Verified Compilation of C Programs with a Nominal Memory Model

ANONYMOUS AUTHOR(S)

Memory models play an important role in verified compilation of imperative programming languages. A representative one is the block-based memory model of CompCert—the state-of-the-art verified C compiler. Despite its success, the abstraction over memory space provided by CompCert's memory model is still primitive and inflexible. In essence, it uses a fixed representation for identifying memory blocks in a global memory space and uses a globally shared state for distinguishing between used and unused blocks. Therefore, any reasoning about memory must work uniformly for the global memory; it is impossible to individually reason about different sub-regions of memory (i.e., the stack and global definitions). This not only incurs unnecessary complexity in compiler verification, but also poses significant difficulty for supporting verified compilation of open or concurrent programs which need to work with contextual memory, as manifested in many previous extensions of CompCert.

To remove the above limitations, we propose an enhancement to the block-based memory model based on *nominal techniques*; we call it the *nominal memory model*. By adopting the key concepts of nominal techniques such as *atomic names* and *supports* to model the memory space, we are able to *1)* generalize the representation of memory blocks to any types satisfying the properties of atomic names and *2)* remove the global constraints for managing memory blocks, enabling flexible memory structures for open and concurrent programs.

To demonstrate the effectiveness of the nominal memory model, we develop a series of extensions of CompCert based on it. These extensions show that the nominal memory model *1)* supports a general framework for verified compilation of C programs, *2)* enables intuitive reasoning of compiler transformations on partial memory; and *3)* enables modular reasoning about programs working with contextual memory. We also demonstrate that these extensions require limited changes to the original CompCert, making the verification techniques based on the nominal memory model easy to adopt.

1 INTRODUCTION

Memory models are critical components for formalizing the semantics of imperative programming languages. They determine an abstraction over memory and provide necessary operations for manipulating memory states at the corresponding level of abstraction. In the setting of verified compilation of imperative programs, a memory model with appropriate abstraction does not only greatly reduce the complexity of verification, but also enables rich forms of semantics preservation. Therefore, a lot of research has been done for developing memory models for verified compilation, ranging from highly abstract models that treat memory states as mappings from addresses to values to highly concrete ones that capture the linear layout of physical memory [Tuch et al. 2007].

Among the memory models developed by researchers, the most representative one is the *block-based memory model* [Leroy et al. 2012; Leroy and Blazy 2008] defined in CompCert—the state-of-the-art verified C compiler [Leroy 2021]. It provides a medium level of abstraction—neither too abstract nor too concrete—by modeling memory space as a collection of contiguous memory regions (also called *blocks*). Isolation between different memory blocks is simply captured by giving each of those memory blocks a unique identifier (called its *block id*). Furthermore, pointers are naturally defined as pairs of block ids and offsets to memory cells, and definitions of pointer operations follow in a straightforward manner. CompCert's block-based memory model has been highly successful. By uniformly applying it to all of CompCert's languages, the developers of CompCert were able to verify over 20 compiler passes containing advanced optimizations. It has also been widely adopted in other verification projects, including various extensions of CompCert

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

for supporting compositionality and concurrency (e.g., [Besson et al. 2017; Jiang et al. 2019; Kang et al. 2016; Sevcík et al. 2011, 2013; Song et al. 2020; Stewart et al. 2015; Wang et al. 2019, 2020]) and formalization of language semantics for program verification and program analysis (e.g., [Appel 2011; Gu et al. 2015, 2018]).

1.1 Deficiency of the Block-Based Memory Model

Despite its previous success, CompCert's block-based memory model is still quite primitive and inflexible, making it difficult to support intuitive and flexible reasoning in verified compilation. This include the following points:

- **Inflexibility 1: Fixed Representation of Block IDs.** In the block-based memory model, block ids are represented by a fixed type, i.e., positive numbers. With this uniform and fixed representation, it is impossible to distinguish between memory regions playing different roles (such as the stack, heap and memory regions for global definitions). Therefore, it is difficult to reason about specific parts of memory in isolation.
- **Inflexibility 2: Sequential Numbering of Valid Blocks.** In any memory state, a special positive number named `nextblock` provides the next available block id. The global memory space is divided by `nextblock` into two parts: blocks with ids less than `nextblock` have already been allocated (called *valid blocks*), while the remaining blocks are waiting to be allocated (called *invalid blocks*). In any program semantics, the ids of valid blocks must be numbered sequentially by $[1, \ldots, \texttt{nextblock} - 1]$. This creates unintuitive and unnecessary dependency between valid blocks playing different roles.
- **Inflexibility 3: Global Constraint for Allocation.** In any memory state, there is only a single `nextblock` for assigning new block ids upon allocation. In the setting where multiple open programs or threads work on the same memory state, every open program or thread must keep track of how other programs or threads modify `nextblock`. This global constraint imposes a complex dependency between open programs and threads.

Because of the complex dependencies created by the above inflexibilities, verification of any compiler transformation must treat the memory space *as a whole*. This incurs significant difficulties both in verified compilation of whole programs and in that of open programs. Specifically, in the former setting, many compiler transformations only work on certain sub-parts of memory (e.g., transformations on stack frames). However, because of **Inflexibilities 1 and 2**, the reasoning must be lifted to the whole memory, therefore becoming significantly more involved and less intuitive. In the latter setting, one would like to get a compositional approach to verifying open or concurrent programs. However, such an approach must be compatible with the evolution of `nextblock` which is inherently non-compositional by **Inflexibilities 2 and 3**. This problem plagues many projects on extending CompCert to support compositionality or concurrency (e.g., CASCompCert [Jiang et al. 2019] and Thread-Safe CompCertX [Gu et al. 2018]). To circumscribe it, various ad hoc modifications to the block-based memory model were invented, leading to verification results that are overly technical and less reusable.

1.2 Nominal Memory Model and Verified Compilation

In this paper, we develop a systematic, clean and lightweight approach to eliminating the above inflexibilities of the block-based memory model. Our approach is based on *nominal techniques* [Gabbay and Pitts 2002; Pitts 2016]. Specifically, by adopting the very basic concepts of nominal techniques to formally model the block-based memory space, we obtain in a natural generalization of the block-based memory model which we call the *nominal memory model*. Our key ideas include the following:

- **Generalizing Block IDs to Atomic Names**. In nominal techniques, the elements of countably infinite sets are used to represent available names; they are called *atomic names*. By generalizing the fixed type of block ids to a type parameter representing countably infinite sets, we allow block ids to be freely instantiated by any kinds of atomic names. For example, because the positive numbers are countably infinite, the original block ids become a special case after the generalization. More importantly, the type of block ids may be instantiated with rich data types, by which we can divide global memory into separate memory regions with clear roles. This eliminates **Inflexibility 1**.
- **Generalizing Valid Blocks to Supports**. In nominal techniques, dependency of objects on names is described via the concept of *supports*. In its most basic form, the *support* of an object is a finite set of atomic names that the object contains. By generalizing valid blocks to a type parameter that satisfies the basic requirements of supports, we allow new names to be freely generated as long as they are fresh w.r.t. the support. This eliminates **Inflexibility 2**.
- **Eliminating Global Constraints via Supports**. Like the type of atomic names, the support type can also be instantiated with sophisticated data types. This enables formalization of memory space with complex structures (e.g., multiple call stacks) and separate growth of memory regions for individual open modules or threads. This eliminates **Inflexibility 3**.

To demonstrate that the nominal memory model can indeed enable more flexible and intuitive reasoning about compiler transformations on both whole programs and open programs, we develop a series of extensions of CompCert based on it. First, we develop *Nominal CompCert*, a full extension of CompCert with a basic nominal memory model that is parameterized over the types of block ids and supports. Nominal CompCert provides a skeleton for developing richer extensions of CompCert that exploit the full power of the above generalization. Second, we instantiate the block ids and supports with rich data types for explicitly dividing global memory into separate regions based on their roles. With this specialized nominal memory space, we give precise definitions of transformations on partial memory as injection functions, resulting in much more intuitive proofs for the key passes of CompCert. Third, we exploit rich instantiation of the support type to verify compilation of open concurrent programs. Specifically, we develop *Multi-Stack CompCert*, a combination of Nominal CompCert and Stack-Aware CompCert [Wang et al. 2019] with further support for multiple call stacks. With the flexibility to independently grow individual stacks, Multi-Stack CompCert becomes the first compiler that supports verified compilation of multi-threaded programs all the way down to multi-stack machines, which provides a new foundation for thread-safe compilation of Certified Concurrent Abstraction Layers [Gu et al. 2018].

1.3 Contributions

We summarize our contributions as follows:

- We introduce the nominal memory model, a natural extension of the block-based memory based on nominal techniques (Sec. 3.2). It enables flexible formalization of block identifiers and memory space and eliminates global assumptions on memory states in the original block-based memory model. The block-based memory model aligns well with such generalization as it becomes an instance of the nominal memory model. Moreover, these enhancements enable intuitive reasoning about compiler passes that perform transformations on partial memory and about compilation of open and concurrent modules.
- We develop Nominal CompCert, an extension to CompCert with its full compilation chain verified based on the nominal memory model (Sec. 3.3). Nominal CompCert is a general framework for verified compilation of C programs in the following sense: with its interface for supporting nominal names established, the compiler and its proofs work regardless what

instances of block ids (names) and valid blocks (supports) are provided through this interface, no matter how complicated they are.
- We develop two extensions of Nominal CompCert that illustrate its power in verified compilation of whole programs and open programs. The first one enables intuitive reasoning of compiler transformations on partial memory (Sec. 4), while the second one enables modular reasoning about open programs working with contextual memory (Sec. 5).
- We demonstrate that the above developments require limited changes to CompCert, making verification techniques based on the nominal memory model lightweight and easy to adopt.

The developments presented in this paper are based on CompCert version 3.8 and fully formalized in the Coq proof assistant version 8.12.0.

1.4 Structure of the Paper

In Sec. 2, we first introduce the necessary background of the block-based memory model and elaborate on its problems. We then present the nominal memory model in its full detail and discuss how to extend CompCert to Nominal CompCert in Sec. 3. In the subsequent two sections, we successively introduce the key applications of the nominal memory model and Nominal CompCert, including verification of transformations on partial memory (in Sec. 4) and verification of open and concurrent programs (in Sec. 5). We provide an evaluation of our proof efforts in Sec. 6. Finally, we give a comparison with existing work and conclude in Sec. 7.

2 BACKGROUND AND APPROACH

We first give a brief introduction to the block-based memory and verified compilation of C programs based on it. We then elaborate on the deficiency of this memory model that prevents it from supporting intuitive proofs for verified compilation of whole programs and open programs.

2.1 The Block-Based Memory Model

In the block-based memory model, the memory space is represented as a countably infinite set of blocks where each block is given a unique identifier called its *block id* (denoted by b). CompCert uses positive numbers to represent block ids. The entire memory space is divided into two parts by a special block id called `nextblock`. A block with id less than `nextblock` has been allocated (called a *valid block*). Otherwise, it has not been allocated yet (called an *invalid block*). `nextblock`—initially with the value 1—denotes the id of next block to be allocated and will be increased after each allocation. A valid memory block is a finite array of bytes with a lower and upper bound. A *memory state* (denoted by m) consists of a mapping from addresses to *in-memory values* in valid blocks and the value of `nextblock`. Its type `mem` is defined in Fig. 1a where `Z` is the type of integers, `memval` is the type of in-memory values, and $m.(\texttt{mem\_contents})\ b\ o = v$ iff b is a valid block in m and v is the in-memory value at the o-th memory cell (byte) of b.

A pointer or a memory address is a pair (b, o) where b is the memory block it points to and o is an index to a memory cell in block b (also called an offset). Pointer arithmetic is represented by adjustments to offsets. For example, $(b, o) + n$ is defined as $(b, o + n)$. This simple block-based abstraction already provides basic support for memory isolation, in the sense that given a pointer to block b_1 we can never reach b_2 by performing pointer arithmetic if $b_2 \neq b_1$.

The main operations over memory are depicted in Fig. 1b where `val` is the type of *regular values* defined as follows:

$$\texttt{Inductive val} := \texttt{Vundef} \mid \texttt{Vint}\ i_{32} \mid \texttt{Vlong}\ i_{64} \mid \texttt{Vsingle}\ f_{32} \mid \texttt{Vfloat}\ f_{64} \mid \texttt{Vptr}\ (b, o).$$

Here, `Vundef` represents undefined values, `Vptr` (b, o) represents a pointer (b, o), and the remaining forms denote 32- and 64-bit integer and floating point values. `chunk` is the type of *memory chunks*

```
Definition block : Type := positive.
Record mem: Type := {
  mem_contents: block → Z → memval;
  nextblock: block;
}.
```

(a) Blocks and Memory States

```
alloc : mem → Z → Z → mem × block
free  : mem → block → Z → Z → ⌊mem⌋
load  : mem → chunk → block → Z → ⌊val⌋
store : mem → chunk → block → Z → val → ⌊mem⌋
```

(b) Memory Operations

Fig. 1. Definitions for the Block-Based Memory Model

containing information necessary for performing conversion between in-memory values and regular values. Note that the option type $\lfloor\rfloor$ is used to describe the possible failure of some operations. Given a memory state m, a lower bound l and a higher bound h, `alloc` $m\ l\ h$ returns a new block whose id is m.(`nextblock`) and whose valid offsets are in the range of $[l, h)$. It also return a new memory state where `nextblock` is increased by 1 and the newly allocated memory cells are updated with undefined values. Note that `alloc` *always succeeds* because the memory space has infinitely countable blocks. The `free` operation frees memory cells in a certain range. Given m, k b and o, `load` $m\ k\ b\ o$ loads a value starting from the address (b, o) such that the size and type of value are determined by k. Conversely, `store` stores a value into a certain location in memory.

Note that we have omitted a discussion of permissions in the block-based memory as they are mostly orthogonal to this research.

2.2 Memory Injections

A main task of compiler is to transform abstract data structures (e.g., stacks) for describing the semantics of the high-level source language into concrete data structures for the low-level target language. These transformations may drastically change the structure of memory.

CompCert introduces *injection functions* to capture such changes. An injection j is a partial function of type $\texttt{block} \to \lfloor \texttt{block} \times \texttt{Z} \rfloor$, such that $j(b) = \lfloor (b', o) \rfloor$ iff a block b is inserted into the offset o in block b' by the transformation. If $j(b) = \emptyset$ (we use $\emptyset$ to represent the `None` constructor), then b is removed from the memory after the transformation.

Given an injection function j, we can relate the source and target values via a binary relation $\hookrightarrow^v_j$ called a *value injection*. $v_1 \hookrightarrow^v_j v_2$ trivially holds iff v_1 is `Vundef`, indicating that undefined values can be relaxed to more concrete values by compiler transformations. If $v_1 = \texttt{Vptr}\ (b, o)$, then $v_1 \hookrightarrow^v_j v_2$ holds iff $j(b) = \lfloor (b', o') \rfloor$ and $v_2 = \texttt{Vptr}\ (b', o + o')$, i.e., the original address is shifted by injection j accordingly. If v_1 is neither undefined nor a pointer, then $v_1 \hookrightarrow^v_j v_2$ holds iff $v_1 = v_2$. A similar injection relation $\hookrightarrow^i_j$ is defined for in-memory values of which we elide a discussion. The in-memory value injection is lifted to an injection relation $\hookrightarrow^m_j$ between memory states. Roughly speaking, $m_1 \hookrightarrow^m_j m_2$ holds if *1)* given any two valid addresses of m_1 and m_2 related by j, the in-memory values at these addresses are related by $\hookrightarrow^i_j$, and *2)* certain well-formedness conditions are satisfied, including that invalid blocks in m_1 must be mapped to $\emptyset$ and if $j(b) = \lfloor (b', o') \rfloor$ then b' must be valid in m_2. A *memory extension* is a specialized memory injection with an identity injection function (i.e., $j(b) = \lfloor (b, 0) \rfloor$) and m_1.(`nextblock`) = m_2.(`nextblock`). Memory extensions are used to describe transformations that do not change the structure of memory.

2.3 Correctness of Compilation in CompCert

CompCert takes C modules (parsed from `.c` files) as input, transforms them through a sequence of compiler passes (about 20 of them) to an architecture-independent language called Mach, from which it finally generates assembly programs on a pre-configured architecture, as depicted in Fig. 2.

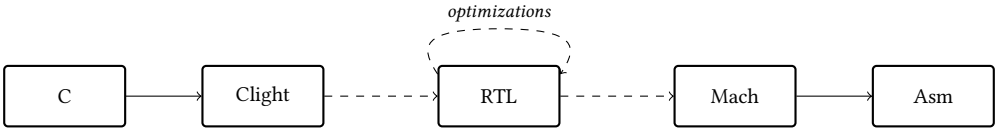


Fig. 2. The Verified Compilation Chain of CompCert

Much of the success of CompCert comes from the block-based memory model used uniformly across the whole compilation and a uniform interface for describing the semantics of all of its languages. In any language $\mathcal{L}$ of CompCert, a program P is represented as a list of global definitions associated with its own identifiers and a main function as its entry point:

```
Record program (F V: Type) := { prog_defs: list (ident * globdef F V); prog_main: ident }.
```

These identifiers are also represented by using positive numbers and have a global scope. To describe the semantics of P, a global environment $\mathcal{G}(P)$ of type genv is generated which provides two pieces of information: a mapping from definition identifiers to memory blocks holding such definitions and a mapping from the ids of these blocks to actual definitions. By using $\mathcal{G}(P)$, the semantics of P is described as a transition system labeled with traces of events, given by the following initialization and step relations where state is the type of abstract machine states:

```
initial_state: mem × state → Prop        step: (mem × state) → trace → (mem × state) → Prop.
```

The memory is initialized by allocating one block for every global definition. As the execution goes on, more blocks will be allocated (e.g., stack blocks created by function calls). We denote this semantics as $[[P]]_{\mathcal{L}}$ and write $[[P]]$ for $[[P]]_{\mathcal{L}}$ when the language is known from the context.

Given a compiler pass C, its correctness property is formulated as follows where $\geq$ is a forward simulation relation between the semantics of the source and target programs:

$$\forall\ P\ P', C(P) = \lfloor P' \rfloor \Longrightarrow [[P']] \geq [[P]].$$

That is, if P' is the result of compiling P by C, then any transitions performed by P can be simulated by corresponding ones by P'. To prove this theorem, one needs to establish an invariant between the source and target program states. A critical component of this invariant is a memory injection between the source and target memories that should hold throughout the entire execution.

The complete CompCert compiler $C_{\texttt{compcert}}$: C.program → ⌊Asm.program⌋ is a transitive composition of its passes $C_1, \ldots, C_n$. By composing the forward simulation proofs of C_i, we get

$$\forall P_s\ P_t, C_{\texttt{compcert}}(P_s) = \lfloor P_t \rfloor \Longrightarrow [[P_t]] \geq [[P_s]].$$

In the end, we would like know if all the behaviors of executable target code are exhibited at the source level. For this, one can flip forward simulations into backward simulations by exploiting the facts that the target semantics of CompCert is *determinate* and the source one is *receptive* [Sevcík et al. 2013]. The final correctness theorem of CompCert is stated as follows:

Theorem 2.1 (Correctness of CompCert).

$$\forall P_s\ P_t, C_{\texttt{compcert}}(P_s) = \lfloor P_t \rfloor \Longrightarrow [[P_t]] \preceq [[P_s]].$$

2.4 Problems

Having discussed the block-based memory model in detail, we can see that it indeed has the inflexibilities described in Sec. 1.1. Now, we elaborate on the problems they cause in verified compilation of whole programs and open programs.

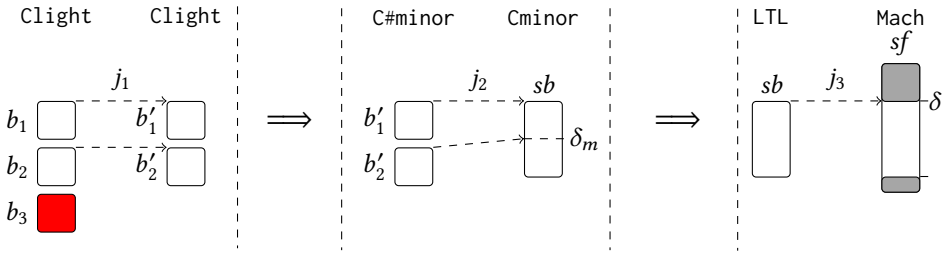


Fig. 3. Compiler Passes Transforming Stack Frames in CompCert

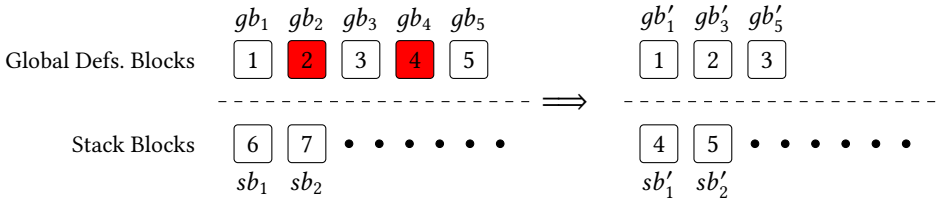


Fig. 4. Transformation of Memory by Unusedglob

2.4.1 Verifying transformation on partial memory. The vanilla CompCert only supports simulation proofs for whole programs. Even in this setting, one compiler pass only operates on certain part of memory and has local effects. More specifically, excluding advanced optimizations such as tail-call elimination and inlining, only four of CompCert's passes change the memory structure, of which three change the stack frame by frame, as depicted in Fig. 3. Those are:

- `SimplLocals`. In a source language called `Clight`, a stack frame consists of a collection of blocks, one for each local variable in the associating function. For example, the left column of Fig. 3 shows a stack frame containing three blocks (b_1, b_2 and b_3) for three local variables in some function. The `SimplLocals` pass transforms programs written in `Clight` by turning scalar local variables whose address are not taken into temporary variables, effectively removing their blocks from the stack frame. For example, in Fig. 3 the local variable associated with b_3 is turned into a temporary one. The remaining local variables are called *stack-allocated variables*.
- `Cminorgen`. This pass transforms programs in `C#Minor` (a variant of `Clight`) into programs in an intermediate language called `Cminor`. The stack-allocated variables are merged into a single block called the *stack-allocated data* (*sb* in Fig. 3). From this point on, a stack frame only contains a single block.
- `Stacking`. This pass lays out the concrete stack frame containing function parameters, return addresses, spilled registers, etc. (the gray areas in Fig. 3). The stack-allocated data is inserted into the middle of the concrete frame.

The remaining pass transforming memory is called `Unusedglob`. It drops *static* global definitions which are unused by the program. Fig. 4 shows an example where the blocks for the second and fourth global definitions (gb_2 and gb_4) are dropped by `Unusedglob` (where the numerals represent block ids). For this pass, the stack memory is completely untouched.

When proving the correctness of the above passes, one would expect that we could exploit the locality of memory transformations to construct intuitive proofs. For example, for `SimplLocals`, `Cminorgen` and `Stacking`, one would like to exploit the facts that *1)* blocks for global definitions are

unchanged, and *2)* the changes to individual stack frames cannot interfere with each other. However, it is impossible to formalize these facts directly because we can neither distinguish blocks for global definitions from blocks in stack frames, nor tell the differences between blocks in different stack frames: they are all indexed by positive numbers. To bypass this problem, CompCert relies on complicated invariants for classifying block ids into different ranges of positive numbers. Furthermore, it introduces general memory injections for relating source and target blocks, essentially lifting reasoning about local stack transformations to a global level. As we can see, the above problems are exactly caused by **Inflexibility 1** introduced in Sec. 1.1.

For `Unusedglob`, because it only drops certain blocks for global definitions, it is reasonable to assume its memory invariant as a *partial identity mapping*. However, in contrast to this intuition, the memory invariant for `Unusedglob` cannot be anything less than a general memory injection. To understand this, note that blocks for global definitions are allocated before any stack blocks. Because valid block ids must be consecutive integers starting from 1, dropping even a single global definition will result in shifting of all the block ids allocated after. This situation is illustrated by the example in Fig. 4 where the removal of gb_2 and gb_4 causes the ids of gb_3 and gb_5 to change from 3 to 2 and 5 to 3, respectively. Moreover, all the stack block ids are shifted by -2. In the end, a non-trivial memory injection is required to connect the source and target blocks. With this injection the correctness proof of `Unusedglob` involves complicated reasoning over stack memory even when the stack is not touched by `Unusedglob` at all. As we can see, the above problems are exactly caused by **Inflexibility 2** in Sec. 1.1.

2.4.2 Verifying transformations on open programs. Verification techniques for open programs are compositional only if the semantics of open programs are compatible with each other. However, the existence of a global `nextblock` makes this compatibility very difficult to establish, even when different open programs operate on completely separate memory regions.

We take the compilation and linking of Certified Concurrent Abstraction Layers (CCAL) as an example to illustrate the above problem [Gu et al. 2018]. A concurrent certified abstraction layer L consists of shared and private memory states, abstract states, and a set of possibly shared primitive operations (like external functions). The semantics of a language (e.g., C or assembly) built on top of L forms an abstract machine in which concurrent programs can be formally described. A CCAL object is a formally verified open program built on top of some layer L. A multi-threaded program is developed by abstracting individually threads into CCAL objects. These objects are then compiled by an extension of CompCert called Thread-Safe CompCertX into assembly code. Finally, CCAL objects need to be linked together to form a complete program.

For the above thread-linking to be possible, the semantics of CCAL objects must be compatible with each other. Achieving this compatibility is a non-trivial task. To understand this, observe that a new stack frame (a block) is allocated for every function call in CompCert's assembly language. To link threads together, it is necessary for each thread to have a compatible view of the stack memory, i.e., having the same sequences of stack frames and the same `nextblock`, meanwhile preventing one thread from accessing the private stack memory of the others.

To solve the above problem, authors of Thread-Safe CompCertX modify the assembly semantics so that when a thread yields to its context, a sequence of dummy stack blocks is allocated to increase `nextblock` in accordance with the actual allocation of stack blocks by the context. Moreover, to avoid any interference between memory operations on the stacks in different threads, the dummy blocks do not carry any read or write permission—they are "dead" memory cells from the perspective of the focused thread. With those devices, it is possible to "thread" the private stack frames of each thread into a single stack. As an example, the linking of stacks for two threads is depicted in Fig. 5. Here, the yield primitive from thread 1 to 2 in the function g allocated two dummy blocks (marked

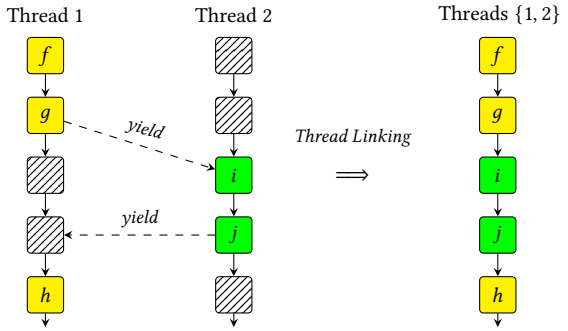


Fig. 5. Linking of Multiple Stacks into a Single Stack in CCAL

with north east lines) for the corresponding execution in thread 2. Access to these dummy blocks are invalid in thread 1.

The solution above has two problems. First, intuitively, each thread should have it own private stack: the context should not interfere with operations on this stack. Contrary to this assumption, the growth of dummy frames depends on how contextual threads change `nextblock`. This introduces unnecessary complication to verification of compilation. Second, in the linked program, stack frames for different threads are interleaved with each other. This not only makes the semantics of linked programs unrealistic, but also makes it extremely difficult to further verify their compilation to actual machine code where each thread should have its own contiguous and private stack. As we can see, the above problems are exactly caused by **Inflexibility 2 and 3** in Sec. 1.1.

2.5 Our Approach

As we have discussed in the introduction, we shall solve the above problems by getting rid of the inflexibilites of the block-based memory model through a generalization of it based on nominal techniques. We shall present this nominal memory model and the development of Nominal CompCert based on it in Sec. 3. After that, we demonstrate how the two kinds of problems discussed above can be solved by further extending Nominal CompCert in Sec. 4 and Sec. 5, respectively. As the following discussion will show, our approach provides systematic, clean and lightweight solutions to these problems.

3 NOMINAL MEMORY MODEL AND VERIFIED COMPILATION

3.1 Key Ideas

Nominal techniques [Gabbay and Pitts 2002; Pitts 2016] provide a mathematical foundation for managing named resources. In this setting, any countably infinite set $\mathbb{A}$ can represent a set of available names. Elements in these sets are called *atomic names* (or simply *names* in short). Dependency of objects upon names is captured by the notion of *supports*. Given a set of objects X and some $x \in X$, $A \subseteq \mathbb{A}$ supports x (or A is a support of x) iff, for any permutation π on $\mathbb{A}$ that is an identity mapping for names in A, we have $\pi(x) = x$. It effectively means that x is independent of any name outside of A. Only objects that can be supported by a finite set of names are of interest to us. A binary relation called *freshness* makes the independence relation concrete. A name $a \in \mathbb{A}$ is fresh w.r.t. x (written as $a\#x$) iff x is supported by some finite set $A \subseteq \mathbb{A}$ and $a \notin A$.

The above concepts can be used to characterize the block-based memory model. By taking memory states as named objects under investigation, we adopt the following analogies:

- Block ids represent names that memory states depend upon;

- Given a memory state m, the set of valid blocks in m represents a support of m;
- Given a memory state m, its `nextblock` is fresh w.r.t. m.

Note that the set of valid blocks is always finite. To some extent, the existing memory operations in CompCert already exploit the properties of atomic names and supports. For example, `alloc` always succeeds because there is always an infinite amount of ids outside the set of valid blocks.

However, the block-based memory also makes rigid assumptions about names and supports:

- Block ids are fixed to positive numbers;
- For any memory state m whose `nextblock` is n, its support must be the fixed set of consecutive numbers $\{1, \ldots, n-1\}$;
- For any memory state m whose `nextblock` is n, its fresh block must be assigned with the id n.

As we have seen in Sec. 2.4, these rigid assumptions cause serious problems in verified compilation. We shall remove those assumptions by generalizing the block-based memory model to work with any valid atomic names, supports and freshness relations.

3.2 The Nominal Memory Model

Given the above ideas, the block-based memory model is generalized to the nominal memory model in a straightforward manner. We first introduce an abstraction of block ids with the following module type:

```
Module Type BLOCK.
  Parameter block : Type.
  Parameter eq_block : ∀ x y : block, {x = y} + {x ≠ y}.
End BLOCK.
```

That is, block ids are names with decidable equality. We then introduce an abstraction of supports:

```
Module Type SUP.
  Parameter sup: Type.
  (** Operations *)
  Parameter sup_empty : sup.  (* Empty support *)
  Parameter fresh_block : sup → block. (* Generation of fresh blocks *)
  Parameter sup_add : block → sup → sup. (* Increment of supports *)
  Parameter sup_in : block → sup → Prop. (* Check validity of block ids *)
  (** Properties *)
  Parameter sup_dec : ∀ b s, {sup_in b s} + {¬sup_in b s}.
  Parameter empty_in: ∀ b, ¬sup_in b sup_empty.
  Parameter freshness : ∀ s, ¬sup_in (fresh_block s) s.
  Parameter sup_add_in : ∀ a s b, sup_in a (sup_add b s) ↔ a = b ∨ sup_in a s.
End SUP.
```

By this definition, a support type must be accompanied by four kinds of operations: checking the membership of blocks in supports (`sup_in`), creating an empty support (`sup_empty`), generating fresh blocks (`fresh_block`) and increasing supports with new blocks (`sup_add`). Furthermore, those operations satisfy some well-formedness properties. Note that after the above abstraction, the rigid assumptions for block-based memory model are completely removed.

We also note that the above generalization does not exactly match with the definitions in nominal techniques. For example, `BLOCK` does not enforce that block ids are from a countably infinite set. Instead, the `freshness` property guarantees that any block fresh w.r.t a support s must not be already in s. Together with the totality of `fresh_block`, it implies the existence of an infinite number of block ids. We also define supports to be any type that has the interface of `SUP`, instead of a finite set of block ids. This generalization allows us to instantiate `sup` with complex data structures for

formalizing memory space in fine-grained styles. We shall exploit this feature extensively in Sec. 4 and Sec. 5.

To make use of the above interfaces, we instantiate block ids and supports as follows:

```
Module Block <: BLOCK. ... End Block.        Module Sup <: SUP. ... End Sup.
```

Then the original `block` type is instantiated with `Block.block`. Now, the memory state carries a support instead of `nextblock`:

```
Record mem: Type := { mem_contents: block → Z → memval; support: Sup.sup;}.
```

The memory operations are adapted accordingly. For example, checking of valid blocks is done by using `sup_in` instead of comparing with `nextblock`:

```
Definition valid_block (m:mem) (b:block) := b < m.(nextblock). (* Old *)
Definition valid_block (m:mem) (b:block) := sup_in b m.(support). (* New *)
```

For another example, `alloc` now invokes `fresh_block` to allocate a new block instead of consulting `nextblock`. The properties of all memory operations can be easily reestablished because they are ignorant of particular instantiations of block ids and supports.

Finally, the block-based memory becomes a special case of the nominal memory model, as follows where `find_max_pos` finds the maximal positive number in a list:

```
Module Block <: BLOCK.
  Definition block := positive.          Definition eq_block := peq.
End Block.
Module Sup <: SUP.
  Definition sup := list block.          Definition sup_in (b: block) (s: sup) : Prop := b ∈ s.
  Definition sup_empty : sup := [].      Definition fresh_block (s: sup) := (find_max_pos s) + 1.
  Definition sup_add (b: block) (s :sup) := b::s.      ...
End Sup.
```

We shall see more complex memory models that exploit the full power of the above abstractions in Sec. 4 and Sec. 5.

3.3 Nominal CompCert

We apply the nominal memory model to the full compilation chain of CompCert to get Nominal CompCert. First, we need to update the semantics of every language of CompCert. This is automatically achieved by replacing the old memory operations with the new ones. We use $[\![P]\!]^{\mathcal{N}}$ to denote the new semantics of a program P. Second, we need to update the simulation proofs. Because the generalization of block ids and supports is mostly orthogonal to the simulation proofs in CompCert, the changes are minimal. The only slightly complicated changes are the results of losing the ability to identify valid blocks via comparison of positive numbers. For example, in the proof of the inlining optimization, the invariant `valid_block` m sp asserts that the stack block sp is valid. It also implies any block in m with id less than sp is also valid. After the generalization, we must make this fact explicit. This is achieved by breaking the above invariant into two: sp = `fresh_block` sps and $\{sp\} \cup sps \subseteq m.(\texttt{support})$. Here, sps denotes the blocks allocated before sp and the second new invariant asserts their validity.

By composing the updated simulation proofs, we get the final correctness theorem of Nominal CompCert, which is almost identical to Theorem. 2.1 except that the semantics of languages are based on the nominal memory model:

Theorem 3.1 (Correctness of Nominal CompCert).

$$\forall P_s\ P_t, C_{\texttt{compcert}}(P_s) = \lfloor P_t \rfloor \Longrightarrow [\![P_t]\!]^{\mathcal{N}} \preceq [\![P_s]\!]^{\mathcal{N}}.$$

We note that Nominal CompCert is a quite lightweight extension to CompCert. The changes made to it amount to about 1% of the code of CompCert 3.8, i.e., about 1.4K lines of Coq code. A majority of these changes are trivial substitutions of `nextblock` with `supports`, except for about 200 lines of code for handling the complication resulting from the inability to compare block ids mentioned before. We also introduce about 200 lines of new code for implementing the nominal memory model as described in Sec. 3.2. A more detailed evaluation of Nominal CompCert can be found in Sec. 6.1.

Finally, Nominal CompCert provides a skeleton for developing further extensions to CompCert that exploit the full power of nominal memory model. In these extensions, block ids and supports will be instantiated with more sophisticated structures. However, there is *zero immediate overhead* to introduce such instantiations because the entire proof of Nominal CompCert is ignorant of and holds for any instances of block ids and supports that meet their interfaces. Only after the user introduces mechanisms for exploiting the internal structures of such instances will any actual overhead be incurred. Even in those cases, the changes are built on well-defined interfaces and confined to local definitions and proofs. We shall witness these benefits in detail in Sec. 4 and Sec. 5.

4 VERIFICATION OF TRANSFORMATIONS ON PARTIAL MEMORY

In this section, we discuss the application of our nominal memory model to verified compilation of whole programs. We shall first introduce an enriched instantiation of this model where the whole memory space is divided into memory regions with well-defined structures and clear roles. Based on this enriched memory model, we get a complete extension of Nominal CompCert. With this extension, we develop an effective solution to the first problem in Sec. 2.4, i.e., how to construct intuitive correctness proofs for compiler passes that operate on partial memory.

4.1 Nominal Memory Model with Structured Space

By the discussion in Sec. 2.4, to model the locality of memory transformations, we first need to divide memory space into local regions with clear roles. Without loss of generality, we shall assume the whole memory consists of stack memory and memory for global definitions, including global variables (data memory) and global functions (code memory). We omit a treatment of heaps by assuming that they reside inside data memory and are managed by some allocator. We further model the stack as a tree such that each of its node corresponds to a stack frame that is either "active" (corresponding to an active call) or "dead" (corresponding to historical calls that have returned). We call it the *stack tree*. Its root corresponds to the stack frame of the main function. With these intuitions in mind, we formalize the new memory model.

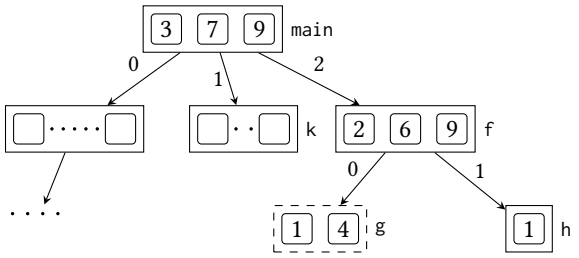
4.1.1 Formalization of Block IDs. We define the type of block ids as follows:

```
Definition fid : Type := ⌊ident⌋.        Definition path : Type := list nat.
Module Block <: BLOCK.
  Inductive block :=
    | Stack : fid → path → positive → block
    | Global : ident → block.
  …
End Block.
```

A block for the global definition named *id* is denoted as `Global` *id*. A block in a stack frame is denoted by `Stack` *fid p b*. It is identified by the path p from the root of stack tree to the frame together with a block id b local to the frame. Here, $p = [i_1, i_2, \ldots, i_n]$ is a list of natural numbers denoting a sequence of indices to child nodes with which the stack frame can be reached from the root, i.e., the frame with path p can be reached by first visiting the i_1-th child of the root frame,

```
extern void g(int*, int*);
void k() {...}
void h(int* w) {...}
void f(int* e, int* f, int* g) {
  int x=*e, y=*f, z=*g;
  ...
  g(&x,&y);
  h(&z);
}
int main() {
  int a,b,c;
  ...
  k();
  f(&a,&b,&c); ...
}
```

(a) The Program



Local block ids: a = 3, b = 7, c = 9, x = 2, y = 6, z = 9, ...

(b) The Stack Tree

Fig. 6. An Example of Stack Trees

then the i_2-th child of it, and so on until p is traversed. The id b is necessary for identifying a block in a frame containing multiple blocks. Such frames occur in higher-level intermediate languages as shown in Fig. 3. To determine whether the frame is allocated by internal function calls or external ones, we also attach the stack block with an optional function id *fid*. When $fid = \lfloor id \rfloor$, the block is allocated by the internal function *id*. Otherwise, it is allocated by some external function.

As an example, Fig. 6 displays a program and a stack tree generated by its execution that reaches the body of `h`. Here, the alphabetic names denote functions associated with frames, the numbers in blocks denote their ids local to a frame, and the numbers besides arrows denote the indices to child nodes. Moreover, frames with dashed boarders represent those allocated by external calls. With this figure we give some examples of formalized ids for stack blocks: the last block in `f` denotes its local variable z and has the id `Stack` $\lfloor$`f`$\rfloor$ [2] 9; the first child of `f` is allocated externally by `g` and its second block has the id `Stack` $\emptyset$ [2, 0] 4; finally, the only block in `h` has the id `Stack` $\lfloor$`h`$\rfloor$ [2, 1] 1.

4.1.2 Formalization of Stack Trees. We would like to keep information about the structured memory space in supports. For this, we need to give a formal definition of stack trees that is compatible with the interface of supports (e.g., `fresh_block` should always succeed when allocating a frame on the stack tree). Therefore, we define stack trees as a data structure that distinguishes between active and dead frames and that allows unbounded growth of active frames (for function calls) and transformation of active frames to dead frames (for function returns). This data structure is named `stree` and formally defined as follows where each node corresponds to a stack frame:

```
Inductive stree : Type :=
  | Node : fid → (list positive) → list stree → ⌊stree⌋ → stree.
```

An `stree` has the form `Node` f *bs dt at* where f denotes an optional name of the function that allocates the current frame (similar to the first argument of `Stack` above), *bs* denotes a list of blocks in the current frame, *dt* denotes a list of child trees containing dead frames, and finally, *at* denotes the only child tree that is still active. By definition of `stree`, we can get the list of active frames by following its last argument. The remaining frames in the tree are all dead. An empty tree (`empty_stree`) is defined as `Node` $\emptyset$ [] [] $\emptyset$.

As an example, consider the tree in Fig. 6 again. Its rightmost path contains the list active frames [`main`, `f`, `h`] and the remaining frames are all dead ones. It is formalized as `t` below where `tj` and `tk` denotes its first and second children:

```
tj := ...   tk := ...   tg := Node ∅ [1, 4] [] ∅   th := Node ⌊h⌋ [1] [] ∅
tf := Node ⌊f⌋ [2, 6, 9] [tg] ⌊th⌋   t := Node ⌊main⌋ [3, 7, 9] [tj, tk] ⌊tf⌋
```

We then formalize the operations over `stree` that are necessary for defining supports, including:

- `next_stree : stree → ident → stree * path`. It is for allocating a new frame. Given a stack tree t and the name *fid* of the allocating function, `next_stree` t *fid* pushes a new active frame onto t and returns the updated tree together with the path to the new frame.
- `next_block_stree : stree → fid * positive * path * stree`. It is for allocating a new block in the newest active frame f. Given a stack tree t, it returns the function name corresponding to f, a new block id that is *locally fresh* in f, the path to f, and the updated tree.
- `return_stree : stree → ⌊stree * path⌋`. It is for deallocating the newest active frame f, meaning that f is moved to the list of dead child trees of its parent. Given a stack tree t, it returns $\emptyset$ if there is no active frame. Otherwise, it returns the path to f and the updated tree.
- `stree_in : fid → path → positive → stree → Prop`. It is for checking the existence of blocks in a stack tree. Given a tree t, a name *fid*, a path p and a block b, `stree_in` *fid* p b t holds when the frame on the path p in t is attached with the name *fid* and contains the block b.

By definition, `next_stree` *always succeeds*, implying the existence of an infinite supply of stack block ids which matches the nominal property of memory space. On the other hand, `return_stree` may fail when there is no active frame on the stack. The full implementation of the above definitions can be found in Appendix.

4.1.3 Formalization of Supports. The type of supports and its operations are defined as follows:

```
Module Sup <: SUP.
  Record sup : Type := mksup { stack : stree; global : list ident; }.

  Definition sup_empty : sup :=
    mksup empty_stree [].

  Definition sup_in (b: block) (s:sup): Prop :=
    match b with
    | Stack fid p i ⇒
       stree_in fid p i s.(stack)
    | Global id ⇒ id ∈ s.(global)
    end.

  Definition fresh_block (s:sup): block :=
    match next_block_stree s.(stack) with
    | (f, i, p, _) ⇒ Stack f p i
    end.

  ....
End Sup.
```

```
Definition sup_incr_glob (i: ident) (s:sup) :=
 mksup (stack s) (i:: s.(global)).

Definition sup_incr_frame (s:sup) (id:ident): sup :=
 let (t', p) := next_stree s.(stack) id in
     mksup t' s.(global).

Definition sup_return_frame (s:sup) : ⌊sup⌋ :=
  match return_stree s.(stack) with
    | ⌊(t', p)⌋ ⇒ ⌊mksup t' s.(global)⌋
    | ∅ ⇒ ∅
  end.

Definition sup_incr_block (s:sup) :=
  let (_, t') := next_block_stree s.(stack) in
    mksup t' s.(global).
```

By definition, a support consists of a stack tree together with a list of names for valid global definitions. Because of the separation of memory into stack and global regions, we introduce four functions for managing supports besides `sup_add`: `sup_incr_glob: ident → sup → sup` for managing global definitions, and `sup_incr_frame: sup → id → sup`, `sup_return_frame: sup → ⌊sup⌋` and `sup_incr_block: sup → sup` for managing the stack. The last three operations are implemented by using the corresponding operations over `stree`. Specifically, `fresh_block` and `sup_incr_block` make use of `next_block_stree` to generate a fresh block in the newest active frame and to update

the stack tree, respectively, `sup_incr_frame` makes use of `next_stree` to allocate new frames, and `sup_return_frame` makes use of `return_stree` to convert active frames to dead ones.

4.1.4 Formalization of Memory Operations. With the new definition of supports, we update the memory operations accordingly. The key change is to introduce four new operations that operate on finer-grained memory regions. They are listed below, where the definition of memory states `mem` stays the same as described in Sec. 3.2.

- `alloc_glob`: `ident` → `mem` → `Z` → `Z` → `mem` ∗ `block`. `alloc_glob` *id m l h* allocates a new block with range $[l, h)$ in m for the global definition *id* by invoking `sup_incr_glob`. It returns the updated memory and the block id `Global` *id*.
- `alloc_frame`: `mem` → `ident` → `mem` ∗ `path`. `alloc_frame` *m id* allocates a new active frame in m by invoking `sup_incr_frame` and returns the path to this new frame.
- `return_frame`: `mem` → ⌊`mem`⌋. `return_frame` *m* deallocates the newest active frame by invoking `sup_return_frame` and returns the updated state.
- `alloc_block` : `mem` → `Z` → `Z` → `mem` ∗ `block`. `alloc_block` *m l h* allocates a fresh block with range $[l, h)$ on the newest active frame of m by invoking `sup_incr_block`. This block's id is obtained by calling `fresh_block` on *m*.(`support`). The updated memory together with this id are returned.

The properties about the new and updated memory operations are proved by following the conventional pattern; we elide a discussion of these changes.

4.2 Nominal CompCert with Structured Memory Space

Even with the previous changes, Theorem. 3.1 still holds as its proof is compatible with any particular instances of `BLOCK` and `SUP` (as we have pointed out by the end of Sec. 3). However, we would like to further exploit the enriched internal structure of the new memory model for simplifying proofs of partial memory transformations. For this, we first need to update the semantics of Nominal CompCert's languages. This is achieved as follows:

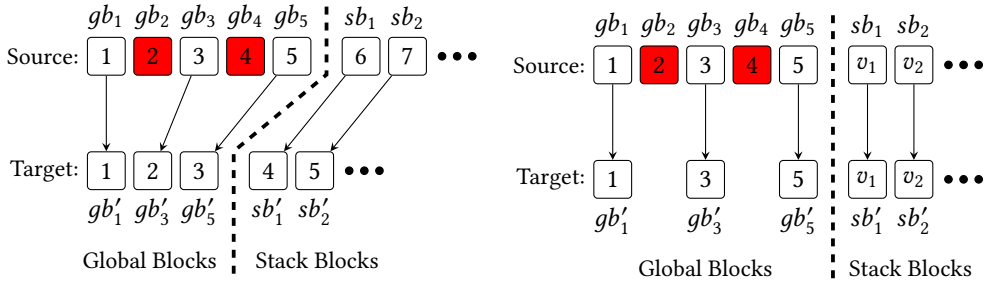
- Make the global environments directly use identifiers of global definitions instead of block ids, because they are the same in the new memory model. Also, make use of `alloc_glob` for initialization of global memory instead of `alloc`.
- At every function call and return, invoke `alloc_frame` and `return_frame`, respectively.
- Replace all the invocations of `alloc` for allocating stack blocks with `alloc_block`.

With the above changes, we reprove the full compilation of Nominal CompCert correct by establishing simulations between the newly introduced operations upon compilation. This requires a few predictable and local modifications to the correctness proof of each pass, indicating that our extension is still very compatible with the abstraction provided by Nominal CompCert. In the end, we got a correctness theorem similar to Theorem. 3.1.

4.3 Intuitive Proofs for Partial Memory Transformations

With the instantiated Nominal CompCert in place, we are ready to show how to construct intuitive correctness proofs for transformations on partial memory. We shall take `Unusedglob` as the main example to illustrate the key ideas. The proof construction for other passes follows a similar pattern, which we shall briefly discuss at the end of this section.

4.3.1 Structural Injection Functions. Previously, simulation proofs relying on general memory injections assume the *existence of some memory injection* that captures the transformation on memory. With our new memory model, it is now possible to explicitly *define the injection function* that precisely describes the transformation on memory. This definition is composed of three pieces of information:



(a) The Original Injection (b) The Structural Injection

Fig. 7. Comparison of Injections for Unusedglob

- the shape of memory transformation that is *statically* known and *independent* of programs being transformed;
- the shape of memory transformation that is *statically* known but *depends* on programs being transformed;
- the *dynamic* information depending on *particular execution* of programs, e.g., the support of memory states.

We call it a *structural injection function*.[1] Unlike the existential injection functions used previously, structural injections are not only computable, but also enable intuitive characterizations of memory transformations.

We take Unusedglob as an example. Recall that, given a module M, it removes statically defined global variables or functions that are never used in M. By intuition, its memory injection should map removed global blocks to $\emptyset$ and any other valid blocks to themselves. The structural injection function struct_meminj exactly captures this intuition. It is defined as follows, where *ge* denotes the global environment of the target program and check_block *s b* checks if the source block *b* should be mapped to a target block given the support *s* of the source program.

```
Variable ge: genv.

Definition struct_meminj (s:sup) (b:block) :=
  if check_block s b
  then ⌊(b, 0)⌋ else ∅.
```

```
Definition check_block (s:sup) (b:block): bool :=
  match b with
  | Global id ⇒ find_symbol ge id
  | Stack _ _ _ ⇒ sup_dec b s
  end.
```

We explain how this definition is composed of the three pieces of information mentioned above. First, we know the injection function is a partial identity function because Unusedglob only drops memory blocks. This information is statically known and independent of programs being transformed. Second, to determine whether a global block *id* should be mapped to the target memory, we check if *id* is in the target environment *ge*. This information is statically known, but dependent of programs resulting from the transformation. Third, to determine if a stack block *b* should be mapped to a target block, we check if it is in the source support *s*. If not, it is an invalid block and should be mapped to $\emptyset$. This information (source support) depends on the dynamic execution of programs.

We illustrate the intuitiveness of the above structural injection by comparing it with the original injection. A concrete example is shown in Fig. 7 where Fig. 7a depicts CompCert's injection for the example in Fig. 4 and Fig. 7b depicts the corresponding structural injection. As we can see, the original injection needs to describe shifting of positive block ids because of deletion of global

[1]It is different from and should not be confused with *structured injection* in Compositional CompCert [Stewart et al. 2015]

definitions, while the new injection is simply a partial identity mapping (we have omitted the concrete ids for stack blocks for simplicity).

4.3.2 Verification Based on Intuitive Proof Invariants. With structural memory injections, we are able to simplify the proofs of partial memory transformations so that they match with our intuition.

We continue with the `Unusedglob` example. The main complexity of the existing proof of `Unusedglob` lies in reasoning about shifted global definitions and stack frames. More specifically, the existing proof is based on an invariant of matching stack frames where each frame is associated with a source and target "bound" for describing the ranges of matching block ids in the source and target stack frames. The proof establishes an initial memory injection and shows it respects these bounds. It then shows that this invariant holds as the injection evolves alongside the allocation and deallocation of new stack blocks. However, since `Unusedglob` does not touch stack at all, we would have not expected such complicated reasoning about stack frames in the first place.

With structural memory injections, the excessive reasoning above about stack frames are simplified. The invariant about stack now assumes the existence of identical stack frames w.r.t. `struct_meminj`. An intuitive simulation proof for `Unusedglob` follows from this simplified invariant.

4.3.3 Verification of Other Transformations. The above ideas can be applied to other transformations in a similar fashion. We take `Cminorgen` as an example whose structural memory injection is defined as follows:

```
Variable ge: genv.

Definition struct_meminj (s:sup) (b:block) :=
  if sup_dec b s
  then unchecked_meminj b
  else ∅.

Definition unchecked_meminj (b:block) :=
  match b with
  | Stack ⌊id⌋ p i ⇒
     do o ← find_frame_offset ge id i ;
     OK ⌊(Stack ⌊id⌋ p 1, o)⌋
  | _ ⇒ ⌊(b, 0)⌋ end.
```

By definition, `Cminorgen` injects stack blocks in each frame into a single block of stack-allocated data for the frame. Therefore, we know that the block ids for global definitions are unchanged. Furthermore, the structure of the stack tree is unchanged—hence the paths to all stack frames remain the same after the transformation. Then, a source stack block b is injected into a block with the same path p, but with an index 1 because there is only one block in each stack frame after the transformation. Finally, the offset o by which the source stack block is inserted into the stack allocated data is obtained by querying the global environment ge of the source program. Like `Unusedglob`, the proof of `Cminorgen` also relies on matching bounds for describing the relations between stack frames. With this structural memory injection, we got rid of these bounds entirely and converted the reasoning about the stack as a whole into that about individual stack frames. This make the simulation proof of `Cminorgen` significantly easier to understand than before. Similar observations can be made for other transformations on partial memory.

5 VERIFIED COMPILATION OF PROGRAMS WITH CONTEXTUAL MEMORY

In this section, we discuss the application of our nominal memory model to verified compilation of programs that work with contextual memory. We achieve this by enriching the nominal memory model with even more finer-grained structures to represent contextual memory. These developments demonstrate an effective solution to the second problem in Sec. 2.4.

5.1 Key Ideas

The idea of *contextual compilation* is widely adopted in the research of verified compilation of open programs (e.g., [Gu et al. 2018; Song et al. 2020; Wang et al. 2019]). It is based on the assumption that,

when compiling multiple open modules or threads, only one of them is compiled while the others (the context) are fixed. The individually compiled modules or threads are then linked together at the target level to form the whole program. We have already seen such an example in Sec. 2.4.2.

To verify contextual compilation, we need to not only keep track of how memory is transformed by internal functions, but also do that for external functions. Since the context is fixed, the transformation on contextual memory should always be described as *an identity mapping* from source to target memory blocks, regardless of how complicated the memory transformation is for internal executions. However, this seemingly simple task is extremely difficult to complete in the original CompCert because it lacks the ability to distinguish memory blocks allocated by internal functions from those by external ones.

We argue that the fine-grained representation of block ids together with structural memory injections provide an elegant solution to contextual compilation of open programs and threads. Let us first take a look at contextual compilation of open programs worked on by *a single thread*. Recall that we previously assumed that the whole memory consists of memory blocks for global definitions and the stack. Because the former is fixed after initialization, contextual programs can only modify the stack by allocating and deallocating new frames. Then, the stack blocks allocated by external calls should always be mapped to themselves. This is indeed the case in our structural memory injections: the external stack blocks has the form $\texttt{Stack}\ \emptyset\ p\ b$ by the definition of block ids in Sec. 4.1.1 and $\texttt{Stack}\ \emptyset\ p\ b$ is always mapped to itself at offset 0 by the definitions of structural injections in Sec. 4.3.1 and Sec. 4.3.3. Based on this observation, we have proved that, given any external calls whose semantics simulate themselves under the identity mapping of contextual memory, they are compatible with the simulation proofs for internal executions. This indicates that the extension to Nominal CompCert in Sec. 4 already supports contextual compilation to a certain extent.

We also would like to support contextual compilation of multi-threaded programs. As we have discussed in Sec. 2.4.2, the previous solutions invented ad hoc mechanisms to cope with the global `nextblock` which prevent further compilation to a realistic machine model in which each thread has its own contiguous and private stack. We show that by extending the instantiation of supports with multiple stack trees, we can grow the stacks individual without interference with each other, thereby eliminating the problems with `nextblock`. Furthermore, by enriching the supports with multiple abstract stacks following the idea of Stack-Aware CompCert [Wang et al. 2019], we are able to compile the multi-threaded programs onto multi-stack machine models. These ideas form a complete solution to thread-safe contextual compilation, which we shall elaborate on in the rest of this section.

5.2 Stack-Aware Nominal CompCert

We first extends the instance of Nominal CompCert in Sec. 4 to support compilation into a contiguous and finite stack by incorporating the key ideas in Stack-Aware CompCert [Wang et al. 2019]. Stack-Aware CompCert explicitly manages the call stack by adding a data type called *abstract stack* to memory states. The abstract stack records the history of memory consumption incurred by stack allocation and maintains fine-grained stack permissions. By exploiting those information, Stack-Aware CompCert achieves contextual compilation of single-threaded C programs into an assembly language called `RealAsm` with a single contiguous stack.

In our structured nominal memory model, the abstract stack can be readily absorbed into the support. Moreover, since there already exists abundant work on managing stack permissions in verified compositional compilation (e.g., [Gu et al. 2018; Jiang et al. 2019; Song et al. 2020]), we decide to separate this issue from compilation to contiguous stacks. Therefore, our abstract stack only contains information about stack consumption by each function call. Like the original Stack-Aware

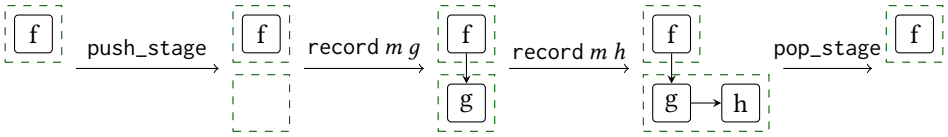


Fig. 8. Effects of the Operations on the Abstract Stack

CompCert, the abstract stack is organized into *stages* of *abstract frames* where a stage corresponds to a continuous sequence of tail-calls. It is defined as follows:

Definition 5.1 (Abstract Stack). Abstract frames are records of the type `frame := {fsize : Z; fsize ≥ 0}` where `fsize` contain the sizes of concrete stack frames. A stage is a list of abstract frames `stage := list frame` allocated by a sequence of tail-calls where its head is the frame for the active tail-call and the remaining frames have been deallocated by previous tail-calls in the sequence. Finally, an abstract stack is a list of stages `stackadt := list stage`.

Given an abstract stack a, its size is the summation of sizes of all the frames in a. We introduce a constant `MAX_STACK` for denoting a maximum size of abstract stacks which is necessary for compilation to a finite stack. We now extend the support type defined in Sec. 4.1.3 with an abstract stack as follows:

```
Record sup := mksup { global: list ident;  stack: stree;  astack: stackadt; }.
```

The operations that modify the structure of the abstract stack are the following: `push_stage : mem → mem` pushes a new stage onto the abstract stack. It is invoked when a regular call happens. `record : mem → frame → ⌊mem⌋` pushes a new frame into the topmost stage of the active stack. It succeeds only if the stack size after pushing this frame does not exceeds the `MAX_STACK` limit. It is invoked either when a regular call happens, or a tailcall happens. Note that only the topmost frame in a stage is "alive". When a tailcall records (pushes) a new frame in the topmost stage, the previous frame becomes "dead", corresponding to the parent frame deallocated by the tail call. `pop_stage` is invoked when a regular call followed by a sequence of tailcall returns. It simply pops the topmost stage from the stack. As an example, Fig. 8 illustrates the effects of these operations applied to the active stack where a regular call to the function g is followed by a tailcall to h which finally returns.

With the enriched memory model, we update the semantics of every language of Nominal CompCert by inserting `push_stage`, `record` and `pop_stage` operations accordingly. The semantics of a program P is also parameterized by an oracle `stackspace` that provides the sizes of concrete stack frames for each function; it is denoted by $[[P, \texttt{stackspace}]]$. Then, we replay the proofs of Stack-Aware CompCert by establishing preservation of stack consumption for every pass, i.e., the size consumption of the target abstract stack is always less than or equal than the source abstract stack. The only complication appears when verifying advanced optimizations such as tail-call recognition and inlining. Because inlining is performed after tail-call recognition, it may lift certain tail-calls back to regular calls, causing complicated changes in stack consumption. We handle this problem by assuming every tail-call consumes stack space like a regular call up to inlining, and inserting an identity transformation (called `RTLmach`) after inlining for shifting from this assumption to an accurate account of stack consumption by tail-calls. This greatly simplifies the proofs of preservation of stack consumption. However, it also means the abstract stack may be out-of-sync with the actual stack tree before inlining. This is exactly why we choose to kept the abstract stack and the stack tree separate in support.

By composing the proofs for its individual passes, we derive the correctness of Stack-Aware Nominal CompCert stated as follows:

Theorem 5.2 (Correctness of Stack-Aware Nominal CompCert). *Let* $C_{\texttt{sa-nominal-compcert}}$ *be Stack-Aware Nominal CompCert that compiles C programs into* `RealAsm` *assembly programs with a finite stack. We have*

$$\forall P_s\ P_t, C_{\texttt{sa-nominal-compcert}}(P_s) = \lfloor P_t, \texttt{stackspace} \rfloor \Longrightarrow [\![P_t, \texttt{stackspace}]\!] \preceq [\![P_s, \texttt{stackspace}]\!].$$

where `stackspace` *is an instance of the oracle (a mapping from functions to sizes of stack frames) generated by the compiler at the Mach level where the concrete layouts of frames are fixed.*

5.3 Multi-Stack CompCert

Multi-Stack CompCert is a straightforward extension of Stack-Aware Nominal CompCert by enriching the support with multiple copies of stack trees and abstract stacks. Each thread is assigned a unique stack. It can only directly work on this stack and is ignorant of the existence of stacks for other threads. This abstraction is achieved by defining the support type as follows:

```
Record sup := mksup { global: list ident;  stack: list stree;  astack: list stackadt; sid: nat}.
```

Compared to the `sup` type of Stack-Aware Nominal CompCert, it now has a list of stack trees and a list of abstract stacks. The field `sid` denotes index to the stack that a thread operates on. The following operations are provided for accessing the stack tree and the abstract stack:

```
Definition get_stacktree (m:mem) := let s := m.(support) in s.(stack)[s.(sid)].
Definition get_abststack (m:mem) := let s := m.(support) in s.(astack)[s.(sid)].
```

With the above abstraction, the entire development of Stack-Aware Nominal CompCert is lifted to work with multiple-stacks, resulting in Multi-Stack CompCert. This is possible because we assume `sid` is bound to a thread which cannot by itself switches to a different stack. Instead, context switching must be completed through external mechanisms (e.g., Certified Concurrent Abstraction Layers [Gu et al. 2018]). With this assumption, a thread simply treats all the stacks except for its own one as part of the contextual memory. Its semantics is completely unchanged except for the uses of `get_stacktree` and `get_abststack` for accessing its own stack. In the end, we get the correctness theorem of Multi-Stack CompCert which is exactly like Theorem. 5.2 except for the updated semantics.

Note that, Multi-Stack CompCert can be very easily proved correct because the modification to contextual stacks is completely irrelevant to the operations on the focus stack. This is in stark contrast to the situation in the original CompCert where such modification affects `nextblock` at a global scope. These developments illustrate the simplicity and power of nominal memory model in formalizing contextual memory, which we shall further exploit.

5.4 Thread-Safe Contextual Compilation

To demonstrate the effectiveness of Multi-Stack CompCert in verified compilation of multi-threaded programs, we apply it to solve the problem of compiling and linking Certified Abstraction Layers (CCAL) as described in Sec. 2.4.2.

5.4.1 Challenges in Compiling and Linking CCAL Objects. To understand the underlying challenges, we first give a more detailed introduction to CCAL. CCAL is a generalization of Certified Abstraction Layers [Gu et al. 2015] for building concurrent programs in a layered style. A certified concurrent layer L provides shared and private memory states and primitive operations for manipulating them. Concurrent objects are built on top of a lower layer L and provide implementations of a higher layer L'. From the point view of the user of L, there is no need to worry about how execution of

objects running on L interleaves in a concurrent setting, because the shared primitives are already abstracted into (proved semantically equivalent to) atomic operations. A predicate of the form $L[A] \vdash_R M : L'[A]$ formally describes the implementation M of the layer L' running on top of the layer L. Given the domain of threads $\mathbb{D}$, it holds iff concurrent execution of M with a focused subset of threads $A \subseteq \mathbb{D}$ on L backward simulates the layer L' where R is the invariant of the simulation. Note that $A \subseteq \mathbb{D}$ indicates that not all threads are known to M. Therefore, M is an open multi-threaded program whose execution trace may be interleaved with that of unknown threads in the context. With the devices above, it is possible to *horizontally*, *vertically* and *parallelly* compose concurrent layers, as described by the following rules:

$$\frac{L[A] \vdash_R M : L_1[A] \quad L[A] \vdash_R N : L_2[A]}{L[A] \vdash_R M \oplus N : L_1[A] \oplus L_2[A]}\ \texttt{HComp}$$

$$\frac{L_1[A] \vdash_R M : L_2[A] \quad L_2[A] \vdash_S N : L_3[A]}{L_1[A] \vdash_{R \circ S} M \oplus N : L_3[A]}\ \texttt{VComp}$$

$$\frac{L_1[A] \vdash_R M : L_2[A] \quad L_1[B] \vdash_R M : L_2[B] \quad \texttt{compat}(L_2[A], L_2[B])}{L_1[A \cup B] \vdash_R M : L_2[A \cup B]}\ \texttt{PComp}$$

The common pattern of developing verified concurrent programs using CCAL is to start with source modules focusing on a single thread, i.e., layer implementations of the form $L[i] \vdash_R M : L'[i]$ where i is the id of the focused thread, then compile these modules as sequential programs, and finally compose the generated programs at the assembly level.

For the above approach to actually work, it is essential that the compilation is *thread safe*, i.e., when compiling modules focusing on a single thread, the compiler C indeed preserves CCAL:

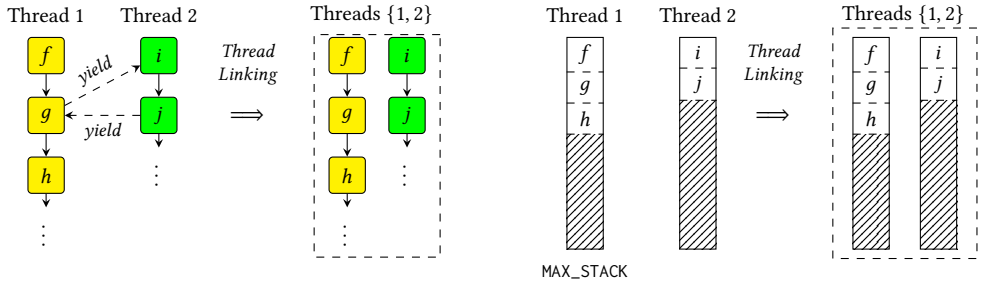
$$\forall\ i\ L\ L'\ M\ R, L[i] \vdash_R M : L'[i] \Longrightarrow L[i] \vdash_R C(M) : L'[i].$$

As we have discussed in Sec. 2.4.2, thread-safeness as currently implemented has serious problems in that it relies on introduction of dummy blocks and does not support compilation to realistic machine models.

5.4.2 Compilation and Linking of CCAL Objects onto Multi-Stack Machine Models. We make immediate use of Multi-Stack CompCert to overcome the above challenges. Given a CCAL object $L[i] \vdash_R M : L'[i]$ written in C, we assign a unique stack i to M in Multi-Stack CompCert. Then, M represents a sequential program that is ignorant of its context in a multi-stack memory model, exactly matching the requirement of source programs for Multi-Stack CompCert. By individually compiling CCAL objects with Multi-Stack CompCert, we can reduce them to `RealAsm` programs that look like sequential assembly codes and that are ignorant of the existence of other threads. At the target level, the stacks are finite and contiguous and the thread-local data are allocated by adjusting the stack pointers to private stacks.

To link the generated `RealAsm` programs, we observe that because the heap is managed by primitives in abstraction layers in CCAL, no new memory blocks will ever be allocated after the initial allocation of global variables and the finite stacks. Therefore, there is no need to allocate dummy blocks or to synchronize any memory structure across threads. Furthermore, we observe that the finite stacks for threads are separated from each other from the beginning and the operations on them never interfere with each other. Therefore, stack merging becomes trivial. For example, the problematic situation described in Fig. 5 becomes the natural merging of multiple private stacks as described in Fig. 9.

With the above changes, we are able to combine Multi-Stack CompCert with CCAL to—for the first time—realize the development of verified concurrent objects operating on a realistic machine model with multiple contiguous and finite stacks.



(a) Linking at CompCert's Asm Level (a) Linking at RealAsm Level

Fig. 9. Realistic Linking of CCAL Objects

Table 1. Changes in Nominal CompCert relative to CompCert 3.8

Files	CompCert 3.8	Nominal CompCert	Changes(*/+)	% of Changes
Values.v	2197	2205	9	0.4%
Memory.v	3838	4033	373	9.71%
Memtype.v	866	894	57	6.58%
Globalenvs.v	1640	1665	135	8.23%
SimplLocalsproof.v	1989	1992	143	7.19%
Cminorgenproof.v	1903	1930	203	10.67%
Inliningproof.v	1161	1178	186	16.02%
ValueAnalysis.v	1805	1847	163	9.03%
Unusedglobproof.v	1259	1272	67	5.32%
Total	136965	137341	1401	1.02%

6 EVALUATION

In this section, we discuss the proof efforts for our developments. We first present an evaluation of the changes introduced into Nominal CompCert. It illustrates the conciseness of the extension for obtaining Nominal CompCert and its compatibility with CompCert. We then present an evaluation of the proof efforts for further extending Nominal CompCert as described in Sec. 4 and Sec. 5. It demonstrates that, thanks to the generality of the nominal interfaces, we can exploit sophisticated instantiations of the nominal memory model to solve complicated problems in verified compilation of whole programs and open programs with minimal efforts.

6.1 Proof Efforts for Nominal CompCert

It took us 1 person month to develop Nominal CompCert, with much of the effort devoted to understanding the details of the proofs for CompCert's compiler passes, especially for those changing the memory structure. Only a small number of files have been modified and their changes are moderate. We give the statistics of our Coq implementation in Table. 1. This table lists all the files with more than 5% changes. The file Values.v contains the definition of block ids and is the only exception. Columns 2 and 3 show the lines of code for each file (counted by using `coqwc`) in CompCert 3.8 and Nominal CompCert, respectively. Columns 4 shows the lines of code that were *modified or added*, while Column 5 shows them in percentage relative to CompCert 3.8. It is also worth noting that a majority of the total changes (of the 1.4k lines of code) are trivial substitutions of `nextblock` with `support` that do not increase the LOC. The files with over 5% changes either

Table 2. Lines of Code for Stack-Aware Nominal CompCert Relative to Nominal CompCert

Files	Spec		Proof	
	NCC	SA-NCC	NCC	SA-NCC
Memory.v	1692	2329	2341	3184
Globalenvs.v	785	815	880	933
Events.v	652	699	709	780
Separations.v	342	377	430	510
RTLmach.v	-	131	-	19
RTLmachproof.v	-	83	-	193
SimplLocalsproof.v	681	696	1311	1452
Cminorgenproof.v	759	770	1171	1329
Tailcallproof.v	173	258	273	423
Inliningproof.v	441	632	737	1093
ValueAnalysis.v	694	730	1153	1215
Stackingproof.v	716	735	1107	1207
Total	71131	72805	66211	69175

contains the implementation of nominal memory model as described in Sec. 3.2 or contains changes to stack pointer and valid blocks as described in Sec. 3.3.

The minimal changes to CompCert for developing Nominal CompCert is the result of successfully reusing most of the framework and proofs already presented in CompCert. In particular, we found that *1)* in essence CompCert's verification does not depend on the specific representation of block ids as positive numbers, and *2)* treating block ids as nominal names is a natural generalization.

6.2 Proof Efforts for Extending Nominal CompCert

It took us about 2 person-months to implement Nominal CompCert with Structured Memory Space as presented in Sec. 4. The main addition is the implementation of structured memory model together with the properties of the newly added memory operations, for which we added about 800 lines of Coq code. We reuse almost all the proofs of Nominal CompCert. As a result, the LOC of compiler passes does not change much compared to Table. 1. On top of this, the improved verification of `Unusedglob` and `Cminorgen` presented in Sec. 4.3 adds about 200 lines of Coq code, respectively. This addition is also minimal because the changes to the structural injection functions are mostly compatible with CompCert's proofs: the memory in essence is still a mapping from block ids to values even when the block ids have sophisticated structures.

It took us an additional 2 person months to implement Stack-Aware Nominal CompCert in Sec. 5.2, for which the main effort is to prove the preservation of stack consumption for every compilation pass. Table. 2 compares the LOC of Stack-Aware Nominal CompCert relative to Nominal CompCert with representative files. As we can see, although we have changed the memory model and added a few new compiler passes, the changes are still moderate. The main reason is that *1)* we have dropped the stack permissions and *2)* we have implemented the abstract stack as part of the support, with which the relation between source and target stack consumptions can be easily absorbed into the memory injection and extension relation, and *3)* we have exploit a technique (introduced in Sec. 5.2) for approximating stack consumptions before inlining and introduced an identity transformation after inlining (`RTLmach.v` and `RTLmachproof.v` in Table. 2) for converting this over approximation into accurate stack consumptions. In total, the change in LOC is under 5k, while the implementation of Stack-Aware CompCert added 21k LOC to CompCert 3.0.1 [Wang et al. 2019]. Moreover, Multi-Stack

CompCert is easily obtained from Stack-Aware Nominal CompCert by changing several lines of code for switching to a multi-stack support (as shown in Sec. 5.3).

The above discussion shows that our extensions are indeed systematic and lightweight instantiations of Nominal CompCert.

7 RELATED WORK AND CONCLUSION

Nominal techniques [Gabbay and Pitts 2002; Pitts 2016] have been widely used to define the semantics of formal calculi with binders (e.g., λ-calculus, π-calculus) using inductive definitions of nominal sets [Pitts 2013]. They have also been used in the game-semantics community [Abramsky et al. 2004; Laird 2004; Murawski and Tzevelekos 2016] to define the nominal games which led to full abstraction results for languages with dynamic generative behaviors, such as ν-calculus [Abramsky et al. 2004], higher-order concurrency [Laird 2006], ML references [Murawski and Tzevelekos 2011], and Interface Middleweight Java [Murawski and Tzevelekos 2014, 2021]. Urban and Berghofer [Urban and Berghofer 2006; Urban and Tasson 2005] have implemented a "nominal datatype" package (in the Isabelle/HOL proof assistant) which has been used to mechanize formal proofs and operational semantics using nominal techniques. However, none of these have attempted to address the memory semantics and verified compilation of low-level C-like languages.

CompCert uses a unified memory model [Leroy et al. 2012; Leroy and Blazy 2008] for all of its compiler intermediate and target languages. The memory model treats global variables, and heap and stack objects as separate memory blocks to enforce isolation. It supports bound-checking (to give semantics to "undefined behaviors") and uses the `Vundef` value to denote results from ill-defined loads. CompCert memory block identifiers are kept relatively abstract and can be renamed, deleted, or injected as sub-blocks of bigger blocks while preserving the observable behaviors of programs. While the abstract block identifiers in CompCert seem to be suitable for a nominal treatment, for a long time, it was not clear how it should be done and what benefits it would bring. CompCert's memory-injection-based simulation proofs are also quite challenging so it is unclear whether nominal techniques can actually help simplify the existing proofs.

What we have shown in this paper is that the nominal techniques can indeed be used to both generalize and simplify the CompCert memory model in a really clean way. Furthermore, the nominal extension is backward compatible in that regardless what representations we use for the block identifiers ("names"), the rest of the compiler (including all the memory-injection-heavy compilation phases and their proofs) will remain valid. The separation of "global-variables-" and "stack-" supports allows us to simplify the memory-injection-related proofs, but the overall CompCert memory remains as a mapping from block identifiers to values as before (so the existing proofs would still work). This is in contrast to previous attempts [Ramananandro et al. 2015; Wang et al. 2019] in which they either had to introduce a "stack tag" for each block [Ramananandro et al. 2015] or add a separate stack component [Wang et al. 2019]; both required a major overhaul over the memory-injection-related proofs in CompCert.

There exists abundant work on extending CompCert to support various memory structures for verified compilation of open and concurrent programs. We shall make comparisons with them from the following two perspectives:

Stack-Awareness. Stack-Awareness means the support of a memory model with an explicit notion of stack memory, in particular, how close the memory model of the final target language is to the actual memory model used by concurrent machine or assembly code.

There has been previous work on translating the unbounded memory of CompCert into some kind of finite memory. CompCertS [Besson et al. 2017] supports low-level manipulation of pointer values in a finite memory space; CompCert-TSO [Sevcík et al. 2013] builds in a notion of finite

memory into all levels of CompCert for certifying concurrency in compilation; A lower-level semantics for CompCert assembly that models pointers as 32-bit values for verifying the peephole optimization is defined by Mullen et al. [2016].

None of the above work tries to model an explicit stack in memory. Quantitative CompCert (QCC) does present a more stack-aware view of CompCert. It relies strongly on reasoning about event traces extended with call/return events, and then reasoning about stack consumption in terms of the *same* structure of such call and return events on traces of the source and target programs. The Cerco C compiler [Amadio et al. 2014] uses a similar approach to QCC for reasoning about stack consumption and uses a different backend from that of CompCert. Because the exact same call/return events are needed throughout the compilation, they cannot support inlining and tailcall optimizations that are essential to the performance of the generated code.

Stack-Aware CompCert [Wang et al. 2019] is the first extension to CompCert that supports merging of stack frames into a finite and contiguous stack, all the optimization passes, and elimination of pseudo instructions for stack manipulation that do not exist in reality. The target code it generates are very close to machine code. CASCompCert [Jiang et al. 2019] does reason about what they call the footprints of programs, and in particular reason about memory ownership. In its memory model, an infinite set of memory locations is used for allocating stack frames, so that it seems challenging to use their framework to merge stack blocks into a finite and contiguous stack.

To support Stack-Awareness, all the above work requires intrusive and sometimes ad hoc changes to CompCert's memory model. On the other hand, Nominal CompCert provides a well-defined and flexible interface for supporting Stack-Awareness through instantiations of block ids and supports. As we have seen before, a complete call stack can be seamlessly integrated into Nominal CompCert (with Structured Memory Space) and compilation to multiple and contiguous machine stacks is supported without intrusive changes (Multi-Stack CompCert).

Contextual and General Compositional Compilation. Contextual compilation is a special case of general compositional compilation. Earlier work on it deals with compilation of layered programs where mutual recursion was not supported [Gu et al. 2015, 2018; Wang et al. 2019]. The most recent advancement in this regard is CompCertM [Song et al. 2020] that supports verified compilation of mixed C and assembly modules. We have shown that, with the nominal memory model and Nominal CompCert, contextual memory can be separated from internal memory and be reasoned about via a well-defined interface. This is the case both for verified compilation of open programs and for that of concurrent programs. This solves an important and basic problem in contextual compilation which was hard to address before.

There is also abundant work on general compositional verification based on CompCert, such as Compositional CompCert [Stewart et al. 2015], CASCompCert [Jiang et al. 2019] and CompCertO [Koenig and Shao 2021]. We believe our nominal memory model and Nominal CompCert provide complementary mechanisms for improving verification results in these projects. For instance, the interaction semantics in Compositional CompCert, CompCertM and CompCertO pass the whole memory state alongside all component interactions. This means the relation between source- and target-level memory states, usually hidden within simulation relations, must now be revealed as part of the interface of simulation proofs. In this context, the nominal approach we have presented can help controlling complexity by eliminating the need for Kripke worlds. First, structural injections eliminate the need for tracking an evolving injection function. Moreover, the techniques we have introduced could help isolating private memory as internal state, for example through the use of a partial commutative monoid structure. Finally, they could provide a seamless way to incorporate stack awareness to compositional compilers of this kind. We shall explore the possibility to implement such enhancements in future work.

REFERENCES

Samson Abramsky, Dan R. Ghica, Andrzej S. Murawski, C.-H. Luke Ong, and Ian David Bede Stark. 2004. Nominal Games and Full Abstraction for the Nu-Calculus. In *19th IEEE Symposium on Logic in Computer Science (LICS 2004), 14-17 July 2004, Turku, Finland, Proceedings*. IEEE Computer Society, 150–159.

Roberto M. Amadio, Nicolas Ayache, Francois Bobot, Jaap P. Boender, Brian Campbell, Ilias Garnier, Antoine Madet, James McKinna, Dominic P. Mulligan, Mauro Piccolo, Randy Pollack, Yann Régis-Gianas, Claudio Sacerdoti Coen, Ian Stark, and Paolo Tranquilli. 2014. Certified Complexity (CerCo). In *Foundational and Practical Aspects of Resource Analysis*, Ugo Dal Lago and Ricardo Peña (Eds.). Springer International Publishing, Cham, 1–18. https://doi.org/10.1007/978-3-319-12466-7_1

Andrew Appel. 2011. Verified Software Toolchain. In *Proc. 20th European Symposium on Programming (ESOP'11)*, Gilles Barthe (Ed.). LNCS, Vol. 6602. Springer, Saarbrucken, Germany, 1–17. https://doi.org/10.1007/978-3-642-19718-5_1

Frédéric Besson, Sandrine Blazy, and Pierre Wilke. 2017. CompCertS: A Memory-Aware Verified C Compiler Using Pointer as Integer Semantics. In *Interactive Theorem Proving (ITP'17)*, Mauricio Ayala-Rincón and César A. Muñoz (Eds.). Springer International Publishing, Cham, 81–97. https://doi.org/10.1007/978-3-319-66107-0_6

Murdoch Gabbay and Andrew M. Pitts. 2002. A New Approach to Abstract Syntax with Variable Binding. *Formal Aspects Comput.* 13, 3-5 (2002), 341–363. https://doi.org/10.1007/s001650200016

Ronghui Gu, Jeremie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan(Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *Proc. 42nd ACM Symposium on Principles of Programming Languages (POPL'15)*. ACM, New York, 595–608. https://doi.org/10.1145/2775051.2676975

Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jeremie Koenig, Vilhelm Sjober, Hao Chen, David Costanzo, and Tahnia Ramananandro. 2018. Certified Concurrent Abstraction Layers. In *Proc. 2018 ACM Conference on Programming Language Design and Implementation (PLDI'18)*. ACM, New York, 646–661. https://doi.org/10.1145/3192366.3192381

Hanru Jiang, Hongjin Liang, Siyang Xiao, Junpeng Zha, and Xinyu Feng. 2019. Towards Certified Separate Compilation for Concurrent Programs. In *Proc. 40th ACM Conference on Programming Language Design and Implementation (PLDI'19)*. ACM, New York, NY, USA, 111–125. https://doi.org/10.1145/3314221.3314595

Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. 2016. Lightweight Verification of Separate Compilation. In *Proc. 43rd ACM Symposium on Principles of Programming Languages (POPL'16)*. ACM, New York, 178–190. https://doi.org/10.1145/2837614.2837642

Jérémie Koenig and Zhong Shao. 2021. CompCertO: compiling certified open C components. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 1095–1109.

James Laird. 2004. A Game Semantics of Local Names and Good Variables. In *Foundations of Software Science and Computation Structures, 7th International Conference, FOSSACS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 2987)*, Igor Walukiewicz (Ed.). Springer, 289–303.

James Laird. 2006. Game Semantics for Higher-Order Concurrency. In *FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science, 26th International Conference, Kolkata, India, December 13-15, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4337)*, S. Arun-Kumar and Naveen Garg (Eds.). Springer, 417–428.

Xavier Leroy. 2005–2021. The CompCert Verified Compiler. https://compcert.org/.

Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. 2012. *The CompCert Memory Model, Version 2*. Research Report RR-7987. INRIA. 26 pages. https://hal.inria.fr/hal-00703441

Xavier Leroy and Sandrine Blazy. 2008. Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformation. *Journal of Automated Reasoning* 41, 1 (2008), 1–31. https://doi.org/10.1007/s10817-008-9099-0

Eric Mullen, Daryl Zuniga, Zachary Tatlock, and Dan Grossman. 2016. Verified Peephole Optimizations for CompCert. In *Proc. 37th ACM Conference on Programming Language Design and Implementation (PLDI'16)*. ACM, New York, NY, USA, 448–461. https://doi.org/10.1145/2980983.2908109

Andrzej S. Murawski and Nikos Tzevelekos. 2011. Game Semantics for Good General References. In *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science, LICS 2011, June 21-24, 2011, Toronto, Ontario, Canada*. IEEE Computer Society, 75–84.

Andrzej S. Murawski and Nikos Tzevelekos. 2014. Game semantics for Interface middleweight Java. In *Proc. 41st ACM Symposium on Principles of Programming Languages (POPL'14)*. 517–528.

Andrzej S. Murawski and Nikos Tzevelekos. 2016. Nominal Game Semantics. *Found. Trends Program. Lang.* 2, 4 (2016), 191–269.

Andrzej S. Murawski and Nikos Tzevelekos. 2021. Game Semantics for Interface Middleweight Java. *J. ACM* 68, 1 (2021), 4:1–4:51.

Andrew M. Pitts. 2013. *Nominal Sets: Names and Symmetry in Computer Science*. Cambridge University Press.

Andrew M. Pitts. 2016. Nominal techniques. *ACM SIGLOG News* 3, 1 (2016), 57–72. https://dl.acm.org/citation.cfm?id=2893594

Tahina Ramananandro, Zhong Shao, Shu-Chun Weng, Jérémie Koenig, and Yuchen Fu. 2015. A Compositional Semantics for Verified Separate Compilation and Linking. In *Proceedings of the 2015 Conference on Certified Programs and Proofs, CPP 2015, Mumbai, India, January 15-17, 2015*, Xavier Leroy and Alwen Tiu (Eds.). ACM, 3–14.

Jaroslav Sevcík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2011. Relaxed-Memory Concurrency and Verified Compilation. In *Proc. 38th ACM Symposium on Principles of Programming Languages (POPL'11)*. ACM, New York, 43–54. https://doi.org/10.1145/1926385.1926393

Jaroslav Sevcík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2013. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *J. ACM* 60, 3 (2013), 22:1–22:50. https://doi.org/10.1145/2487241.2487248

Youngju Song, Minki Cho, Dongjoo Kim, Yonghyun Kim, Jeehoon Kang, and Chung-Kil Hur. 2020. CompCertM: CompCert with C-Assembly Linking and Lightweight Modular Verification. *Proc. ACM Program. Lang.* 4, POPL, Article 23 (Jan. 2020), 31 pages. https://doi.org/10.1145/3371091

Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. 2015. Compositional CompCert. In *Proc. 42nd ACM Symposium on Principles of Programming Languages (POPL'15)*. ACM, New York, 275–287. https://doi.org/10.1145/2676726.2676985

Harvey Tuch, Gerwin Klein, and Michael Norrish. 2007. Types, bytes, and separation logic. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, Martin Hofmann and Matthias Felleisen (Eds.). ACM, 97–108.

Christian Urban and Stefan Berghofer. 2006. A Recursion Combinator for Nominal Datatypes Implemented in Isabelle/HOL. In *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4130)*, Ulrich Furbach and Natarajan Shankar (Eds.). Springer, 498–512.

Christian Urban and Christine Tasson. 2005. Nominal Techniques in Isabelle/HOL. In *Automated Deduction - CADE-20, 20th International Conference on Automated Deduction, Tallinn, Estonia, July 22-27, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3632)*, Robert Nieuwenhuis (Ed.). Springer, 38–53.

Yuting Wang, Pierre Wilke, and Zhong Shao. 2019. An Abstract Stack Based Approach to Verified Compositional Compilation to Machine Code. *Proc. ACM Program. Lang.* 3, POPL, Article 62 (Jan. 2019), 30 pages. https://doi.org/10.1145/3290375

Yuting Wang, Xiangzhe Xu, Pierre Wilke, and Zhong Shao. 2020. CompCertELF: Verified Separate Compilation of C Programs into ELF Object Files. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 197 (2020), 28 pages. https://doi.org/10.1145/3428265