Memory bijections: reasoning about exact memory transformations induced by refactorings in CompCert C

Igor Zhirkov, Julien Cohen, Rémi Douence

Memory bijections: reasoning about exact memory transformations induced by refactorings in CompCert C

Igor Zhirkov[1,3] Julien Cohen[2,3] Rémi Douence[1,3]

March 2019

1: Institut Mines Telecom
2: Université de Nantes
3: LS2N – INRIA

Abstract

This document reports on our work in extending CompCert memory model with a relation to model relocations. It preserves undefined values unlike similar relations defined in CompCert. This relation commutes with memory operations. Our main contributions are the relation itself and mechanically checked proofs of its commutation properties. We intend to use this extension to construct and verify a refactoring tool for programs written in C.

Contents

1 Introduction **2**

2 Language semantics and proofs of correctness **4**
2.1 Non-determinism in C 4
2.2 Compilation and refactoring 4

3 CompCert memory model **7**
3.1 Overview 7
3.2 Memory contents 8
3.3 Permissions 11
3.4 High-level abstract model 12
3.5 Axiomatic memory semantics 15

4 Memory injections in CompCert **16**
4.1 Injection usage during compilation 16
4.2 Defining memory injection 17
4.3 Formal definition 21
4.4 Commutation properties 23

5 Memory bijection **28**
5.1 Inverse embedding functions 29
5.2 Relation on memories 30
5.3 Relation on expression values 31
5.4 Relation on memory values 31
5.5 Properties 32
5.6 Usage 40

6 Related work **42**
6.1 Refactoring correctness 42
6.2 Formal C semantics 44
6.3 Extensions to CompCert memory model 47

7 Conclusion **48**

1 Introduction

A refactoring is a modification of the source-code of a program which does not alter its observable behavior. It usually tends to improve some non-functional code characteristics of that program, in particular its maintainability. Renaming functions, renaming variables, moving functions between modules are common refactorings, but there are tens of them that are identified (see for instance [11]).

Very often, a refactoring does not change the *observable* behavior, but changes the way the program produces this behavior. Consider for instance the removal of an usused local variable in a function. If a function `f` contains a variable `x` that is never used, that variable can be safely removed without altering the result of the program (Fig. 1). The two versions of that function will give the same result, but the second one is more readable and can use less memory at execution time.

```
void f() {
  int x;
  int y = 42;
}
```

(a) The function `f` in the initial program p

```
void f() {

  int y = 42;
}
```

(b) The function `f` in the refactored program p'

Figure 1: Removing an unused variable `x`.

Checking that a refactoring does not change the observable behavior of a program is generally difficult. Automatic refactoring tools are well known to be unreliable [34, 26]. Indeed, it is hard to craft a reliable tool which works with general-purpose languages such as C or Java: such industrial-scale languages have complex semantics, described on hundreds of pages of standards ([4, 12]) with a large number of interactions between different language features that are not well anticipated by tool developpers and testers.

For this reason, we are interested in proving the correctness of such tools, or, more precisely, in building tools that we can prove correct.

To be able to prove that two programs have the same behavior we need a formalization of the underlying programming language. Here we rely on CompCert C [22] which provides a formal semantics of a large subset of C mechanized in Coq. CompCert C covers the features of C99 language standard, except for a couple of rarely used ones: `longjmp`/`setjmp` and unstructured `switch` [1]. This subset covers the industrial standard MISRA C.

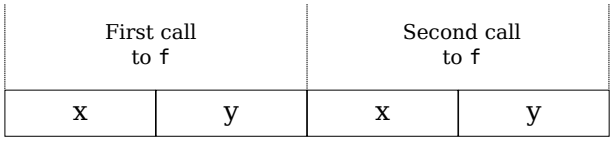
CompCert describes program semantics as a state transition system. It is defined by an operational semantics where small-step transitions denote the reduction of an expression or the execution of a statement. Some transitions issue an observable event, such as a system call. The observable behavior for a program execution then boils down to a sequence of observable events. We call those sequences *traces*.

Compcert C semantics also accounts for the non-determinism of C programs: a given program has a set of possible behaviors.

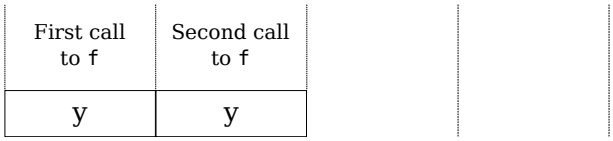
The concept of semantic preservation, the preservation of the observable behavior, is not well defined in the refactoring community because of the multiplicity of the practices. Here we simply aim at ensuring that two programs have the same set of possible behaviors and, so, that they can produce the same set of traces. This is refered to as *bisimulation* in [20].

Such a bisimulation has been formally established for a tool that can rename some global variables [6]. But the renaming of variables does not result in changes in the memory layout at execution time, so the bisimulation proofs in that work does not deal with changes in memory.

In fact, few refactoring operations do not alter the memory layout. Let us come back to the refactoring that removes an used local variable in a function. While syntactically removing an unused local variable is local to a function, it has an impact on the whole memory during program execution. Allocating one less local variable in a function (say `f`) makes corresponding memory states of the initial program (say p) and the modified program (say p') diverge at each call of that function. Fig. 2 shows memory states of programs p and p' of Fig. 1 after `f` is called twice. When executing p, each call to `f` allocates two blocks for the instances of the local variables `x` and `y`. When executing p', each call only allocates one block for `y`.



(a) Memory state for p.



(b) Memory state for p'.

Figure 2: Corresponding memory states for p and p' executions.

CompCert provides a tool to deal with memory relocation called *memory injection* (Sec. 4). It allows to relate values in program execution traces before and after refactoring but it specifies only the preservation of defined values. Whereas a compiler can replace an undefined behavior by any behavior, we are interested in preserving undefinedness to ensure that a refactoring does not change the set of possible traces. So we cannot directly reuse memory injection in proofs of correctness of refactorings (Sec. 2.2.2).

Our goal in this report is to provide a tool similar to memory injection but that preserves undefined values. To that end, we craft a stronger relation by associating two inverse memory injections (Sec. 5). We call this *memory bijection*. We also give simulation properties on high-level memory operators for memories that are in bijection (Sec. 5.5). Then we explain in section 5.6 how this relation and its properties can be used in proofs of correctness for refactoring operations.

Our results are mechanically checked in Coq; the code is shipped with this report.

2 Language semantics and proofs of correctness

In this section we recall that a C program can have many behaviors and how strictly compilers and refactoring tools preserve them. The differences in the concepts of behavior preservation explains why we are not reusing as-is the relation between memories routinely used in CompCert proofs (memory injection).

2.1 Non-determinism in C

The C language is non-deterministic: several behaviors are possible for a C program. A compiler will select a single behavior for the compiled program. There are three kinds of non-determinism in C:

Unspecified behavior. In case of unspecified behavior the compiler gets a set of possible behaviors to chose from. It cannot chose an arbitrary behavior outside of this set. For example, the following program can output either `"ab"` or `"ba"` because the order in which the operator + evaluates its operands is not specified by the semantics of the language.

```
void main() {
    printf("a") + printf("b");
}
```

Both orders are valid, it is up to compiler to select one. Using a different compiler or changing some compilation options can lead to a different behavior being selected.

Implementation-defined behavior. In case of implementation-defined behavior the compiler selects a behavior from a set of possible behaviors and explicitly documents it. This choice may often be altered by using specific compiler options. For example, the size of `int` type varies between compilers; the compiler might be asked to select a different data model with a different `int` size.

Undefined behavior. In case of undefined behavior the compiler is free to silently assign any behavior to the program. For example, freshly-allocated dynamic memory has undefined contents; it means, that reading such memory can lead to any result.

2.2 Compilation and refactoring

Compilation and refactoring are transformations that bear a similarity: both are static code transformations that preserve semantics, but they preserve it in different ways. There are multiple concepts of behavior preservation, such as forward simulation, backward simulation and bisimulation [20]. Compilers for non-deterministic languages into assembly can not preserve semantics in a sense of bisimulation because they have to select a single behavior from multiple ones; they use weaker notions of semantic preservation such as backward simulation. On the other side, refactorings are generally expected to preserve program semantics in a bisimulation sense, preserving the whole set of possible behaviors.

2.2.1 Compilation correctness

To certify its correctness, CompCert proves that for any input C programs the program semantics, i.e. its external behavior, is preserved: the behavior of the executable program conforms to one of the possible behaviors of the source program (backward simulation). Recall that an external behavior is a sequence of events, a trace, describing volatile reads/writes and system calls occuring during execution.

CompCert transforms programs through several stages down to assembly level. The first stage is a translation from C into *Clight*, an intermediate language with similar semantics but which is deterministic. The behavior of a Clight program is proven to be one of possible behaviors of the corresponding C program. Following stages are transforming a Clight program into assembly code with an equivalent behavior. The behavior of the assembly program will thus correspond to one of the possible behaviors of the source C program [2].

CompCert proves two types of correctness theorems for all input programs:

- The behavior of the output assembly program is contained in the set of behaviors of the input C program. This is a backward simulation.
- The behavior of the assembly program matches exactly the behavior of the Clight program (the C program with a fixed order of execution). This is a bisimulation.

Proving backwards simulation between C and assembly. In CompCert, the execution of all programs is modeled as state transition systems; a small-step semantics is used. The assembly program is deterministic, so its execution can be represented as a sequence of states $s'_0, s'_1, \ldots$ (Fig. 3).

In order to prove that the behavior of an assembly program is contained in the set of behaviors of a C program, we have to provide a corresponding sequence of C program states $s_0, s_1, \ldots$ with transitions between them (Fig. 3). The initial memory states s_0 for a the C program and s'_0 for the assembly program should be related by a so-called simulation relation $\mathcal{R}$, which persists when a transition allowed by semantics occurs. Between two pairs of corresponding states, the sequences of transitions `step*` and `asm_step*` must trigger the same observable events.

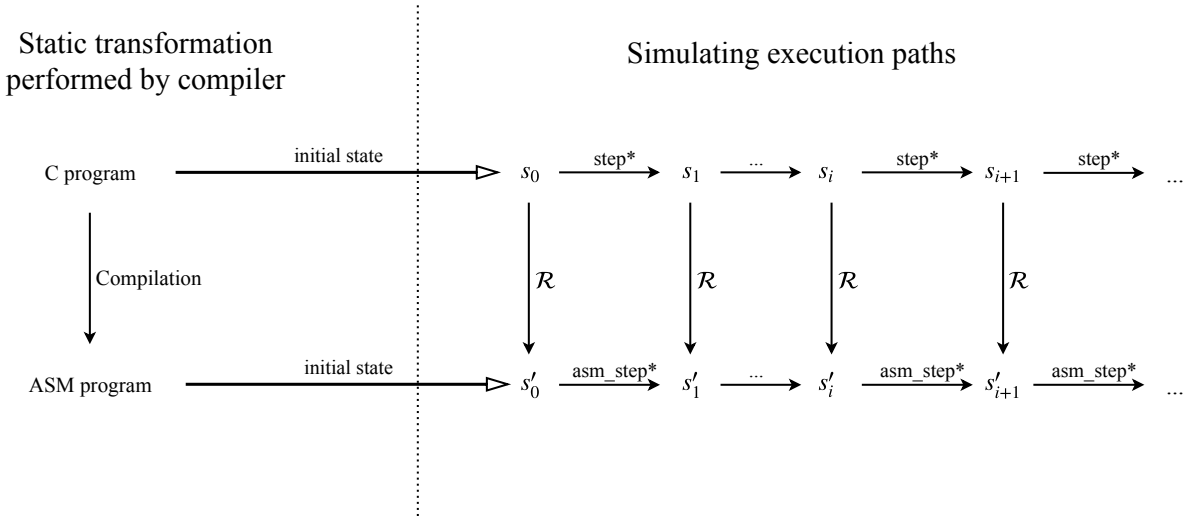


Figure 3: Proving backward simulation between C and assembly program for a compiler.

The relation $\mathcal{R}$ connects all parts of abstract machine states s_i and s'_i including their memories. In practice, the compilation is performed in multiple passes and different passes relate memories in a different way. *Memory injection* is such a relation and is frequently used in CompCert. It models data relocation: some data might be moved to different addresses and the pointers to these data are changed accordingly. Memory injection is detailed in section 4.

2.2.2 Refactoring correctness

A refactoring tool also performs a static code transformation. To certify its correctness we prove that for all input programs the sequences of events in that program are the same as in refactored program. This proof follows the same scheme as the proof of correctness of the compilation; it also has to build a relation $\mathcal{R}$ to connect the corresponding states s_i and s'_i (Fig. 4). Refactorings correctness corresponds to a bisimulation semantic preservation property.

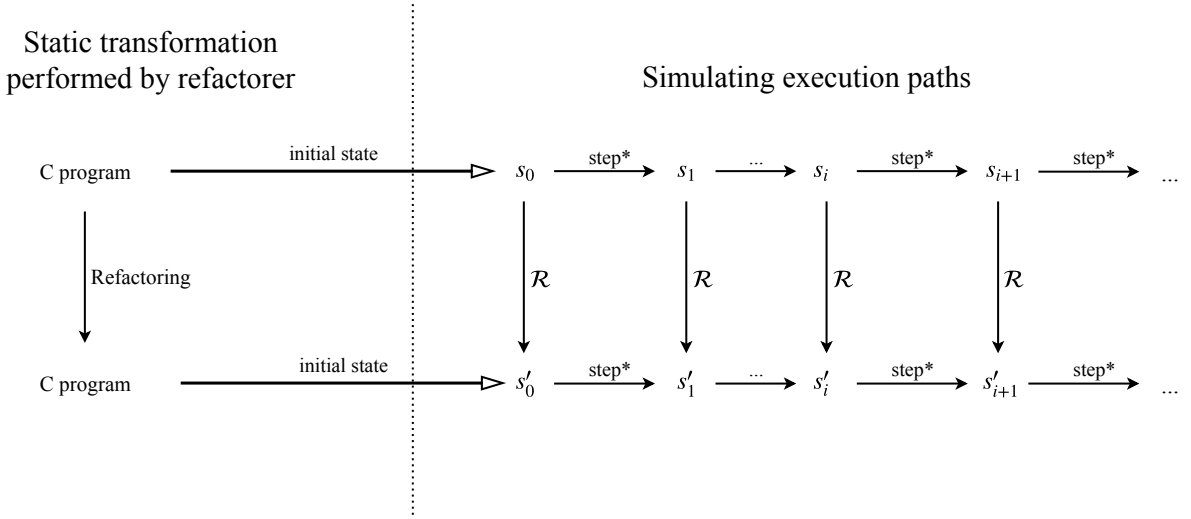


Figure 4: Proving bisimulation between two C programs for a refactoring.

At some point, we have to encode a bisimulation relation R between the memory in s_i and the memory in s'_i. The nature of our key example of refactoring – removing an unused local variable – is that some blocks that exist in s_i will not exist in s'_i. The other blocks are relocated.

Reusing memory injection as-is is not sufficient for our needs. Memory injection seems like a close fit for the refactoring correctness proof because:

- The refactoring correctness proof resembles the compilation correctness proof: both of them use bisimulation;
- Memory in refactored program looks like memory in source program with relocations and with some blocks discarded.

However, memory injection as defined by CompCert only distinguishes memory blocks that contain defined values, which is not strong enough for our purposes. Let us see more precisely the problem. Suppose we have two memory states with one block each (coming for example from programs p_1 and p_2 of Fig. 5):

- m_1 has a block containing an undefined value,
- m_2 has a block of the same size containing an integer value `42`.

While m_2 can not be obtained from m_1 by choosing different indexes for its blocks, m_1 and m_2 are still related by memory injection. This relation allows undefined values to be projected into any value so a block with an undefined value can be mapped into the block with value `42`.

```
void main() {
  int x;
  print(x);
}
```

(a) Program p_1

```
void main() {
  int x = 42;
  print(x);
}
```

(b) Program p_2

Figure 5: Example of refinement. In p_1, the local variable `x` is not initialized so it can contain any value.

Now, let us compare the possible behaviors of the programs p_1 and p_2. The program p_1 can legally output any value, and it will be a correct behavior. In p_2, the variable `x` is explicitly initialized to `42` so it makes for only one valid behavior: this program should output `42`.

The possible traces for these programs are thus:

1. `[ call:Print <any value> ]` (an infinite number of correct behaviors)
2. `[ call:Print 42 ]`

From the compiler's point of view, it is correct to pass from the first program to the second one (Sec. 2.1). But we consider that refactorings should preserve all behaviors so undefined values should be preserved too.

This fact makes memory injection too loose for us to extract enough information for a refactoring correctness proof; this is the reason we define another relation, memory bijection (Sec. 5). That relation will guarantee that undefined values are preserved, enabling us to preserve all behaviors in the refactored program.

Before presenting our memory bijection, we recall how memory works in CompCert (Sec. 3) and then how memory injection works (Sec. 4) since we build memory bijection on top of memory injection.

3 CompCert memory model

The translation process in CompCert incorporates multiple stages, all of which share a single memory model. In this section, we describe this model in necessary details. From this point onward, listings will either be showing Coq or C code.

3.1 Overview

In the source code of CompCert, the record describing memory looks as follows:

```
(* File: compcert.common.Memory *)
Record mem := {

  mem_contents : Map ℤ (Map ℤ memval);
  mem_access   : Map ℤ (ℤ → perm_kind → option permission);
  nextblock    : ℤ ;

  access_max : ...
  nextblock_noaccess : ...
  contents_default : ...
}.
```

We are interested in three fields of this record:

- `mem_contents` stores the actual bytes in memory. We explore how values are encoded, decoded, stored, and loaded from memory in in Section 3.2.
- `mem_access` stores a permission map that associates two permission levels to each allocated byte. We explain the permissions mechanism in Section 3.3.
- `nextblock` is an integer showing the next available block index. Memory allocation relies on it to index the fresh block. See Section 3.4.3.

The last three fields contain proofs of invariants that are required to construct a valid instance of `mem`. We never construct new instances of `mem` in our work so these properties are not relevant to us.

3.2 Memory contents

The first field of `mem` record is `mem_contents` which is a map from block indices (integers) into blocks. A block maps offsets (integers) into bytes.

An exemplary instance of `mem_contents` is shown on Figure 6. We adopt this notation to illustrate the contents of a particular memory instance throughout this document. Each byte is modeled using `memval` type, as described in Section 3.2.2.

There is no distinction between heap, stack and other kinds of memory regions: all blocks are allocated and treated alike.

In this memory model, an address is a pair of integers representing block index and offset inside block. Fetching a single byte from memory consists of two parts:

- We use the block index to get a block from `mem_contents`. The result is a map from $\mathbb{Z}$ to `memval`.
- We use the offset inside block to get a `memval` instance from this map.

The two-component nature of pointers in this model makes blocks separated by construction. All pointer arithmetic operations happen with the offset, so no can change the block index, redirecting a pointer to a different block. We are able to increment the offset past the block size, but an out-of-bound access will not lead us to a different block. This behavior conforms to the C standard which divides memory in blocks and does not provide any legit ways to pass from one block to another by performing pointer arithmetic without triggering undefined behavior. Reading outside block bounds is undefined in C[1].

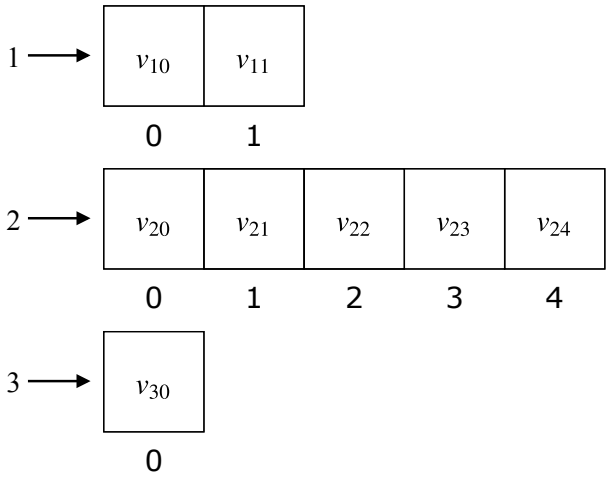


Figure 6: An example of a memory instance with three blocks.

Blocks are used to store:

- Global variables.
- Instances of local variables.
- Functions.
- Dynamic memory regions allocated by `malloc`.

[1] In real world we can surely get past the block by incrementing a pointer to it. However there are no guarantees to what exact bytes we will encounter there or will the resulting address be valid.

Arrays are stored in a single block, where elements are placed contiguously without gaps. The machine instructions contained in functions are not encoded and stored in memory; instead, an empty block exists for each function. The address of such block corresponds to the address of the function and is used to call the function by pointer.

Blocks are never retired from memory, but they can be marked as deallocated by changing their permissions accordingly, as described in Section 3.3. Such deallocated blocks stay in memory until the end of execution.

Now let us investigate how different kind of values are represented in memory. CompCert has two kinds of values: expression values and memory values.

3.2.1 Expression values

The first type of values is used in the AST expressions as immediate values and in language semantics. Evaluating expressions (for example, numeric addition or calling a function) results in values of this type. Its definition Their type is `val`

```
Inductive val :=
  | Vundef : val
  | Vint   : int     → val
  | Vlong  : int64   → val
  | Vfloat : float   → val
  | Vsingle : float32 → val
  | Vptr   : ℤ → ℤ → val
```

In the following example, an immediate value `42` occurs as a part of AST:

```
int z = 42;
```

The right hand side of the assignment will be represented as `Eval (Vint 42)` where `Vint 42` is an expression value of type `val`, and `Eval` is an expression constructor to box an immediate value.

A value can be undefined (`Vundef`) as a result of triggering undefined behavior such as signed integer overflow[2]. A pointer value `Vptr` stores a pair of block index and offset.

3.2.2 Memory values

The second type of values is used to model bytes stored in memory. CompCert semantics describes the computations with expression values, but does not allow to store them directly into memory. Instead, these values are encoded into one or multiple bytes which are then put in memory. This allows us to reason about the value of each byte.

```
Inductive memval :=
  | Undef   : memval
  | Byte    : int → memval
  | Fragment : val → quantity → nat → memval
```

Memory values can be undefined (`Undef`). Such are the memory bytes of a freshly allocated dynamic memory region.

Defined values are encoded as follows:

- Numeric values are encoded as sequences of concrete bytes (`Byte`) depending on the target architecture and taking endianness into account. For example, a 4-byte hexadecimal number `0x102F` on big-endian architecture will be encoded as shown on Figure 7:

[2]When we increase a signed integer past its maximal value, the result is undefined by the C standard. In real world the resulting integer usually becomes negative due to how signed machine arithmetics work.

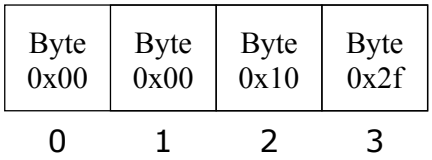


Figure 7: A multibyte number stored into memory.

- The pointers are abstracted; they are encoded as several `Fragments`. Each `Fragment` models a single byte of a pointer but does not provide its concrete value. The parameters of a constructed `Fragment v q n` have the following meaning:
  - `v:val` is the pointer value being encoded;
  - `q:quantity` can be 32-bit or 64-bit;
  - `n:`$\mathbb{N}$ shows which byte of the multibyte pointer value are we looking at. It ranges from 0 to 3 for 32-bit values and from 0 to 7 for 64-bit values.

Let us look at an example.

Example 1. Suppose that a variable `x` resides in a 7-th block and the pointer size is 32 bits. Then executing this assignment results in storing four memory values into a block associated with variable `px` as shown on Figure 8.

```
int32_t x;
int32_t* px;
...

px = &x
```

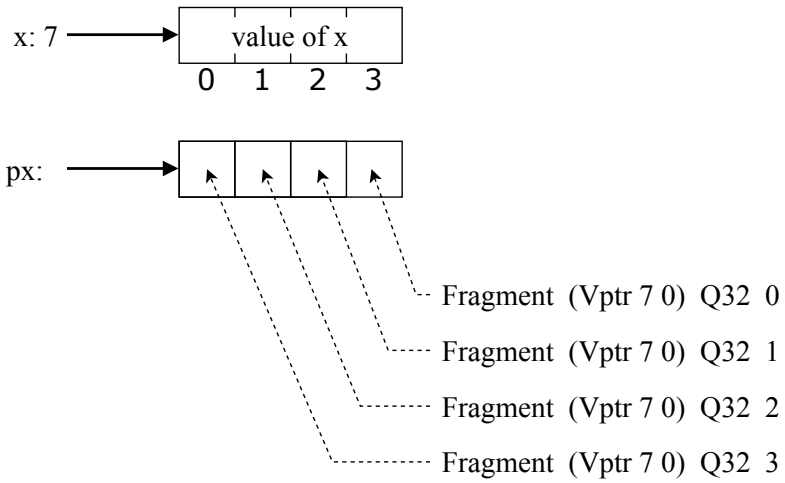


Figure 8: How an encoded pointer is stored in memory

Only pointers are encoded using `Fragments`. Making fragments of non-pointer values is not forbidden by construction, it is a legacy feature from the prior version of CompCert memory model (used until CompCert 1.7).

3.3 Permissions

The memory model we have described is augmented to express some properties seen in the language standard, such as:

- Constant variables, function code and string literals are read-only; trying to modify them triggers undefined behavior.
- Pointers can be compared using operators like > or <=. The result of this operation is only defined if both pointers point inside the same allocated block. So we need to be able to tell whether a pointer points to an existing block at all and if so, at which one.

To account for these properties, CompCert uses a permissions mechanism.
Programmers can perform four basic operations on memory addresses:

1. Compare pointers to them with a well-defined result. Recall that we are only able to compare addresses inside the same memory block.
2. Read the data stored by these addresses.
3. Overwrite the data stored by these addresses.
4. Free these addresses.

CompCert defines four corresponding permission kinds of type `perm` listed in the table below. The columns indicate which exact actions can we perform with the corresponding address given a current permission level.

Permission	can compare?	can read?	can write?	can free?
`Nonempty`	yes	no	no	no
`Readable`	yes	yes	no	no
`Writable`	yes	yes	yes	no
`Freeable`	yes	yes	yes	yes

The permissions are listed in the order of inclusion. Having a permission p implies having all previous permissions, so `Freeable` is the most permissive, while `Nonempty` is the least.

Every correct address has two associated permissions (access rights) stored in `mem_access` field of `mem` record (see Section 3.1). We can use `perm_kind` parameter to select which permission do we want to get: `Max` or `Current`.

- Maximal permission is set to `Freeable` on allocation. It never increases; it can be lowered during execution using `drop_perm` operation.
- Current permission is varying between `Nonempty` and maximal permission value.

The main purpose of this distinction is to provide a mechanism for various extensions of CompCert such as shared memory concurrency[21].

Incorrect addresses are not in this map. Such are non-allocated and already freed addresses.

We will only be interested in current address permissions because they are being checked during load and store operations. They are also relevant to the properties of memory memory transformations, as we see in Section 4.

3.4 High-level abstract model

CompCert encourages us to operate on memory through a dedicated interface rather than working directly with the fields of the `mem` record. This interface consists of several operations described in details in Section 3.4.3, which:

1. Load and store values (`load`, `loadbytes`, `store`, `storebytes`).
2. Allocate and free memory blocks (`alloc`, `free`).
3. Manage permissions (`drop_perm`).

They are used as part of language semantics to give meaning to C language constructions. These operations are also a base for axiomatic memory semantics described in Section 3.5.

3.4.1 Data chunk size

Before we address the memory operations themselves, we need to introduce the notion of memory chunk. In C programs, memory reads/writes differ by the size of value read/written. The exact pointer type defines how many bytes we refer to when we dereference it using $*$ operator. For example, `char*` implies a reference to a single byte, while `int64_t*` refers to an eight byte value.

- According to the language standard, reading too many bytes from a smaller block will yield an undefined result (there are no guarantees as to which bytes are located after the block end if any at all).

```
char data = 42;
int64_t res = *( (int64_t*) &data ); //res is undefined
```

- Reading a smaller integer from a big multibyte number produces different results depending on the target architecture and its endianness. In the following example, `data` can be stored as either `0x00 0x28` (big endian) or `0x28 0x00` (little endian). So, reading its first byte will yield different results on different platforms.

```
int16_t data = 0x28 ;
char res = *( (char*) &data); //res can be 0 or 0x28, implementation-dependent
```

To differentiate these reads, CompCert parameterizes them with a `chunk` parameter of type `memory_chunk` described below. Writes are parameterized in the same way.

```
Inductive memory_chunk :=
  | Mint8signed
  | Mint8unsigned
  | Mint16signed
  | Mint16unsigned
  | Mint32
  | Mint64
  | Mfloat32
  | Mfloat64.
```

CompCert semantics selects an appropriate `memory_chunk` constructor depending on the exact pointer type (if we access memory by pointer) or variable type (when we address a variable directly).

3.4.2 Natural alignment

Memory reads and writes should not only respect permissions, but also alignment constraints[3].

CompCert uses a notion of natural alignment for a memory chunk, which is usually equal to its size. Only naturally aligned reads and writes are allowed.

```
Definition align_chunk (chunk: memory_chunk) : Z :=
  match chunk with
  | Mint8signed ⇒ 1
  | Mint8unsigned ⇒ 1
  | Mint16signed ⇒ 2
  | Mint16unsigned ⇒ 2
  | Mint32 ⇒ 4
  | Mint64 ⇒ 8
  | Mfloat32 ⇒ 4
  | Mfloat64 ⇒ 4
  | Many32 ⇒ 4
  | Many64 ⇒ 4
  end.
```

As the natural alignment is required for reads, we can not relocate an aligned value to an unaligned address without breaking the program semantics. We will see how this restriction is accounted for during memory transformations in Section 4.

3.4.3 Operations on memory

CompCert defines the following basic operations on memory:

`load` Reads and decodes a value from memory.

```
load : mem → chunk → ℤ → ℤ → option val
```

All memory bytes in range should have sufficient permissions for reading, otherwise `None` is returned.

If the permission check succeeds, we fetch one or several consecutive `memval`s from memory and try to decode them into an expression value. The parameter of type `chunk` determines how many bytes are fetched.

If decoding fails, `Some Vundef` is returned. Decoding integers and floats is architecture-dependent, decoding pointers is described in Section 3.1.

`loadbytes` Reads raw bytes from memory without decoding them. The Section 3.4.4 gives the use cases for this operation.

```
loadbytes: mem → ℤ → ℤ → ℤ → option (list memval)
```

All bytes in range should be readable, otherwise `None` is returned. If the permission check succeeds, we take the `memval`s from memory and return them as a list. Unlike `load`, no decoding is involved.

[3]On some architectures such as SPARC, unaligned reads lead to hardware exceptions.

`store` Stores a value of a given chunk size in memory into a block with an offset.

```
store: mem → chunk → ℤ → ℤ → val → option mem
```

All memory bytes in range should have sufficient permissions for writing, otherwise the action fails and `None` is returned.

If the permission check succeeds, we encode `val` into a list of `memvals`. The encoding of different data formats is different; an appropriate type is selected based on `chunk` value. Encoding integers and floats is also architecture-dependent, encoding pointers is described in Section 3.1.

`storebytes` Stores several consecutive bytes into memory. The Section 3.4.4 gives the use cases for this operation.

```
storebytes: mem → Z → Z → list memval → option mem
```

This is a raw operation like `loadbytes` in a sense no encoding is involved, unlike `store`. On attempt to overwrite non-writable bytes, `None` is returned.

`alloc` Allocates a new block and set its limits: the base offset (always equal to zero) and the latest correct offset[4].

```
alloc : mem → ℤ → ℤ → mem * ℤ
```

All memory allocations (static, dynamic or automatic) are performed by `alloc` operation. It creates a fresh memory block and takes its index from the `nextblock` field. It is the index of the next block being allocated.

In real C, a particular case of allocating dynamic memory using `malloc` can indeed fail if we fall short of memory. In CompCert allocation never fails because memory is infinite.

`alloc` returns a pair of the new memory instance and a fresh block index. The `nextblock` field is used as a new block index; in the new memory it is incremented. This way if `alloc` is the only way to allocate memory all blocks from the first one until `nextblock-1` inclusive are allocated. This constraint is however not enforced by memory construction.

Definition 1. A *valid block* is the block whose index is less than `nextblock`.

We will see how it is important to preserve the validity of block indices during memory transformations in Section 4.

`free` Frees a range of bytes in a given block. It accepts three integer parameters: the block index, the lower and upper bound of the range to free. This operation fails if the said range contains bytes that are already freed.

Note that as with `alloc` this is an all-purpose memory freeing that is applied to local variables and dynamic memory alike. It is also different from the standard C function `free` because the latter frees the entire block, not a range of addresses inside it.

```
free: mem → ℤ → ℤ → ℤ → option mem
```

[4]The base offset can be non-zero. However, CompCert does not currently use this functionality.

`drop_perm` Lowers the (maximal) permission level for a range of addresses to a new value. It accepts three integer parameters: the block index, the lower and upper bound of the range of address for which the permissions will be changed.

`drop_perm: mem` $\to \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z} \to$ `permission` $\to$ `option mem`

For example, it can be used to mark memory as read-only after initializing it. `None` is returned if we attempt to increase permission level for some address.

3.4.4 The purpose of `loadbytes` and `storebytes` operations

Operations `loadbytes` and `storebytes` are used to give semantics:

- to the C standard function `memcpy`. It copies a range of bytes to a new location.
- to the assignment operation. C allows to assign one structure to another, like this:

```
struct s { int field; } ;
struct s inst1, inst2;

void main() {
  inst1 = inst2;
}
```

This involves copying all bytes of `inst2` into `inst1`.

Definition 2. *Structure padding* is a concept of inserting one or more empty bytes between structure members to align them. It happens during memory allocation.

The values of bytes inserted by structure padding are not defined; however, copying a structure usually copies these values as well for performance reasons. So instead of copying structures on a field-to-field basis, `loadbytes` and `storebytes` are used to copy all fields and all bytes in between.

3.5 Axiomatic memory semantics

Reasoning about an exact memory state by working with individual byte values is often too verbose, especially if the memory is only modified through an interface described in Section 3.4.3. CompCert provides a set of theorems describing the interaction of different memory operations. They allow us to for example simulate an operation on a modified memory state by an operation on the original one.

This set of theorems does not cover all possible combinations of operations – only those with defined behavior. For example, there is no theorem describing the result of freeing memory that is already freed, because in this case the language standard postulates an undefined behavior.

A full list of such theorems can be found in [5]; we will only give an example to illustrate how the semantics look like, but we will not use it later.

3.5.1 Example: Load after alloc

If `alloc` m_1 *low high* `=` (*newblock*`,` m_2) then for all $b \neq$ *newblock* the following holds:

$\forall$ *type offset*`,` `load` *type* m_2 *b offset* `=` `load` *type* m_1 *b offset*

We have allocated a new block in memory state m_1 with index *newblock* and some bounds *low*, *high*; the new memory state is m_2. In m_2 any load is equivalent to a load from m_1 except for a load from freshly allocated block *newblock*

Note, that this law can not be applied to reason about memory contents of a fresh block; its contents are not yet initialized and reading them triggers undefined behavior.

3.5.2 Example: Load after free

If `free` m b low $high$ `=` m' then two cases are defined:

Load outside of freed range If one of the following conditions holds:

- $b' \neq b$ (load from a different block)
- $offset + $ `size` $type \leq low$ (load from an affected block but before freed range)
- $high \leq offset$ (load from an affected block but after freed range)

then

`load` m' $type'$ b' $offset'$ `=` `load` m $type$ b' $offset'$

This is an obvious case of fully independent `free` and `load` working with different memory areas.

Load intersects with the freed range If, on the other hand, $low \leq offset \leq high -$ `size` $type$ then

`load` m' $type$ b $offset$ `=` `None`

If at least one of the bytes fetched from memory by `load` was in the freed region, it is no longer valid to fetch it. Hence the `load` is incorrect.

4 Memory injections in CompCert

Each compilation pass transforms the input program. Program transformation induces the transformation of runtime states that appear during program execution; these states contain memory layouts. The transformed program should behave in states with transformed memory like the input program behaves in its own states. To prove that CompCert defines several classes of relations between memory layouts such as memory injections (studied in this section), memory extensions [21] etc. These relations are used to prove correctness of compilation stages through bisimulation.

In our work we build our own relation using memory injection relation defined in CompCert. This part gives an intuition behind memory injection (Section 4.1 provides a use case from CompCert), recalls its definition (Section 4.2) and its properties (Section 4.4).

4.1 Injection usage during compilation

One of the compilation stages in CompCert is the translation from C#minor to Cminor intermediate languages. The difference between them is in their memory layouts: while a program in C#minor allocates all local variables in their distinct memory blocks, a Cminor program allocates them in a single block per function instance.

Example 2. Let us take a look at the following C#minor program:

```
void g() {
  int32_t x, y;
}
```

If we interpret this code as a C#minor program, we should allocate the following blocks for a call to `g`:

- Block b_1 for `x` of size 4.
- Block b_2 for `y` of size 4.

On the other hand, if we interpret this code as a Cminor program, we should allocate only one block for a call to `g`:

- Block b'_1 for `x` and `y` of size 8.

The block b'_1 models the stack frame of `g`.

Suppose we execute this program as a C#minor program and as a Cminor program; we have just called `g`. Now we want to map values from C#minor memory state m_1 to the values in Cminor memory state m_2. Then:

- All pointers to the block b_1 in m_1 should point to the same block b'_1 in m_2.
- All pointers to the block b_2 in m_1 should point to the block b'_1 with additional offset 4 in m_2.

When these conditions are satisfied, the pointers to `x` and `y` in m_1 and the same pointers in m_2 become equivalent for the respective programs. In CompCert such memory states m_1 and m_2 are related by memory injection defined below; it is used to prove external behavior preservation in several compilation stages.

4.2 Defining memory injection

Memory injection is a relation between memory states m_1 and m_2 that captures the idea of C program semantics being invariant to data relocation.

4.2.1 Data relocation

Definition 3. *Data relocation* is the process of assigning new addresses for data and adjusting data and code to reflect these changes.

In a C program, there is only one valid way of obtaining data address: by using & operator. As variable `x` gets relocated, the value of expression `&x` follows the updated address of `x`, always yielding its correct address. This makes the choice of exact address for data and code arbitrary.

Hard-coded addresses Using hard-coded pointers in programs does not allow us to place code and data at arbitrary addresses. What if a variable is relocated so that it overlaps such hard-coded address?

```
char* ptr = 0xABFF;
*ptr = 42; // writing to address 0xABCC
```

However hard-coded addresses should not be dereferenced in valid C programs, otherwise an implementation-defined behavior is triggered. A standard-conforming way to address data at specific addresses is to instruct linker to place global variables to these addresses, and then work with these variables.

Address arithmetic and relocation We might argue that we can always perform address arithmetic to get the data address in other ways, for example:

```
void foo() {
  int x;
  int y;

  int* y_addr = &x + sizeof( x ) ;
  int z = *y_addr;
}
```

In this code fragment we assume that local variables are stored in a stack frame for the function `foo` and thus are placed consecutively in memory. We then conclude that the address of `y` can be computed based on address of `x`, and computing the same address after relocation will not yield a correct result.

However this assumption is wrong: the language standard marks all out-of-bound accesses as undefined behavior. This allows the compiler to chose the exact stack frame layout as it sees fit, for example optimizing some local variables out. The function `foo` thus triggers undefined behavior in the first place when `y_addr` is dereferenced.

4.2.2 Assigning new addresses to data

For the sake of simplicity in this subsection we suppose that the data itself is not changed by relocation; in the next section we elaborate how it actually changes. In figures referenced in this subsection capital letters correspond to `memval`s.

In CompCert assigning new addresses to data happens on the block level. Each block can be assigned a new index and a new starting offset.

To encode a mapping between addresses in memories m_1 and m_2 CompCert uses embedding functions.

Definition 4. *Embedding function* (or just embedding) is a function of type `meminj`:

```
Definition meminj := ℤ → option ℤ × ℤ
```

An embedding f that maps addresses of m_1 to addresses of m_2 has the following meaning:

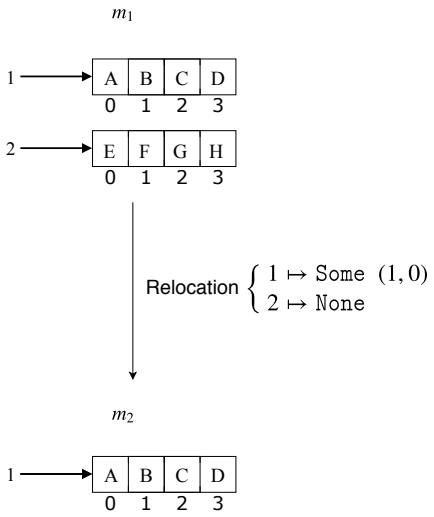
- $f(b) =$ `Some` (b', δ) means that the block with index b is mapped to a block with index b' with offset δ. An address (b, i) of m_1 is mapped to $(b', i + \delta)$ of m_2.
- $f(b) =$ `None` means either:
  - that a block with index b does not exist in m_1 or
  - that a block with index b is discarded;

Embeddings allow us to coalesce multiple blocks into one by mapping different blocks of m_1 into the same block of m_2 but with different starting offsets.

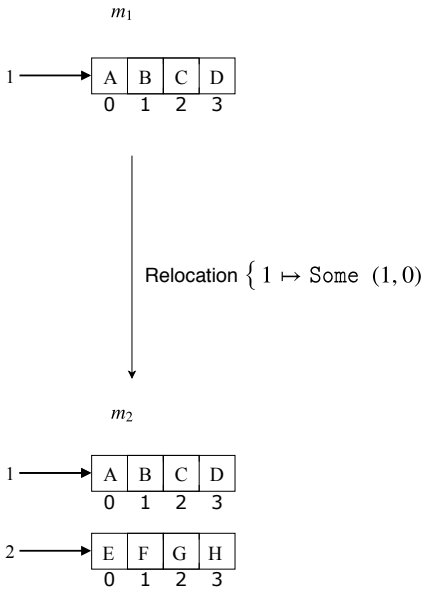
The effects on data addresses described by f can be also described by composing any number of the following four basic operations:

1. Discard blocks of m_1 (Figure 9a).
2. Add new blocks not present in m_1 (Figure 9b).
3. Change the block indices in m_1, for example, swapping two blocks (Figure 10a; see also Figure 11).
4. Coalesce multiple blocks into one (Figure 10b). Note, that we can not break blocks themselves into parts, so unlike the first three operations, this operation is not invertible.

Because of non-invertibility of operation (4) memory embeddings are not necessarily symmetric. The summary of asymmetric aspects of memory injections is placed in Section 4.3.

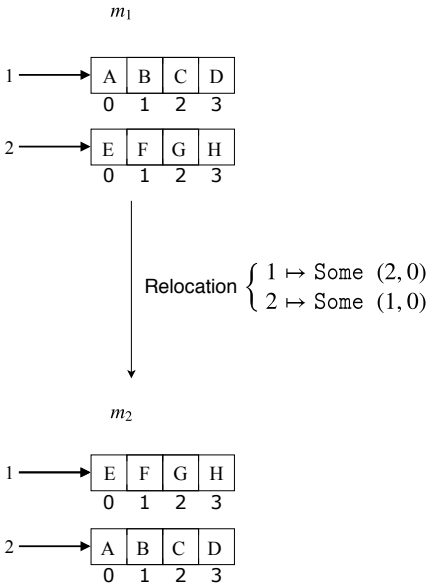


(a) Discard block 2 in m_1.

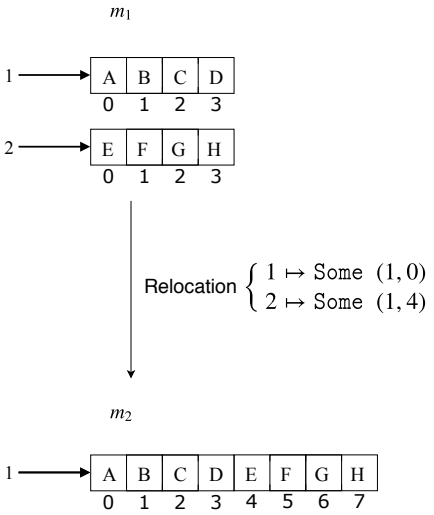


(b) Introduce block 2 in m_2.

Figure 9: First two basic operations, inverses of each other.



(a) Swap blocks 1 and 2 in m_2.



(b) Coalesce blocks 1 and 2 in m_1 into 1 in m_2.

Figure 10: Another two basic operations; swap is inverse of itself, coalesce has no inverse.

4.2.3 Adjusting the code and data

If blocks are relocated, it is necessary to adjust pointers to these blocks.

Example 3. Figure 11 shows two memory states m_1 and m_2; m_2 depicts the stat Suppose m_1 holds a numeric value `ABCD` in block 2; it is unchanged by relocation because it is not a pointer. In m_2 this value is relocated to the block 3. In both memories the block 1 holds a pointer containing the address of `ABCD`. In m_1 its value is `Vptr 2 0`.

As the numeric value is relocated, we have to redirect the pointer in block 1 of m_2 changing its value to `Vptr 3 0`. Otherwise this pointer will no longer point at the same block as in m_1.

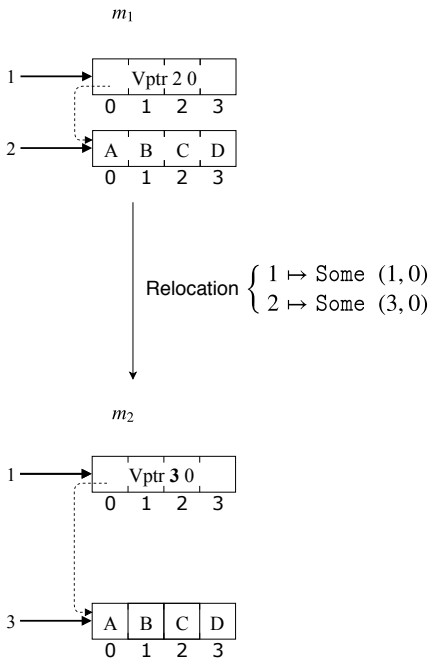


Figure 11: Block 2 is relocated to index 3, the pointer to it (inside block 1) is actualized.

The pointers can be found:

- in program code, where they are modeled by `Vptr` of type `val`; In CompCert C semantics it also occurs in continuations, global and local environments.
- in memory, where their bytes are modeled as `Fragment`s of type `memval`; each such fragment holds a piece of `Vptr` of type `val`.

CompCert introduces two relations parameterized by the embedding f to describe mapping of expression values and memory values by relocation:

Expression values. For the values of type `val` this relation is `Val.inject` :

```
Inductive Val.inject (f : meminj) : val → val → Prop :=
    inject_int   : ∀ i , inject f (Vint i) (Vint i)
  | inject_long  : ∀ i , inject f (Vlong i) (Vlong i)
  | inject_float : ∀ d , inject f (Vfloat d) (Vfloat d)
  | inject_single : ∀ f , inject f (Vsingle f) (Vsingle f)
  | inject_ptr   : ∀ b1 ofs1 b2 δ,
                     f b1 = Some (b2, δ) →
                     inject f (Vptr b1 ofs1) (Vptr b2 (ofs1 + δ))
  | inject_undef : ∀ v, inject f Vundef v
```

- For numeric values, this relation acts as an identity (`inject_int`, `inject_long`, `inject_float`, `inject_single`).
- Pointers are redirected using the embedding f (`inject_ptr`).
- Undefined values are mapped to any values (`inject_undef`).

Compilers treat undefined values as wildcards: they can be substituted for any values without violating program semantics. Because of this `Val.inject` is constructed to allow mapping undefined values to any values, without providing no guarantees for them. As the consequence `Val.inject` loses information about `Vundef` and is not symmetric.

Memory values. A similar relation for `memval` is `memval_inject` :

```
Inductive memval_inject (f: meminj)
                : memval → memval → Prop :=
  | memval_inject_byte :
      ∀ n, memval_inject f (Byte n) (Byte n)
  | memval_inject_frag :
      ∀ v1 v2 q n,
       Val.inject f v1 v2 →
       memval_inject f (Fragment v1 q n) (Fragment v2 q n)
  | memval_inject_undef :
      ∀ mv, memval_inject f Undef mv
```

- Bytes are preserved as they are (`memval_inject_byte`);
- Pointer values in fragments are injected using `Val.inject`[5] (`memval_inject_frag`).
- Analogously to `val`s, undefined memory values can become more defined (`memval_inject_undef`). If m_1 holds `Undef` by an address, and m_2 holds `Byte 42` by the corresponding address instead, the compilation will still be considered correct.

Similarly to `Val.inject` this relation loses information about `Undef` and is thus not symmetric. A Figure 12 shows an example with two corresponding memory blocks of m_1 and m_2 such that $m_1 \hookrightarrow m_2$.

Figure 12: Two blocks where corresponding bytes are related by `memval_inject`.

4.3 Formal definition

Finally, having studied the intuition behind correct data relocations we can give it a formal definition[6].

Definition 5. *Memory injection* A relocation is encoded by a triplet of two memory states m_1, m_2 and an embedding function f. A memory injection is a relation `inject` that states that a specific relocation satisfies the following constraints :

[5] Recall that only pointers are stored in Fragments. See Section 3.1.

[6] A note on technical implementation details: CompCert defines two types of relations between two memory states on top of embedding: `mem_inj` (containing `mi_perm`, `mi_align` and `mi_memval` properties of the definition above) and a stronger `inject` (containing also all other properties). It happened for historical reasons; we will not distinguish these relations and will always use `inject`, which implies `mem_inj`.

```
(* In module common.Memory *)
Record inject (f: meminj) (m1 m2: mem) : Prop := {
    mi_perm:
      ∀ b1 b2 δ ofs k p,
      f b1 = Some( b2, δ) →
      perm m1 b1 ofs k p →
      perm m2 b2 (ofs + δ) k p;

    mi_align:
      ∀ b1 b2 δ chunk ofs p,
      f b1 = Some (b2, δ) →
      range_perm m1 b1 ofs (ofs + size_chunk chunk) Max p →
      (align_chunk chunk | δ);

    mi_memval:
      ∀ b1 ofs b2 δ,
      f b1 = Some (b2, δ) →
      perm m1 b1 ofs Cur Readable →
      read_byte b1 ofs m1 ↪^f read_byte b2 (ofs + δ) m2

    mi_freeblocks:
      ∀ b, ¬ valid_block m1 b → f b = None;

    mi_mappedblocks:
      ∀ b b' δ,
         f b = Some(b',δ) → valid_block m2 b';

    mi_no_overlap: ...

    mi_representable: ...

    mi_perm_inv: ...

  }.
```

This record collects the following properties which constraint both the embedding function f and memories m_1 and m_2:

- `mi_perm`: Permissions of corresponding addresses match. They do not need to be equal: because having higher permissions implies having all weaker permissions, we can map addresses with lower permissions into ones with higher permissions.
- `mi_align`: If a block is mapped to a sub-block, the latter is *naturally aligned* (see Section 3.4.2).
- `mi_memval`: The values of corresponding readable bytes are related by `memval_inject`.
- `mi_freeblocks`: the embedding does not map invalid blocks into anything. Blocks with indices greater than `nextblock` are invalid; see Section 3.4.3.
- `mi_mappedblocks`: the embedding only produces valid blocks.
- `mi_no_overlap`: the blocks do not overlap: two distinct blocks are either mapped to distinct blocks, or into non-overlapping sub-blocks of the same block.
- `mi_representable` and `mi_perm_inv` are not useful in the context of our work.

We are only interested in non-trivial memory injections. Trivially any two memory states m_1 and m_2 can be related by injection with an empty embedding f_ϵ such that $\forall b, f(b) = \texttt{None}$.

Notation. We use an overloaded notation $\hookrightarrow^f$ to denote injection of `val`s and `memval`s parameterized with an embedding f as well as memory injection itself:

- if v_1 and v_2 are of type `val` and `Val.inject`, $f\ v_1\ v_2$ holds, then we write $v_1 \hookrightarrow^f v_2$
- if mv_1 and mv_2 are of type `memval` and `memval_inject` $f\ mv_1\ mv_2$ holds, then we also write $mv_1 \hookrightarrow^f mv_2$
- if m_1 and m_2 are of type `mem` and `inject` $f\ m_1\ m_2$ holds, then we also write $m_1 \hookrightarrow^f m_2$.

When the exact value of embedding function is not important to us we omit it, like this: $m_1 \hookrightarrow m_2$. This notation appears in [21].

Inversibility. There are three aspects that contribute to the asymmetry of `inject`:

- Blocks can be coalesced, but we can not break them into parts.
- Undefined values (`val`s as well as `memval`s) can be substituted by any values, but defined values can not be mapped into undefined ones.
- The constraints on mapped values are only applied to the readable addresses. It means that an address that is not readable and contains value `Byte 42` can be mapped into an address that is readable and contains a different value `Byte 99`. This can not break the program semantics because the program can not legally access the first value.

The consequences of this are studied in the next section.

4.4 Commutation properties

The properties described by `inject` entail some properties on high-level operations on memory:

- `load` and `loadbytes`
- `alloc`
- `drop_perm`
- `store` and `storebytes`
- `free`

In this section we study the commutation of memory operations with memory injection relation. For convenience we will omit the `chunk` parameter of memory operations such as `load` or `store`; its value will be the same across all operations mentioned in any theorem definition.

4.4.1 Commutation of loads

Suppose m_1 and m_2 are memories such that $m_1 \hookrightarrow m_2$. Then loads from m_2 are simulated by loads from m_1 at corresponding addresses.

```
Theorem load_inject:
  ∀ f m1 m2 b1 ofs b2 δ v1,

  m1 ↪[f] m2 →
  load m1 b1 ofs = Some v1 →
  f b1 = Some (b2, δ) →

  ∃ v2, load m2 b2 (ofs + δ) = Some v2 ∧ v1 ↪[f] v2
```

This theorem states that if we have successfully read a value v_1 from some address in m_1, we can read a similar value v_2 (such as $v_1 \hookrightarrow^f v_2$) from a corresponding address in m_2.

This statement differs from `mi_memval`: while `mi_memval` is a property about encoded bytes of type `memval`, stored in memory, `load_inject` is a property about `vals`, fetched and decoded from several `memvals`.

There is a similar theorem for `loadbytes` operation, it is called `loadbytes_inject`.

4.4.2 Kinds of store commutation

In this section we introduce three different kinds of store commutation. We will later show how they can be generalized to other kinds of memory modifications such as `alloc`.

Suppose m_1 and m_2 are memories and $m_1 \hookrightarrow^f m_2$.
There are three ways of performing stores in either of them that will yield memory injection:

- corresponding stores in m_1 and m_2;
- store in a discarded block of m_1;
- store outside of mapped area in m_2.

Corresponding stores. Suppose we write a value v_1 in m_1 at (b_1, ofs) and the resulting memory state is n_1. The existence of n_1 means that the write succeeded, otherwise the `store` operation would have returned `None`. The address (b_1, ofs) is therefore valid and its permissions are sufficient for writing. Then we can perform `store` of an appropriately injected value v_2 at a corresponding address of m_2; this will always succeed with a correct result n_2 as shown on Figure 13 and on Figure 14 in more details.

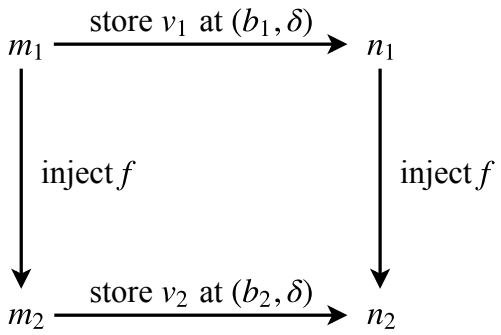


Figure 13: Corresponding stores preserve memory injection; $f(b_1) = (b_2, 0)$ is mapped to b_2.

The theorem `store_mapped_inject` formalizes this property.

```
Theorem store_mapped_inject:
  ∀ f m1 b1 ofs v1 n1 m2 b2 δ v2,
  m1 ↪[f] m2 →
  store m1 b1 ofs v1 = Some n1 →
  f b1 = Some (b2, δ) →
  v1 ↪[f] v2 →

  ∃ n2, ( store m2 b2 (ofs + δ) v2 = Some n2 ) ∧ ( n1 ↪[f] n2 ).
```

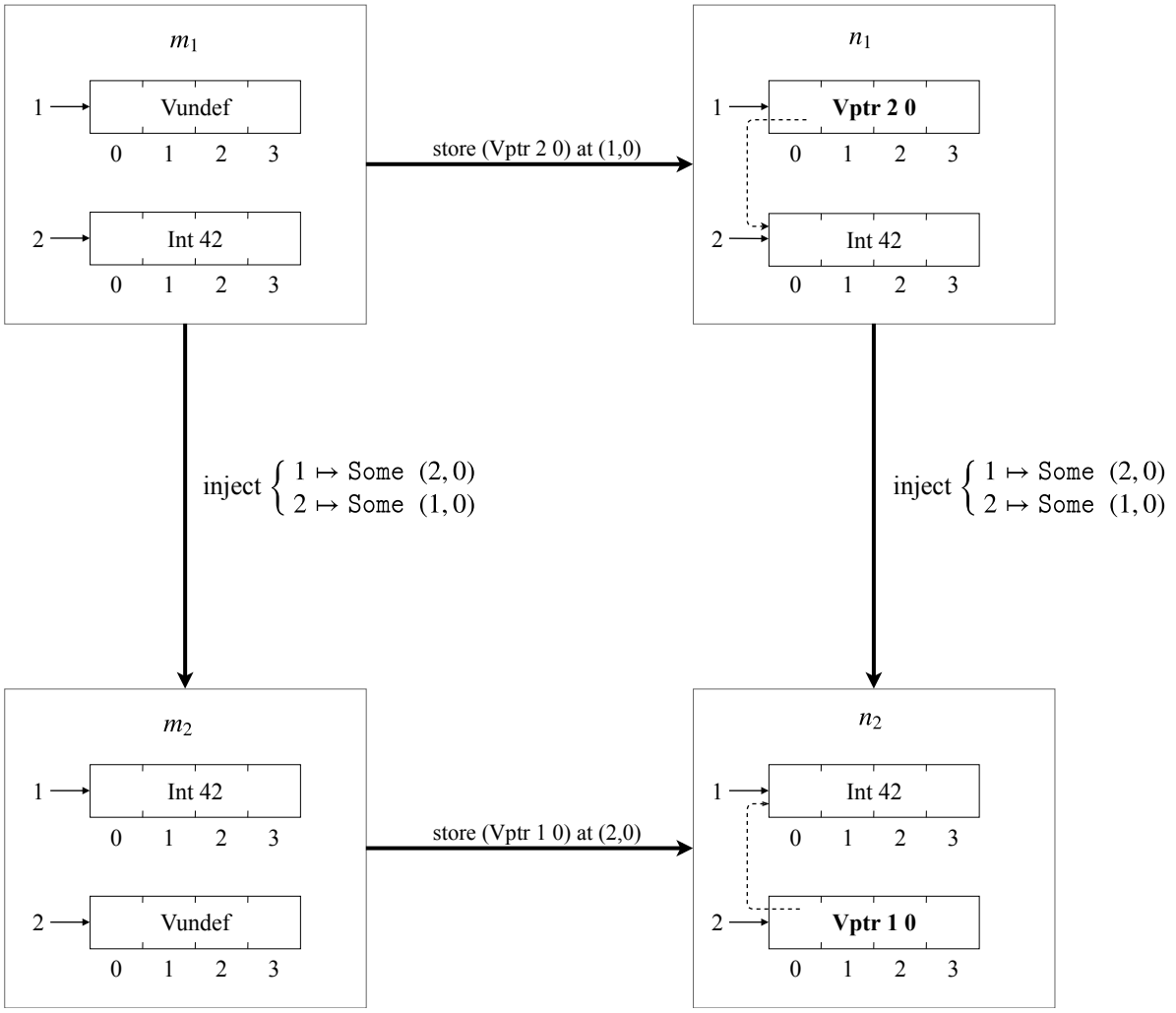


Figure 14: Corresponding stores preserve memory injection: `Vptr 2 0` $\hookrightarrow^f$ `Vptr 1 0`.

Store in a discarded block of m_1. Suppose again that we write a value v in m_1 at (b, ofs) and the resulting memory state is n_1; this time we do not modify m_2. If `store` is performed into a block discarded by f (that is, $f(b) =$ `None`) then $n_1 \hookrightarrow^f m_2$, as shown on Figure 15.

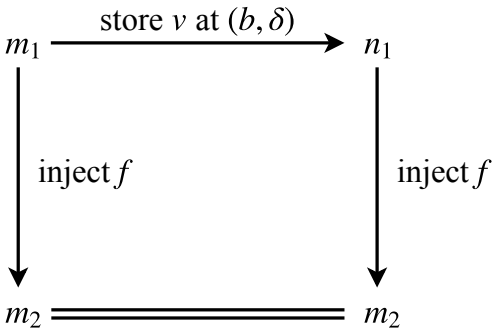


Figure 15: Store in a discarded block preserves memory injection.

The theorem `store_unmapped_inject` formalizes this property:

```
Theorem store_unmapped_inject: ∀ f m1 b ofs v n1 m2,
  m1 ↪[f] m2 →
  store m1 b ofs v = Some n1 →
  f b = None →
  n1 ↪[f] m2
```

Store outside of mapped area in m_2. Now suppose that we write a value v in m_2 at an address (b, ofs) and the resulting memory state is n_2; this time we do not modify m_1. The relation $m_1 \hookrightarrow^f n_2$ holds as shown on Figure 16 if either of these conditions are met:

1. The target block b has no preimage in m_1 by f.
2. The target block b has a preimage b' in m_1 but b' is not readable in m_1.

A block that is not readable in m_1 can correspond to a writable block in m_2. Suppose we have an address A in m_1 and a corresponding address A' in m_2. Memory injection does not force permissions at A and A' to match exactly: permissions at A' are allowed to be higher. It means that a non-readable A can correspond to readable and writable A'. By injection definition, only readable memory values in m_1 are bound by `memval_inject` with values in m_2. If permissions at A are only `Nonempty` (and thus not readable nor writable), the value at A' can be arbitrary.

For brevity we call regions of such addresses in m_2 *indifferent*.

Definition 6. *Indifferent address.* Suppose $m_1 \hookrightarrow^f m_2$. An address A' of m_2 is called indifferent if it has at least `Readable` permissions and its preimage address A in m_1 either does not exist or has only `Nonempty` permissions. In the Coq definition below, $A' = (b, ofs + \delta)$, its preimage $A = (b', ofs')$.

```
Definition indifferent f b ofs :=
  ∀ b' δ ofs',
  f b' = Some (b, δ) →
  perm m1 b' ofs' Cur Readable →
  ofs ≠ ofs' + δ.
```

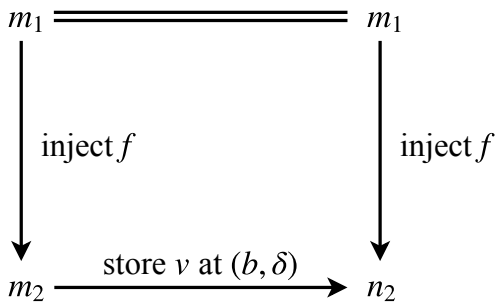


Figure 16: Store in an indifferent region of m_2 preserves memory injection.

The theorem `store_outside_inject` formalizes this commutation property.

```
Theorem store_outside_inject:
  ∀ f m1 m2 b ofs v n2 chunk,
```

```
m1 ↪f m2 →

(* if store is performed to an indifferent region *)
indifferent_range f b ofs (ofs+ size_chunk chunk) →

store chunk m2 b ofs v = Some n2 →

m1 ↪f n2.
```

The `store` in m_2 affects addresses in range from (b, ofs) to $(b, ofs+$`size_chunk chunk`$)$. No readable address (b', ofs') should be mapped into an address $(b, ofs' + \delta)$ falling into this range. An auxiliary definition `indifferent_range` similar to `indifferent` is used to encapsulate this premise .

```
Definition indifferent_range f b lo hi :=
  ∀ b' δ ofs',
  f b' = Some (b, δ) →
  perm m1 b' ofs' Cur Readable →
  ¬ lo ≤ ofs' + δ < hi
```

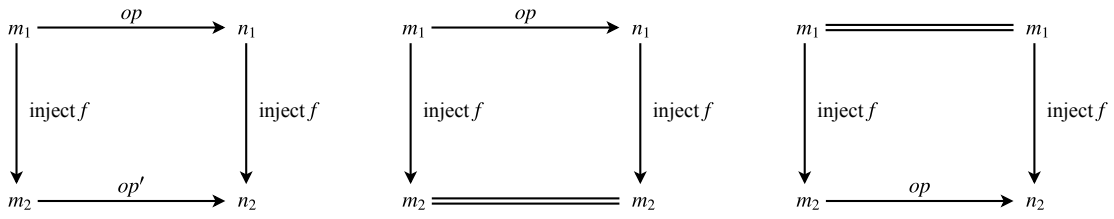
Otherwise, if (b', ofs') is a readable address in m_1, the corresponding address in m_2 is *not indifferent*, and modifying its contents will break memory injection.

4.4.3 Commutation with other modifications

We have elaborated three kinds of commutation properties by using the `store` operation as an example:

- Parallel modifications;
- Modifying an unmapped block;
- Modifying memory region outside injection (an *indifferent* region).

These kinds can be generalized to all other memory mutating operations such as `alloc` or `storebytes` as seen on Figure 17.



(a) Parallel modification. ***(b)*** Modifying m_1. ***(c)*** Modifying m_2.

Figure 17: Kinds of commutation properties; arguments of operation *op* are adjusted in op'.

Parallel modifications. In this case m_1 is modified and m_2 is modified accordingly. The same operation is performed on m_2, its arguments are adjusted using $\hookrightarrow$ relation.

The following theorems formulate these commutation properties:

- For `store`: `store_mapped_inject`.
- For `storebytes`: `storebytes_mapped_inject`.
- For `alloc`: `alloc_parallel_inject` when a similar fresh block is allocated in m_1 and m_2. The embedding function f is upgraded to map the fresh block in m_1 to the fresh block in m_2 using `inject_incr` relation.
- For `free`: `free_parallel_inject`
- For `drop_perm`: `drop_mapped_inj`

Modifying an unmapped block. The modification made in m_1 is local to a certain block. If memory injection discards this block, we can inject a modified memory into the original m_2. In other words, a block *unmapped* by memory injection is modified. The following theorems formulate these commutation properties:

- For `store`: `store_unmapped_inject`.
- For `storebytes`: `storebytes_unmapped_inject`
- For `alloc`: `alloc_unmapped_inject`.
- For `free`: `free_left_inject`

Modifying memory region outside injection. As in the `store` example provided in Section 4.4.2 acting on a block of m_2 without preimage in m_1 or modifying an indifferent memory region in m_2 allow us to preserve memory injection relation. In this case the modification occurs *outside* of the area of m_2 bound by $\hookrightarrow$ relation. The following theorems defined formulate these commutation properties:

- For `store`: `store_outside_inject`.
- For `storebytes`: `storebytes_outside_inject`.
- For `alloc`: `alloc_right_inject`.
- For `free`:free_right_inject‘
- For `drop_perm`: `drop_outside_inject`.

5 Memory bijection

When $m_1 \hookrightarrow m_2$, all undefined values in m_1 are unbound: they can become anything in m_2. In order to construct refactoring correctness proofs we need a stricter version of memory injection that would preserve undefined values. We construct such relation using two opposite injections; we call it *memory bijection*.

Memory injection is different from memory bijection in the following way. First, memory injection does not preserve undefined values unlike memory bijection. Second, injection can coalesce several blocks of m_1 together to form a composite block in m_2 (see Figure 10b). Memory bijection does not allow that: it only allows changing the block index, but not mapping it into a subblock. This guarantees an invertible one-to-one mapping between some blocks of m_1 and some blocks of m_2. Some blocks in m_1 can have no correspondence in m_2; some blocks in m_2 can also have no correspondence in m_1. This relation is exemplified on Figure 18.

As a consequence or undefined value preservation, memory bijection implies a stronger version of `load_inject` (Section 4.3). This stronger theorem `load_biject` is given in Section 5.5.

Simulation and commutation properties mentioned in Section 4.4 hold for memory bijection as it is a subcase of memory injection. These properties also have their stronger versions for memory bijection; these versions are studied in Section 5.5.

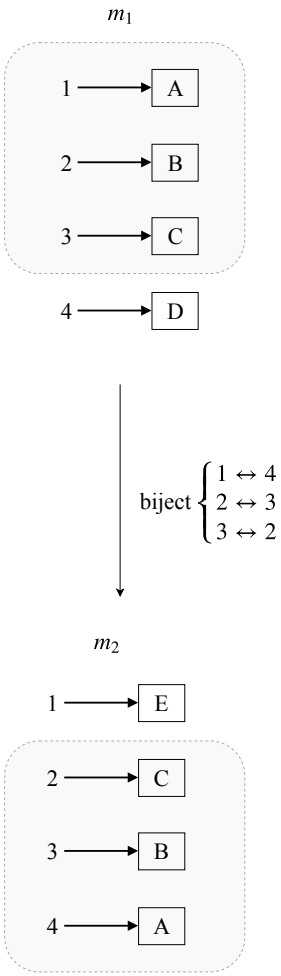


Figure 18: An example of memory bijection between m_1 and m_2. Blocks 4 of m_1 and 1 of m_2 are outside the bijection.

5.1 Inverse embedding functions

Definition 7. *Inverse embeddings.* Embedding functions f and g are *inverses* of each other when the following two conditions are satisfied:

1. For all arguments, the second component of values of f and g is zero. The following should hold for f and g:

```
Definition no_offset (mi:meminj) :=
  ∀ b δ, mi b = Some (b', δ) →
  δ = 0
```

2. f projects $b \mapsto b'$ if and only if g projects $b' \mapsto b$:

```
Definition bidir (f g: meminj): Prop :=
   ∀ b b',
   f b = Some (b',0) ↔
   g b' = Some (b,0).
```

The record `inverse` collects these properties for a pair of embeddings f and g:

```
Record inverse (f g: meminj) :=
  {
    mb_no_offset1: no_offset f;
    mb_no_offset2: no_offset g;
    mb_consistent: bidir f g
  }.
```

With the additional offset of f and g fixed at zero, their codomain `option` $(\mathbb{Z}*\mathbb{Z})$ becomes isomorphic to `option` $\mathbb{Z}$. We are now able to think about them as partial mappings between block indices.

Note, that `no_offset` still allows an embedding to discard any block. Our intention is to allow both m_1 and m_2 to have extra blocks.

Symmetry. The relation `inverse` is symmetric:

$$\texttt{inverse}\ f\ g \to \texttt{inverse}\ g\ f \tag{symmetry}$$

This follows from its definition.

5.2 Relation on memories

Definition 8. *Memory bijection.* A memory bijection between memories m_1 and m_2 is a pair of two injections: $m_1 \hookrightarrow^f m_2$ and $m_2 \hookrightarrow^g m_1$ such that f and g are inverses.

```
Record bject (f g: meminj) (m1 m2: mem):=
  {
    bi_fwd: m1 ↪^f m2;
    bi_bwd: m2 ↪^g m1;
    bi_bidir: inverse f g
  }.
```

It is possible to derive `g` by using `f` and a set of constraints appearing in definitions for $m_1 \hookrightarrow^f m_2$ and $m_2 \hookrightarrow^g m_1$. We chose to provide `g` explicitly for convenience in proofs.

By memory bijection definition any two memories are related by trivial bijection if f and g always return `None`, but this case is not interesting to us.

Invertibility. The relation `bject` is invertible:

$$\texttt{bject}\ f\ g\ v\ v' \to \texttt{bject}\ g\ f\ v'\ v \tag{bject_inv}$$

This follows from its definition.

Relations defined directly and using `inject`. We have defined memory bijection through the `inject` relation. Defining it directly akin to `inject` would be less easy: `inject` encapsulates too many constraints on corresponding memory bytes, their permissions and the embedding function (see `inject` definition in Section 4.3). It would also render impossible the direct reuse of proven complex properties of memory injections.

In order to formulate simulation and commutation properties of memory bijection (Section 5.5) we also need to formalize how the expression values and memory values are transformed by memory bijection. We have to craft relations that would act like `Val.inject` and `memval_inject`. Unlike `inject` we chose to define the correspondence between values directly for the following reasons:

- Directly encoded mapping of values are simpler to use in proofs.
- The relations `Val.inject` and `memval_inject` are simpler than `inject`.
- There are less proven properties of `Val.inject` and `memval_inject` in CompCert than of `inject`.

Thus we provide three new relations analogous to injection relations:

Relation between:	Memory injection	Memory bijection
memories	`inject`	`biject`
expression values	`Val.inject`	`biject_value`
memory values	`memval_inject`	`biject_memval`

The following sections study `biject_value` and `biject_memval` relations.

5.3 Relation on expression values

A relation `biject_value` relates two expression values fetched and decoded from m_1 and m_2 by a corresponding address. It is similar to `Val.inject` but preserves `Vundef` values; this can be observed by comparing constructors `inject_undef` of `Val.inject` (p. 20) and `val_biject_undef` below.

```
Inductive biject_value (mi: meminj) : val → val → Prop :=
    val_biject_int   : ∀ i, biject_value mi (Vint i) (Vint i)
  | val_biject_long  : ∀ i, biject_value mi (Vlong i) (Vlong i)
  | val_biject_float : ∀ f, biject_value mi (Vfloat f) (Vfloat f)
  | val_biject_single : ∀ f, biject_value mi (Vsingle f) (Vsingle f)
  | val_biject_undef : biject_value mi Vundef Vundef.

  | val_biject_ptr   : ∀ b1 b2 ofs,
                       mi b1 = Some (b2, 0) →
                       biject_value mi (Vptr b1 ofs) (Vptr b2 ofs)
```

Connection to $\hookrightarrow$. Relations `Val.inject` and `biject_value` are connected by theorems:

$$\texttt{biject\_value}\ f\ v\ v' \rightarrow v \hookrightarrow^{f} v' \qquad \text{(biject\_value\_inject)}$$

and

$$\left.\begin{array}{l} v \hookrightarrow^{f} v' \\ v' \hookrightarrow^{g} v \\ \texttt{inverse}\ f\ g \end{array}\right\} \rightarrow \texttt{biject\_value}\ f\ v\ v' \qquad \text{(biject\_inject\_value)}$$

Both theorems are proven by case analysis on v and v'.

Invertibility. The relation `biject_value` is invertible if f has an inverse function g:

$$\left.\begin{array}{l} \texttt{inverse}\ f\ g \\ \texttt{biject\_value}\ f\ v\ v' \end{array}\right\} \rightarrow \texttt{biject\_value}\ g\ v'\ v \qquad \text{(biject\_value\_inv)}$$

The invertibility is proven by case analysis on v and v'.

5.4 Relation on memory values

The predicate `biject_memval` lifts `biject_value` from `val` to `memval`. It shows how a single `memval` of m_1 is transformed in m_2; it is similar to `memval_inject`, but preserves `Undef`. This can be observed by comparing constructors `memval_inject_undef` of `memval_inject` (p. 21) and `memval_biject_undef` below.

```
Inductive biject_memval (mi: meminj) : memval → memval → Prop :=
    memval_biject_byte: ∀ i , biject_memval mi (Byte i) (Byte i)
  | memval_biject_undef: biject_memval mi Undef Undef
  | memval_biject_frag: ∀ v v' i n,
      biject_value mi v v' →
      biject_memval mi (Fragment v i n) (Fragment v' i n).
```

Connection to $\hookrightarrow$. Relations `memval_inject` and `biject_memval` are connected by the following theorems:

$$\texttt{biject\_memval}\ f\ mv\ mv' \to mv \hookrightarrow^f mv' \qquad \text{(biject\_memval\_inject)}$$

and

$$\left.\begin{array}{l} mv \hookrightarrow^f mv' \\ mv' \hookrightarrow^g mv \\ \texttt{inverse}\ f\ g \end{array}\right\} \to \texttt{biject\_memval}\ f\ mv\ mv' \qquad \text{(inject\_biject\_memval)}$$

Both theorems are proven by case analysis on mv and mv'.

Invertibility. The relation `biject_memval` is invertible if f has an inverse function g:

$$\left.\begin{array}{l} \texttt{inverse}\ f\ g \\ \texttt{biject\_memval}\ f\ mv\ mv' \end{array}\right\} \to \texttt{biject\_memval}\ g\ mv'\ mv \qquad \text{(biject\_memval\_inv)}$$

The invertibility is proven by case analysis on mv and mv'.

5.5 Properties

We now consider the basic operations on memory state: `load`, `store`, `alloc`, `free`, `loadbytes`, `storebytes` and `drop_perm`. As we have shown in Section 4.4, CompCert's $\hookrightarrow$ relation is compatible with its axiomatic memory model: if $m_1 \hookrightarrow m_2$, then the result of an operation in m_2 can be simulated by the same operation in m_1. In this section we explore analogous properties of memory bijection.

5.5.1 `load` simulation

The key property of CompCert `inject` relation is the ability to simulate `load` operation in injected memory: `load_inject` (p. 23). For `biject` a similar property is formulated as follows:

```
Theorem load_biject:
  ∀ f g m1 m2 chunk b1 ofs b2 v,
  biject f g m1 m2 →
  load chunk m1 b1 ofs = Some v →
  f b1 = Some (b2,0) →
  ∃ v', biject_value f v v' ∧ load chunk m2 b2 ofs = Some v'.
```

Proof. The proof is constructed by composing lemmas `load_inject` and `biject_inject_value`. It consists of the following steps:

- We have:

  $$m_1 \hookrightarrow^f m_2 \qquad \text{(Hfwd)}$$

  and

  $$m_2 \hookrightarrow^g m_1 \qquad \text{(Hbwd)}$$

  from `biject` $f\ g\ m_1\ m_2$ (premise).

- There exists a value v_2 such that:

  $$v \hookrightarrow^f v_2 \qquad \text{(Hinj2)}$$

  and

$$\texttt{load}\ \mathit{chunk}\ m_2\ b_2\ \mathit{ofs} = \texttt{Some}\ v_2. \tag{Hload2}$$

This follows from `load_inject` (p. 23) applied to $m_1 \hookrightarrow^f m_2$ (Hfwd) and $f\ b_1 = \texttt{Some}\ (b_2, 0)$ (premise).

- There exists a value v_3 such that:

$$v_2 \hookrightarrow^g v_3 \tag{Hinj3}$$

and

$$\texttt{load}\ \mathit{chunk}\ m_1\ b_1\ \mathit{ofs} = \texttt{Some}\ v_3. \tag{Hload3}$$

To prove that we first deduce:

$$g\ b_2 = \texttt{Some}\ (b_1, 0) \tag{Hg}$$

by `biject` definition. Then we apply `load_inject` (p. 23) to $m_2 \hookrightarrow^g m_1$ (Hbwd) and (Hg).

- We conclude $v_3 = v$ from the theorem premise $\texttt{load}\ \mathit{chunk}\ m_1\ b_1\ \mathit{ofs}\ \texttt{=}\ \texttt{Some}\ v$ and $\texttt{load}\ \mathit{chunk}\ m_1\ b_1\ \mathit{ofs}\ \texttt{=}\ \texttt{Some}\ v_3$ (Hload3).
- We get

$$v \hookrightarrow^f v_2 \tag{Hinj2}|_{v_3 \mapsto v}$$

and

$$v_2 \hookrightarrow^g v \tag{Hinj3}|_{v_3 \mapsto v}$$

from (Hinj2) and (Hinj3) after substituting v_3 for v.

- We derive

$$\texttt{biject\_value}\ f\ v\ v_2 \tag{Hmap2}$$

by applying `biject_inject_value` (p. 31) to (Hinj2)$|_{v_3 \mapsto v}$ and (Hinj3)$|_{v_3 \mapsto v}$.

- The proof is finished by choosing v_2 as a witness for v'; (Hmap2) and (Hload2) then match the required conclusion.

□

5.5.2 `loadbytes` simulation

A simulation property similar to `load` exists for raw `loadbytes` operation. It is analogous to `loadbytes_inject` and is encoded as follows:

```
Theorem loadbytes_biject:
  ∀ f g m1 m2 b1 ofs len b2 bytes1,

  biject f g m1 m2 →

  loadbytes m1 b1 ofs len = Some bytes1 →
  f b1 = Some (b2,0) →

  ∃ bytes2,
    loadbytes m2 b2 ofs len = Some bytes2 ∧
    list_forall2 (biject_memval f) bytes1 bytes2.
```

Proof. The key ideas of this proof are:

- We have to derive:

$$\begin{cases} \texttt{loadbytes}\ m_2\ b_2\ ofs\ len\ =\ \texttt{Some}\ bytes_2 \\ \texttt{list\_forall2}\ (\texttt{memval\_inject}\ f)\ bytes_1\ bytes_2 \\ \texttt{loadbytes}\ m_3\ b_3\ ofs\ len\ =\ \texttt{Some}\ bytes_3 \\ \texttt{list\_forall2}\ (\texttt{memval\_inject}\ f)\ bytes_2\ bytes_3 \end{cases}$$

  by using `loadbytes_inject` theorem.

- We have to prove for all xs and ys that

$$\left.\begin{array}{l} \texttt{list\_forall2}\ (\texttt{memval\_inject}\ f)\ xs\ ys \\ \texttt{list\_forall2}\ (\texttt{memval\_inject}\ g)\ ys\ xs \end{array}\right\} \rightarrow \texttt{list\_forall2}\ (\texttt{memval\_biject}\ f\ g)\ xs\ ys$$

  by induction on `xs`.

□

5.5.3 `alloc` commutation

Allocation changes embedding functions: the new functions should either ignore the new blocks or add a mapping for them. We start by introducing two auxiliary definitions to transform embedding functions; they return an updated embedding valid for new memories.

1. When a new block is allocated in both m_1 and m_2 and we have to include a new mapping into the embedding function:

```
Definition alloc_mapper f m1 m1' : meminj :=
  λ b ⇒ if b == nextblock m1
        then Some (nextblock m1', 0)
        else f b.
```

2. When a new block is allocated in m but we do not want to map it to anything:

```
Definition alloc_mapper_skip f m : meminj :=
   λ b ⇒ if b == nextblock m
         then None
         else f b.
```

Now we can state commutation properties for `alloc`.

The first theorem states that allocating block in m_1 preserves memory bijection; this new block stays unmapped.

```
Theorem alloc_right_biject:
  ∀ f g m1 m2 lo hi b2 m2' ,
    biject f g m1 m2 →
    alloc m2 lo hi = (m2', b2) →
    biject f (alloc_mapper_skip g m2) m1 m2'.
```

Proof. The bijection is constructed by applying `alloc_right_inject` to $m_1 \hookrightarrow m_2$ and analogously `alloc_left_unmapped_exact_inject` to $m_2 \hookrightarrow m_1$. □

The second theorem states that allocating block in m_2 preserves memory bijection; this new block has no preimage in m_1.

```
Theorem alloc_left_unmapped_biject:
  ∀ f g m1 m1' m2 lo hi nb,
    biject f g m1 m1' →
    alloc m1 lo hi = (m2, nb) →
    biject (alloc_mapper_skip f m1) g m2 m1'.
```

Proof. The bijection is constructed by applying `alloc_left_unmapped_inject` to $m_1 \hookrightarrow m_2$ and analogously `alloc_left_right_inject` to $m_2 \hookrightarrow m_1$. □

The last theorem states that allocating block in both m_1 and m_2 and mapping fresh blocks to one another preserves memory bijection.

```
Theorem alloc_biject_parallel:
  ∀ f g m1 m1' m2 lo hi nb,
    biject f g m1 m1' →
    alloc m1 lo hi = (m2, nb) →
    ∃ m2' nb', alloc m1' lo hi = (m2', nb') ∧
             biject
               (alloc_mapper f m1 m1')
               (alloc_mapper g m1' m1)
               m2 m2'.
```

Proof. The bijection is constructed by applying `alloc_inject_parallel` to $m_1 \hookrightarrow m_2$ and analogously `alloc_inject_parallel` to $m_2 \hookrightarrow m_1$. □

5.5.4 `store` commutation

There are three properties describing commutation of `biject` with store. The first property is analogous to `store_mapped_inject`.

```
Theorem store_mapped_biject:
  ∀ f g chunk m1 b1 ofs v1 n1 m2 b2 v2,

  biject f g m1 m2 →

  store chunk m1 b1 ofs v1 = Some n1 →
  f b1 = Some (b2, 0) →
  biject_value f v1 v2 →

  ∃ n2,
    store chunk m2 b2 ofs v2 = Some n2 ∧
    biject f g n1 n2.
```

Proof. The proof is decomposed into following steps:

- We have:

$$m_1 \hookrightarrow^{f} m_2 \tag{Hfwd}$$

  and

$$m_2 \hookrightarrow^{g} m_1 \tag{Hbwd}$$

and

$$\texttt{inverse}\ f\ g \tag{Hinverse}$$

from $\texttt{bject}\ f\ g\ m_1\ m_2$ (premise).

- We have:

$$v_1 \hookrightarrow^f v_2 \tag{Hinj}$$

This follows from (biject_value_inject) applied to $\texttt{bject\_value}\ f\ v_1\ v_2$ (premise).

- There exists n_2 such that:

$$n_1 \hookrightarrow^g n_2 \tag{Hinject2}$$

and

$$\texttt{store chunk}\ m_2\ b_2\ ofs\ v_2 = \texttt{Some}\ n_2 \tag{Hstore2}$$

This follows from `store_mapped_inject` (p. 24) applied to:

  - $m_1 \hookrightarrow^f m_2$ (Hfwd);
  - $f\ b_1 = \texttt{Some}\ (b_2, 0)$ (premise);
  - $v_1 \hookrightarrow^f v_2$ (Hinj).

- We derive:

$$g\ b_2 = \texttt{Some}\ (b_1, 0) \tag{Hg}$$

by applying $\texttt{inverse}\ f\ g$ (Hinverse) to $f\ b_1 = \texttt{Some}\ (b_2, 0)$ (premise).

- We have:

$$\texttt{biject\_value}\ g\ v_2\ v_1 \tag{Hmap2}$$

by applying (biject_value_inv) to $\texttt{bject\_value}\ f\ v_1\ v_2$ (premise).

- We have:

$$v_2 \hookrightarrow^g v_1 \tag{Hinj2}$$

by applying (biject_value_inject) to (Hmap2)

- There exists n_3 such that:

$$n_2 \hookrightarrow^g n_3 \tag{Hinject3}$$

and

$$\texttt{store chunk}\ m_1\ b_1\ ofs\ v_1 = \texttt{Some}\ n_3 \tag{Hstore3}$$

This follows from `store_mapped_inject` (p. 24) applied to:

  - $m_2 \hookrightarrow^f m_1$ (Hbwd).
  - $\texttt{store chunk}\ m_2\ b_2\ ofs\ v_2 = \texttt{Some}\ n_2$ (Hstore2).
  - $g\ b_2 = \texttt{Some}\ (b_1, 0)$ (Hg).

– $v_1 \hookrightarrow^f v_2$ (Hinj).

- We derive that $n_1 = n_3$ because:

$$\begin{cases} \texttt{store chunk } m_1\ b_1\ ofs\ v_1 = \texttt{Some } n_1 & \text{(Hstore2)} \\ \texttt{store chunk } m_1\ b_1\ ofs\ v_1 = \texttt{Some } n_3 & \text{(Hstore3)} \end{cases}$$

- We get:

$$n_2 \hookrightarrow^g n_1 \qquad \text{(Hinject3}|_{n_3 \mapsto n_1}\text{)}$$

  by substituting n_3 for n_1 in (Hinject3).

- We construct a new memory bijection:

$$\texttt{bject } f\ g\ n_1\ n_2 \qquad \text{(Hbiject)}$$

  by combining:

  – $n_1 \hookrightarrow^f n_2$ (Hinject2).
  – $n_2 \hookrightarrow^g n_1$ (Hinject3$|_{n_3 \mapsto n_1}$).
  – `inverse` $f\ g$ (Hinverse).

- We finish the proof by choosing n_2 as the witness for the memory state in the conclusion; then (Hstore2) and (Hbiject) match the conclusion.

□

The second commutation property for `bject` and `store` is analogous to `store_unmapped_inject` (Section 4.4):

```
Theorem store_unmapped_biject:
  ∀ f g chunk m1 b1 ofs v1 n1 m2,

    bject f g m1 m2 →
    store chunk m1 b1 ofs v1 = Some n1 →
    f b1 = None →
    bject f g n1 m2.
```

Proof. This theorem is proven by breaking `bject` relation into two inverse memory injections, applying `store_unmapped_inject` to the first one and `store_outside_inject` to the second one.

□

The third commutation property for `store` is analogous to `store_outside_inject`.

```
Theorem store_outside_biject:
  ∀ f g m1 m2 chunk b ofs v m2',

  bject f g m1 m2 →

  (∀b' ofs',
      f b' = Some (b,0) →
      perm m1 b' ofs' Cur Readable →
      ofs ≤ ofs' < ofs+ size chunk → False) →

    store chunk m2 b ofs v = Some m2' →
    bject f g m1 m2'.
```

Proof. To prove this theorem we start by breaking `biject` relation into two inverse memory injections. Then we apply `store_outside_inject` to the first one and `store_unmapped_inject` to the second one. To do that we require $g\ b$ `= None`, that is in turn proven by contradiction using `store_valid_access_3` lemma defined in CompCert. □

5.5.5 `storebytes` commutation

There are three properties describing commutation of `biject` with `storebytes`. The first property is analogous to `storebytes_mapped_inject`.

```
Theorem storebytes_mapped_biject:
  ∀ f g m1 b1 ofs bytes1 n1 m2 b δ bytes2,
  biject f g m1 m2 →
  storebytes m1 b1 ofs bytes1 = Some n1 →
  f b1 = Some (b2,δ) →
  list_forall2 (biject_memval f) bytes1 bytes2 →

  ∃ n2, storebytes m2 b2 (ofs + δ) bytes2 = Some n2 ∧ biject f g n1 n2.
```

Proof. The proof schema is analogous to the one for `store_mapped_biject`. □

The second commutation property for `biject` and `storebytes` is analogous to `storebytes_unmapped_inject` (Section 4.4):

```
Theorem storebytes_unmapped_biject:
    ∀ f g m1 b1 ofs bytes1 n1 m2,
      biject f g m1 m2 →
      storebytes m1 b1 ofs bytes1 = Some n1 →
      f b1 = None →
      biject f g n1 m2.
```

Proof. The proof schema is analogous to the one for `store_unmapped_biject`. □

The third commutation property for `store` is analogous to `storebytes_outside_inject`. The definition for `indifferent_range` is providef on p. 27.

```
  Theorem storebytes_outside_biject:
    ∀ f g m1 m2 b ofs bytes2 m2',
      biject f g m1 m2 →

      indifferent_range f b ofs (ofs+ length bytes2) →

      storebytes m2 b ofs bytes2 = Some m2' →
      biject f g m1 m2'.
```

Proof. The proof schema is analogous to the one for `store_outside_biject`.

□

5.5.6 `drop_perm` commutation

```
Theorem drop_mapped_biject_weak:
  ∀ f g m1 m2 b1 b2 δ lo hi p m1',
  biject f g m1 m2 →
  drop_perm m1 b1 lo hi p = Some m1' →
  meminj_no_overlap f m1 →
  f b1 = Some (b2, δ) →
  ∃ m2',
    drop_perm m2 b2 (lo + δ) (hi + δ) p = Some m2' ∧ biject f g m1' m2'.
```

Proof. The proof is performed by using `drop_mapped_inj` with $m_1 \hookrightarrow^f m_2$ and $m_2 \hookrightarrow^g m_1$. □

5.5.7 `free` commutation

There are three properties describing commutation of `biject` with `free`. The first property is analogous to `free_parallel_inject`:

```
Theorem free_parallel_biject:
  ∀ f g m1 m2 b lo hi m1' b' δ,
    biject f g m1 m2 →
    free m1 b lo hi = Some m1' →
    f b = Some (b', δ) →
    ∃ m2', free m2 b' (lo + δ) (hi + δ) = Some m2' ∧ biject f g m1' m2'.
```

Proof. This theorem is proven by applying `free_parallel_inject` in both directions. □

The second property is analogous to `free_left_inject`. The definition for `indifferent_range` is providef on p. 27.

```
Theorem free_left_biject:
  ∀ f g m1 m2 b lo hi m1',

    biject f g m1 m2 →

    indifferent_range g b ofs (ofs+ size_chunk chunk) →

    free m1 b lo hi = Some m1' →

    biject f g m1' m2.
```

Proof. This theorem is proven by applying `free_left_inject` with an injection $m_1 \hookrightarrow m_2$ and then `free_right_inject` in the other direction. □

The third property is analogous to `free_right_inject`: The definition for `indifferent_range` is providef on p. 27.

```
Theorem free_right_biject:
  ∀ f g m1 m2 b lo hi m2',
    biject f g m1 m2 →
    free m2 b lo hi = Some m2' →

    indifferent_range g b ofs (ofs+ size_chunk chunk) →

    biject f g m1 m2'.
```

Proof. This theorem is proven by applying `free_left_inject` with an injection $m_1 \hookrightarrow^f m_2$ and then `free_right_inject` with an injection $m_2 \hookrightarrow^g m_1$. □

5.6 Usage

In this section we show an example of how the simulation property `load_biject` is intended to be used as a part of refactoring correctness proof.

Suppose we want to establish a bisimulation between two programs (source and refactored) to prove that they behave in the same way. This is usually done by establishing how the refactored program simulates the source program and the opposite.

Forward-Simulation proof. In order to prove that the refactored program p' simulates the source program p (all the behaviors of p have a corresponding behavior in p') we craft a simulation relation $\mathcal{R}$ such that:

1. $\mathcal{R}$ holds for the initial states of p and p' and,
2. if two states s_i and s'_i are related by $\mathcal{R}$, and if there is a possible transition form s_i to s_{i+1} then we can have a transition from s'_i to a state s'_{i+1} such that $\mathcal{R}$ is preserved (Fig 19).

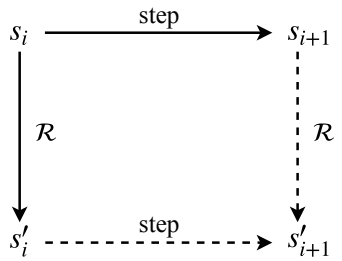


Figure 19: Single transition. We have show that s'_{i+1} exists.

The second statement is formalized as follows. For states s_1, s_2 from p and s'_1 from p' we consider we have:

- a transition $s_1 \rightarrow^e s_2$ of source program (e is an observable event);
- $\mathcal{R}(s_1, s'_1)$ holds.

We should show that there exists a state s'_2 such that:

- a transition $s'_1 \rightarrow^e s'_2$ is possible (with the same observable event e);
- $\mathcal{R}(s_2, s'_2)$ holds.

After that, we can then derive by induction that $\mathcal{R}$ is preserved after any number of transitions.

Connection with memory bijection. When memories are related by memory bijection, commutation and simulation theorems given in section 5.5 describe the results of memory operations such as `load` or `alloc`. We craft $\mathcal{R}$ so that, given arbitrary states s and s', $\mathcal{R}(s, s')$ implies `bject` between their memories. Memory operations preserve `bject` as shown in section 5.5; in CompCert C semantics memory is only modified through dedicated operations (`load`, `store`...). So when similar transitions occur in executions of p and p', their memories keep being related by memory bijection, enabling us to reason about memory in further transitions.

Example of a case involving load. Consider a transition in a source program p that occurs when a value is read from memory. This transition corresponds to evaluating an expression `Evalof (Eloc` b δ`)` (immediate address (b, δ)) obtaining `Eval` v (a value stored by this address). The memory is not changed by this transition.

Figure 20 depicts how this transition is encoded at different levels of abstraction, the most abstract at the bottom (`step`), the most concrete at the top (in this case, concrete memory operation `load`). Each level is defined through the next level: the definition of `step` uses `estep` and so on.

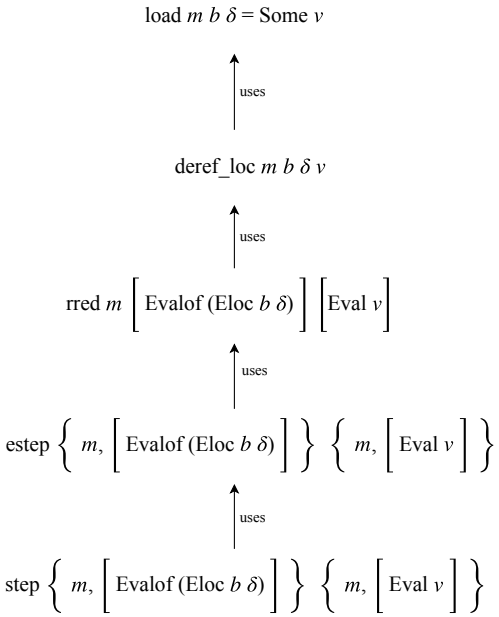


Figure 20: A transition that performs `load`. Notation: states are enclosed with { } and expressions with [ ].

Finally, on the last level we have:

$$\texttt{load}\ \ m\ b\ \delta = \texttt{Some}\ \ v$$

To construct a similar step in refactored program p' we have to construct a matching `load` first in memory m'. This is possible because memory bijection holds for m and m'.

Having:

$$\begin{cases} \texttt{load}\ \ m\ b\ \delta = \texttt{Some}\ \ v \\ \texttt{biject}\ \ f\ m\ m' \end{cases}$$

by applying load simulation property `load_biject` (p. 32) we derive:

$$\exists v', \begin{cases} \texttt{load}\ \ m'\ b'\ \delta = \texttt{Some}\ \ v' \\ \texttt{biject\_value}\ \ f\ v\ v' \end{cases}$$

Using this new `load` we are able to gradually build a matching instance of `step` for refactored program, starting with `deref_loc`.

We also provide some results on high level operations `deref_loc` and `assign_loc` but we do not explain them in the report : see for instance `biject_deref_loc` and `biject_assign_loc` in the provided Coq source code.

6 Related work

We are interested in verifying correctness of a C refactoring: removing a local unused variable. There are different approaches to ascertain that a refactoring does what it should do; we explore them in Section 6.1.

Our approach is to construct a machine-checked proof of correctness. Such proof requires a formalized C language semantics. We have chosen the version of C semantics provided by CompCert, a certified C compiler written in Coq. Different formalizations of C are explored in Section 6.2. We prove a bisimulation property to ascertain semantic preservation by refactoring. This has required us to construct a bisimulation relation between memory states of source and refactored programs. In order to do so we have extended the memory model of CompCert with a new memory relation preserved by memory operations. Section 6.3 explores other projects that extend CompCert memory and introduce new memory relations.

6.1 Refactoring correctness

Most mainstream general-purpose programming languages such as C, Java, C++ are so complex that their descriptions take hundreds of pages in language standards. Creating and implementing a correct refactoring algorithm for such languages is error-prone.

To increase our confidence to the refactoring algorithm we can validate it by testing or apply formal methods.

6.1.1 Testing

Using tests for software validation is a common choice in industry. However, applying this technique to validate refactoring algorithm correctness poses particular challenges due to the complexity of input programs. While our main approach is to apply formal methods instead, we believe that testing and formal verification are complimentary processes [6]. We are using a small number of tests for sanity checks before diving into proofs.

Mongiovi et al (2014) A work of Mongiovi et al. [27] relies on automated testing to check user-provided invariants after refactoring. After performing an impact analysis with the tool SAFIRA [26] the analysis results are used to automatically generate tests. A human needs to provide refactoring correctness invariants checked by the tests. The tests invoke functions with randomized parameters and check if the invariants still hold.

Ichii et al (2016) The work of Ichii et al. [15] tests the composition of a refactoring and its inverse for being an identity transformation. The authors suppose that doing mutually exclusive errors in a refactoring and its inverse is unlikely. This method actually only tests the inversibility of a refactoring, but not whether it does what a programmer expects.

6.1.2 Formal methods

Formal methods are an alternative to validation by testing that is usually seen in safety-critical software or in cases where crafting an efficient test suite is hard. There are not a lot of works applying formal methods to verify refactorings of general purpose languages due to the complexity of languages. They differ by the extent they approximate the target language (subset or a model) and by proven refactoring algorithm's properties. We are inspired by the work of J. Cohen [6] who certified renaming of a global variable for a full C semantics described in CompCert and proved full semantics preservation (bisimulation property) for his transformation.

Model language or subset To prove refactoring correctness formally a formal semantics for the target language is required. It is hard to formalize even a significant subset of an industrial scale language, capturing most language features. CompCert defines a small-step semantics with over 60 reduction rules for expressions and statements. The semantics of C in K framework [10] needs 77 rules to cover the behavior of statements, and another 163 for expressions. To our knowledge, only the work of J. Cohen [6] is based on a quasi-complete formalization of C; we base our work on the same semantics.

A different approach is to create a model language that has the most relevant features of the target language. This language will have a simpler semantics allowing us to study its properties and prove them formally with less effort. Of course, the target language does not necessarily have the properties proven for the model language. However, studying a model can reveal corner cases where the refactoring algorithm does not behave as intended.

Sanity check properties or full semantic preservation Whether we have selected the formalization of the target language itself or a model language, we then have to write a specification for the refactoring algorithm and prove the algorithm correct with regards to it. The specification can be complete and describe the full semantics preservation [6], or partial and only include some necessary, but not sufficient properties [24, 35]. Such properties can guarantee, for example, that some parts of the program stay unmodified but do not guarantee that new parts

Notions of semantics preservation There are multiple notions of semantic preservations such as forward, backward simulation, specification preservation and bisimulation [20]. Bisimulation is the strongest among them and implies all others. For a program and its transformed counterpart, bisimulation guarantees that the whole set of behaviors is preserved and no new behaviors appear. Proofs of refactoring correctness are proving the bisimulation property.

Proving full semantics preservation by a code transformation is usually done in one of two ways:

- Explicit bisimulation construction [14].
- Double-pushout rewritings with borrowed contexts; see [9, 31].

A work [14] contains a comparison of these techniques.

CompCert uses bisimulation to prove that the a determinized C program and its compiled counterpart match [22]; In order to do that, the forward simulation is performed, and then determinism of internal compiler languages is used to derive bisimulation (see lemma `forward_to_backward_simulation` in `Smallstep` module). CompCert also uses backwards simulation to prove that compiled program has a behavior selected from all valid behaviors of a source C program.

Double-pushout rewritings with borrowed contexts is a more recent and less studied approach, that has only been tested on toy models. By adapting this approach it is possible to automatically find which transitions in the refactored program should correspond to the transitions in the source program. However, the parameters of these transitions are not derived.

In our case, the correspondence between the transitions in the source program and its refactoring counterpart is trivial: our refactoring produces small local changes in the program's source code and does not change its control flow. Because of that, the refactored program will always mimic the transitions of the original program. This neglects for us the possible benefits of adapting this technique for the proof of correctness.

Li et al (2005) A work by Li et al. [24] studies a refactoring of Haskell programs that moves a definition into another module. It approximates Haskell with a simple lambda-calculus with module system. Then they prove certain invariants preserved by this refactoring, but not the full semantics preservation. For example, they prove, that after refactoring all definitions stay unchanged (but each one could be moved between modules as whole). Their proof is performed on paper and is not machine-checked.

Sultana et al. The work of Sultana et al. [35] studies several formal refactorings of typed and untyped λ-calculus in HOL/Isabelle, including add/drop a redundant definition. This refactoring is an analogue of ours.

6.2 Formal C semantics

6.2.1 Language standard

Starting from 1989, the C language is described by the C language standard [4], a document created by a International Organization for Standardization (ISO). New standards appear from time to time, expanding and precising the language syntax and semantics. Some bits of language are described formally, such as the language syntax, which is defined by a formal grammar using BNF. Most parts, however, are described in plain English, and thus are prone to misinterpretation. The size of the language standard makes a formalization that accounts for all language features and accurately describes all corner cases hard to achieve. The informality poses an additional challenge for that. When the standard can be interpreted in several different ways, the creators of formalization should chose one interpretation. Sometimes the committee clarifies the ambiguity in ways contradictory to the common practices and programmers' expectations [2].

The following features currently pose a particular challenge for a modern C formalization:

- Full syntax support, including rarely used features such as non-structured `switch` (see Duff's device [8]), arrays with size guarantees by means of `static` keyword as function arguments [4, p.132], keywords `alignof` and `alignas` etc.
- Support for byte-grained memory model to reason about unions and reinterpreting data; strict aliasing rules should be taken into account.

  The standard allows us to write a value of type `int` into a `char` array, and then get this value back as an `int` through pointer dereferencing and type conversions.
- Support for multithreading and different kinds of memory models, not only the usual sequentially consistent one [30].

The standard purposefully leaves the language underspecified to make it portable to a wide variety of platforms and for performance reasons (enabling more compiler optimizations). For example, it states that the number of bits in a byte is defined by the macrodefinition `CHAR_BIT`. It is guaranteed that `CHAR_BIT` is *at least* 8, but its exact value is not fixed. This makes formalizing harder in case we want to create an executable semantics. Executable semantics should have such non-exact values as parameters, so that the programs would be able to use them in computations.

Having an executable semantics is beneficial for refactoring correctness checks. CompCert provides an executable semantics [5] that allows us to simulate all execution orders for the source C programs and their refactored counterparts. This semantics was extensively tested by specialized tools [36] and by industrial use [23, 16]. Comparing these two sets of traces allows us to test the refactoring correctness of exemplary programs.

6.2.2 History of C language formalization

The history of formalizing C language semantics to use in machine-checked proofs spans over several decades. It starts with the simpler and smaller formalizations [7, 29], that leave many parts of the language uncovered. Then through a series of approximations, language semantics has approached the description given in the language standard. Different styles of semantics were used:

- Denotational semantics [7]
- Operational semantics
  - small-step for expressions, big-step for statements, based on Logic for Computable Functions [29];

  - small-step for expressions and statements, based on Calculus of Inductive Constructions [17, 23, 2].
  - based on Rewriting Logic [10].

- Axiomatic semantics based on a variant of Separation Logic [17].

Today, even the most advanced solutions still diverge from the standard in some situations. There are several kinds of such situations, none of which means that the formalization is *incorrect*:

1. The program triggers an undefined behavior, but the formal semantics gives a defined behavior to the program. When the standard implies undefined behavior, any behavior is considered correct, so formal semantics does not contradict the standard.

   The following example has an undefined behavior according to the C standard and [18], but has a defined behavior in the CompCert C semantics that we use. This is due to the variable `i` being modified twice between two consecutive sequence points. The reasons for this will be explained later in this section.

```
void f() {
  int i = 0;
  i = ++ i + i ++;
}
```

2. The program has a defined behavior, but the formal semantics assigns an undefined behavior to it. In this case formal semantics can only be used to reason about subset of all valid programs.

   For example, the C language standard describes the functions `longjmp` and `setjmp`. However, they are not described in CompCert C semantics. and using them triggers undefined behavior according to it. These two functions were formalized as a part of other semantics like the one proposed by Ellison et al. [10].

Recent works [10, 17, 2, 19] concentrate on eliminating specific situations where existing semantics diverge from the language standard.

We are now going to overview the related works.

Cook and Subramanian (1994) To our knowledge, the first attempt to formalize a realistic semantics of C using proof assistant was performed by Cook and Subramanian [7]. It was a semantics for a subset of C embedded in the theorem prover Nqthm [3] (a precursor to ACL2). This subset included `int` and `void` types, most statements except for `switch`, `do while` and `for`, expressions for integer and pointer arithmetic and array support. The formalization was based on denotational semantics of C90 on top of a custom-made temporal logic [7]. The authors have succeeded in verifying at least two functions: one that takes two pointers and swaps the values at each, and another computing the factorial. They have also proved properties of statements, for example, that `p`=&`a`[`n`] puts the address of the nth element of the array `a` into `p`. Comparing to the semantics of CompCert that we use it covers a significantly smaller part of the language. The denotational approach might be beneficial to prove program equivalence, which is the cornerstone of verifying refactorings.

Norrish (1998) A major step in formalizing C was performed by Norrish in HOL[29]. His work formalized most control structures, expression evaluation, memory accesses, using small-step semantics for expressions and big-step semantics for statements. CompCert also provides an operational semantics for C, but it has a small-step semantics for statements. This makes it suitable to reason about programs that do not terminate. Norrish does not provide an executable semantics; in some cases, he specifies semantics in an axiomatic way, which is an incompatible approach. For example, in Norrish's semantics, the said `CHAR_BIT` macro's value can not be known, but a proposition "`CHAR_BIT`≥ 8" can be derived. Such approach follows the standard closer, but does not allow the executability. This formalization has a number of limitations such as:

- No support for `switch` and `goto`.
- No IO support.
- Type system does not model `enum` and `union` types, `volatile` and `const` qualifiers, `float`s.

Norrish has used this semantics to prove some properties of the small (4-5 lines) programs, but was unable to apply his work for larger programs. As the semantics is not executable, it has not been tested against actual programs.

Ellison et al. (2010) A work of Ellison et al. [13] presents an executable C semantics encoded inside K framework[32].

K framework is a rewrite-based executable semantic [25] framework used to define type systems, programming languages and formal analysis tools. Its core concepts are _configurations, *computations* and *rules*. Configurations organize the system state in labeled units (called cells), which can be nested. Computations carry the "computational meaning" sequentializing computational tasks as sequences of program fragments. Rules generalize rewriting rules by making explicit which parts of the term they read, write or do not care about.

The major goal of Ellison is to formalize C in K proving that K is capable of handling semantics for industrial-scale languages. It makes a special focus on formalizing *negative* semantics of C, which means explicitly identifying the cases of undefined behavior. The standard approach is to make undefined behavior implicit in case no transition is possible from the current program state. We could not benefit from negative formalization in scope of our work because we want to preserve undefined behavior, but not differentiate between contexts where it occurs. As in CompCert, this semantics is also executable. Unlike CompCert, Ellison's semantics supports reasoning about `longjmp` and `setjmp` and unstructured `switch`, from which we could have benefited in future refactoring proofs.

It seems to us that using this semantics for refactoring correctness proofs is adequate. However in doing so we would lose compatibility with the work of J. Cohen [6] performed in Coq on base of CompCert C semantics. We want to preserve the compatibility to obtain more complex certified refactorings with small effort by composing already certified refactorings.

Krebbers et al (2014): formalizing aliasing rules The work of Krebbers et al. [19] concentrates on formalizing strict aliasing rules – a part of the standard that enables compilers to do more aggressive optimizations.

A pointer is *aliasing* another pointer if they refer to the same location in memory. In C it is illegal to create an alias of a different type than the original. This is known as *strict aliasing rule*.

The purpose of the strict aliasing rule is to help compiler in generating efficient code. Optimized code often caches a value read from memory in registers for later reuse. Following strict aliasing rule guarantees that writing by a pointer of some type T will not overwrite data of another type U. As a consequence, all data of type U in registers remains up-to-date. For data of type T, further analysis is required to optimize reads.

Dereferencing an aliased pointer of different type triggers undefined behavior. With optimizations related to type-based alias analysis turned on, such programs might not execute correctly. Disabling these optimizations will force compiler to forget about strict aliasing rules and produce a program that behaves as intended.

Krebbers et al. provide a semantics with memory model that keeps track of the types of data stored in memory. Accessing data of some type through a pointer of different type triggers undefined behavior.

The current memory model of CompCert does not account for aliasing rules, so its semantics assign a defined behavior to programs that should actually be undefined. The compilation correctness is unaffected because CompCert does not perform type-based alias analysis.

Moreover, CompCert's memory model is not expressive enough to capture all information needed to enforce strict aliasing rules: To enforce them, it is necessary to know the exact type of data stored in memory.

First, data of numeric types stored in memory have their types erased and represented as raw `Byte`s.

Second, we do not know whether data is a part of union or not. According to the standard we can only reinterpret data by accessing it through different members of a union type. Krebbers provides an example of two functions `f` and `g` that seem equivalent but of which only `f` has a defined behavior:

```
union U { int x; float y; };
int f() { union U t;                  t.y = 3.0; return t.x; }
int g() { union U t; int *p = &t.x; t.y = 3.0; return *p;  }
```

The function `g` performs seemingly the same actions as `f`, but the access to the field `x` is performed through a distinct pointer, not through the union. The information about data being part of union is lost in CompCert memory model, so making a distinction between accesses through union and through a direct pointer is not possible.

Considering refactoring correctness proofs, we do not want a program that follows strict aliasing rules to break them after refactoring. In this context, supporting reasoning about strict aliasing rules is beneficial for other refactorings that might change the types of data or the ways data is accessed. As CompCert semantics considers correct both functions `f` and `g` in the example above, it might be possible that a refactoring transforms `f` into `g` and produces a program that violates strict aliasing rules. However, our current refactoring – removing an unused local variable – is not capable of breaking any aliasing rule because it has no effect on memory reads and writes, only allocations. Because of that in our case CompCert semantics is just as good for refactoring correctness proofs.

Krebbers et al (2014): formalizing sequence points In other work Krebbers et al. [17] concentrate on formalizing sequence points – particular places in code where all effects of preceding computations are guaranteed to happen, and none of the effects of later computations have happened.

Despite the seeming sequential nature of C, it does not force a single fixed order of computations. Compiler has a right to reorder them to a great extent to generate faster code. Sequence points limit reordering: compiler can not move computations further than the next sequence point occurs. Some of the most frequently seen sequence points are semicolon operator and function calls.

No object in memory should be modified twice between two consecutive sequence points, because such modifications will occur in an undefined order. Violating this rule leads to undefined behavior.

The solution proposed by Krebbers et al. is based on modified small step separation logics by Krebbers and Wiedijk [18]. This is a separation logic extended with non-deterministic expressions with side-effects and the restriction on sequence points. Memory model marks objects to catch when the object might be modified more than once before the next sequence point is reached.

CompCert does not detect the undefined behavior in all such cases. For example, it considers the statement `i=i++` to have a defined behavior whereas the language standard considers it undefined.

Complex refactorings might change program so that some sequence points are omitted or new ones are introduced. It is not trivial to find out whether that would introduce undefined behavior or not. However, our current refactoring (removal of unused variable) modifies no computations or expressions. CompCert C semantics is just as good for refactoring correctness proofs.

6.3 Extensions to CompCert memory model

We needed to describe transformations like memory injection but preserving undefined values in memory. We have extended the CompCert memory model with an additional memory transformation class.

There are two limitations to the current memory model that motivated other authors to extend the memory model in different ways. These limitations are:

1. The inability to reason about concrete pointer values.

   As of CompCert 3.3, pointers are represented as a pair of integers: block index and offset. Such pointers are not representable as fixed-range integer numbers. In practice, programmers are often interested in treating pointers like integers. It is natural because in many popular platforms (Intel 64, ARM, MIPS) pointers are represented as multibyte integers[7]. The examples of real world situations when such need occurs are tagged pointers (storing bits of data in unused bits of pointers themselves) and xor-lists [33].

2. Modeled memory is infinite, so allocations never fail. In real world, allocating dynamic memory can fail with `malloc` returning `NULL`. Additionally, if pointers were representable as finite integers, it would impose a natural limit on the memory size.

[7] It is not the only way; for example, in older Intel architectures pointers were pairs of segment starting address and offset.

Several projects are extending CompCert memory model to alleviate these limitations. They also have to provide their own ways of matching memory states akin to memory injections in CompCert.

Mullen et al. (2016) Peek [28] is a framework to perform verified assembly-level optimizations in CompCert, such as peephole optimizations.

Peephole optimizations analyze the assembly and try to recognize a known instruction pattern, for example `add x, 42` followed by `sub x, 42`. Their occurrences are then replaced with equivalent instruction sequences, to make overall code faster.

In CompCert, many correct peephole optimizations are not provable because arithmetic operations are undefined (or partially defined) over pointer values. To support them, the authors add a new CompCert assembly semantics where pointers are represented as concrete 32-bit integers so more operations will be defined over pointers. This change is propagated to memory model and memory injections.

However, Peek does not define memory transformations that preserve undefined values. Thus it does not solve the problem, for which we have defined memory bijection.

Peek is plugged in CompCert on the backend level. The C language semantics defined in CompCert can not work with Peek's memory model out of the box. We only need to work with the C semantics; adapting Peek's memory model would require us to modify it.

Besson et al. CompCertS is a modified version of CompCert [2], featuring a stronger memory model. It modifies CompCert on all compilation levels to work with symbolic values. A symbolic value is essentially a C expression, whose evaluation is deferred until needed. The evaluation occurs e.g. when memory accesses are performed or when the program has to decide between execution branches (`if`, `while` and similar statements).

Such modification allows us to reason about previously undefined values. For example, suppose `x` is an uninitialized local variable. In CompCert, the value of expression `x-x` is undefined, because `x` is undefined. However, by using the symbolic values we exploit the idea of undefined values being stable. It means that repeated readings from the same memory locations will yield the same value, albeit an arbitrary one. This implies that even without knowing the exact value of `x`, the symbolic expression `x-x` can be evaluated to zero.

Modifying the value domain of CompCert C propagates to all language semantics through all compilation stages until assembly. The memory model and memory transformation classes used to prove their correctness are updated accordingly. Memory injection is used to certify several compilation stages, so it is modified to work with symbolic values as well. However, this new memory injection still relates undefined values with any values (albeit there are less undefined values now). So, this memory model modification complicates the reasoning, but does not help us in constructing a bisimulation memory relation that preserves undefined values.

Additionally, the changes in semantics make these properties incompatible with the work of J. Cohen without heavily modifying the latter. Our intention is however to reuse his work to be able to statically compose his refactorings with ours, obtaining more complex refactorings, for which the additional certification is not needed.

7 Conclusion

Contributions. In this document we have studied memory bijection: a crafted relation between memories to model relocations that preserve undefined values. It is intended to be used to prove the semantic preservation of refactorings through bisimulation. We have proven some properties of simulation and commutation for memories related by bijection in Coq.

Future work. Many refactorings induce an effect on memories that we can describe using memory bijections: "Add/Remove a Global Variable", "Add/Remove a Local Variable", to name a few. We plan to implement and verify several such refactoring operations. Memory bijection is to be reused in their proofs.

References

[1] The CompCert C Compiler. `http://compcert.inria.fr/compcert-C.html`.

[2] Frédéric Besson, Sandrine Blazy, and Pierre Wilke. CompCertS: A Memory-Aware Verified C Compiler using Pointer as Integer Semantics. In *ITP 2017 - 8th International Conference on Interactive Theorem Proving*, volume 10499 of *ITP 2017: Interactive Theorem Proving*, pages 81–97, Brasilia, Brazil, September 2017. Springer.

[3] Robert S. Boyer and J Strother Moore. *A computational logic handbook: Formerly notes and reports in computer science and applied mathematics*. Elsevier, 2014.

[4] C11 language standard – Committee Draft. `http://www.open-std.org/jtc1/sc22/wg14/www/standards`, April 2011.

[5] Brian Campbell. An executable semantics for CompCert C. In *International Conference on Certified Programs and Proofs*, pages 60–75. Springer, 2012.

[6] Julien Cohen. Renaming global variables in C mechanically proved correct. *Electronic Proceedings in Theoretical Computer Science*, 216:5064, Jul. 2016.

[7] Jeffrey Cook and Sakthi Subramanian. A formal semantics for C in Nqthm. Technical report, Trusted Information Systems, 1994.

[8] Tom Duff. The description of Duff's device – a mail in comp.lang.c. `http://www.lysator.liu.se/c/duffs-device.html`, August 1988.

[9] Hartmut Ehrig and Barbara König. Deriving bisimulation congruences in the DPO approach to graph rewriting. In *International Conference on Foundations of Software Science and Computation Structures*, pages 151–166. Springer, 2004.

[10] Chucky Ellison and Grigore Rosu. An executable formal semantics of C with applications. *ACM SIGPLAN Notices*, 47(1):533–544, 2012.

[11] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[12] James Gosling, Bill Joy, and Guy Steele. *The Java language specification*. Addison-Wesley Professional, 2000.

[13] Hathhorn, Chris and Ellison, Chucky and Rosu, Grigore. Defining the undefinedness of C, 2015.

[14] Mathias Hülsbusch, Barbara König, Arend Rensink, Maria Semenyak, Christian Soltenborn, and Heike Wehrheim. Showing full semantics preservation in model transformation – a comparison of techniques. In *International Conference on Integrated Formal Methods*, pages 183–198. Springer, 2010.

[15] Makoto Ichii, Daisuke Shimbara, Yasufumi Suzuki, and Hideto Ogawa. Refactoring Verification Using Model Transformation. In *Proceedings of the 1st International Workshop on Software Refactoring*, IWoR 2016, pages 17–24, New York, NY, USA, 2016. ACM.

[16] Daniel Kästner, Jörg Barrho, Ulrich Wünsche, Marc Schlickling, Bernhard Schommer, Michael Schmidt, Christian Ferdinand, Xavier Leroy, and Sandrine Blazy. CompCert: Practical experience on integrating and qualifying a formally verified optimizing compiler. *ERTS 2018: Embedded Real Time Software and Systems*, January 2018.

[17] Robbert Krebbers. An Operational and Axiomatic Semantics for Non-determinism and Sequence Points in C. *SIGPLAN Not.*, 49(1):101–112, January 2014.

[18] Robbert Krebbers and Freek Wiedijk. Separation logic for non-local control flow and block scope variables. In *International Conference on Foundations of Software Science and Computational Structures*, pages 257–272. Springer, 2013.

[19] Krebbers, Robbert. Aliasing Restrictions of C11 Formalized in Coq. In Gonthier, Georges and Norrish, Michael, editor, *Certified Programs and Proofs*, pages 50–65. Springer, 2013.

[20] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363, 2009.

[21] Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. *The CompCert Memory Model, Version 2*. Research report, June 2012.

[22] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. CompCert - A Formally Verified Optimizing Compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, Toulouse, France, January 2016.

[23] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. CompCert-A Formally Verified Optimizing Compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, 2016.

[24] Li, Huiqing and Thompson, Simon J. Formalisation of Haskell refactorings. volume 20, page 160, 2005.

[25] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical computer science*, 96(1):73–155, 1992.

[26] Melina Mongiovi. Safira: A Tool for Evaluating Behavior Preservation. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '11, pages 213–214, New York, NY, USA, 2011. ACM.

[27] Melina Mongiovi, Rohit Gheyi, Gustavo Soares, Leopoldo Teixeira, and Paulo Borba. Making refactoring safer through impact analysis. *Science of Computer Programming*, 93:39–64, 2014.

[28] Eric Mullen, Daryl Zuniga, Zachary Tatlock, and Dan Grossman. Verified Peephole Optimizations for CompCert. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 448–461, New York, NY, USA, 2016. ACM.

[29] Norrish, Michael. C formalised in HOL. Technical report, University of Cambridge, Computer Laboratory, 1998.

[30] Jeff Preshing. Weak vs. Strong Memory Models. Blog post. `http://preshing.com/20120930/weak-vs-strong-memory-models/`, 2012.

[31] Guilherme Rangel, Leen Lambers, Barbara König, Hartmut Ehrig, and Paolo Baldan. Behavior preservation in model refactoring using DPO transformations with borrowed contexts. In *International Conference on Graph Transformation*, pages 242–256. Springer, 2008.

[32] Grigore Rosu. An overview of the K semantic framework. *The Journal of Logic and Algebraic Programming*, 2010.

[33] Prokash Sinha. A Memory-Efficient Doubly Linked List. `https://www.linuxjournal.com/article/6828`, 2011.

[34] Gustavo Soares Soares. Automated Behavioral Testing of Refactoring Engines. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, pages 49–52, New York, NY, USA, 2012. ACM.

[35] Nik Sultana and Simon Thompson. Mechanical Verification of Refactorings. In *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '08, pages 51–60, New York, NY, USA, 2008. ACM.

[36] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. *SIGPLAN Not.*, 46(6):283–294, June 2011.