Verified Compilation of C Programs with a Nominal Memory Model

Yuting Wang\textsuperscript{1}, Ling Zhang\textsuperscript{1}, Zhong Shao\textsuperscript{2} and Jérémie Koenig\textsuperscript{2}

\textsuperscript{1}Shanghai Jiao Tong University, China
\textsuperscript{2}Yale University, U.S.A.

Philadelphia (Virtually), POPL, Jan 2022
Background

- Memory Models in Verified Compilation
  - Semantics for languages based on some memory model
  - Prove preservation of semantics with memory invariants

The Structure of Verified Compilers
The State-of-the-Art

• **Block-Based Memory Model**
  • Memory model for CompCert
  • Pointers:
    - a pair \((b, \delta)\) of block id \(b\) and offset \(\delta\)
  • Pointer Arithmetic:
    - \((b, \delta) + n = (b, \delta + n)\)
  • Memory isolation by definition

• **Injections as Memory Invariants**
  • An injection function \(j\) is a partial mapping for blocks
  • \(j(b) = \text{Some}(b', \delta)\) if \(b\) is embedded into \(b'\) at offset \(\delta\)
  • \(j(b) = \text{None}\) if \(b\) is pulled out of the memory

\[ j(b_1) = \text{Some}(b', 0) \]
\[ j(b_2) = \text{Some}(b', \delta) \]
\[ j(b_3) = \text{None} \]
Restrictions

- **Concrete Numbering of Memory Blocks**
  a) Block identifiers are positive numbers: 1, 2, ..., \( n \), ...
  b) A special identifier called `nextblock` for allocating fresh blocks
  c) Valid blocks are \{1, 2, ..., nextblock \- 1\}

- \( b_1 = 1 \)
- \( b_2 = 2 \)
- \( b_3 = 3 \)
- \( \text{nextblock} = 4 \)
Problems

1. No distinction between different memory regions
2. Contiguous numbering brings unnecessary dependency
3. Global constraint imposed by $\texttt{nextblock}$

Elimination of Unused Global Variables

Linking of Multi-Threaded Programs
Big Picture

Treatment of **Named Resources in Formal Verification**

1. Is there a more flexible representation of memory space?

2. What benefits does it bring to compiler verification?
Our Contributions

• **Nominal Memory Model**: Generalization of Block-Based Memory Model
  • Flexible representation of blocks based on nominal techniques
  •Eliminates unnecessary dependency and global constraints
  • *Compatible with all existing mechanisms in BBMM*

• **Nominal CompCert**: A General Framework for Verified Compilation of C
  • Proofs are abstracted over the interface of nominal memory model
  • Supports complex memory structures through instantiation

• **Application of Nominal CompCert**
  • Verified compilation with structured memory
  • Verified contextual compilation to multi-stack machines
Memory Representation with Nominal Names

• **Background: Nominal Techniques for Managing Named Objects**
  • Names are represented as atoms in countably infinite sets
  • Renaming is described as permutations (bijection) on atoms
  • A set $A$ of atoms supports an object $x$ if
    \[ \forall \pi, \pi(x) = x \quad (\pi \text{ denotes a permutation on atoms that is identity for } A) \]
  • A name $a$ (atom) is fresh to $x$ if $a$ is not in some support $A$

• **Key Ideas:**
  • Atoms to generalize block ids
  • Permutation is equivalent to (renaming-based) memory injection
  • Supports to generalize valid block ids
  • Freshness to generalize $nextblock$

• **Note:** We do not yet exploit the analogy between permutation and injection
Nominal Memory Model

An Abstraction of Block-Based Memory Model with a Nominal Interface

(* Block ADT *)
Module Type BLOCK.
  Parameter block : Type.
  Parameter eq_block : ∀ x y : block, {x = y} + {x ≠ y}
End BLOCK.

(* Support ADT *)
Module Type SUP.
  Parameter sup : Type.
  Parameter sup_empty : sup.
  Parameter fresh_block : sup → block.
  Parameter sup_incr : sup → sup.
  Parameter valid_block : block → sup → bool.
End SUP.

(* Block ADT *)
Module Block <: BLOCK.
  Definition block := positive.
  Definition eq_block := peq.
End Block.

(* Support ADT *)
Module Sup <: SUP.
  Definition sup := list block.
  Definition sup_empty : sup = [].
  Definition fresh_block (s: sup) := (max s) + 1.
  Definition sup_incr (s: sup) := (fresh_block s) :: s.
  Definition valid_block (b: sup) (s: sup) := b ∈ s.
End Sup.

Interface of the Nominal Memory Model
Block-Based Memory Model
Benefits

**Problems:**
1. No Distinction of Memory Regions
2. Contiguous Numbering of Blocks
3. Global Constraint from $nextblock$

**Solutions:**
1. Block Type for Classifying Memory
2. Support Type for Separating Memory
3. $fresh\_block$ for Localized Allocation

All operations, properties and proofs remain (almost) unchanged!
Nominal CompCert

A Complete Extension of CompCert with the Nominal Memory Model

- **Abstraction:** Proofs hold under any instantiation of nominal interface

The Structure of Nominal CompCert
Enhanced Verified Compilation

1. Verified Compilation with Structured Memory
2. Verified Contextual Compilation to Multi-Stack Machines
Structured Memory Space

• **Key Idea:** Rich memory structures via instantiating blocks and supports
• **Memory Space = Global Space + Stack Space**

```
Record sup := {global ; stack }.
```

• Global blocks are given static names
• Stack space is organized into a tree of frames
• Note: Heap is part of global memory

• **Block Type:**

```
Inductive block :=
| Global : ident → block.
| Stack : option ident → list nat → positive → block;
```

Stack (Some g) [2,0] 1
Structural Injection Functions

- Represent memory invariant by static injection functions
- **Example:** Elimination of Unused Global Variables

```
Variable ge: genv. (** target environment **)

Definition check_block (s:sup) (b:block): bool :=
match b with
  | Stack _ _ _ ⇒ valid_block b s
  | Global i ⇒ match (find_symbol ge i) with
    | None ⇒ false | Some _ ⇒ true
end
end.

Definition struct_meminj (s:sup) (b:block) :=
if check_block s b then Some (b, 0) else None.
```
Reasoning about Local Memory Transformations

• **Observation:** Many transformation focuses on local memory regions
• Structural injections capture local memory transformations
• **Example:** Merging of Stack-Allocated Variables

```
Variable ge : genv. (* source environment *)

Definition unchecked_meminj (b:block) :=
  match b with
  | Global _ ⇒ Some (b, 0)
  | Stack (Some id) p i ⇒
    offset ← find_frame_offset ge id i ;
    Some (Stack (Some id) p 1, offset)
  end.

Definition struct_meminj (s:sup) (b:block) :=
  if valid_block b s
  then unchecked_meminj b
  else None.
```
Nominal CompCert with Structured Memory

- Complete Extension to Nominal CompCert with
  - Structured Memory Space
  - Intuitive Proofs with Concrete Memory Injections

Nominal Memory Model with Structured Memory Space
Contextual Compilation with Multiple Stacks

- **Contextual Compilation**
  - Open modules compiled in contextual memory
  - Investigated extensively for verified compilation

- **Problems with Contextual Compilation of Multiple Threads**
  1. Independent Stacks
  2. Finite and Continuous Stacks

Certified Concurrent Abstraction Layers (Gu et. al, PLDI’18)
New Approach to Support Finite Stacks

• **Background:** Stack-Aware CompCert [Wang et al, POPL 2019]:
  • First extension with a finite and contiguous stack
  • No increase of stack consumption in compilation
  • Key Technique: Abstract stack in the memory model

• **Observation:** Abstract stack describes properties of memory space

• **Stack-Aware Nominal CompCert**
  • Absorb the abstract stack into support:
    
    ```
    Record sup := {global: list ident; stack: stree; astack: stackdt}.
    ```

  • Significantly simplified proofs for preservation of stack consumption
Multi-Stack CompCert

1. Merge stack frames into finite and contiguous stacks
2. Add multiple stacks that grow independently

Record $sup := \{\text{global: list ident; stack: list stree; astack: list stackadt; thread_id: nat}\}$. 
Contextual Compilation to Multi-Stack Machine

- Direct Application of Multi-Stack CompCert

Thread 1

\[ \begin{align*}
&b_1 \\
&b_2 \\
&b_3
\end{align*} \]

Thread 2

\[ \begin{align*}
&b_1' \\
&b_2' \\
&b_3
\end{align*} \]

Threads \{1,2\}

\[ \begin{align*}
&b_1 \\
&b_1' \\
&b_2 \\
&b_2' \\
&b_3
\end{align*} \]

Compile to Independent and Finite Stacks

Max_Stack_Size

\[ b_1 \]

\[ b_1' \]

\[ b_2 \]

\[ b_2' \]

\[ b_3 \]
Evaluation

• Development is based on CompCert v3.8 in Coq

• Nominal CompCert
  • Time: 1 Person Month
  • LOC: 1.4K (0.5% addition to CompCert v3.8)

• Nominal CompCert with Structured Memory Space
  • Time: 2 Person Month
  • LOC: 3.5K (2.5% addition to Nominal CompCert)

• Multi-Stack CompCert (including Stack-Aware Nominal CompCert)
  • Time: 3 Person Month
  • LOC: 15K (10.6% addition to Nominal CompCert)

• Artifact: https://github.com/SJTU-PLV/nominal-compcert-popl22-artifact
Conclusion

• **Nominal Memory Model**: A Principled Generalization over BBMM

• **Nominal CompCert**: A Framework for Verified Compilation of C programs

• **Principled Instantiation of Nominal CompCert**

• **Note**: Regardless the complexity of instances, the existing proofs for all the memory-injection phases remain valid.

• **Future Work:**
  • Combination of Nominal Memory Model with General Compositional Verification
  • Support for Transportation of Proofs between Different Memory Structures
  • Application to Program Verification in General