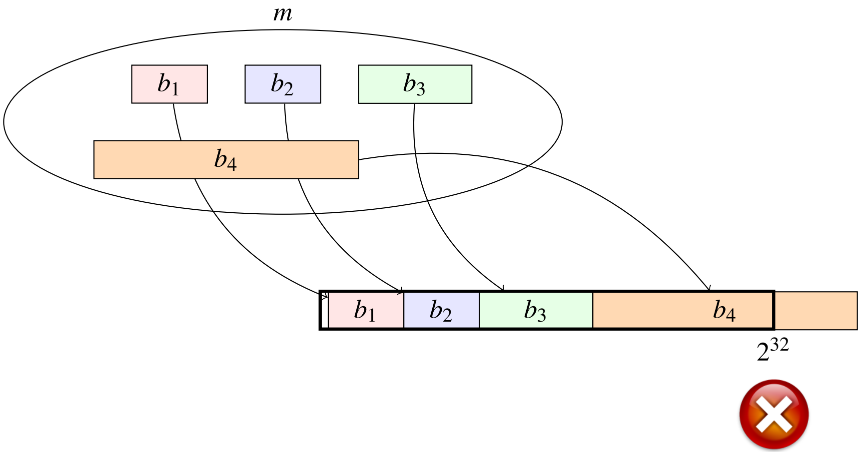


As a result, memory allocation **fails** when the memory is full

The need for finite memory

Because we map (an unbounded set of) blocks onto a (finite) address space, we model a **finite memory**.

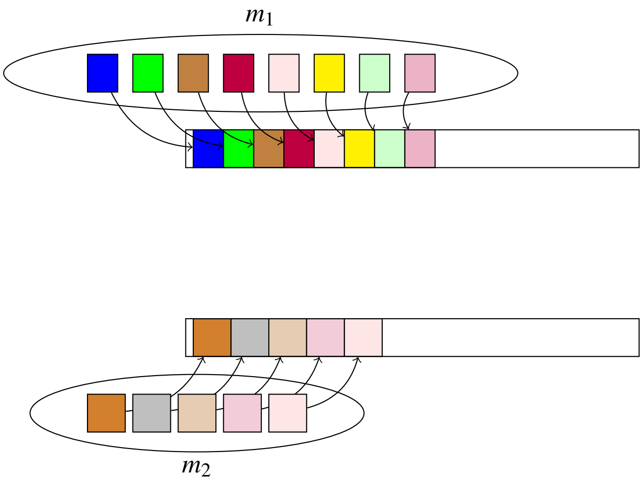
For every memory m, there must exist a concrete memory.



As a result, memory allocation **fails** when the memory is full

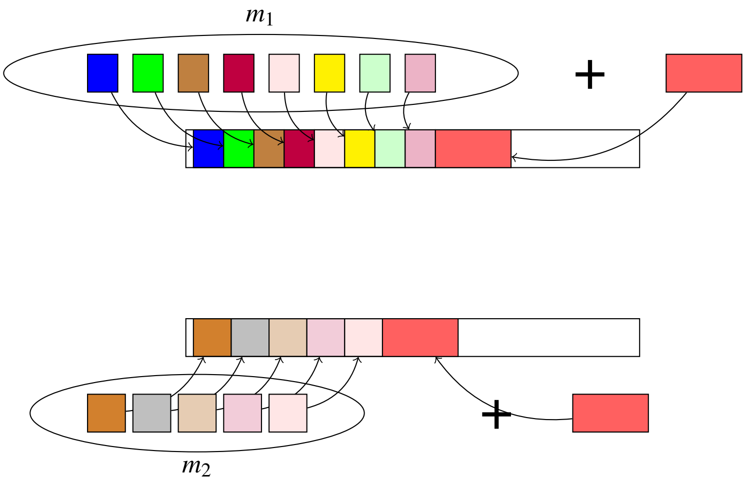
Preservation of the absence of memory overflows

If the allocation of a block succeeds before the transformation,
then it also succeeds after.



Preservation of the absence of memory overflows

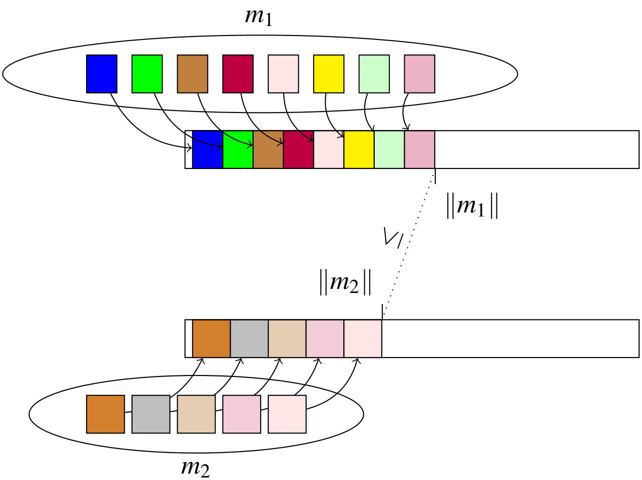
If the allocation of a block succeeds before the transformation,
then it also succeeds after.



Decreasing memory size: invariant

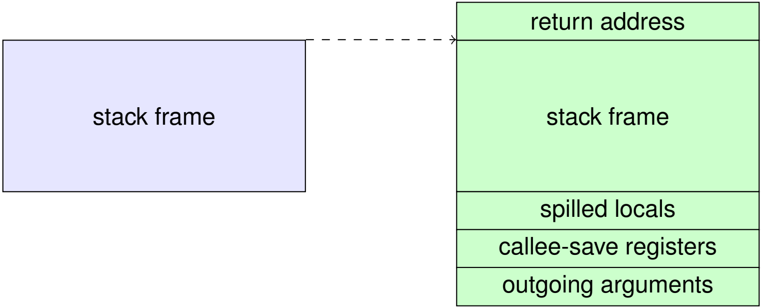
For every compiler pass that transforms memory state m_1 into m_2:

$$\|m_2\| \leq \|m_1\|$$



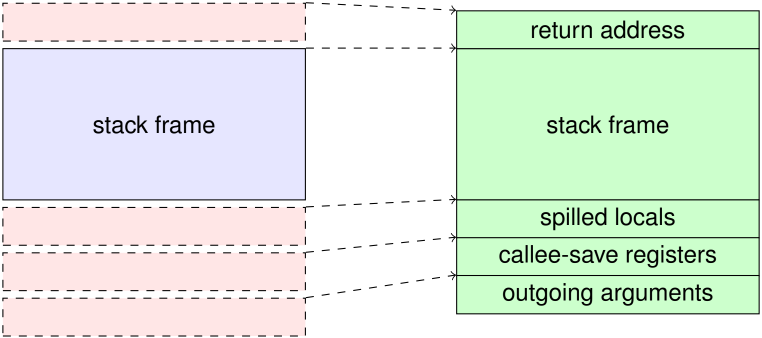
$\Rightarrow$ preservation of the absence of memory overflows

Problem: the `Stacking` pass



This compiler pass makes memory usage **grow**.

Problem: the `Stacking` pass

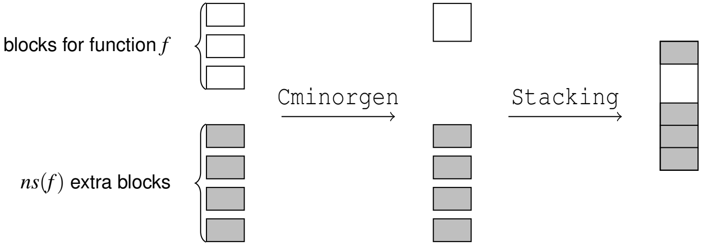


This compiler pass makes memory usage **grow**.

Solution: we preallocate for every function additional memory

Parameterizing the semantics with oracles for finite memory

Semantics are parameterized with **oracles** ns : `fn_name` $\rightarrow \mathbb{Z}$



The compiler outputs such an oracle.

New semantic preservation theorem

Theorem (`transf_c_program_preservation`)

Let P be a C program.
If CompCert successfully generates an assembly program P' and an oracle ns,
*If P has **defined semantics** with oracle ns,*
*then P' **behaves as** P.*

This new theorem gives us the additional guarantee that for well-defined C programs, the compiled program will not run out of memory.

3. Compiler proofs: Optimizations

Compiler optimizations: constant propagation

```
int main(){
  int x = 1;
  //
  uintptr_t p = &x >> 1;
  //
  f(p);
  //
  return x;
}
```

It is sound to optimize the return statement into `return 1;` in CompCert

Compiler optimizations: constant propagation

```
int main(){
  int x = 1;
  // [x ↦ 1]
  uintptr_t p = &x >> 1;
  //
  f(p);
  //
  return x;
}
```

It is sound to optimize the return statement into `return 1;` in CompCert

Compiler optimizations: constant propagation

```
int main(){
  int x = 1;
  // [x ↦ 1]
  uintptr_t p = &x >> 1;
  // [x ↦ 1; p ↦ Cst]
  f(p);
  //
  return x;
}
```

It is sound to optimize the return statement into `return 1;` in CompCert

Compiler optimizations: constant propagation

```
int main(){
  int x = 1;
  // [x ↦ 1]
  uintptr_t p = &x >> 1;
  // [x ↦ 1;p ↦ Cst]
  f(p);
  // [x ↦ 1;p ↦ Cst]
  return x;
}
```

It is sound to optimize the return statement into `return 1;` in CompCert

Compiler optimizations: pointer dependence

In our symbolic semantics:

```
int main(){
  int x = 1;
  // [x ↦ 1]
  uintptr_t p = &x >> 1;
  // [x ↦ 1;p ↦ dep(&x)]
  f(p);
  //
  return x;
}
```

We enrich the abstract domain: $dep(\&x)$

- symbolic values from which a pointer to `x` may be derived

Because our semantics are more permissive, our optimizations are more conservative.

Compiler optimizations: pointer dependence

In our symbolic semantics:

```
int main(){
  int x = 1;
  // [x ↦ 1]
  uintptr_t p = &x >> 1;
  // [x ↦ 1;p ↦ dep(&x)]
  f(p);
  // [x ↦?;p ↦ dep(&x)]
  return x;
}
```

We enrich the abstract domain: $dep(\&x)$

- symbolic values from which a pointer to `x` may be derived

Because our semantics are more permissive, our optimizations are more conservative.

Compiler optimizations – conclusion

In the existing CompCert, optimizations are written with prudence in order to avoid counterintuitive behaviors.

Our symbolic semantics provide a **formal safeguard** to avoid those “miscompilations”.

Conclusion

CompCertS: a Memory-Aware Verified C Compiler using Pointer as Integer Semantics

- formal guarantees on the memory consumption of programs
  - the compiler does not introduce memory overflow
- formal guarantees for programs that perform bitwise operations on pointers
  - optimizations are more conservative, in a formal way

Possible applications:

- formal verification of system code (Linux red-black trees, implementations of `malloc`)
- formal verification of obfuscations (variable splitting)
- software fault isolation (masking pointers)