# LLVM verification in Isabelle

Research Topics

Thomas Kas
t.s.kas@student.utwente.nl

2026-02-10

## Contents

## 1 Topic

LLVM is an intermediate language for compilers that serves as a backend for compilation to many different CPU architectures. It abstracts from architecture-specific CPU instruction to provide a set of common instructions for which it is relatively easy to create a compiler for programming languages. From there, LLVM has compilers to many different architectures with specific optimizations. This means creating a compiler from your language to LLVM is all that is needed to be able to compile to most existing architectures, rather than having to create compilers for each architecture. The aim of my thesis is to import LLVM snippets into Isabelle in order to create formal proofs for arbitrary programs written in languages that can be compiled to LLVM.

## 2 Related work

Much has been written about software verification. This section covers the basic principles defined by Hoare and Floyd (Section 2.1 and Section 2.2), extensions opens those bases (Section 2.3 and Section 2.4), different ways of modeling memory in proofs (Section 2.5), and finally different existing program verifiers and their underlying principles (Section 2.6).

### 2.1 Hoare Logic

Hoare logic forms the basis of a lot of program verification. It defines methods of reasoning about programs through the connections between preconditions $P$, programs $Q$ (now commonly $c$), and results of execution (or postconditions) $R$ (now commonly $Q$). This means that as long as the intended execution of the program can be defined in terms of assertions about the values of variables at some point in execution, Hoare logic can be used to prove the partial correctness of such programs. However, the logic defined by Hoare cannot be used to proof termination of programs, and as such can only prove partial correctness.[1] As such, extensions and alternate methods have been defined, such as separation logic (see Section 2.3) and verification condition generators (see Section 2.4).

### 2.2 Floyd's Method

However, Hoare logic is not the only basis to be used for program verification. Another method was defined by Floyd independently from Hoare. Rather than defining an algebra like Hoare, Floyd's method views programs as flowcharts where each vertex represents a command being executed. Verification conditions are defined as $V_c(P; Q)$ which assert that if $P$ holds and command $c$ is executed, then $Q$ will hold afterwards.[2]

### 2.3 Separation Logic

- [3]
- Extension with better reasoning for programs with pointers/dynamic memory allocation

## 2.4 Verification Condition Generators

## 2.5 Memory Models

- Axiomatic specification of memory operations [4]

## 2.6 Existing imperative program verifiers

- Framework for VCGs using theorem provers [5]
- Exporting from Isabelle to LLVM[6]
- Verifying x86 binaries[7]

### 2.6.1 Deductive verifiers

There exist many deductive verifiers for different purposes, based on different principles. Some are based on separation logic, while others use verification condition generators.

Ones based on VCGs often use a frontend/backend architecture, where the frontend translates a verification problem into some intermediate language used in verification, and the backend extracts proof obligations and proves them.[8] With this architecture, it becomes easier to support new languages for verification, as only a new frontend needs to be created rather than reimplementing all verification infrastructure. This architecture aligns with that of LLVM-based compilers, which also use a separate frontend (translating the source language to the LLVM-IR) and backend (compiling the LLVM-IR to the target architecture).

This architecture is brought to separation-logic based verifiers by Viper. It is a verification infrastructure, consisting of an intermediate language and two internal verifiers. Its aim is to bring the frontend/backend architecture to separation logic based verifiers. It supports many different languages through separate frontends, including Rust, Java, and C.[8] An example of such a front-end is VerCors. This is a verifier aimed at concurrent programs written in languages such as Java, OpenCL, and OpenMP.[9] It has a prototype implementation to support verification of LLVM-IR programs called VCLLVM, which is not yet production-ready.[10] Its purpose is similar to that of this project: create a verifier for LLVM so that all languages compiling to LLVM are immediately supported.

Another separation-logic based verifier is VeriFast, aimed at verifying single- and multi-threaded C and Java programs.[11]

Dafny has a similar approach with a key difference: instead of compiling programs to Dafny from their source language, programmers instead create programs in the Dafny language, verify them there, and then compile them from Dafny to their preferred language.[12]

- RESOLVE[13]
- Whiley[14]
- Frama-C[15]
- KIV[16]
- OpenJML[17]

## 3 Preliminary results

Work has already begun defining the semantics of LLVM in Isabelle and creating a verification condition generator based on Floyd's assertion method using weakest precondition mechanics.

### 3.1 LLVM semantics

Part of the LLVM-IR's AST has been defined as datatypes in Isabelle. With this part of the AST, functions made up of instruction blocks can be executed.

The operational semantics are defined as follows:

- Functions contain multiple labeled instruction blocks and one unlabeled block to be executed first.
- Instruction blocks have a list of regular instructions to be executed in order and one terminator instruction that impacts execution flow.
- Instructions are executed according to their specification in the LLVM Language Reference Manual.[1]

Execution of any single instruction takes in some state and produces a state. This state is defined as follows:

- It is a triple of single static assignment (SSA) values, a stack, and a heap.
- The stack and heap have the same underlying memory definition: a list of values. They are addressable using indices in these lists.
- SSA values are a mapping from a name to a value. Although LLVM only allows assigning them once per function, this only applies to their static definition, not their value in execution. As such, no such limitation is posed here. This means the verifier works on a superset of LLVM.

The regular instructions currently (partly) implemented are:

- alloca - allocates a new address in the stack, and keeps track of the address with some SSA name.
- store - stores a value at some address in the stack or heap.
- load - loads a value from some address in the stack or heap and keep track of it with an SSA name.
- icmp - compares two values (32/64 bit signed/unsigned integers) and keep track of the result as a single bit (boolean) with an SSA name.
- add - adds two values (32/64 bit signed/unsigned integers) and keep track of the result with an SSA name (another 32/64 bit integer or a poison value if an overflow occurred).
- phi - stores an SSA value from a list of possible values depending on the instruction block that lead to this block.

The terminator instructions implemented are:

[1]https://llvm.org/docs/LangRef.html

- br - branch to a different instruction given its label (could always be the same label, or switch based on a boolean SSA value).
- ret - return from the function with some value.

With this implementation, basic programs consisting of these instructions can already be executed. For example, take the following C program:

```
int main() {
    int y = 1;
    int x = y?1:0;
    return x;
}
```

This might produce the following LLVM-IR code:

```
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 0, ptr %1, align 4
  store i32 1, ptr %2, align 4
  %5 = load i32, ptr %2, align 4
  %6 = icmp ne i32 %5, 0
  br i1 %6, label %7, label %9

7:
  store i32 1, ptr %4, align 4
  %8 = load i32, ptr %4, align 4
  br label %10

9:
  br label %10

10:
  %11 = phi i32 [ %8, %9 ], [ 0, %9 ]
  store i32 %11, ptr %3, align 4
  %12 = load i32, ptr %3, align 4
  ret i32 %12
}
```

Encoded into the current AST representation, this becomes:

```
definition pmain :: "llvm_instruction_block" where
  "pmain = ([
    alloca ''1'' i32 (Some 4),
    alloca ''2'' i32 (Some 4),
    alloca ''3'' i32 (Some 4),
    alloca ''4'' i32 (Some 4),
    store i32 (val (vi32 0)) (ssa_val ''1'') (Some 4),
    store i32 (val (vi32 1)) (ssa_val ''2'') (Some 4),
    load ''5'' i32 (ssa_val ''2'') (Some 4),
    icmp ''6'' False comp_ne i32 (ssa_val ''5'') (val (vi32 0))],
    br_i1 (ssa_val ''6'') ''7'' ''9''
  )"

definition p7 :: "llvm_instruction_block" where
  "p7 = ([
    store i32 (val (vi32 1)) (ssa_val ''4'') (Some 4),
    load ''8'' i32 (ssa_val ''4'') (Some 4)],
    br_label ''10''
  )"

definition p9 :: "llvm_instruction_block" where
  "p9 = ([],
    br_label ''10''
  )"

definition p10 :: "llvm_instruction_block" where
  "p10 = ([
    phi ''11'' i32 [(''7'', ssa_val ''8''), (''9'', val (vi32 0))],
    store i32 (ssa_val ''11'') (ssa_val ''3'') (Some 4),
    load ''12'' i32 (ssa_val ''3'') (Some 4)],
    ret i32 (ssa_val ''12'')
  )"

definition phi_main :: "llvm_function" where
  "phi_main = func (func_def ''main'' i32) pmain [(''7'', p7), (''9'', p9),
(''10'', p10)]"
```

This can be executed as follows, which gives the proper return value:

```
value "execute_function empty_state phi_main"

= "ok (Some (vi32 1))" :: "llvm_value option result"
```

### 3.2 Weakest precondition mechanics

- Intro rules done for abstractions, some instructions

## 4 Rough planning

- Better LLVM semantics
- Finish intro rules for instructions
- Support verification condition generation at the level of blocks
- Add method of specifying pre- and post-conditions for code blocks
- Create flowchart view of code blocks
- Match up post-condition of one block with pre-condition of subsequent block for correctness proofs

- Support more LLVM instructions
- Support arrays
- Import/export from/to direct LLVM code

## Bibliography

[1] C. A. R. Hoare, “An axiomatic basis for computer programming,” *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969, doi: 10.1145/363235.363259.

[2] R. W. Floyd, “Assigning Meanings to Programs,” in *Program Verification*, Springer Netherlands, 1993, pp. 65–81. doi: 10.1007/978-94-011-1793-7_4.

[3] J. Reynolds, “Separation logic: a logic for shared mutable data structures,” in *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, in LICS-02. IEEE Comput. Soc, pp. 55–74. doi: 10.1109/lics.2002.1029817.

[4] W. Mansky, D. Garbuzov, and S. Zdancewic, “An Axiomatic Specification for Sequential Memory Models,” in *Computer Aided Verification*, Springer International Publishing, 2015, pp. 413–428. doi: 10.1007/978-3-319-21668-3_24.

[5] J. Matthews, J. S. Moore, S. Ray, and D. Vroon, “Verification Condition Generation Via Theorem Proving,” in *Logic for Programming, Artificial Intelligence, and Reasoning*, Springer Berlin Heidelberg, 2006, pp. 362–376. doi: 10.1007/11916277_25.

[6] P. Lammich, “Generating Verified LLVM from Isabelle/HOL,” Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019, pp. 22:1–22:19. doi: 10.4230/LIPICS.ITP.2019.22.

[7] J. A. Bockenek, F. Verbeek, P. Lammich, and B. Ravindran, “Formal Verification of Memory Preservation of x86-64 Binaries,” in *38th International Conference on Computer Safety, Reliability and Security*, Sept. 2019.

[8] P. Müller, M. Schwerhoff, and A. J. Summers, “Viper: A Verification Infrastructure for Permission-Based Reasoning,” in *Verification, Model Checking, and Abstract Interpretation*, Springer Berlin Heidelberg, 2015, pp. 41–62. doi: 10.1007/978-3-662-49122-5_2.

[9] S. Blom, S. Darabi, M. Huisman, and W. Oortwijn, “The VerCors Tool Set: Verification of Parallel and Concurrent Software,” in *Integrated Formal Methods*, Springer International Publishing, 2017, pp. 102–110. doi: 10.1007/978-3-319-66845-1_7.

[10] D. van Oorschot, M. Huisman, and Ö. Şakar, “First Steps towards Deductive Verification of LLVM IR,” in *Fundamental Approaches to Software Engineering*, Springer Nature Switzerland, 2024, pp. 290–303. doi: 10.1007/978-3-031-57259-3_15.

[11] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens, “VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java,” in *NASA Formal Methods*, Springer Berlin Heidelberg, 2011, pp. 41–55. doi: 10.1007/978-3-642-20398-5_4.

[12] K. R. M. Leino, “Dafny: An Automatic Program Verifier for Functional Correctness,” in *Logic for Programming, Artificial Intelligence, and Reasoning*, Springer Berlin Heidelberg, 2010, pp. 348–370. doi: 10.1007/978-3-642-17511-4_20.

[13] M. Sitaraman and B. W. Weide, “A Synopsis of Twenty Five Years of RESOLVE PhD Research Efforts,” *ACM SIGSOFT Software Engineering Notes*, vol. 43, no. 3, p. 17, Dec. 2018, doi: 10.1145/3229783.3229794.

[14] D. J. Pearce, M. Utting, and L. Groves, “An Introduction to Software Verification with Whiley,” in *Engineering Trustworthy Software Systems*, Springer International Publishing, 2019, pp. 1–37. doi: 10.1007/978-3-030-17601-3_1.

[15] A. Blanchard, F. Loulergue, and N. Kosmatov, “Towards Full Proof Automation in Frama-C Using Auto-active Verification,” in *NASA Formal Methods*, Springer International Publishing, 2019, pp. 88–105. doi: 10.1007/978-3-030-20652-9_6.

[16] G. Ernst, J. Pfähler, G. Schellhorn, D. Haneberg, and W. Reif, “KIV: overview and VerifyThis competition,” *International Journal on Software Tools for Technology Transfer*, vol. 17, no. 6, pp. 677–694, Apr. 2014, doi: 10.1007/s10009-014-0308-3.

[17] D. R. Cok, “OpenJML: JML for Java 7 by Extending OpenJDK,” in *NASA Formal Methods*, Springer Berlin Heidelberg, 2011, pp. 472–479. doi: 10.1007/978-3-642-20398-5_35.