# FORMALIZING AN SSA-BASED COMPILER FOR VERIFIED ADVANCED PROGRAM TRANSFORMATIONS

Jianzhou Zhao

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2013

Steve Zdancewic, Associate Professor of Computer and Information Science
Supervisor of Dissertation

Jianbo Shi, Associate Professor of Computer and Information Science
Graduate Group Chairperson

Dissertation Committee

Andrew W. Appel, Professor, Princeton University

Milo M. K. Martin, Associate Professor of Computer and Information Science

Benjamin Pierce, Professor of Computer and Information Science

Stephanie Weirich, Associate Professor of Computer and Information Science

## ABSTRACT

## FORMALIZING AN SSA-BASED COMPILER FOR VERIFIED ADVANCED PROGRAM TRANSFORMATIONS

Jianzhou Zhao

Supervisor: Steve Zdancewic

Compilers are not always correct due to the complexity of language semantics and transformation algorithms, the trade-offs between compilation speed and verifiability, *etc.* The bugs of compilers can undermine the source-level verification efforts (such as type systems, static analysis, and formal proofs) and produce target programs with different meaning from source programs. Researchers have used mechanized proof tools to implement verified compilers that are guaranteed to preserve program semantics and proved to be more robust than ad-hoc non-verified compilers.

The goal of the dissertation is to make a step towards verifying an industrial strength modern compiler—LLVM, which has a typed, SSA-based, and general-purpose intermediate representation, therefore allowing more advanced program transformations than existing approaches. The dissertation formally defines the sequential semantics of the LLVM intermediate representation with its type system, SSA properties, memory model, and operational semantics. To design and reason about program transformations in the LLVM IR, we provide tools for interacting with the LLVM infrastructure and metatheory for SSA properties, memory safety, dynamic semantics, and control-flow-graphs. Based on the tools and metatheory, the dissertation implements verified and extractable applications for LLVM that include an interpreter for the LLVM IR, a transformation for enforcing memory safety, translation validators for local optimizations, and verified SSA construction transformation.

This dissertation shows that formal models of SSA-based compiler intermediate representations can be used to verify low-level program transformations, thereby enabling the construction of high-assurance compiler passes.

# Contents

# List of Figures

# List of Tables

# List of Abbreviations

| | |
|---|---|
| AC | Allen-Cocke. |
| ADCE | Aggressive dead code elimination. |
| AH | Aycock and Horspool. |
| CFG | Control-flow graph. |
| CHK | Cooper-Harvey-Kennedy. |
| DAE | Dead alloca elimination. |
| DFS | Depth first search. |
| DSE | Dead store elimination. |
| GVN | Global value numbering. |
| IR | Intermediate representation. |
| LAA | Load after alloca. |
| LAS | Load after store. |
| LICM | Loop invariant code motion. |
| LT | Lengauer-Tarjan. |
| PO | Postorder. |
| PRE | Partial redundancy elimination. |
| SAS | Store after store. |
| SCCP | Sparse conditional constant propagation. |
| SSA | Static Single Assignment. |

# Chapter 1

# Introduction

Compiler bugs can manifest as crashes during compilation or even result in the silent generation of incorrect program binaries. Such mis-compilations can introduce subtle errors that are difficult to diagnose and generally puzzling to the software developers. A recent study [73] used random test-case generation to expose serious bugs in mainstream compilers including GCC [2], LLVM [38], and commercial tools. Whereas few bugs were found in the front end of the compiler, various optimization phases of the compiler that aim to make the generated program faster was a prominent source of bugs.

Improving the correctness of compilers is a worthy goal. Large-scale source-code verification efforts (such as the seL4 OS kernel [36] and Airbus's verification of fly-by-wire software [61]), program invariants checked by sophisticated type systems (such as Haskell and OCaml), and sound program synthesis (for example, Matlab/Simulink parallelizes high-level languages into C to achieve high performance [3]) can be undermined by an incorrect compiler. The need for correct compilers is amplified when compilers are parts of the trusted computing base in modern computer systems that include mission-critical financial servers, life-critical pacemaker firmware, and operating systems.

Verified Compilers are tackling the problem of compiler bugs by giving a rigorous proof that a compiler preserves the behavior of programs. The CompCert project [42, 68, 69, 70] first implemented a realistic and mechanically verified compiler that is programmed and *mechanically verified* in the Coq proof assistant [25] and generates compact and efficient assembly code for a large fragment of the C language. The aforementioned study [73] supports the effectiveness of this approach. Whereas the study uncovered many bugs in other compilers, the only bugs found in CompCert were in those parts of the compiler not formally verified:

> "The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users."

Despite CompCert's groundbreaking compiler-verification efforts, there still remain many challenges in applying its technology to industrial-strength compilers. In particular, the original CompCert development and the bulk of the subsequent work—with the notable exception of CompCertSSA [14] (which is concurrent with our work)—did not use a *static single assignment (SSA)* [28] intermediate representation (IR), as Leroy [42] explains:

> "Since the beginning of CompCert we have been considering using SSA-based intermediate languages, but were held off by two difficulties. First, the dynamic semantics for SSA is not obvious to formalize. Second, the SSA property is global to the code of a whole function and not straightforward to exploit locally within proofs."

In SSA, each variable is assigned statically only once and each variable definition must dominate all of its uses in the control-flow graph. These SSA properties simplify or enable many compiler optimizations [49, 71] including: sparse conditional constant propagation (SCCP), aggressive dead code elimination (ADCE), global value numbering (GVN), common subexpression elimination (CSE), global code motion, partial redundancy elimination (PRE), inductive variable analysis (indvars) and *etc.* Consequently, open-source and commercial compilers such as GCC [2], LLVM [38], Java HotSpot JIT [57], Soot framework [58], and Intel CC [59] use SSA-based IRs.

Despite their importance, there are few mechanized formalizations of the correctness properties of SSA transformations. This dissertation tackles this problem by developing formal semantics and proof techniques suitable for mechanically verifying the correctness of SSA-based compilers. We do so in the context of our Vellvm framework, which formalizes the operational semantics of programs expressed in LLVM's SSA-based IR [43] and provides Coq [25] infrastructure to facilitate mechanized proofs of properties about transformations on the LLVM IR. Moreover, because the LLVM IR is expressive to represent arbitrary program constructors, maintain properties from high-level programs, and hide details about target platforms, we define Vellvm's memory model to encode data along with high-level type information and to support arbitrary bit-width integers, padding, and alignment issues.

The Vellvm infrastructure, along with Coq's facility for extracting executable code from constructive proofs, enables Vellvm users to manipulate LLVM IR code with high confidence in the results. For example, using this framework, we can extract verified LLVM transformations that plug directly into the LLVM compiler.

In summary, **Thesis statement:** Formal models of SSA-based compiler intermediate representations can be used to verify low-level program transformations, thereby enabling the construction of high-assurance compiler passes.

**Contributions** The specific contributions of the dissertation include:

- The dissertation formally defines the sequential semantics of the industrial strength modern compiler intermediate representation—the LLVM IR that includes its type system, SSA properties, memory model, and operational semantics.

- To design and reason about program transformations in the IR, the dissertation designs tools for interacting with the LLVM infrastructure, and metatheory for SSA properties, memory safety, dynamic semantics, and control-flow-graphs.

- Based on the tools and metatheory, we implement verified and extractable applications for LLVM that include the interpreter of the LLVM IR, a transformation for enforcing memory safety, translation validators for local optimizations, and SSA construction.

The dissertation is based on our published work [75, 76, 77]. The rest of the dissertation is organized as follows: Chapter 2 presents the background and preliminaries used in the dissertation. To streamline the formalization of the SSA-based transformations, Chapter 2 also describes Vminus, a simpler subset of our full LLVM formalization—Vellvm [75], but one that still captures the essence of SSA. Chapter 3 formalizes one crucial component of SSA-based compilers—computing dominators [77]. Chapter 4 shows the dynamic and static semantics of Vminus. Chapter 5 describes the proof techniques we have developed for formalizing properties of SSA-style intermediate representations in the context of Vminus [76]. To demonstrate that our proof techniques can be used for practical compiler optimizations, Chapter 6 shows the syntax of the full LLVM IR—Vellvm. Then, Chapter 6 formalizes the semantics of Vellvm. Chapter 7 presents an application of Vellvm—a verified program transformation that hardens C programs against spatial memory safety violations (*e.g.,* buffer overflows, array indexing errors, and pointer arithmetic errors). Chapter 8 demonstrates that our proof techniques developed in Chapter 5 can be used for practical compiler optimizations in Vellvm: verifying the most performance-critical optimization pass in LLVM's compilation strategy—the `mem2reg` pass [76]. Chapter 9 summarizes our Coq development. Finally, Chapter 10 discusses the related work, and Chapter 11 concludes.

# Chapter 2

# Background

This chapter presents the background and preliminaries used in the dissertation.

## 2.1 Program Refinement

In this dissertation, we prove the correctness of a compiler by showing that its output program $P'$ preserves the semantics of its original program $P$: informally, $P'$ cannot do more than what $P$ does, although $P'$ can have fewer behaviors than $P$. With this correctness, a compiler ensures that the analysis and verification results for source programs still hold after compilation.

Formally, we use program refinement to formalize semantic preservation. Following the CompCert project [42], we define program refinement in terms of programs' external behaviors (which include program traces of input-output events, whether a program terminates, and the returned value if a program terminates): a transformed program *refines* the original if the behaviors of the original program include all the behaviors of the transformed program. We define the operational semantics using traces of a labeled transition system.

$$\begin{array}{llcl}
\text{Events} & e & ::= & v = \mathit{fid}(\overline{v_j}^{j}) \\
\text{Finite traces} & t & ::= & \varepsilon \mid e, t \\
\text{Finite or infinite traces} & T & ::= & \varepsilon \mid e, T \quad \text{(coinductive)}
\end{array}$$

We denote one small-step of evaluation as $\mathit{config} \vdash S \xrightarrow{t} S'$: in program environment *config*, program state $S$ transitions to the state $S'$, recording events $e$ of the transition in the trace $t$. An event $e$ describes the inputs $v_j$ and output $v$ of an external function call named *fid*. $\mathit{config} \vdash S \xrightarrow{t}{}^{*} S'$ denotes the reflexive, transitive closure of the small-step evaluation with a finite trace $t$. $\mathit{config} \vdash S \xrightarrow{T} \infty$ denotes a diverging evaluation starting from $S$ with a finite or infinite trace $T$. Program refinement is given by the following definition.

**Definition 1** (Program refinement)**.**

1. **init** $(\mathit{prog}, \mathit{fid}, \overline{v_j}^j, S)$ *means $S$ is the initial program state of the program prog with the main entry fid and inputs $v_j$.*

2. **final** $(S, v)$ *means $S$ is the final state with the return value $v$.*

3. $\Downarrow(\mathit{prog}, \mathit{fid}, \overline{v_j}^j, t, v)$ *means* $\exists S\, S'$. **init** $(\mathit{prog}, \mathit{fid}, \overline{v_j}^j, S)$, $\mathit{config} \vdash S \xrightarrow{t}{}^{*} S'$ *and* **final** $(S', v)$.

4. $\Uparrow(\mathit{prog}, \mathit{fid}, \overline{v_j}^j, T)$ *means* $\exists S$. **init** $(\mathit{prog}, \mathit{fid}, \overline{v_j}^j, S)$ *and* $\mathit{config} \vdash S \xrightarrow{T} \infty$.

5. $\not\Downarrow(\mathit{prog}, \mathit{fid}, \overline{v_j}^j, t)$ *means* $\exists S\, S'$. **init** $(\mathit{prog}, \mathit{fid}, \overline{v_j}^j, S)$, $\mathit{config} \vdash S \xrightarrow{t}{}^{*} S'$ *and $S'$ is stuck.*

6. **defined** $(\mathit{prog}, \mathit{fid}, \overline{v_j}^j)$ *means* $\forall\, t$, $\neg \not\Downarrow(\mathit{prog}, \mathit{fid}, \overline{v_j}^j, t)$

7. *$\mathit{prog}_2$ **refines** program $\mathit{prog}_1$, written $\mathit{prog}_1 \supseteq \mathit{prog}_2$, if*
$$\begin{array}{lll} (a) & \textbf{defined}\,(\mathit{prog}_1, \mathit{fid}, \overline{v_j}^j) & \\ (b) & \Downarrow(\mathit{prog}_2, \mathit{fid}, \overline{v_j}^j, t, v) & \Rightarrow \Downarrow(\mathit{prog}_1, \mathit{fid}, \overline{v_j}^j, t, v) \\ (c) & \Uparrow(\mathit{prog}_2, \mathit{fid}, \overline{v_j}^j, T) & \Rightarrow \Uparrow(\mathit{prog}_1, \mathit{fid}, \overline{v_j}^j, T) \\ (d) & \not\Downarrow(\mathit{prog}_2, \mathit{fid}, \overline{v_j}^j, t) & \Rightarrow \not\Downarrow(\mathit{prog}_1, \mathit{fid}, \overline{v_j}^j, t) \end{array}$$

Note that refinement requires only that a transformed program preserves the semantics of a well-defined original program, but does not constrain the transformation of undefined programs.

We use the simulation diagrams in Figure 2.1 to prove that a program transformation satisfies the refinement property. Note that in Figure 2.1, we use $S$ to denote program states of a source program and use $\Sigma$ to denote program states of a target program. The backward simulation diagrams imply program refinement for both deterministic and non-deterministic semantics. The forward simulation diagrams (which are similar to the diagrams the CompCert project [42] uses) imply program refinement for deterministic semantics. In each diagram, the program states of original and compiled programs are on the left and right respectively. A line denotes a relation $\sim$ between program states. Solid lines or arrows denote hypotheses; dashed lines or arrows denote conclusions.

At a high-level, we first need to find a relation $\sim$ between program states and their transformed counterparts. The relation must hold initially, imply equivalent returned values finally, and imply that stuck states

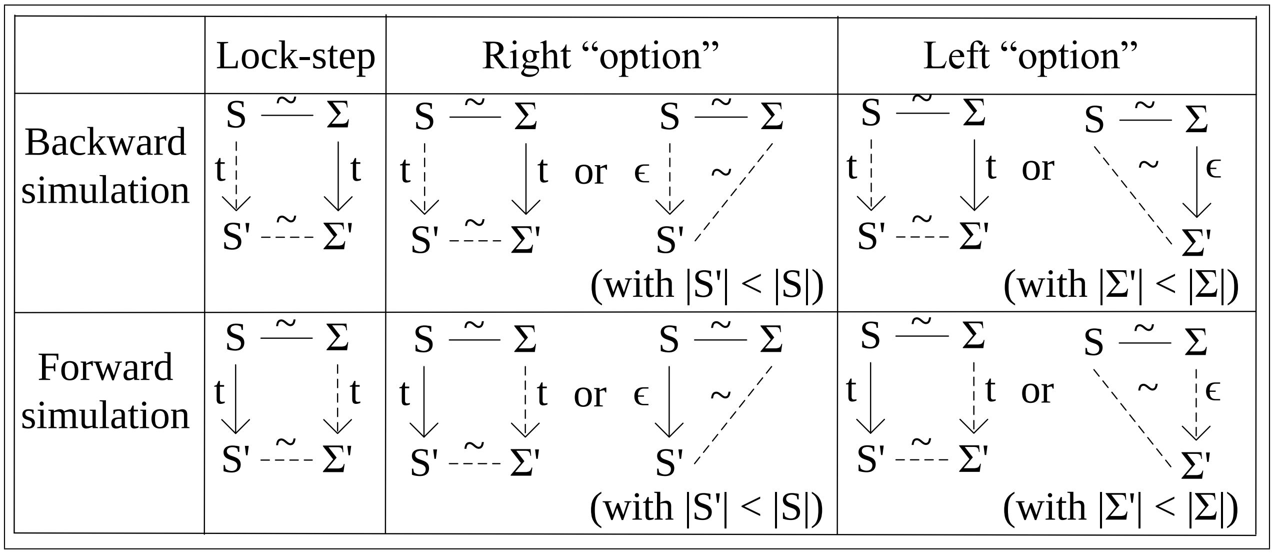


Figure 2.1: Simulation diagrams that imply program refinement.

are related. Then, depending on the transformation, we prove that a specific diagram holds: lock-step simulation is for variable substitution, right “option” simulation is for instruction removal, and left “option” simulation is for instruction insertion. Because the existence of a diagram implies that the source and target programs share traces, we can prove the equivalence of program traces by decomposing program transitions into matched diagrams. To ensure that an original program terminates iff the transformed program terminates, the “option” simulations are parameterized by a measure of program states $|S|$ that must decrease to prevent “infinite stuttering” problems.

## 2.2 Static Single Assignment

One of the crucial analysis in compiler design is determining values of temporary variables statically. With the analysis, compilers can reason about equivalence among variables and expressions, and then eliminate redundant computation to reduce the runtime overhead. However, the analysis for an ordinary imperative language is not trivial: a temporary variable can be defined more than once; therefore, at runtime its value introduced at one definition is alive only by the next definition of the variable. Moreover, because program transformations can add or remove temporary variables, change control flow graphs, compilers have to rerun the analysis after transformations.

To address the issue, *Static Single Assignment* (SSA) form [28] [1] was proposed to enforce referential transparency syntactically [9], therefore simplifying program analysis for compilers. Informally, SSA form is an intermediate representation distinguished by its treatment of temporary variables—each such variable may be defined only once, statically, and each use of the variable must be dominated by its definition with respect to the control-flow graph of the containing function. Informally, the variable definition dominates a use if all possible execution paths to the use go through the definition first.

To maintain these invariants in the presence of branches and loops, SSA form uses $\phi$-instructions, which act like control-flow dependent move operations. Such $\phi$-instructions appear only at the start of a basic block and, crucially, they are handled specially in the dominance relation to "cut" apparently cyclic data dependencies.

The left part of Figure 2.2 shows an example program in SSA form, written using the stripped-down notation of Vminus (defined more formally in Section 2.4). The temporary $r_3$ at the beginning of the block labeled $l_2$ is defined by a $\phi$-instruction: if control enters the block $l_2$ by jumping from basic block $l_1$, $r_3$ will get the value 0; if control enters from block $l_2$ (via the back edge of the branch at the end of the block), then $r_3$ will get the value of $r_5$.

The SSA form is good for implementing optimizations because it identifies variable names with the program points at which they are defined. Maintaining the SSA invariants thus makes definition and use information of each variable more explicit. Also, because each variable is defined only once, there is less mutable state to be considered (for purposes of aliasing, *etc.*) in SSA form, which makes certain code transformations easier to implement.

Program transformations like the one in Figure 2.2 are correct if the transformed program refines the original program (in the sense described above) and the result is well-formed SSA. Proving that such code transformations are correct is nontrivial because they involve non-local reasoning about the program. Chapter 5 describes how such optimizations can be formally proven correct by breaking them into micro transformations, each of which can be shown to preserve the semantics of the program and maintain the SSA invariants.

| Original | Transformed |
|---|---|
| $l_1 : \cdots$ | $l_1 : \cdots$ |
| $\quad \cdots$ | $\quad r_4 := r_1 * r_2$ |
| $\quad \mathbf{br}\, r_0\, l_2\, l_3$ | $\quad \mathbf{br}\, r_0\, l_2\, l_3$ |
| $l_2 : r_3 = \mathbf{phi}\,\mathbf{int}[0, l_1][r_5, l_2]$ | $l_2 : r_3 = \mathbf{phi}\,\mathbf{int}[0, l_1][r_5, l_2]$ |
| $\quad r_4 := r_1 * r_2$ | |
| $\quad r_5 := r_3 + r_4$ | $\quad r_5 := r_3 + r_4$ |
| $\quad r_6 := r_5 \geq 100$ | $\quad r_6 := r_5 \geq 100$ |
| $\quad \mathbf{br}\, r_6\, l_2\, l_3$ | $\quad \mathbf{br}\, r_6\, l_2\, l_3$ |
| $l_3 : r_7 = \mathbf{phi}\,\mathbf{int}[0, l_1][r_5, l_2]$ | $l_3 : r_7 = \mathbf{phi}\,\mathbf{int}[0, l_1][r_5, l_2]$ |
| $\quad r_8 := r_1 * r_2$ | |
| $\quad r_9 := r_8 + r_7$ | $\quad r_9 := r_4 + r_7$ |

In the original program (left), $r_1 * r_2$ is a partial common expression for the definitions of $r_4$ and $r_8$, because there is no domination relation between $r_4$ and $r_8$. Therefore, eliminating the common expression directly is not correct. For example, we cannot simply replace $r_8 := r_1 * r_2$ by $r_8 := r_4$ since $r_4$ is not available at the definition of $r_8$ if the block $l_2$ does not execute before $l_3$ runs. To transform this program, we might first move the instruction $r_4 := r_1 * r_2$ from the block $l_2$ to the block $l_1$ because the definitions of $r_1$ and $r_2$ must dominate $l_1$, and $l_1$ dominates $l_2$. Then we can safely replace all the uses of $r_8$ by $r_4$, because the definition of $r_4$ in $l_1$ dominates $l_3$ and therefore dominates all the uses of $r_8$. Finally, $r_8$ is removed, because there are no uses of $r_8$.

Figure 2.2: An SSA-based optimization.

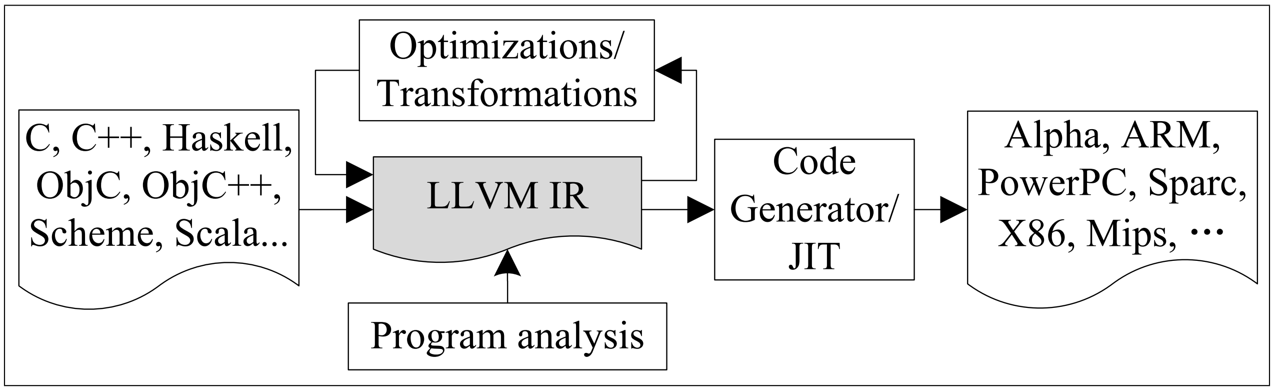

Optimizations/ Transformations
C, C++, Haskell, ObjC, ObjC++, Scheme, Scala...
LLVM IR
Code Generator/ JIT
Alpha, ARM, PowerPC, Sparc, X86, Mips, ···
Program analysis

Figure 2.3: The LLVM compiler infrastructure

## 2.3 LLVM

LLVM [43] (*Low-Level Virtual Machine*) is a robust, industrial-strength, and open-source compilation framework. LLVM uses a typed, platform-independent SSA-based IR originally developed as a research

[1] In the literature, there are different variants of SSA forms [16]. We use the LLVM SSA form: for example, memory locations are not in SSA form; LLVM does not maintain any connection between a variable in LLVM and its original name in imperative form; and the live ranges of variables can overlap.

| | | | |
|---|---|---|---|
| Types | $typ$ | $::=$ | $\mathbf{int}$ |
| Constants | $cnst$ | $::=$ | $Int$ |
| Values | $val$ | $::=$ | $r \mid cnst$ |
| Binops | $bop$ | $::=$ | $+ \mid * \mid \&\& \mid = \mid \geq \mid \leq \mid \cdots$ |
| Right-hand-sides | $rhs$ | $::=$ | $val_1\, bop\, val_2$ |
| Commands | $c$ | $::=$ | $r := rhs$ |
| Terminators | $tmn$ | $::=$ | $\mathbf{br}\, val\, l_1\, l_2 \mid \mathbf{ret}\, typ\, val$ |
| Phi Nodes | $\phi$ | $::=$ | $r = \mathbf{phi}\, typ\, \overline{[val_j, l_j]}^{j}$ |
| Instructions | $insn$ | $::=$ | $\phi \mid c \mid tmn$ |
| Non-$\phi$s | $\psi$ | $::=$ | $c \mid tmn$ |
| Blocks | $b$ | $::=$ | $l\, \overline{\phi}\, \overline{c}\, tmn$ |
| Functions | $f$ | $::=$ | $\mathbf{fun}\, \{\overline{b}\}$ |

Figure 2.4: Syntax of Vminus

tool for studying optimizations and modern compilation techniques [38]. The LLVM project has since blossomed into a robust, industrial-strength, and open-source compilation platform that competes with GCC in terms of compilation speed and performance of the generated code [38]. As a consequence, it has been widely used in both academia and industry [2].

An LLVM-based compiler is structured as a translation from a high-level source language to the LLVM IR (see Figure 2.3). The LLVM tools provide a suite of IR to IR translations, which provide optimizations, program transformations, and static analyses. The resulting LLVM IR code can then be lowered to a variety of target architectures, including x86, PowerPC, and ARM (either by static compilation or dynamic JIT-compilation). The LLVM project focuses on C and C++ front-ends, but many source languages, including Haskell, Scheme, Scala, Objective C and others have been ported to target the LLVM IR.

## 2.4 The Simple SSA Language—Vminus

To streamline the formalization of the SSA-based transformations, we describe the properties and proof techniques of SSA in the context of Vminus, a simpler subset of our full LLVM formalization—Vellvm [75], but one that still captures the essence of SSA.

Figure 2.4 gives the syntax of Vminus. Every Vminus expression is of type integer. Operations in Vminus compute with values *val*, which are either identifiers *r* naming temporaries or constants *cnst* that must be integer values. We use *R* to range over sets of identifiers.

[2] See `http://llvm.org/ProjectsWithLLVM/`

All code in Vminus resides in a top-level function, whose body is composed of blocks $b$. Here, $\overline{b}$ denotes a list of blocks; we also use similar notation for other lists. As is standard, a basic block consists of a labeled entry point $l$, a series of $\phi$ nodes, a list of commands $c$s, and a terminator instruction $tmn$. In the following, we also use the label $l$ of a block to denote the block itself.

Because SSA ensures the uniqueness of variables in a function, we use $r$ to identify instructions that assign temporaries. For instructions that do not update temporaries, such as terminators, we introduce "ghost" identifiers to identify them—$r : \textbf{br}\, val\, l_1\, l_2$. Ghost identifiers satisfy uniqueness statically but do not have dynamic semantics, and are not shown when we do not distinguish instructions.

The set of blocks making up the top-level function constitutes a control-flow graph with a well-defined entry point that cannot be reached from other blocks. We write $f[l] = \lfloor b \rfloor$ if there is a block $b$ with label $l$ in function $f$. Here, the $\lfloor\,\rfloor$ (pronounced "some") indicates that the function is partial (might return "none" instead).

As usual in SSA, the $\phi$ nodes join together values from a list of predecessor blocks of the control-flow graph—each $\phi$ node takes a list of (value, label) pairs that indicates the value chosen when control transfers from a predecessor block with the associated label. The commands $c$ include the usual suite of binary arithmetic or comparison operations (*bop*—*e.g.,* addition $+$, multiplication $*$, and &&, equivalence $=$, greater than or equal $\geq$, less than or equal $\leq$, *etc.*). We denote the right-hand-sides of commands by *rhs*. Block terminators (**br** and **ret**) branch to another block or return a value from the function. We also use metavariable *insn* to range over $\phi$-nodes, commands and terminators, and non-phinodes $\psi$ to represent commands and terminators.

# Chapter 3

# Mechanized Verification of Computing Dominators

One crucial component of SSA-based compilers is computing dominators—on a control-follow-graph, a node $l_1$ dominates a node $l_2$ if all paths from the entry to $l_2$ must go through $l_1$ [8]. Dominance analysis allows compilers to represent programs in the SSA form [28] (which enables many advanced SSA-based optimizations), optimize loops, analyze memory dependency, and parallelize code automatically, *etc.* Therefore, one prerequisite to the formal verification of SSA-based compilers is formalizing computing dominators.

In this chapter, we present the formalization of dominance analysis used in the Vellvm project. To the best of our knowledge, this is the first mechanized verification of dominator computation for LLVM. Although the CompCertSSA project [14] also formalized dominance analysis to prove the correctness of a global value numbering optimization, as we explain in Chapter 10, our results are more general: beyond soundness, we establish completeness and related metatheory results that can be used in other applications. Because different styles of formalization may also affect the cost of proof engineering, we also discuss some tradeoffs in the choices of formalization.

To simplify the formal development, we describe the work in the context of Vminus in this section. The following sections describe how to extend the work for the full Vellvm. Following LLVM, we distinguish dominators at the block level and at the instruction level. Given the former one, we can easily compute the latter one. Therefore, we will focus on the block-level analysis. Section 4.2 discusses the instruction-level analysis, Section 4.3 shows how to use the dominance analysis to design a type checker for the SSA form, and Chapter 5 describes how to verify SSA-based optimizations by the metatheory of the dominance analysis.

Concretely, we present the following specific contributions:

1. Section 3.1 gives an abstract and succinct specification of computing dominators at the block level.

2. We instantiate the specification by two algorithms. Section 3.2 shows the standard dominance analysis [7] (AC). Section 3.3 presents an extension of the standard algorithm [24] (CHK) that is easy to implement and verify, but still fast. We verify the correctness of both algorithms. In the meanwhile, we provide a verified depth first search algorithm (Section 3.2.1).

3. Then, Section 3.4 constructs dominator trees that compilers traverse to transform programs.

4. Section 3.6 evaluates performance of the algorithms, and shows that in practice CHK runs nearly as fast as the sophisticated algorithm used in LLVM.

5. We formalize all the claims of the paper for Vminus and the full Vellvm in Coq (available at `http://www.cis.upenn.edu/~stevez/vellvm/`).

Note that in this chapter we present definitions and proofs in Coq; the later chapters use mathematical notations.

## 3.1 The Specification of Computing Dominators

This section first defines dominators in term of the syntax of Vminus, then gives an abstract and succinct specification of algorithms that compute dominators.

### 3.1.1 Dominance

The set of blocks making up the top-level function $f$ constitutes a control-flow graph (CFG) $G = (e, succs)$ where $e$ is the entry point (the first block) of $f$; $succs$ maps each label to a list of its successors. On a CFG, we use $G \models l_1 \rightarrow^* l_2$ to denote a path $\rho$ from $l_1$ to $l_2$, and $l \in \rho$ to denote that $l$ is in the path $\rho$. By `wf f` (which Section 4.3 formally defines), we require that a well-formed function must contain an entry point that cannot be reached from other blocks, all terminators can only branch to blocks within $f$, and that all labels in $f$ are unique. In this section, we only consider well-formed functions to streamline the presentation.

**Definition 2** (Domination (Block-level)). *Given $G$ with an entry $e$,*

- *A block $l$ is **reachable**, written $G \to^* l$, if there exists a path $G \models e \to^* l$.*
- *A block $l_1$ **dominates** a block $l_2$, written $G \models l_1 \gg= l_2$, if for every path $\rho$ from $e$ to $l_2$, $l_1 \in \rho$.*
- *A block $l_1$ **strictly dominates** a block $l_2$, written $G \models l_1 \gg l_2$, if for every path $\rho$ from $e$ to $l_2$, $l_1 \neq l_2 \wedge l_1 \in \rho$.*

Because the dominance relations of a function at the block level and in its CFG are equivalent, in the following we do not distinguish $f$ and $G$. The following consequence of the definitions are useful to define the specification of computing dominators. First of all, we can convert $\gg$ and $\gg=$:

**Lemma 1.**

- *If $G \models l_1 \gg l_2$, then $G \models l_1 \gg= l_2$.*
- *If $G \models l_1 \gg= l_2 \wedge l_1 \neq l_2$, then $G \models l_1 \gg l_2$.*

For all labels in $G$, $\gg=$ and $\gg$ are transitive.

**Lemma 2** (Transitivity).

- *If $G \models l_1 \gg= l_2$ and $G \models l_2 \gg= l_3$, then $G \models l_1 \gg= l_3$.*
- *If $G \models l_1 \gg l_2$ and $G \models l_2 \gg l_3$, then $G \models l_1 \gg l_3$.*

However, because there is no path from the entry to unreachable labels, $\gg=$ and $\gg$ relate every label to any unreachable labels.

**Lemma 3.** *If $\neg(G \to^* l_2)$, then $G \models l_1 \gg= l_2$ and $G \models l_1 \gg l_2$.*

If we only consider the reachable labels in $V$, $\gg$ is acyclic.

**Lemma 4** ($\gg$ is acyclic). *If $G \to^* l$, then $\neg G \models l \gg l$.*

Moreover, all labels that strictly dominate a reachable label are ordered.

**Lemma 5** ($\gg$ is ordered). *If $G \to^* l_3$, $l_1 \neq l_2$, $G \models l_1 \gg l_3$ and $G \models l_2 \gg l_3$, then $G \models l_1 \gg l_2 \vee G \models l_2 \gg l_1$.*

```
Module Type ALGDOM.
  Parameter sdom: f -> l -> set l.
  Definition dom f l1 := l1 {+} sdom f l1.
  Axiom entry_sound: forall f e, entry f = Some e -> sdom f e = {}.
  Axiom successors_sound: forall f l1 l2,
    In l1 ((succs f) !!! l2) -> sdom f l1 {<=} dom f l2.
  Axiom complete: forall f l1 l2,
    wf f -> f |= l1 >> l2 -> l1 `in` (sdom f l2).
End ALGDOM.

Module AlgDom_Properties(AD: ALGDOM).
  Lemma sound: forall f l1 l2,
    wf f -> l1 `in` (AD.sdom f l2) -> f |= l1 >> l2.
  (*******************************************************************)
  (* Properties: conversion, transitivity, acyclicity, ordering and ... *)
  (*******************************************************************)
End AlgDom_Properties.
```

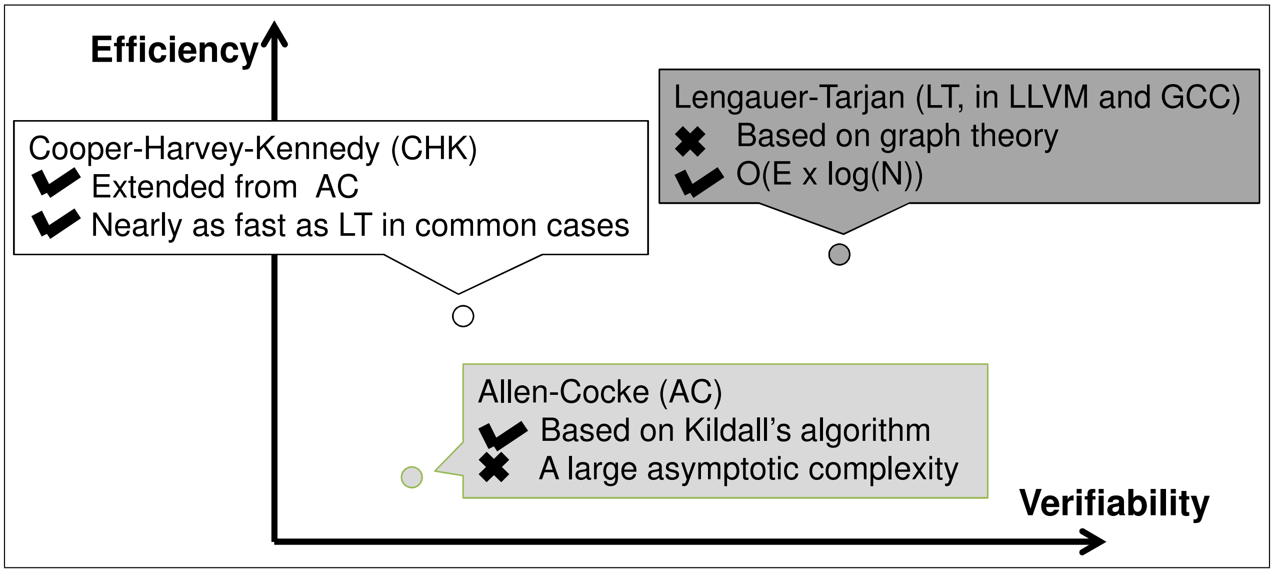
Figure 3.1: The specification of algorithms that find dominators.

### 3.1.2 Specification

**Coq Notations.** We use `{}` to denote an empty set; use `{+}`, `{<=}`, `` `in` ``, `{\/}` and `{/\}` to denote set addition, inclusion, membership, union and intersection respectively. Our developments reuse the basic tree and map data structures implemented in the CompCert project [42]: `ATree.t` and `PTree.t` are trees with keys of type $l$ and `positive` respectively; `PMap.t` is a map with keys of type `positive`. We use `!` and `!!` to denote tree and map lookup respectively. A tree lookup is partial, while a map lookup returns a default value when the key to search does not exist. *succs* are defined by trees. `!!!` is a special tree lookup for *succs*, and it returns an empty list when a searched-for key does not exist. `[x]` is a list with one element `x`.

Figure 3.1 gives an abstract specification of algorithms that compute dominators using a Coq module interface `ALGDOM`. First of all, `sdom` defines the signature of a dominance analysis algorithm: given a function $f$ and a label $l_1$, (`sdom` $f$ $l_1$) returns the set of strict dominators of $l_1$ in $f$; `dom` defines the set of dominators of $l_1$ by adding $l_1$ into $l_1$'s strict dominators.

To make the interface simple, `ALGDOM` requires only basic properties that ensure that `sdom` is correct: it must be both `sound` and `complete` in terms of the declarative definitions (Definition 2). Given the correctness of `sdom`, the `AlgDom_Properties` module can 'lift' properties (conversion, transitivity, acyclicity, ordering, *etc.*) from the declarative definitions to the implementations of `sdom` and `dom`. Sec-

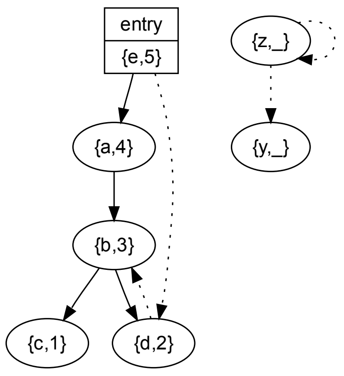

Efficiency
Lengauer-Tarjan (LT, in LLVM and GCC)
✖ Based on graph theory
✓ O(E x log(N))
Cooper-Harvey-Kennedy (CHK)
✓ Extended from AC
✓ Nearly as fast as LT in common cases
Allen-Cocke (AC)
✓ Based on Kildall's algorithm
✖ A large asymptotic complexity
Verifiability
Figure 3.2: Algorithms of computing dominators

tion 3.4, Section 3.5, Section 4.3 and Chapter 8 show how clients of `ALGDOM` use the properties proven in `AlgDom_Properties` by examples.

`ALGDOM` requires completeness of the algorithm directly. Soundness of the algorithm can be proven by two more basic properties: `entry_sound` requires that the entry has no strict dominators; `successors_sound` requires that if $l_1$ is a successor of $l_2$, then $l_2$'s dominators must include $l_1$'s strict dominators. Given an algorithm that establishes the two properties, `AlgDom_Properties` proves that the algorithm is sound by induction over any path from the entry to $l_2$.

### 3.1.3 Instantiations

In the literature, there is a long history of algorithms that find dominators (See Figure 3.2), each making different trade-offs between efficiency and simplicity. Most of the industrial compilers, such as LLVM and GCC, use the classic Lengauer-Tarjan algorithm [40] (LT) that has a complexity of $O(E * log(N))$ where $N$ and $E$ are the number of nodes and edges respectively, but is complicated to implement and reason about because it is base on complicated graph theory. The Allen-Cocke algorithm [7] (AC) based on iteration is easier to design, but suffers from a large asymptotic complexity of $O(N^3)$. Moreover, LT explictly creates dominator trees that provide convenient data structures for compilers whereas AC needs an additional tree construction algorithm with more overhead. The Cooper-Harvey-Kennedy algorithm [24] (CHK) extends from AC with careful engineering and runs nearly as fast as LT in common cases [24, 31], but is still simple



| stk | visited | PO_l2p po |
|---|---|---|
| e[a d] | e | |
| e[d]; a[b] | e a | |
| e[d]; a[]; b[c d] | e a b | |
| e[d]; a[]; b[d]; c[] | e a b c | (c,1) |
| e[d]; a[]; b[]; d[b] | e a b c d | (c,1) |
| e[d]; a[]; b[]; d[] | e a b c d | (c,1); (d,2) |
| e[d]; a[]; b[]; | e a b c d | (c,1); (d,2); (b,3) |
| e[d]; a[]; | e a b c d | (c,1); (d,2); (b,3); (a,4) |
| e[] | e a b c d | (c,1); (d,2); (b,3); (a,4); (e,5) |

Figure 3.3: The postorder (left) and the DFS execution sequence (right).

to implement and reason about. Moreover, CHK generates dominator trees implicitly, and provides a faster tree construction algorithm.

Because CHK gives a relatively good trade-off between verifiability and efficency, we present CHK as an instance of `ALGDOM`. In the following sections, we first review the AC algorithm, and then study its extension CHK.

## 3.2 The Allen-Cocke Algorithm

The Allen-Cocke algorithm (AC) is an instance of the forward worklist-based Kildall's algorithm [35] that computes program fixpoints by iteration. The number of iterations that a worklist-based algorithm takes to meet a fixpoint depends on the order in which nodes are processed: in particular, forward algorithms can converge relatively faster when visiting nodes in reverse postorder (PO) [33].

At the high-level, our Coq implementation of AC works in three steps: 1) calculate the PO of a CFG by depth-first-search (DFS); 2) compute strict dominators for PO-numbered nodes in Kildall; 3) finally relate the analysis results to the original nodes. We omit the 3rd step's proofs here.

This section first presents a verified DFS algorithm that computes PO, then reviews Kildall's algorithm as implemented in the CompCert project [42], and finally it studies the implementation and metatheory of AC.

```
Record PostOrder := mkPO { PO_cnt: positive; PO_l2p: LTree.t positive }.

Record Frame := mkFr { Fr_name: l; Fr_scs: list l }.

Definition dfs_F_type : Type := forall (succs: LTree.t (list l))
  (visited: LTree.t unit) (po:PostOrder) (stk: list Frame), PostOrder.

Definition dfs_F (f: dfs_F_type) (succs: LTree.t (list l))
  (visited: LTree.t unit) (po:PostOrder) (stk: list Frame): PostOrder :=
  match find_next succs visited po stk with
  | inr po' => po'
  | inl (next, visited', po', stk') => f succs visited' po' stk'
  end.
```

Figure 3.4: The DFS algorithm.

### 3.2.1 DFS: PO-numbering

DFS starts at the entry, visits nodes as deep as possible along each path, and backtracks when all deep nodes are visited. DFS generates PO by numbering a node after all its children are numbered. Figure 3.3 gives a PO-numbered CFG. In the CFG, we represent the depth-first-search (DFS) tree edges by solid arrows, and non-tree edges by dotted arrows. We draw the entry node in a box, and other nodes in circles. Each node is labeled by a pair with its original label name on the left, and its PO number on the right. Because DFS only visits reachable nodes, the PO numbers of unreachable nodes are represented by '_'.

Figure 3.4 shows the data structures and auxiliary functions used by a typical DFS algorithm that maintains four components to compute PO. `PostOrder` takes the next available PO number and a map from nodes to their PO numbers with type `positive`. The map from a node to its successors is represented by `succs`. To facilitate reasoning about DFS, we represent the recursive information of DFS explicitly by a list of `Frame` records that each contains a node `Fr_name` and its unprocessed successors `Fr_scs`. To prevent the search from revisiting nodes, the DFS algorithm uses `visited` to record visited nodes. `dfs_F` defines one recursive step of DFS.

Figure 3.3 (on the right) gives a DFS execution sequence (by running `dfs_F` until all nodes are visited) of the CFG in Figure 3.3 (on the left) . We use $l[l_1 \cdots l_n]$ to denote a frame with the node $l$ and its unprocessed successors $l_1$ to $l_n$; $(l, p)$ to denote a node $l$ and its PO $p$. Initially the DFS adds the entry and its successors to the stack. At each recursive step, `find_next` finds the next available node that is the unvisited node in the `Fr_scs` of the latest node $l'$ of the stack. If the next available node exists, the DFS pushes the node with

```
Fixpoint iter (A:Type) (n:nat) (F:A->A) (g:A) : A :=
  match n with
  | O => g
  | S p => F (iter A p F g)
  end.

Definition wf_stk succs visited stk :=
  stk_in_succs succs stk /\ incl visited succs

Program Fixpoint dfs_tmn succs visited po stk
  (Hp: wf_stk succs visited stk) {measure (size succs - size visited)}:
  { po':PostOrder | exists p:nat,
                    forall k (Hlt: p < k) (g:dfs_F_type),
                    iter _ k dfs_F g succs visited po stk = po' } :=
  match find_next succs visited po stk with
  | inr po' => po'
  | inl (next, visited', po', stk') =>
      let _ := dfs_tmn succs visited' po' stk' _ in _
  end.

Program Definition dfs succs entry : PostOrder :=
  fst (dfs_tmn succs empty (mkPO 1 empty) (mkFr entry [(succs!!!entry)]) _).
```

Figure 3.5: Termination of the DFS algorithm.

its successors to the stack, and makes the node to be visited. `find_next` pops all nodes in front of $l'$, and gives them PO numbers. If `find_next` fails to find available nodes, the DFS stops.

We can see that the straightforward algorithm is not a structural recursion. To implement the algorithm in Coq, we must show that it terminates. Although in Coq we can implement the algorithm by well-founded recursion, such designs are hard to reason about [17]. One of possible alternatives is implementing DFS with a ‘strong’ dependent type to specify the properties that we need to reason about DFS. However, this design is not modular because when the type of DFS is not strong enough—for example, if we need a new lemma about DFS—we must extend or redesign its implementation by adding new invariants. Instead, following the ideas in Coq’Art [17], we implement DFS by iteration and prove its termination and inductive principle separately. By separating implementation and specification, the DFS design is modular and easier to reason about.

Figure 3.5 presents our design. Similar to bounded iteration, the top-level entry is `iter`, which needs a bounded step `n`, a fixpoint `F` and a default value `g`. `iter` only calls `g` when `n` reaches zero, and otherwise recursively calls one more iteration of F. If F is terminating, we can prove that there must exist a final value and a bound n, such that for any bound k that is greater than or equal to n, iter always stops and generates the same final value. In other words, F must reach a fixpoint with less than n steps. In fact, the proof of the existence of n is erasable; the computation part of the proof provides a terminating algorithm for free, not requiring the bound step at runtime.

Figure 3.5 proves that the DFS must terminate, as shown by dfs_tmn, which is implemented by well-founded recursion over the number of unvisited nodes. Intuitively, this follows because after each iteration, the DFS visits more nodes. The invariant that the number of unvisited nodes decreases holds only for well-formed recursion states (wf_stk), which requires that all visited nodes and unprocessed nodes in frames must be in the CFG. We implemented dfs_tmn by Coq's Program Fixpoint, which allows programmers to leave *holes* for which Program Fixpoint automatically generates obligations to solve. Using dfs_tmn, dfs defines the final definition of DFS.

To reason about dfs, Figure 3.6 shows a well-founded inductive principle for dfs. In Module Ind, to prove that the final result has the property wf_po and the property wf_stack holds for all its intermediate states, we need to show that the initial state satisfies wf_stack, and that find_next preserves wf_stack when it can find a new available node, and produces a well-formed final result when no available nodes exist. With the inductive principle, we proved the following properties of DFS that are useful to establish the correctness of AC and CHK.

```
Variable (succs: ATree.t (list l)) (entry:l) (po:PostOrder).
Hypothesis Hdfs: dfs succs entry = po.
```

First of all, a non-entry node must have at least one predecessor that has a greater PO number than the node's. This is because 1) DFS must visit at least one predecessor of a node before visiting the node; 2) PO gives greater numbers to the nodes visited earlier:

```
Lemma dfs_order: forall l1 p1, l1 <> entry -> (PO_l2p po)!l1 = Some p1,
  exists l2, exists p2,
  In l2 ((make_preds succs)!!!l1) /\ (PO_l2p po)!l2 = Some p2 /\ p2 > p1.
(* Given succs, (make_preds succs) computes predecessors of each node. *)
```

Second, a node is PO-numbered iff the node is reachable:

```
Lemma dfs_reachable:forall l,(PO_l2p po)!l <> None <-> (entry,succs)->* l.
```

Moreover, different nodes do not have the same PO number.

```
Module Ind.
Section Ind.
  Variable (succs: ATree.t (list l)) (entry:l) (po:PostOrder).

  Hypothesis find_next__wf_stack: forall ... (Hwf: wf_stack visited po stk)
    (Heq: find_next succs visited po stk = inl (next, visited', po', stk')),
    wf_stack visited' po' stk'.

  Hypothesis wf_stack__find_next__wf_order: forall ...,
    (Hwf: wf_stack visited po1 stk)
    (Heq: find_next succs visited po1 stk = inr po2), wf_po po2.

  Hypothesis entry__wf_stack:
    wf_stack empty (mkPO 1 empty) (mkFr entry [(succs!!!entry)]).

  Lemma dfs_wf: dfs succs entry = po -> wf_po po.
End Ind.
End Ind.
```

Figure 3.6: Inductive principle of the DFS algorithm.

```
Lemma dfs_inj: forall l1 l2 p,
  (PO_l2p po)!l2 = Some p -> (PO_l2p po)!l1 = Some p -> l1 = l2.
```

### 3.2.2 Kildall's algorithm

Figure 3.7 summarizes the Kildall module used in the CompCert project. The module is parameterized by the following components: `NS` that provides the order to process nodes, and a lattice `L` that defines `top`, `bot`, equality (`eq`), least upper bound (`lub`) and order (`ge`) of the abstract domain of an analysis; `succs` that is a tree that maps a node to their successors; `transf` that is the transfer function of Kildall analysis; `inits` that initializes the analysis. Given the inputs, `state` records the iteration states that include `sin` that records analysis states of each node, and a work list `swrk` hat contains nodes to process.

`fixpoint` implements iterations by `Iter.iter`—bounded recursion with a maximal step number (`num`) [17]. `Iter.iter` is partial if an analysis does not stop after the maximal number of steps. A monotone analysis must reach its fixpoint after a fixed number of steps. Therefore, we can alway pick a large enough number of steps for a monotone analysis.

Initially Kildall's algorithm calls `start_st` to initialize iteration states. Nodes not in `inits` are initialized to be the bottom of `L`. Then `start_st` adds all nodes into the worklist and starts the loop. `step` defines

```
Module Kildall (NS: PNODE_SET) (L: LATTICE).
Section Kildall.
  Variable succs: PTree.t (list positive).

  Variable transf : positive -> L.t -> L.t.

  Variable inits: list (positive * L.t).

  Record state : Type := mkst { sin: PMap.t L.t; swrk: NS.t }.

  Definition start_st := mkst (start_state_in inits) (NS.init succs).

  Definition propagate_succ (out: L.t) (s: state) (n: positive) :=
    let oldl := s.(sin) !! n in
    let newl := L.lub oldl out in
    if L.eq newl oldl
    then mkst (PMap.set n newl s.(sin)) (NS.add n s.(swrk)) else s.

  Definition step (s: state): PMap.t L.t + state :=
    match NS.pick s.(swrk) with
    | None => inl s.(sin)
    | Some(n, rem) => inr (fold_left
                            (propagate_succ (transf n s.(sin) !! n))
                            (succs !!! n) (mkst s.(sin) rem))
  end.

  Variable num : positive.

  Definition fixpoint : option (PMap.t L.t):= Iter.iter step num start_st.
End Kildall.
End Kildall.
```

Figure 3.7: Kildall's algorithm.

the loop body. At `step`, Kildall's algorithm checks if there are still unprocessed nodes in the worklist. If the worklist is empty, the algorithm stops. Otherwise, `step` picks a node from the worklist in term of the order provided by `NS`, and then propagates its information (computed by `transf`) to all the node's successors by `propagate_succ`. In `propagate_succ`, the new value of a successor is `L.lub` of its old value and the propagated value from its predecessor. The algorithm only adds a successor into the worklist when its value is changed.

Kildall's algorithm satisfies the following properties:

```
Variable res: PMap.t L.t.
Hypothesis Hfix: fixpoint = Some res.
```

First of all, the worklist contains nodes that have unstable successors in the current state. Formally, each state st preserves the following invariant:

```
forall n, NS.In n st.(swrk) \/
  (forall s, In s (succs!!!n) -> L.ge st.(sin)!!s (transf n st.(sin)!!n)).
```

Each iteration may only remove the picked node n from the worklist. If none of n's successors' values are changed, no matter whether n belongs to its successors, n won't be added back to the worklist. Therefore, the above invariant holds. This invariant implies that when the analysis stops, all nodes hold the in-equations:

```
Lemma fixpoint_solution: forall s,
  In s (succs!!!n) -> L.ge res!!s (transf n res!!n).
```

The second property of Kildall's algorithm is *monotonicity*. At each iteration, the value of a successor of the picked node can only be updated from oldl to newl. Because newl is the least upper bound of oldl and out, newl is greater than or equal to oldl. Therefore, iteration states are always monotonic:

```
Lemma fixpoint_mono: incr (start_state_in inits) res.
```

where incr is a pointwise lift of L.ge for corresponding nodes. In particular, the final states must be greater than or equal to the initial states. When an iteration does not change states, no nodes will be added back to the worklist, but the size of worklist must decrease. Therefore, a monotonic analysis must reach its fixpoint with less than $N^2 * H$ steps where $N$ is the number of nodes; $H$ is the height of the lattice of the analysis [33].

### 3.2.3 The AC algorithm

AC instantiates Kildall with PN that picks nodes in reverse PO (by picking the maximal nodes from the worklist), and LDoms that defines the lattice of AC. Dominance analysis computes a set of strict dominators for each node. We represent the domain of LDoms by option (set l). The top and bot of LDoms are Some nil and None respectively. The least upper bound, order and equality of LDoms are lifted from set intersection, set inclusion, and set equality to option: None is smaller than Some x for any x. This design leads to better performance by providing shortcuts for operations on None. Note that using None as bot does not make the height of LDoms to be infinite, because any non-bot element can only contain nodes in the CFG, and the height of LDoms is $N$.

AC uses the following transfer function and initialization:

```
Definition transf l1 input := l1 {+} input.
Definition inits := [(e, LDoms.top)].
```

Initially AC sets the strict dominators of the entry to be empty, and other nodes' strict dominators to be all labels in the function. The algorithm will iteratively remove non-strict-dominators from the sets until the conditions below hold (by Lemma `fixpoint_mono` and Lemma `fixpoint_solution`):

```
(forall s, In s (succs!!!n) ->
           L.ge (st.(sin))!!s (n{+}(st.(sin))!!n)) /\ (st.(sin))!!e = {}.
```

which proves that AC satisfies `entry_sound` and `successors_sound`.

To show that the algorithm is complete, it is sufficient to show that each iteration state `st` preserves the following invariant:

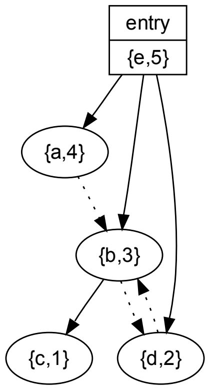
```
  forall n1 n2, ~ n1 `in` st.(sin)!!n2 -> ~ (e, succs) |= n1 >> n2.
```

In other words, AC only removes non-strict dominators. Initially, AC sets the entry's strict dominators to be empty. Because in a well-formed CFG, the entry has no predecessors, the invariant holds at the very beginning. At each iteration, suppose that we pick a node `n` and update one of its successors `s`. Consider a node `n'` not in `LDoms.lub st.(sin)!!s (n {+} st.(sin)!!n)`. If `n'` is not in `LDoms.lub st.(sin)!!s`, then `n'` does not strictly dominate `s` because `st` holds the invariant. If `n'` is not in `(n {+} st.(sin)!!n)`, then `n'` does not strictly dominate `n` because `st` holds the invariant. Appending the path from the entry to `n` that bypasses `n'` with the edge from `n` to `s` leads to a path from the entry to `s` that bypasses `n'`. Therefore, `n'` does not strictly dominate `s`, either.

## 3.3 Extension: the Cooper-Harvey-Kennedy Algorithm

The CHK algorithm is based on the following observation: when AC processes nodes in a reversed post-order (PO), if we represent the set of strict dominators in a list, and always add a newly discovered strict dominator at the head of the list (on the left in Figure 3.8), the list must be sorted by PO. Figure 3.8 (on the right) shows the execution of the algorithm for the CFG in Figure 3.3.

Because lists of strict dominators are always sorted, we can implement the set intersection (`lub`) and the set comparison (`eq`) of two sorted lists by traversing the two lists only once. Moreover, the algorithm only calls `eq` after `lub`. Therefore, we can group `lub` and `eq` into `LDoms.lub` together. The following defines a `merge` function used by `LDoms.lub` that intersects two sorted lists and returns whether the final result equals to the left one:



| Nodes | sin | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 5 | [] | [] | [] | [] | [] | [] | [] | [] | [] |
| 4 | . | [5] | [5] | [5] | [5] | [5] | [5] | [5] | [5] |
| 3 | . | . | [45] | [45] | [45] | [5] | [5] | [5] | [5] |
| 2 | . | . | . | [345] | [345] | [345] | [35] | [35] | [35] |
| 1 | . | [5] | [5] | [5] | [5] | [5] | [5] | [5] | [5] |
| swrk | [54321] | [4321] | [321] | [21] | [1] | [3] | [21] | [1] | [] |

Figure 3.8: The dominator trees (left) and the execution of CHK (right).

```
Program Fixpoint merge (l1 l2: list positive) (acc:list positive * bool)
  {measure (length l1 + length l2)}: (list positive * bool) :=
  let '(rl, changed) := acc in
  match l1, l2 with
  | p1::l1', p2::l2' =>
      match (Pcompare p1 p2 Eq) with
      | Eq => merge l1' l2' (p1::rl, changed)
      | Lt => merge l1' l2 (rl, true)
      | Gt => merge l1 l2' (rl, changed)
      end
  | nil, _ => acc
  | _::_, nil => (rl, true)
  end.
(* (Pcompare p1 p2 Eq) returns whether p1 = p2, p1 < p2 or p1 > p2. *)
```

### 3.3.1 Correctness

To show that CHK is still correct, it is sufficient to show that all lists are well-sorted at each iteration, which ensures that the above `merge` correctly implements intersection and comparison. First, if a node with number `n` still maps to `bot`, the worklist must contain one of its predecessors that has a greater number.

```
forall n, in_cfg n succs -> (st.(sin))!!n = None ->
  exists p, In p ((make_preds succs)!!!n) /\ p > n /\ PN.In p st.(st_wrk).
(* in_cfg checks if a node is in CFG. *)
```

This invariant holds in the beginning because all nodes are in the worklist. At each iteration, the invariant implies that the picked node `n` with the maximal number in `st.(st_wrk)` is not `bot`. Suppose it is `bot`, there cannot be any node with greater number in the worklist. This property ensures that after each iteration, the successors of `n` cannot be `bot`, and that the new nodes added into the worklist cannot be `bot`, because they must be those successors. Therefore, the predecessors of the remaining `bot` nodes still in the worklist cannot be `n`. Since only `n` is removed, the rest of the `bot` nodes still hold the above invariant.

In the algorithm, a node's value is changed from `bot` to non-`bot` when one of its non-`bot` predecessors is processed. With the above invariant, we know that the predecessor must be of larger number. Once a node turns to be non-`bot`, no new elements will be added in its set. Therefore, this implies that, at each iteration, if the value of a node is not `bot`, then all its candidate strict dominators must be larger than the node:

```
forall n sdms, (st.(sin))!!n = Some sdms -> Forall (Plt n) sdms.
(* Plt is the less-than of positive. *)
```

Moreover, a node `n` is considered as a candidate of strict dominators originally by `tranf` that always cons `n` at the head of `(st.(sin))!!n`. Therefore, we proved that the non-`bot` value of a node is always sorted:

```
forall n sdms, (st.(sin))!!n = Some sdms -> Sorted Plt (n::sdms).
```

## 3.4 Constructing Dominator Trees

In practice, compilers construct dominator trees from dominators, and analyze or optimize programs by recursion on dominator trees.

**Definition 3.**

- *A block $l_1$ is an* ***immediate dominator*** *of a block $l_2$, written $G \models l_1 \ggg l_2$, if $G \models l_1 \gg l_2$ and* $(\forall G \models l_3 \gg l_2, G \models l_3 \gg= l_1)$.
- *A tree is called a* ***dominator tree*** *of $G$ if the tree has an edge from $l$ to $l'$ iff $G \models l \ggg l'$.*

Figure 3.8 shows the dominator tree of the CFG in Figure 3.3. In Figure 3.8, solid edges represent tree edges, and dotted edges represent non-tree but CFG edges.

```
Inductive DTree : Set :=
| DT_node : l -> DTrees -> DTree
with DTrees : Set :=
| DT_nil : DTrees
| DT_cons : l -> DTrees -> DTrees.

Variable (f: function) (entry:l).

Inductive wf_dtree : DTree -> Prop :=
| Wf_DT_node : forall l0 dts (Hrd: f |= entry ->* l0)
               (Hnotin: ~ l0 `in` (dtrees_dom dts)) (Hdisj: disjoint_dtrees dts)
               (Hidom: forall_children idom l0 dts) (Hwfdts: wf_dtrees dts),
               wf_dtree (DT_node l0 dts)
  (* (dtrees_dom dts) returns all labels in dts.                          *)
  (* (disjoint_dtrees dts) ensures that labels of dts are disjointed.     *)
  (* (forall_children idom l0 dts)) checks that l0 immediate-dominates all *)
  (*   roots of dts.                                                       *)
with wf_dtrees : DTrees -> Prop :=
| Wf_DT_nil : wf_dtrees DT_nil
| Wf_DT_cons : forall dt dts (Hwfdt: wf_dtree dt) (Hwfdts: wf_dtrees dts),
               wf_dtrees (DT_cons dt dts).
```

Figure 3.9: The definition and well-formedness of dominator trees.

Formally, we define dominator trees in Figure 3.9 that has the inductive **well-formed** (`wf_dtree`) property with which we can reason about recursion on dominator trees: given a tree node $l$, 1) $l$ is reachable; 2) $l$ is different from all labels in $l$'s descendants; 3) labels of $l$'s subtrees are disjointed; 4) $l$ immediate-dominates its children; 5) $l$'s subtrees are well-formed.

Consider the final analysis results of CHK in Figure 3.8, we can see that for each node, its list of strict dominators exactly presents a path from root to the node on the dominator tree. Therefore, we can construct a dominator tree by merging the paths. We proved that the algorithm correctly constructs a well-formed dominator tree (See our code). For the sake of space, we only present that each tree edge represents $\ggg$ by showing that for any node $l$ in the final state, the list of $l$'s dominators must be sorted by $\ggg$.

We first show that the list is sorted by $\gg$. Consider two adjacent nodes in the list, $l_1$ and $l_2$, such that $l_1 < l_2$. Because of soundness, $G \models l_1 \gg= l$ and $G \models l_2 \gg= l$. By Lemma 5, $G \models l_2 \gg l_1 \vee G \models l_1 \gg l_2$. Suppose $G \models l_1 \gg l_2$, by completeness, $l_1$ must be in the strict dominators computed for $l_2$, and therefore, be greater than $l_2$. This is a contradiction. Then, we prove that the list is sorted by $\ggg$. Suppose $G \models l_3 \gg l_1$. By Lemma 1 and Lemma 2, $G \models l_3 \gg l$. By completeness, $l_3$ must be in the list. We have two cases:

1. $l_3 \geq l_2$: Because the list is sorted by $\gg$, $G \models l_3 \gg= l_2$.

2. $l_3 \leq l_1$: Similarly, $G \models l_1 \gg= l_3$. This is a contradiction by Lemma 4.

## 3.5 Dominance Frontier

Another application of computing dominators is the calculation of dominance frontiers that has applications to SSA construction algorithms, computing control dependence, and *etc.*

Cytron *et al.* define the dominance frontier of a node, *b*, as:

> ... the set of all CFG nodes, *y*, such that *b* dominates a predecessor of *y* but does not strictly dominate *y* [28].

They propose finding the dominance frontier set for each node in a two step manner. They begin by walking over the dominator tree in a bottom-up traversal. At each node, *b*, they add to *b*'s dominance-frontier set any CFG successors not dominated by *b*. They then traverse the dominance-frontier sets of *b*'s dominator-tree children each member of these frontiers that is not dominated by *b* is copied into *b*'s dominance frontier.

We follow an algorithm designed by Cooper, Harvey and Kennedy [24] that approaches the problem from the opposite direction, and tends to run faster than Cytron *et al.*'s algorithm in practice. The algorithm is based on three observations. First, nodes in a dominance frontier represent join points in the graph, nodes into which control flows from multiple predecessors. Second, the predecessors of any join point, *j*, must have *j* in their respective dominance-frontier sets, unless the predecessor dominates *j*. This is a direct result of the definition of dominance frontiers, above. Finally, the dominators of *j*'s predecessors must themselves have *j* in their dominance-frontier sets unless they also dominate *j*.

These observations lead to a simple algorithm. First, we identify each join point, *j*—any node with more than one incoming edge is a join point. We then examine each predecessor, *p*, of *j* and walk up the dominator tree starting at *p*. We stop the walk when we reach *j*'s immediate dominator—*j* is in the dominance frontier of each of the nodes in the walk, except for *j*'s immediate dominator. Intuitively, all of the rest of *j*'s dominators are shared by *j*'s predecessors as well. Since they dominate *j*, they will not have *j* in their dominance frontiers.

As shown previously [24], this approach tends to run faster than Cytron *et al.*.'s algorithm in practice, almost certainly for two reasons. First, the iterative algorithm has already built the dominator tree. Second,

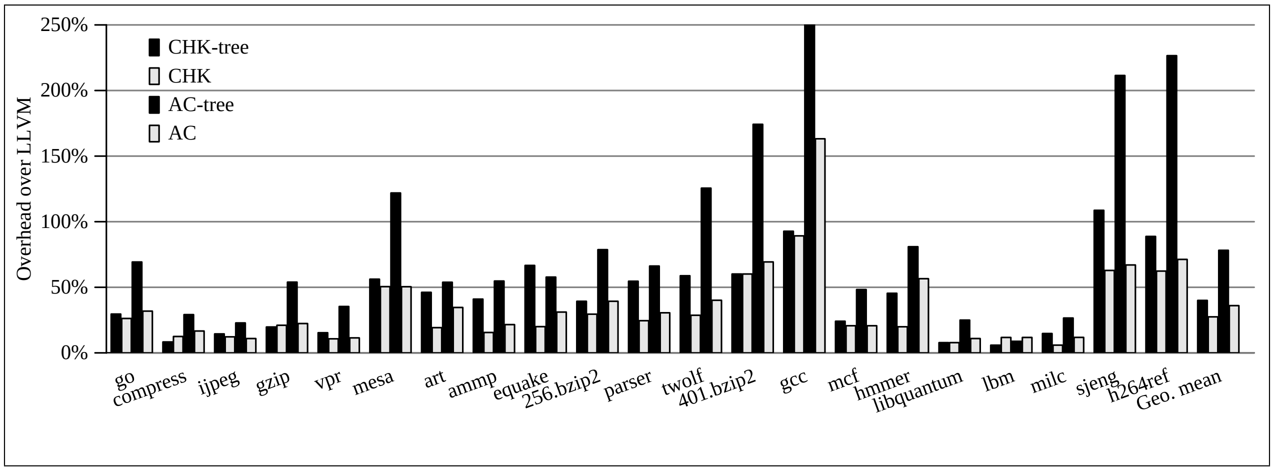


Figure 3.10: Analysis overhead over LLVM's dominance analysis for our extracted analysis.

the algorithm uses no more comparisons than are strictly necessary. Section 8.5.2 will revisit the implementation of the algorithm.

## 3.6 Performance Evaluation

As we discussed, computing dominators is crucial in SSA-based compilers. Therefore, we use the Coq extraction to obtain a certified implementation of AC and CHK and evaluate the performance of the resultant code on a 1.73 GHz Intel Core i7 processor with 8 GB memory running benchmarks selected from the SPEC CPU benchmark suite that consist of over 873k lines of C source code.

Figure 3.10 reports the analysis time overhead (smaller is better) over the C++ version of LLVM dominance analysis (which uses LT) baseline. LT only generates dominator trees. Given a dominator tree, the strict dominators of a tree node are all the node's ancestors. The second left bar of each group shows the overhead of CHK, which provides an average overhead of 27%. The right-most bar of each group is the overhead of AC, which provides 36% on average.

To study the asymptotic complexity, Table 3.1 shows the result of graphs that elicit the worst-case behavior used previously [31]. On average, CHK is 86 times slower than LT. The '_' indicates that the running time is too long to collect. For the testcases on which AC stops, AC is 226 times slower than LT.

The results of CHK match earlier experiments [24, 31]: in common cases, CHK runs nearly as fast as LT. For programs with reducible CFGs, a forward iteration analysis in reverse PO will halt in no more than size passes [33], and most CFGs of the common benchmarks are reducible. The worst-case tests contain

| Instance | | | Analysis Times (s) | | | | |
|---|---|---|---|---|---|---|---|
| Name | Vertices | Edges | LT | CHK | CHK-tree | AC | AC-tree |
| idfsquad | 6002 | 10000 | 0.08 | 10.54 | 24.87 | – | – |
| ibfsquad | 4001 | 6001 | 0.14 | 11.38 | 13.16 | 12.43 | 30.00 |
| itworst | 2553 | 5095 | 0.14 | 8.47 | 11.22 | 19.16 | 69.72 |
| sncaworst | 3998 | 3096 | 0.19 | 17.03 | 32.08 | 205.07 | 740.53 |

Table 3.1: Worst-case behavior.

huge irreducible CFGs. Different from these experiments, AC does not provide large overhead, because we use `None` to represent `bot`, which provides shortcuts for set operations.

As shown in Section 3.4, CHK computes dominator trees implicitly, while AC needs additional costs to create dominator trees. Figure 3.10 and Table 3.1 also report the performance of the dominator tree construction. CHK-tree stands for the algorithm that first computes dominators by CHK and then runs the tree construction defined in Section 3.4. AC-tree stands for the algorithm that first computes dominators by AC, sorts strict dominators for each node, and then runs the same tree construction. For common programs, on average, CHK-tree provides an overhead 40% over the baseline; AC-tree provides an overhead 78% over the baseline. Note that in Figure 3.10 the testcase gcc's overhead for AC-tree is 361%. The additional overhead of AC-tree is from its sorting algorithm. For worst-case programs, on average, CHK-tree is 104 times slower than LT. For the testcases on which AC-tree stops, on average, AC-tree is 738 times slower than LT.

These results match the previous evaluation [24] and indicate that CHK makes a good trade-off between simplicity and efficiency.

# Chapter 4

# The Semantics of Vminus

Given the formalism in Chapter 3, this chapter presents the semantics of Vminus. Chapter 6 extends the semantics for the full Vellvm.

## 4.1 Dynamic Semantics

The operational semantics rules in Figure 4.1 are parameterized by the top-level function $f$, and relate evaluation frames $\sigma$ before and after an evaluation step. An evaluation frame keeps track of the integer values $v$ bound to local temporaries $r$ in $\delta$ and current program counter. We also use $\sigma.pc$ and $\sigma.\delta$ to denote the program counter and locals of $\sigma$ respectively. Because Vminus has no function calls, the rules ignore program traces. This simplification does not affect the essence of the proof techniques. Section 6.4 shows the full Vellvm semantics with traces.

Instruction positions are denoted by program counters $pc$: $l.i$ indicates the $i$-th command in the block $l$; $l.\mathbf{t}$ indicates the terminator of the block $l$. We write $f[pc] = \lfloor insn \rfloor$ if some $insn$ is at the program counter $pc$ of function $f$. We also use $l.(i+1)$ to denote the next program counter of $l.i$. When $l.i$ is the last command of block $l$, $l.(i+1) = l.\mathbf{t}$. To simplify presentation of the operational semantics, we use $l, \overline{c}, tmn$ to "unpack" the instructions at a program counter in function $f$. Here, $l$ is the current block, $\overline{c}$ and $tmn$ are the instructions of $l$ that are not executed yet. "block & offset" specification is equivalent to the "continuation commands" representation. To streamline some presentations, we also use temporaries or ghost identifiers to represent program counters.

$$\begin{array}{llllll} \text{Values } v & ::= & Int & \text{Locals } \delta & ::= & r \mapsto v \\ \text{Frames } \sigma & ::= & (pc, \delta) & \text{Prog Counters } pc & ::= & l.i \mid l.\mathbf{t} \end{array}$$

$$\frac{\begin{array}{c} [\![val]\!]_\delta = \lfloor v \rfloor \quad l_3 = (v ? l_1 : l_2) \\ f[l_3] = \lfloor (l_3\, \overline{\phi_3}\, \overline{c_3}\, tmn_3) \rfloor \quad [\![\overline{\phi_3}]\!]^l_\delta = \lfloor \delta' \rfloor \end{array}}{f \vdash (l, \emptyset, \mathbf{br}\, val\, l_1\, l_2, \delta) \longrightarrow (l_3, \overline{c_3}, tmn_3, \delta')} \quad \text{E\_BR}$$

$$\frac{\begin{array}{c} [\![val_1]\!]_\delta = \lfloor v_1 \rfloor \quad [\![val_2]\!]_\delta = \lfloor v_2 \rfloor \\ c = r := val_1\, bop\, val_2 \quad \mathbf{eval}\,(bop, v_1, v_2) = v_3 \end{array}}{f \vdash (l, (c, \overline{c}), tmn, \delta) \longrightarrow (l, \overline{c}, tmn, \delta\{v_3/r\})} \quad \text{E\_BOP}$$

Figure 4.1: Operational Semantics of Vminus (excerpt)

Most of the Vminus commands have straight-forward interpretation. The arithmetic and logic instructions are all unsurprising (as shown in rule E_BOP)—the $[\![val]\!]_\delta$ function computes a value from the local state $\delta$ and *val*, looking up the meanings of variables in the local state as needed; **eval** implements arithmetic and logic operations. We use $[\![rhs]\!]_\delta$ to denote evaluating the right-hand-side *rhs* in the state $\delta$.

There is one wrinkle in specifying the operational semantics when compared to a standard environment-passing call-by-value language. All of the $\phi$ instructions for a block must be executed atomically and with respect to the "old" local value mapping due to the possibility of self loops and dependencies among the $\phi$ nodes. For example the well-formed code fragment below has a circular dependency between $r_1$ and $r_2$.

$$\begin{array}{ll} l_0: & \cdots \\ l_1: & r_1 = \mathbf{phi}\,\mathbf{int}[r_2, l_1][0, l_0] \\ & r_2 = \mathbf{phi}\,\mathbf{int}[r_1, l_1][1, l_0] \\ & r_3 := r_1 \ = \ r_2 \\ & \mathbf{br}\, r_3\, l_1\, l_2 \\ l_2: & \cdots \end{array}$$

Although front-ends usually do not generate codes with the circular dependency, optimizations, such as copy propagation, may produce the above code [16]. In the code fragment, if control enters this block from $l_0$, $r_1$ will map to 0 and $r_2$ to 1, which causes the conditional branch to fail, jumping back to the label $l_1$. The new values of $r_1$ and $r_2$ should be 1 and 0, and not 1 and 1 as might be computed if they were handled sequentially. This atomic update of the local state, similar to "parallel assignment", is handled by the $[\![\overline{\phi_3}]\!]^l_\delta$ function as shown in rule E_BR.

## 4.2 Dominance Analysis

Dominance analysis plays an important role in the type system. To check that a program is in SSA form, we need to extend domination relations from the block-level (Chapter 3) to the instruction-level. Instruction positions are denoted by program counters $pc$. We write $f[pc] = \lfloor insn \rfloor$ if $insn$ is at $pc$ of $f$.

**Definition 4** (Instruction-level domination)**.**

- $val\,\mathbf{uses}\,r \triangleq val = r$.
- $insn\,\mathbf{uses}\,r \triangleq \exists val.\, val\,\mathbf{uses}\,r \wedge val$ *is an operand of insn.*
- *A variable $r$ is defined at a program counter $pc$ of function $f$, written $f\,\mathbf{defines}\,r\ @\ pc$ if and only if $f[pc] = \lfloor insn \rfloor$ and $r$ is the left-hand side of insn.*
- *In function $f$, $pc_1$* ***strictly dominates*** *$pc_2$, written $f \models pc_1 \gg pc_2$, if $pc_1$ and $pc_2$ are at distinct blocks $l_1$ and $l_2$ respectively and $f \models l_1 \gg l_2$; if $pc_1$ and $pc_2$ are in the same block, and $pc_1$ appears earlier than $pc_2$.*
- $\mathbf{sdom}_f(pc)$ *is the set of variables strictly dominating pc:*

$$\mathbf{sdom}_f(pc) = \{r \mid f\,\mathbf{defines}\,r\ @\ pc' \text{ and } f \models pc' \gg pc\}$$

We prove the following lemmas about the instruction-level domination relations, which are needed to establish the SSA-based program properties in the following sections.

**Lemma 6** (Domination is transitive)**.** *If $f \vdash pc_1 \gg pc_2$ and $f \vdash pc_2 \gg pc_3$, then $f \vdash pc_1 \gg pc_3$.*

**Lemma 7** (Strict domination is acyclic)**.** *If $f \rightsquigarrow pc$ (pc is reachable), then $\neg f \vdash pc \gg pc$.*

By Lemma 6, $\mathbf{sdom}_f(pc)$ has the following properties:

**Lemma 8** (sdom step)**.**

1. *If $l.i$ and $l.(i+1)$ are valid program counters of $f$, then $\mathbf{sdom}_f(l.(i+1)) = \mathbf{sdom}_f(l.i) \cup \{r\}$ where $f\,\mathbf{defines}\,r\ @\ l.i$.*
2. *If $l.\mathbf{t}$ and $l'.0$ are valid program counters of $f$, and $l'$ is a successor of $l$, then $\mathbf{sdom}_f(l'.0) - \mathbf{defs}\,(\overline{\phi}) \subseteq \mathbf{sdom}_f(l.\mathbf{t})$ where $\overline{\phi}$ are from the block $l'$ and $\mathbf{defs}\,(\overline{\phi})$ denotes all variables defined by $\overline{\phi}$.*

$\boxed{f \vdash_{\gg} \psi \mathbin{@} pc}$

$$\frac{\forall r.(\psi\ \mathbf{uses}\ r \Longrightarrow r \in \mathbf{sdom}_f(pc))}{f \vdash_{\gg} \psi \mathbin{@} pc}\ \text{NONPHI}$$

$\boxed{f, l \vdash_{\gg} \phi}$

$$\frac{\begin{array}{c}\mathbf{uniq}\,(\overline{l_j}^{\,j}) \quad \overline{l_j}^{\,j} = \mathbf{preds}\,(f, l) \\ \overline{\forall r_j.(val_j\ \mathbf{uses}\ r_j \Longrightarrow r_j \in \mathbf{sdom}_f(l_j.\mathbf{t}))}^{\,j} \quad \mathbf{len}\,(\overline{[val_j, l_j]}^{\,j}) > 0 \quad \overline{f \vdash val_j : typ}^{\,j}\end{array}}{f, l \vdash_{\gg} r = \mathbf{phi}\, typ\, \overline{[val_j, l_j]}^{\,j}}\ \text{PHI}$$

$\boxed{f \vdash \psi}$

$$\frac{f \vdash val_1 : \mathbf{int} \quad f \vdash val_2 : \mathbf{int}}{f \vdash r := val_1\, bop\, val_2}\ \text{WF\_BOP}$$

$$\frac{f \vdash val : \mathbf{int} \quad f[l_1] = \lfloor b_1 \rfloor \quad f[l_2] = \lfloor b_2 \rfloor}{f \vdash \mathbf{br}\, val\, l_1\, l_2}\ \text{WF\_BR}$$

$$\frac{f \vdash val : typ}{f \vdash \mathbf{ret}\, typ\, val}\ \text{WF\_RET}$$

$\boxed{f \vdash \psi \mathbin{@} pc}$

$$\frac{f \vdash_{\gg} \psi \mathbin{@} pc \quad f \vdash \psi}{f \vdash \psi \mathbin{@} pc}\ \text{WF\_NONPHI}$$

$\boxed{f \vdash b}$

$$\frac{f \rightsquigarrow l \Longrightarrow (\overline{f, l \vdash_{\gg} \phi_j}^{\,j} \wedge \overline{f \vdash c_i \mathbin{@} l.i}^{\,i} \wedge f \vdash tmn \mathbin{@} (l.\mathbf{t}))}{f \vdash l\, \overline{\phi_j}^{\,j}\, \overline{c_i}^{\,i}\, tmn}\ \text{WF\_B}$$

$\boxed{\vdash f}$

$$\frac{\mathbf{uniq}\,(\mathbf{defs}\,(f)) \quad \mathbf{uniq}\,(\mathbf{labels}\,(f)) \quad f = \mathbf{fun}\,\{\overline{b_j}^{\,j}\} \quad \overline{f \vdash b_j}^{\,j} \quad \mathbf{wf\_entry}\, f}{\vdash f}\ \text{WF\_F}$$

Figure 4.2: Static Semantics of Vminus (excerpt)

## 4.3 Static Semantics

Vminus requires a program satisfy certain invariants to be considered well formed: every variable in the top-level function must dominate all its uses and be assigned exactly once statically. At a minimum, any reasonable Vminus transformation must preserve these invariants; together they imply that the program is in SSA form [28].

Figure 4.2 shows the judgments to check the SSA invariants with respect to the control-flow graph and program points of the function $f$.

Rule WF_F ensures that variables **defs** $(f)$ defined in the top function must be unique, which enforces the single-assignment part of the SSA property; additionally all block labels **labels** $(f)$ in the function must also be unique for a well-formed control-flow graph; the entry block has no predecessors (**wf_entry**$f$).

Rule WF_B checks that all instructions in reachable blocks (written $f \leadsto l$) satisfy the SSA domination invariant. Because unreachable blocks have no effects at runtime, the rule does not check them. Rule NONPHI ensures that a $\psi$ at $pc$ must be strictly dominated by the definitions of all variables used by $\psi$; the rule PHI ensures that the number of incoming values is not zero, that all incoming labels are unique, and that the current block's predecessors is the same as the set of incoming gables. If an incoming value $val_j$ from a predecessor block $l_j$ uses a variable $r_j$ at $pc_j$, then $pc_j$ must strictly dominate the terminator of $l_j$. Importantly, this rule allows "cyclic" uses of SSA variables of the kind used in the example above (Section 4.1).

Given the semantics in this chapter, the next chapter presents the proof techniques for reasoning about SSA-based program properties and transformations of Vminus.

# Chapter 5

# Proof Techniques for SSA

This section describes the proof techniques we have developed for formalizing properties of SSA-style intermediate representations. To most clearly articulate the approach, we present the results using Vminus (see Chapter 4).

The key idea of the technique is to generalize the invariant used for Vminus's preservation lemma for proving safety to other predicates that are also shown to be invariants of the operational semantics. Crucially, these predicates all share the same form, which only constrains variable definitions that *strictly dominate* the current program counter. Because Vminus is such a stripped-down language, the relevant lemmas are relatively straightforward to establish; Chapter 8 shows how to scale the proof technique to the full Vellvm model of LLVM to verify the `mem2reg` pass.

Instances of this idea are found in the literature (see, for example, Menon, *et al.* [48]), and related proof techniques have been recently used in the CompCertSSA [14] project, but as we explain in Chapter 10, our results are more general: we provide proof techniques applicable to many SSA-based optimizations and transformations.

The remainder of this section first proves safety (which in this context simply amounts to showing that all variables are well-scoped). We then show how to generalize the safety invariant to a form that is useful for proving program transformations correct and demonstrate its applicability to a number of standard optimizations.

We mechanically verified all the claims in this chapter for Vminus in Coq.[1]

[1] Annotated Coq source available at `http://www.cis.upenn.edu/~stevez/vellvm/`.

## 5.1 Safety of Vminus

There are two ways that a Vminus program might get stuck. First, it might try to jump to an undefined label, but this property is ruled out statically by WF_BR. Second, it might try to access a variable whose value is not defined in $\delta$. We can prove that this second case never happens by establishing the following safety theorem:

**Theorem 9** (Safety). *If $\vdash f$ and $f \vdash (l.0, \emptyset) \longrightarrow^* \sigma$, then $\sigma$ is not stuck. (Here, $l$ is the entry block of function $f$ and $\emptyset$ denotes an empty mapping for identifiers.)*

The proof takes the standard form using preservation and progress lemmas with the invariant for frames shown below:

$$\frac{pc \in f \quad \forall r.(r \in \mathbf{sdom}_f(pc) \Longrightarrow \exists v.\delta[r] = \lfloor v \rfloor)}{f \vdash (pc, \delta)} \text{ WF\_FR}$$

This is similar to the predicate used in prior work for verifying the type safety of an SSA-based language [48]. The invariant WF_FR shows that a frame $(pc, \delta)$ is well-formed if every definition that strictly dominates $pc$ is defined in $\delta$. The initial program state satisfies this invariant trivially:

**Lemma 10** (Initial State). *If $\vdash f$ then $f \vdash (l.0, \emptyset)$, where $l$ is the entry block of $f$.*

The preservation and progress lemmas are straightforward—but note that they crucially rely on the interplay between the invariant on $\delta$ "projected" onto $\mathbf{sdom}_f(pc)$ (Lemma 8), and the PHI and NONPHI rules of the static semantics.

**Lemma 11** (Preservation). *If $\vdash f$, $f \vdash \sigma$ and $f \vdash \sigma \longrightarrow \sigma'$, then $f \vdash \sigma'$.*

*Proof.* The proof proceeds by case analysis on the reduction rule. At the E_BOP case: Let $\sigma = (l.i, \delta)$, $\sigma' = (l.(i+1), \delta\{v_3/r\})$, and $f[l.i] = \lfloor r := val_1\, bop\, val_2 \rfloor$. The conclusion holds by Lemma 8.

At the E_BR case: Let $\sigma = (l.\mathbf{t}, \delta)$, $\sigma' = (l_3.0, \delta')$, $f[l.\mathbf{t}] = \lfloor \mathbf{br}\, val\, l_1\, l_2 \rfloor$, $[\![\overline{\phi}_3]\!]^l_\delta = \lfloor \delta' \rfloor$, and $\overline{\phi}_3$ is from the block $l_3$. Suppose $r \in \mathbf{sdom}_f(l_3.0)$. If $r \in \mathbf{defs}(\overline{\phi}_3)$, then $r$ must be defined in $\delta'$ by the definition of $[\![\,]\!]^l_{\overline{\phi}_3}$. Otherwise, if $\neg r \in \mathbf{defs}(\overline{\phi}_3)$, the conclusion holds by Lemma 8. $\square$

**Lemma 12** (Progress). *If $\vdash f$, $f \vdash \sigma$, then $\sigma$ is not stuck.*

*Proof.* Assume that $\sigma = (pc, \delta)$. Since $pc \in f$, then $\exists insn.\, f[pc] = \lfloor insn \rfloor$. The proof proceeds by case analysis on the *insn*. At the case when $insn = r := val_1\, bop\, val_2$: The rule NONPHI ensures that the definitions of the variables used by $val_1$ and $val_2$ strictly dominate $pc$, so are in $\mathbf{sdom}_f(pc)$. Therefore, $\sigma$ is not stuck.

At the case when $insn = \mathbf{br}\, val\, l_1\, l_2$: First, the rule NONPHI ensures that the *val* must use the variable defined in $\mathbf{sdom}_f(pc)$. Therefore, $[\![val]\!]_\delta = \lfloor v \rfloor$. Suppose $l_3 = (v?l_1 : l_2)$, $f[l_3] = \lfloor (l_3\, \overline{\phi}_3\, \overline{c}_3\, tmn_3) \rfloor$, and *insn* is at block $l_j$. The rule PHI ensures that the definitions of the $j$-th incoming variables dominate $l_j.\mathbf{t}$, so are in $\mathbf{sdom}_f(pc)$. Therefore, $[\![\overline{\phi}_3]\!]^l_\delta = \lfloor \delta' \rfloor$.

At the case when $insn = \mathbf{ret}\, typ\, val$: The program terminates. □

## 5.2 Generalizing Safety to Other SSA Invariants

The main feature of the preservation proof, Lemma 11, is that the constraint on $\mathbf{sdom}_f(pc)$ is an invariant of the operational semantics. But—and this is a key observation—we can parameterize rule WF_FR by a predicate $P$, which is an arbitrary proposition about functions and frames:

$$\boxed{\dfrac{\sigma.pc \in f \qquad P\, f\, (\sigma|_f)}{f, P \vdash \sigma}\ \text{GWF\_FR}}$$

Here, $\sigma|_f$ is $(\sigma.pc, (\sigma.\delta)|_{(\mathbf{sdom}_f(\sigma.pc))})$ and we write $(\delta|_R)[r] = \lfloor v \rfloor$ iff $r \in R$ and $\delta[r] = \lfloor v \rfloor$ and observe that $\mathbf{dom}\,(\delta|_R) = R$. These restrictions say that we don't need to consider *all* variables: Intuitively, because SSA invariants are based on dominance properties, when reasoning about a program state we need consider only the variable definitions that strictly dominate the program counter in a given state.

For proving Theorem 9, we instantiated $P$ to be:

$$P_{\text{safety}} \triangleq \lambda f.\, \lambda \sigma.\, \forall r. r \in \mathbf{dom}\,(\sigma.\delta) \Longrightarrow \exists v.(\sigma.\delta)[r] = \lfloor v \rfloor$$

For safety, it is enough to show that each variable in the domination set is well defined at its use. To prove program transformations correct, we instantiate $P$ with a different predicate, $P_{\text{sem}}$, that relates the syntactic definition of a variable with the semantic value:

$$\lambda f.\, \lambda \sigma.\, \forall r. f[r] = \lfloor rhs \rfloor \Longrightarrow (\sigma.\delta)[r] \neq \mathbf{none} \Longrightarrow (\sigma.\delta)[r] = [\![rhs]\!]_{(\sigma.\delta)}$$

This predicate ensures that if a definition $r$ is in scope, the value of $r$ must equal to the value to which the right-hand-side of its definition evaluates.

Just as we proved preservation for $P_{\text{safety}}$, we can also prove preservation for $P_{\text{sem}}$ (using Lemma 4):

**Theorem 13.** *If* $\vdash f$ *and* $f, P_{\text{sem}} \vdash \sigma$ *and* $f \vdash \sigma \longrightarrow \sigma'$*, then* $f, P_{\text{sem}} \vdash \sigma'$.

**P**roof (sketch): Suppose a command $r := rhs$ is defined at a program counter $pc_1$. The NONPHI rule ensures that all variables used by *rhs* must strictly dominate $pc_1$. Because strict domination relation is acyclic (Lemma 4), at any program counter $pc_2$ that $pc_1$ strictly dominates, the program cannot define $r$ and any variable used by *rhs*. In other words, the values of $r$ and *rhs* are not changed between $pc_1$ and $pc_2$. The result follows immediately. □

Theorem 13 shows the dynamic property of an SSA variable: the value of $r$ is invariant in any execution path that its definition strictly dominates. As we show next, Theorem 13 can be used to justify the correctness of many SSA-based transformations. Instantiating $P$ with other predicates can also be useful—Section 8.3 shows how.

## 5.3 The Correctness of SSA-based Transformations

Consider again the example code transformation from Figure 2.2. It, and many other SSA-based optimizations, can be defined by using a combination of simpler transformations: deleting an unused definition, substituting a constant expression for a variable, substituting one variable by another, or moving variable definitions. Each such transformation is subject to the SSA constraints—for example, we can't move a definition later than one of its uses—and each transformation preserves the SSA invariants. By pipelining these basic transformations, we can define more sophisticated SSA-based program transformations whose correctness is established by the composition of the proofs for the basic transformations.

In general, an SSA-based transformation from $f$ to $f'$ is *correct* if it preserves both well-formedness and program behavior.

1. Preserving well-formedness: if $\vdash f$, then $\vdash f'$.

2. Program refinement: if $\vdash f$, then $f \supseteq f'$.

Here, behaviors of a Vminus program include whether the program terminates, and the returned value if it does (see Section 2.1).

Each of the basic transformations mentioned above can be proved correct by using Theorem 13. Here we present only the correctness of variable substitution (although we proved correct all the mentioned transformations in our Coq development). Chapter 8 shows how to extend the transformations to implement memory-aware optimizations in the full Vellvm.

**Variable substitution** Consider the step of the program transformation from Figure 2.2 in which the use of $r_8$ on the last line is replaced by $r_4$ (this is valid only after hoisting the definition of $r_4$ so that it is in scope). This transformation is correct because both $r_4$ and $r_8$ denote the same value, and the definition of $r_4$ (after hoisting) strictly dominates the definition of $r_8$. In Figure 2.2, it is enough to do *redundant variable elimination*—this optimization lets us replace one variable by another when their definitions are syntactically equal; other optimizations, such as *global value numbering*, allow a coarser, more semantic, equality to be used. Proving them correct follows the same basic pattern as the proof shown below.

**Definition 5** (Redundant Variable)**.** *In a function $f$, a variable $r_2$ is **redundant** with variable $r_1$ if:*

1. $f\,\textbf{defines}\,r_1 \;@\; pc_1$, $f\,\textbf{defines}\,r_2 \;@\; pc_2$ *and* $f \models pc_1 \gg pc_2$

2. $f[pc_1] = \lfloor c_1 \rfloor$, $f[pc_1] = \lfloor c_2 \rfloor$ *and $c_1$ and $c_2$ have syntactically equal right-hand-sides.*

We would like to prove that eliminating a redundant variable is correct, and therefore must relate a program $f$ with $f\{r_1/r_2\}$, in which all uses of $r_2$ have been substituted by $r_1$.

Since substitution does not change the control-flow graph, it preserves the domination relations.

**Lemma 14.**

1. $f \models l_1 \gg= l_2 \iff f\{r_2/r_1\} \models l_1 \gg= l_2$

2. $f \models pc_1 \gg pc_2 \iff f\{r_2/r_1\} \models pc_1 \gg pc_2$

Applying Lemma 2 and Lemma 14, we have:

**Lemma 15** ($f\{r_2/r_1\}$ preserves well-formedness)**.** *Suppose that in $f$, $r_1$ is redundant with $r_2$. If $\vdash f$, then $\vdash f\{r_2/r_1\}$.*

Let two program states simulate each other if they have the same local state $\delta$ and program counter. We assume that the original program and its transformation have the same initial state.

**Lemma 16.** *If $\vdash f$, $r_2$ is redundant with $r_1$ in $f$, and $(pc, \delta)$ is a reachable state, then*

1. *If val is an operand of a non-phinode at program counter pc, then* $\exists v. [\![val]\!]_\delta = \lfloor v \rfloor \wedge [\![val\{r_1/r_2\}]\!]_\delta = \lfloor v \rfloor$.

2. *If pc is* $l_i.\mathbf{t}$*, and* $l_i$ *is a previous block of a block with* $\phi$*-nodes* $\overline{\phi_j}^j$*, then* $\exists \delta'. [\![\overline{\phi_j}^j]\!]_\delta^{l_i} = \lfloor \delta' \rfloor \wedge [\![\overline{\phi_j\{r_1/r_2\}}^j]\!]_\delta^{l_i} = \lfloor \delta' \rfloor$.

**P**roof (sketch): The proof makes crucial use of Theorem 13. For example, to show part 1 for a source instruction $r := rhs$ (with transformed instruction $r := rhs\{r_1/r_2\}$) located at program counter $pc$, we reason like this: if $r_2$ is an operand used by $rhs$, then $r_2 \in \mathbf{sdom}_f(pc)$ and by Theorem 13, property $P_{\text{sem}}$, implies that $\delta[r_2] = [\![rhs_2]\!]_\delta$ for some $rhs_2$ defining $r_2$. Since $r_1$ is used as an operand in $rhs\{r_1/r_2\}$, similar reasoning shows that $\delta[r_1] = [\![rhs_1]\!]_\delta$, but since $r_2$ is redundant with $r_1$, we have $rhs_2 = rhs_1$, and the result follows immediately. □

Using Lemma 16, we can easily show the lock-step simulation lemma, which completes the correctness proof:

**Lemma 17.** *If* $\vdash f$*,* $r_2$ *is redundant with* $r_1$ *in* $f$*,* $f\{r_1/r_2\} \vdash \sigma_1 \longrightarrow \sigma_2$*, then* $f \vdash \sigma_1 \longrightarrow \sigma_2$*.*

This chapter showed the proof techniques for reasoning about SSA-based program properties and transformations of Vminus. To demonstrate that our proof techniques can be used for practical compiler optimizations, the following chapters present how to verify program transformations of the full LLVM IR.

# Chapter 6

# The formalism of the LLVM IR

Vminus provides a convenient minimal setting in which to study SSA-based optimizations, but it omits many features necessary in a real intermediate representation. To demonstrate that our proof techniques can be used for practical compiler optimizations, we next show how to apply them to the LLVM IR.

The Vellvm infrastructure provides a Coq implementation of the full LLVM intermediate language and defines (several) operational semantics along with some useful metatheory about the memory model. Vellvm's formalization is based on the LLVM release version 3.0, and the syntax and semantics are intended to model the behavior as described in the LLVM Language Reference [1], although we also used the LLVM IR reference interpreter and the x86 backend to inform our design. The chapter describes the syntax and semantics of the LLVM IR, emphasizing those features that are either unique to the LLVM or have non-trivial implications for the formalization.

## 6.1 The Syntax

Figure 6.1 and Figure 6.2 show the abstract syntax for the subset of the LLVM IR formalized in Vellvm. The metavariable *id* ranges over LLVM identifiers, written `%X`, `%T`, `%a`, `%b`, *etc.*, which are used to name local types and temporary variables, and `@a`, `@b`, `@main`, *etc.*, which name global values and functions.

Each source file is a module *mod* (which is also called a program *P*) that includes data layout information *layout* (which defines sizes and alignments for types; see below), named types, and a list of *prod*s that can

[1]See `http://llvm.org/releases/3.0/docs/LangRef.html`

| | | | |
|---|---|---|---|
| Floats | $fp$ | $::=$ | $\mathbf{float} \mid \mathbf{double}$ |
| Types | $typ$ | $::=$ | $\mathbf{i}sz \mid fp \mid \mathbf{void} \mid typ* \mid [sz \times typ] \mid \{\overline{typ_j}^j\} \mid typ\ \overline{typ_j}^j \mid id \mid \mathbf{opaque}$ |
| Bin ops | $bop$ | $::=$ | $\mathbf{add} \mid \mathbf{sub} \mid \mathbf{mul} \mid \mathbf{udiv} \mid \mathbf{sdiv} \mid \mathbf{urem} \mid \mathbf{srem} \mid \mathbf{shl} \mid \mathbf{lshr} \mid \mathbf{ashr}$ |
| | | $\mid$ | $\mathbf{and} \mid \mathbf{or} \mid \mathbf{xor}$ |
| Float ops | $fbop$ | $::=$ | $\mathbf{fadd} \mid \mathbf{fsub} \mid \mathbf{fmul} \mid \mathbf{fdiv} \mid \mathbf{frem}$ |
| Extension | $eop$ | $::=$ | $\mathbf{zext} \mid \mathbf{sext} \mid \mathbf{fpext}$ |
| Trunc ops | $trop$ | $::=$ | $\mathbf{trunc}_{int} \mid \mathbf{trunc}_{fp}$ |
| Cast ops | $cop$ | $::=$ | $\mathbf{fptoui} \mid \mathbf{fptosi} \mid \mathbf{uitofp} \mid \mathbf{sitofp} \mid \mathbf{ptrtoint} \mid \mathbf{inttoptr} \mid \mathbf{bitcast}$ |
| Conditions | $cond$ | $::=$ | $\mathbf{eq} \mid \mathbf{ne} \mid \mathbf{ugt} \mid \mathbf{uge} \mid \mathbf{ult} \mid \mathbf{ule} \mid \mathbf{sgt} \mid \mathbf{sge} \mid \mathbf{slt} \mid \mathbf{sle}$ |
| Float conditions | $fcond$ | $::=$ | $\mathbf{oeq} \mid \mathbf{ogt} \mid \mathbf{oge} \mid \mathbf{olt} \mid \mathbf{ole} \mid \mathbf{one} \mid \mathbf{ord} \mid \mathbf{fueq} \mid \mathbf{fugt} \mid \mathbf{fuge} \mid \cdots$ |
| Constants | $cnst$ | $::=$ | $\mathbf{i}sz\,Int$ |
| | | $\mid$ | $fp\,Float$ |
| | | $\mid$ | $typ * id$ |
| | | $\mid$ | $(typ*)\,\mathbf{null}$ |
| | | $\mid$ | $typ\,\mathbf{zeroinitializer}$ |
| | | $\mid$ | $typ[\overline{cnst_j}^j]$ |
| | | $\mid$ | $\{\overline{cnst_j}^j\}$ |
| | | $\mid$ | $typ\,\mathbf{undef}$ |
| | | $\mid$ | $bop\,cnst_1\,cnst_2$ |
| | | $\mid$ | $fbop\,cnst_1\,cnst_2$ |
| | | $\mid$ | $trop\,cnst\,\mathbf{to}\,typ$ |
| | | $\mid$ | $eop\,cnst\,\mathbf{to}\,typ$ |
| | | $\mid$ | $cop\,cnst\,\mathbf{to}\,typ$ |
| | | $\mid$ | $\mathbf{getelementptr}\,cnst\,\overline{cst_j}^j$ |
| | | $\mid$ | $\mathbf{select}\,cnst_0\,cnst_1\,cnst_2$ |
| | | $\mid$ | $\mathbf{icmp}\,cond\,cnst_1\,cnst_2$ |
| | | $\mid$ | $\mathbf{fcmp}\,fcond\,cnst_1\,cnst_2$ |
| | | $\mid$ | $\mathbf{extractvalue}\,cnst\,\overline{cnst_j}^j$ |
| | | $\mid$ | $\mathbf{insertvalue}\,cnst\,cnst'\,\overline{cnst_j}^j$ |

Figure 6.1: Syntax for LLVM (1).

be function declarations, function definitions, and global variables. Figure 6.3 shows a small example of LLVM syntax (its meaning is described in more detail in Section 6.3).

$$
\begin{array}{llcl}
\text{Modules} & mod, P & ::= & \overline{layout}\,\overline{namedt}\,\overline{prod} \\
\text{Layouts} & layout & ::= & \textbf{bigendian} \mid \textbf{littleendian} \mid \textbf{ptr}\, sz\, align_0\, align_1 \mid \textbf{int}\, sz\, align_0\, align_1 \\
 & & \mid & \textbf{float}\, sz\, align_0\, align_1 \mid \textbf{aggr}\, sz\, align_0\, align_1 \mid \textbf{stack}\, sz\, align_0\, align_1 \\
\text{Products} & prod & ::= & id = \textbf{global}\, typ\, const\, align \mid \textbf{define}\, typ\, id(\overline{arg})\{\overline{b}\} \mid \textbf{declare}\, typ\, id(\overline{arg}) \\
\text{Values} & val & ::= & id \mid cnst \\
\text{Blocks} & b & ::= & l\,\overline{\phi}\,\overline{c}\,tmn \\
\phi\ \text{nodes} & \phi & ::= & id = \textbf{phi}\, typ\, \overline{[val_j, l_j]}^{j} \\
\text{Tmns} & tmn & ::= & \textbf{br}\, val\, l_1\, l_2 \\
 & & \mid & \textbf{br}\, l \\
 & & \mid & \textbf{ret}\, typ\, val \\
 & & \mid & \textbf{ret}\,\textbf{void} \\
 & & \mid & \textbf{unreachable} \\
\text{Commands} & c & ::= & id = bop(\textbf{int}\, sz)\, val_1\, val_2 \\
 & & \mid & id = fbop\, fp\, val_1\, val_2 \\
 & & \mid & \textbf{store}\, typ\, val_1\, val_2\, align \\
 & & \mid & id = \textbf{load}\, (typ*)\, val_1\, align \\
 & & \mid & id = \textbf{malloc}\, typ\, val\, align \\
 & & \mid & \textbf{free}\, (typ*)\, val \\
 & & \mid & id = \textbf{alloca}\, typ\, val\, align \\
 & & \mid & id = trop\, typ_1\, val\, \textbf{to}\, typ_2 \\
 & & \mid & id = eop\, typ_1\, val\, \textbf{to}\, typ_2 \\
 & & \mid & id = cop\, typ_1\, val\, \textbf{to}\, typ_2 \\
 & & \mid & id = \textbf{icmp}\, cond\, typ\, val_1\, val_2 \\
 & & \mid & id = \textbf{select}\, val_0\, typ\, val_1\, val_2 \\
 & & \mid & id = \textbf{fcmp}\, fcond\, fp\, val_1\, val_2 \\
 & & \mid & option\, id = \textbf{call}\, typ_0\, val_0\, \overline{param} \\
 & & \mid & id = \textbf{getelementptr}\, (typ*)\, val\, \overline{val_j}^{j} \\
 & & \mid & id = \textbf{extractvalue}\, typ\, val\, \overline{cnst_j}^{j} \\
 & & \mid & id = \textbf{insertvalue}\, typ\, val\, typ'\, val'\, \overline{cnst_j}^{j}
\end{array}
$$

Figure 6.2: Syntax for LLVM (2).

Every LLVM expression has a type, which can easily be determined from type annotations that provide sufficient information to check an LLVM program for type compatibility. The LLVM IR is not a type-safe language, however, because its type system allows arbitrary casts, calling functions with incorrect signatures, accessing invalid memory, *etc.* The LLVM type system ensures only that the size of a runtime value in a

```
%ST = type { i10 , [10 x i8*] }

define %ST* @foo(i8* %ptr) {
entry:
  %p = malloc %ST, i32 1
  %r = getelementptr %ST* %p, i32 0, i32 0
  store i10 648, %r        ; decomposes as 136, 2
  %s = getelementptr %ST* %p, i32 0, i32 1, i32 0
  store i8* %ptr, %s
  ret %ST* %p
}
```

Here, `%p` is a pointer to a single-element array of structures of type `%ST`. Pointer `%r` indexes into the first component of the first element in the array, and has type `i10*`, as used by the subsequent **store**, which writes the 10-bit value 648. Pointer `%s` has type `i8**` and points to the first element of the nested array in the same structure.

Figure 6.3: An example use of LLVM's memory operations.

well-formed program is compatible with the type of the value—a well-formed program can still be stuck (see Section 6.4.3).

Types *typ* include arbitrary bit-width integers **i**8, **i**16, **i**32, *etc.*, or, more generally, **i***sz* where *sz* is a natural number. Types also include **float**, **void**, pointers *typ*∗, arrays $[sz \times typ]$ that have a statically-known size *sz*. Anonymous structure types $\{\overline{typ_j}^j\}$ contain a list of types. Functions $typ\ \overline{typ_j}^j$ have a return type, and a list of argument types. Here, $\overline{typ_j}^j$ denotes a list of *typ* components; we use similar notation for other lists throughout the paper. Finally, types can be named by identifiers *id* which is useful to define recursive types.

The sizes and alignments for types, and endianness are defined in *layout*. For example. $\mathbf{int}\,sz\,align_0\,align_1$ dictates that values with type **i***sz* are $align_0$-byte aligned when they are within an aggregate and when used as an argument, and $align_1$-byte aligned when emitted as a global.

Operations in the LLVM IR compute with values *val*, which are either identifiers *id* naming temporaries, or constants *cnst* computed from statically-known data, using the compile-time analogs of the commands described below. Constants include base values (*i.e.,* integers or floats of a given bit width), and zero-values of a given type, as well as structures and arrays built from other constants.

To account for uninitialized variables and to allow for various program optimizations, the LLVM IR also supports a type-indexed **undef** constant. Semantically, **undef** stands for a *set* of possible bit patterns, and LLVM compilers are free to pick convenient values for each occurrence of **undef** to enable aggressive optimizations or program transformations. As described in Section 6.4, the presence of **undef** makes the LLVM operational semantics inherently nondeterministic.

All code in the LLVM IR resides in top-level functions, whose bodies are composed of block $b$s. As in classic compiler representations, a basic block consists of a labeled entry point $l$, a series of $\phi$ nodes, a list of commands, and a terminator instruction. As is usual in SSA representations, the $\phi$ nodes join together values from a list of predecessor blocks of the control-flow graph—each $\phi$ node takes a list of (value, label) pairs that indicates the value chosen when control transfers from a predecessor block with the associated label. Block terminators (**br** and **ret**) branch to another block or return (possibly with a value) from the current function. Terminators also include the **unreachable** marker, indicating that control should never reach that point in the program.

The core of the LLVM instruction set is its *commands* ($c$), which include the usual suite of binary arithmetic operations (*bop*—e.g., **add**, **lshr**, etc.), memory accessors (**load**, **store**), heap operations (**malloc** and **free**), stack allocation (**alloca**), conversion operations among integers, floats and pointers (*eop*, *trop*, and *cop*), comparison over integers (**icmp** and **select**), and calls (**call**). Note that a call site is allowed to ignore the return value of a function call. Finally, **getelementptr** computes pointer offsets into structured datatypes based on their types; it provides a platform- and layout-independent way of performing array indexing, struct field access, and pointer arithmetic.

**Omitted details** This dissertation does not discuss all of the LLVM IR features that the Vellvm Coq development supports. Most of these features are uninteresting technically but necessary to support real LLVM code: (1) The LLVM IR provides aggregate data operations (**extractvalue** and **insertvalue**) for projecting and updating the elements of structures and arrays; (2) the LLVM **switch** instruction, which is used to compile jump tables, is lowered to the normal branch instructions that Vellvm supports by a LLVM-supported pre-processing step.

**Unsupported features** Some features of LLVM are not supported by Vellvm. First, the LLVM provides *intrinsic* functions for extending LLVM or to represent functions that have well known names and semantics and are required to follow certain restrictions—for example, functions from standard C libraries, handling variable argument functions, *etc.* Second, the LLVM functions, global variables, and parameters can be decorated with attributes that denote linkage type, calling conventions, data representation, *etc.* which provide more information to compiler transformations than what the LLVM type system provides. Vellvm does not statically check the well-formedness of these attributes, although they should be obeyed by any valid program transformation. Third, Vellvm does not support the *invoke* and *unwind* instructions, which are used to implement exception handling, nor does it support variable argument functions. Forth, Vellvm does not support vector types, which allow for multiple primitive data values to be computed in parallel using a single instruction.

## 6.2 The Static Semantics

Following the LLVM IR specification, Vellvm requires that every LLVM program satisfy certain invariants to be considered well formed: every variable in a function is well-typed, well-scoped, and assigned exactly once. At a minimum, any reasonable LLVM transformation must preserve these invariants; together they imply that the program is in SSA form [28].

All the components in the LLVM IR are annotated with types, so the typechecking algorithm is straightforward and determined only by local information.The only subtlety is that types themselves must be well formed. All *typ*s except **void** and function types are considered to be *first class*, meaning that values of these types can be passed as arguments to functions. A set of first-class type definitions is well formed if there are no degenerate cycles in their definitions (*i.e.,* every cycle through the definitions is broken by a pointer type). This property ensures that the physical sizes of such *typ*s are positive (non-zero), finite, and known statically.

The LLVM IR has two syntactic scopes—a global scope and a function scope—and does not have nested local scopes. In the global scope, all named types, global variables and functions have different names, and are defined mutually. In the scope of a function *fid* in module *mod*, all the global identifiers in *mod*, the names of arguments, locally defined variables and block labels in the function *fid* must be unique, which enforces the single-assignment part of the SSA property.

The set of blocks making up a function constitute a control-flow graph with a well-defined entry point. All instructions in the function must satisfy the SSA scoping invariant with respect to the control-flow graph: the instruction defining an identifier must *dominate* all the instructions that use it. These well-formedness constraints must hold only of blocks that are *reachable* from a function's entry point—unreachable code may contain ill-typed and ill-scoped instructions. Chapter 5 described the proof techniques we have developed for formalizing the invariant in the context of Vminus. We applied the idea in the full Vellvm.

## 6.3 A Memory Model for the LLVM IR

### 6.3.1 Rationale

Vminus does not include memory operations because the LLVM IR does not represent memory in SSA. However, understanding the semantics of LLVM's memory operations is crucial for reasoning about LLVM programs. LLVM developers make many assumptions about the "legal" behaviors of such LLVM code, and they informally use those assumptions to justify the correctness of program transformations.

There are many properties expected of a reasonable implementation of the LLVM memory operations (especially in the absence of errors). For example, we can reasonably assume that the **load** instruction does not affect which memory addresses are allocated, or that different calls to **malloc** do not inappropriately reuse memory locations. Unfortunately, the LLVM Language Reference Manual does not enumerate all such properties, which should hold of *any* "reasonable" memory implementation.

On the other hand, details about the particular memory management implementation *can* be observed in the behavior of LLVM programs (*e.g.,* you can print a pointer after casting it to an integer). For this reason, and also to address error conditions, the LLVM specification intentionally leaves some behaviors undefined. Examples include: loading from an unallocated address; loading with improper alignment; loading from properly allocated but uninitialized memory; and loading from properly initialized memory but with an incompatible type.

Because of the dependence on a concrete implementation of memory operations, which can be platform specific, there are *many* possible memory models for the LLVM. One of the challenges we encountered in formalizing the LLVM was finding a point in the design space that accurately reflects the intent of the LLVM documentation while still providing a useful basis for reasoning about LLVM programs.

In this dissertation we adopt a memory model that is based on the one implemented for CompCert [42]. This model allows Vellvm to accurately implement the LLVM IR and, in particular, detect the kind of errors mentioned above while simultaneously justifying many of the "reasonable" assumptions that LLVM programmers make. The nondeterministic operational semantics presented in Section 6.4 takes advantage of this precision to account for much of the LLVM's under-specification.

Although Vellvm's design is intended to faithfully capture the LLVM specification, it is also partly motivated by pragmatism: building on CompCert's existing memory model allowed us to re-use a significant amount of their Coq infrastructure. A benefit of this choice is that our memory model is compatible with CompCert's memory model (*i.e.,* our memory model implements the CompCert `Memory` signature).

This Vellvm memory model inherits some features from the CompCert implementation: it is single threaded (in this paper we consider only single-threaded programs); it assumes that pointers are 32-bits wide, and 4-byte aligned; and it assumes that the memory is infinite. Unlike CompCert, Vellvm's model must also deal with arbitrary bit-width integers, padding, and alignment constraints that are given by layout annotations in the LLVM program, as described next.

### 6.3.2 LLVM memory commands

The LLVM supports several commands for working with heap-allocated data structures:

- **malloc** and **alloca** allocate array-structured regions of memory. They take a type parameter, which determines layout and padding of the elements of the region, and an integral size that specifies the number of elements; they return a pointer to the newly allocated region.

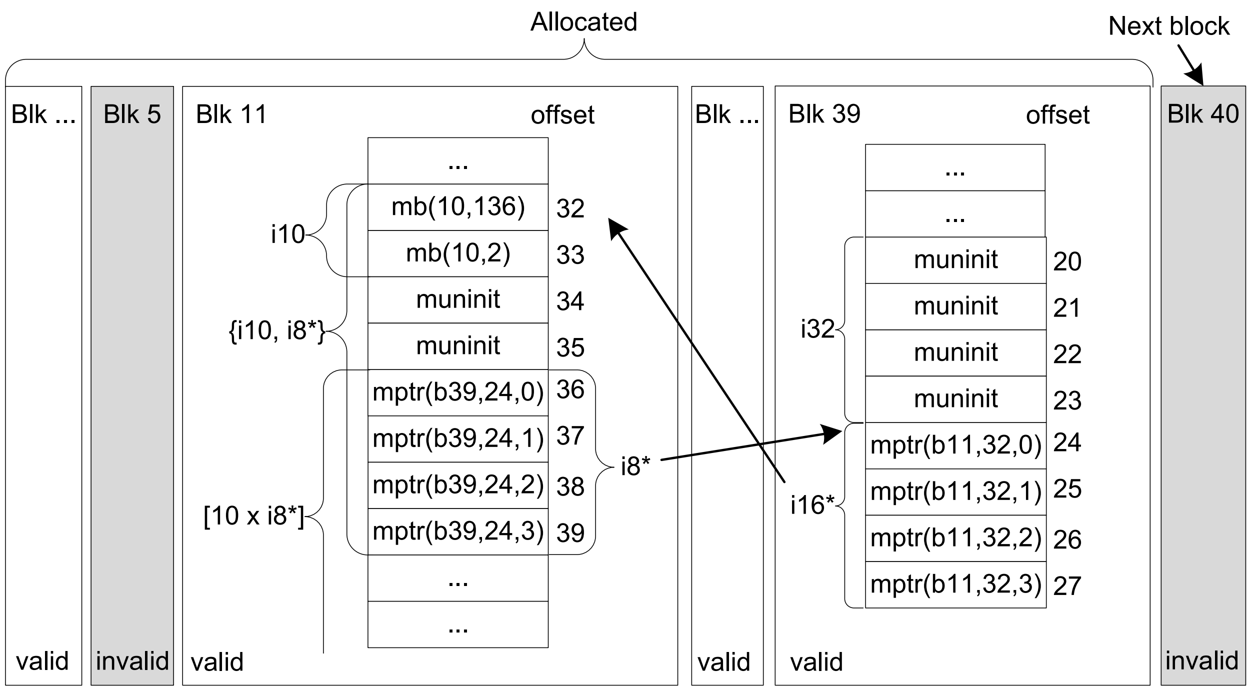
- **free** deallocates the memory region associated with a given pointer (which should have been created by **malloc**). Memory allocated by **alloca** is implicitly freed upon return from the function in which **alloca** was invoked.

- **load** and **store** respectively read and write LLVM values to memory. They take type parameters that govern the expected layout of the data being read/written.

- **getelementptr** indexes into a structured data type by computing an offset pointer from another given pointer based on its type and a list of indices that describe a path into the datatype.

Figure 6.3 gives a small example program that uses these operations. Importantly, the type annotations on these operations can be any first-class type, which includes arbitrary bit-width integers, floating point values, pointers, and aggregated types—arrays and structures. The LLVM IR semantics treats memory as though it is dynamically typed: the sizes, layout, and alignment, of a value read via a **load** instruction must be consistent with that of the data that was **stored** at that address, otherwise the result is undefined.

This approach leads to a memory model structured in two parts: (1) a low-level byte-oriented representation that stores values of basic (non-aggregated) types along with enough information to indicate physical size, alignment, and whether or not the data is a pointer, and (2) an encoding that flattens LLVM-level structured data with first-class types into a sequence of basic values, computing appropriate padding and alignment from the type. The next two subsections describe these two parts in turn.



This figure shows (part of) a memory state. Blocks less than 40 were allocated; the next fresh block to allocate is 40. Block 5 is deallocated, and thus marked invalid to access; fresh blocks ($\geq$ 40) are also invalid. Invalid memory blocks are gray, and valid memory blocks that are accessible are white. Block 11 contains data with structure type `{i10, [10 x i8*]}` but it might be read (due to physical subtyping) at the type `{i10, i8*}`. This type is flattened into two byte-sized memory cells for the i10 field, two uninitialized padding cells to adjust alignment, and four pointer memory cells for the first element of the array of 32-bit `i8*` pointers. Here, that pointer points to the 24th memory cell of block 39. Block 39 contains an uninitialized i32 integer represented by four muninit cells followed by a pointer that points to the 32nd memory cell of block 11.

Figure 6.4: Vellvm's byte-oriented memory model.

## 6.3.3 The byte-oriented representation

The byte-oriented representation is composed of blocks of memory cells. Each cell is a byte-sized quantity that describes the smallest chunk of contents that a memory operation can access. Cells come in several flavors:

$$\text{Memory cells}\, mc \quad ::= \quad \mathsf{mb}(sz, byte) \mid \mathsf{mptr}(blk, ofs, idx) \mid \mathsf{muninit}$$

The memory cell $\mathsf{mb}(sz, byte)$ represents a byte-sized chunk of numeric data, where the LLVM-level bit-width of the integer is given by $sz$ and whose contents is $byte$. For example, an integer with bit-width 32 is represented by four mb cells, each with size parameter 32. An integer with bit-width that is not divisible by 8 is encoded by the minimal number of bytes that can store the integer, *i.e.,* an integer with bit-width 10 is encoded by two bytes, each with size parameter ten (see Figure 6.4). Floating point values are encoded similarly.

Memory addresses are represented as a block identifier *blk* and an offset *ofs* within that block; the cell $\mathsf{mptr}(blk, ofs, idx)$ is a byte-sized chunk of such a pointer where *idx* is an index identifying which byte the chunk corresponds to. Because Vellvm's implementation assumes 32-bit pointers, four such cells are needed to encode one LLVM-pointer, as shown in Figure 6.4. Loading a pointer succeeds only if the 4 bytes loaded are sequentially indexed from 0 to 3.

The last kind of cell is muninit, which represents uninitialized memory, layout padding, and bogus values that result from undefined computations (such as might arise from an arithmetic overflow).

Given this definition of memory cells, a memory state $M = (N, B, C)$ includes the following components: $N$ is the next fresh block to allocate, $B$ maps a valid block identifier to the size of the block; $C$ maps a block identifier and an offset within the block to a memory cell (if the location is valid). Initially, $N$ is 1; $B$ and $C$ are empty. Figure 6.4 gives a concrete example of such a memory state for the program in Figure 6.3.

There are four basic operations over this byte-oriented memory state: **alloc**, **mfree**, **mload**, and **mstore**. **alloc** allocates a fresh memory block $N$ with a given size, increments $N$, fills the newly allocated memory cells with muninit. **mfree** simply removes the deallocated block from $B$, and its contents from $C$. Note that the memory model does not recycle block identifiers deallocated by a **mfree** operation, because this model assumes that a memory is of infinite size.

The **mstore** operation is responsible for breaking non-byte sized *basic values* into chunks and updating the appropriate memory locations. Basic values are integers (with their bit-widths), floats, addresses, and padding.

$$\begin{array}{lrcl} \text{Basic values} & bv & ::= & Int\,sz \mid Float \mid blk.ofs \mid \mathbf{pad}\,sz \\ \text{Basic types} & btyp & ::= & \mathbf{i}sz \mid fp \mid typ* \end{array}$$

**mload** is a partial function that attempts to read a value from a memory location. It is annotated by a basic type, and ensures compatibility between memory cells at the address it reads from and the given type. For example, memory cells for an integer with bit-width *sz* cannot be accessed as an integer type with a different bit-width; a sequence of bytes can be accessed as floating point values if they can be decoded as a floating point value; pointers stored in memory can only be accessed by pointer types. If an access is type incompatible, **mload** returns **pad***sz*, which is an "error" value representing an arbitrary bit pattern with the bitwidth *sz* of the type being loaded. **mload** is undefined in the case that the memory address is not part of a valid allocation block.

### 6.3.4 The LLVM flattened values and memory accesses

LLVM's structured data is flattened to lists of basic values that indicate its physical representation:

$$\text{Flattened Values}\, v \quad ::= \quad bv \mid bv, v$$

A constant *cnst* is flattened into a list of basic values according to it annotated type. If the *cnst* is already of basic type, it flattens into the singleton list. Values of array type $[sz \times typ]$ are first flattened element-wise according to the representation given by *typ* and then padded by uninitialized values to match *typ*'s alignment requirements as determined by the module's *layout* descriptor. The resulting list is then concatenated to obtain the appropriate flattened value. The case when a *cnst* is a structure type is similar.

The LLVM **load** instruction works by first flattening its type annotation *typ* into a list of basic types, and mapping **mload** across the list; it then merges the returned basic values into the final LLVM value. Storing an LLVM value to memory works by first flattening to a list of basic values and mapping **mstore** over the result.

This scheme induces a notion of dynamically-checked *physical subtyping*: it is permitted to read a structured value at a different type from the one at which it was written, so long as the basic types they flatten into agree. For non-structured data types such as integers, Vellvm's implementation is conservative—for example, reading an integer with bit width two from the second byte of a 10-bit wide integer yields `undef` because the results are, in general, platform specific. Because of this dynamically-checked, physical subtyping, pointer-to-pointer casts can be treated as the identity. Similar ideas arise in other formalizations of low-level language semantics [54, 55].

The LLVM **malloc** and **free** operations are defined by **alloc** and **mfree** in a straightforward manner. As the LLVM IR does not explicitly distinguish the heap and stack and function calls are implementation-specific, the memory model defines the same semantics for stack allocation (**alloca**) and heap allocation (**malloc**) — both of them allocate memory blocks in memory. However, the operational semantics (described next) maintains a list of blocks allocated by **alloca** for each function, and it deallocates them on return.

## 6.4 Operational Semantics

Vellvm provides several related operational semantics for the LLVM IR, as summarized in Figure 6.5. The most general is LLVM$_{ND}$, a small-step, nondeterministic evaluation relation given by rules of the form $\mathit{config} \vdash S \twoheadrightarrow S'$ (see Figure 6.6). This section first motivates the need for nondeterminism in understanding

$$\begin{array}{c} \text{LLVM}_{ND} \\ \Cup \\ \text{LLVM}_{Interp} \approx \text{LLVM}_D \gtrsim \text{LLVM}^*_{DFn} \gtrsim \text{LLVM}^*_{DB} \end{array}$$

Figure 6.5: Relations between different operational semantics, which are justified by proofs in Vellvm.

the LLVM semantics and then illustrates LLVM$_{ND}$ by explaining some of its rules. Next, we introduce several equivalent deterministic refinements of LLVM$_{ND}$—LLVM$_D$, LLVM$^*_{DB}$, and LLVM$^*_{DFn}$—each of which has different uses, as described in Section 6.4.4. All of these operational semantics must handle various error conditions, which manifest as partiality in the rules. Section 6.4.3 describes these error conditions, and relates them to the static semantics of Section 6.2.

Vellvm's operational rules are specified as transitions between *machine states* $S$ of the form $M, \overline{\Sigma}$, where $M$ is the memory and $\overline{\Sigma}$ is a stack of *frames*. A frame keeps track of the current function *fid* and block label $l$, as well as the "continuation" sequence of commands $\overline{c}$ to execute next ending with the block terminator $tmn$. The map $\Delta$ tracks bindings for the local variables (which are not stored in $M$), and the list $\alpha$ keeps track of which memory blocks were created by the **alloca** instruction so that they can be marked as invalid when the function call returns.

$$\begin{array}{llll llll} \text{Value sets} & V & ::= & \{v \mid \Phi(v)\} & \text{Locals} & \Delta & ::= & id \mapsto V \\ \text{Allocas} & \alpha & ::= & [] \mid blk, \alpha & \text{Frames} & \Sigma & ::= & fid, l, \overline{c}, tmn, \Delta, \alpha \\ \text{Call stacks} & \overline{\Sigma} & ::= & [] \mid \Sigma, \overline{\Sigma} & \text{Program states} & S & ::= & M, \overline{\Sigma} \end{array}$$

### 6.4.1 Nondeterminism in the LLVM operational semantics

There are several sources of nondeterminism in the LLVM semantics: the **undef** value, which stands for an arbitrary (and ephemeral) bit pattern of a given type, various memory errors, such as reading from an uninitialized location. Unlike the "fatal" errors, which are modeled by stuck states (see Section 6.4.3), we choose to model these behaviors nondeterministically because they correspond to choices that would be resolved by running the program with a concrete memory implementation. Moreover, the LLVM optimization passes use the flexibility granted by this underspecificity to justify aggressive optimizations.

Nondeterminism shows up in two ways in the LLVM$_{ND}$ semantics. First, stack frames bind local variables to sets of values $V$; second, the $\twoheadrightarrow$ relation itself may relate one state to many possible successors. The semantics teases apart these two kinds of nondeterminism because of the way that the **undef** value interacts with memory operations, as illustrated by the examples below.

$\boxed{config \vdash S \twoheadrightarrow S'}$

$$\frac{\begin{array}{c}\mathbf{eval}_{ND}(g,\Delta,val) = \lfloor V \rfloor \quad \mathbf{findfdef}\,(mod,\theta,v) = \lfloor \mathbf{define}\, typ\, fid'\,(\overline{arg})\{(l'[]\overline{c}'tmn'),\overline{b}\}\rfloor \\ v \in V \quad \mathbf{initlocals}\,(g,\Delta,\overline{arg},\overline{param}) = \lfloor \Delta' \rfloor \quad c_0 = (option\, id\, =\, \mathbf{call}\, typ\, val\, \overline{param})\end{array}}{mod,g,\theta \vdash M,((fid,l,(c_0,\overline{c}),tmn,\Delta,\alpha),\overline{\Sigma}) \twoheadrightarrow M,((fid',l',\overline{c}',tmn',\Delta',[]),(fid,l,(c_0,\overline{c}),tmn,\Delta,\alpha),\overline{\Sigma})} \quad \text{NDS\_CALL}$$

$$\frac{\mathbf{eval}_{ND}(g,\Delta,val) = \lfloor V \rfloor \quad c_0 = (option\, id\, =\, \mathbf{call}\, typ\, val\, \overline{param}) \quad \mathbf{freeallocas}\,(M,\alpha') = \lfloor M' \rfloor}{mod,g,\theta \vdash M,((fid',l',[],\mathbf{ret}\, typ\, val,\Delta',\alpha'),(fid,l,(c_0,\overline{c}),tmn,\Delta,\alpha),\overline{\Sigma}) \twoheadrightarrow M',((fid,l,\overline{c},tmn,\Delta\{id \leftarrow V\},\alpha),\overline{\Sigma})} \quad \text{NDS\_RET}$$

$$\frac{\mathbf{eval}_{ND}(g,\Delta,val) = \lfloor V \rfloor \quad \mathbf{true} \in V \quad \mathbf{findblock}\,(mod,fid,l_1) = (l_1\overline{\phi}_1\overline{c}_1 tmn_1) \quad \mathbf{computephinodes}_{ND}(g,\Delta,l,l_1,\overline{\phi}_1) = \lfloor \Delta' \rfloor}{mod,g,\theta \vdash M,((fid,l,[],\mathbf{br}\, val\, l_1\, l_2,\Delta,\alpha),\overline{\Sigma}) \twoheadrightarrow M,((fid,l_1,\overline{c}_1,tmn_1,\Delta',\alpha),\overline{\Sigma})} \quad \text{NDS\_BR\_TRUE}$$

$$\frac{\mathbf{eval}_{ND}(g,\Delta,val) = \lfloor V \rfloor \quad v \in V \quad c_0 = (id = \mathbf{malloc}\, typ\, val\, align) \quad \mathbf{malloc}\,(M,typ,v,align) = \lfloor M',blk \rfloor}{mod,g,\theta \vdash M,((fid,l,(c_0,\overline{c}),tmn,\Delta,\alpha),\overline{\Sigma}) \twoheadrightarrow M',((fid,l,\overline{c},tmn,\Delta\{id \leftarrow \{blk.0\}\},\alpha),\overline{\Sigma})} \quad \text{NDS\_MALLOC}$$

$$\frac{\mathbf{eval}_{ND}(g,\Delta,val) = \lfloor V \rfloor \quad v \in V \quad c_0 = (id = \mathbf{alloca}\, typ\, val\, align) \quad \mathbf{malloc}\,(M,typ,v,align) = \lfloor M,blk \rfloor}{mod,g,\theta \vdash M,((fid,l,(c_0,\overline{c}),tmn,\Delta,\alpha),\overline{\Sigma}) \twoheadrightarrow M',((fid,l,\overline{c},tmn,\Delta\{id \leftarrow \{blk.0\}\},(blk,\alpha)),\overline{\Sigma})} \quad \text{NDS\_ALLOCA}$$

$$\frac{\mathbf{eval}_{ND}(g,\Delta,val_1) = \lfloor V_1 \rfloor \quad \mathbf{eval}_{ND}(g,\Delta,val_2) = \lfloor V_2 \rfloor \quad \mathbf{evalbop}_{ND}(bop,sz,V_1,V_2) = V_3}{mod,g,\theta \vdash M,((fid,l,(id = bop(\mathbf{int}\, sz)val_1\, val_2,\overline{c}),tmn,\Delta,\alpha),\overline{\Sigma}) \twoheadrightarrow M,((fid,l,\overline{c},tmn,\Delta\{id \leftarrow V_3\},\alpha),\overline{\Sigma})} \quad \text{NDS\_BOP}$$

Figure 6.6: LLVM$_{ND}$: Small-step, nondeterministic semantics of the LLVM IR (selected rules).

From the LLVM Language Reference Manual: "Undefined values indicate to the compiler that the program is well defined no matter what value is used, giving the compiler more freedom to optimize." Semantically, LLVM$_{ND}$ treats **undef** as the set of *all* values of a given type. For some motivating examples, consider the following code fragments:

```
(a)  %z = xor i8 undef undef

(b)  %x = add i8 0 undef
     %z = xor i8 %x %x

(c)  %z = or i8 undef 1

(d)  br undef %l1 %l2
```

The value computed for `%z` in example (a) is the set of all 8-bit integers: because each occurrence of **undef** could take on any bit pattern, the set of possible results obtained by `xor`ing them still includes all 8-bit integers. Perhaps surprisingly, example (b) computes the *same* set of values for `%z`: one might reason that no matter which value is chosen for **undef**, the result of `xor`ing `%x` with itself would always be `0`, and therefore `%z` should always be `0`. However, while that answer is *compatible* with the LLVM language reference (and hence allowed by the nondeterministic semantics), it is also safe to replace code fragment (b) with `%z = undef`. The reason is that the LLVM IR adopts a liberal substitution principle: because `%x = undef` would be a legitimate replacement for first assignment in (b), it is allowed to *substitute* **undef** for `%x` throughout, which reduces the assignment to `%z` to the same code as in (a).

Example (c) shows why the semantics needs arbitrary sets of values. Here, `%z` evaluates to the set of odd 8-bit integers, which is the result of `or`ing 1 with each element of the set $\{0,\ldots,255\}$. This code snippet could therefore *not* safely be replaced by `%z = undef`; however it could be optimized to `%z = 1` (or any other odd 8-bit integer).

Example (d) illustrates the interaction between the set-semantics for local values and the nondeterminism of the $\twoheadrightarrow$ relation. The control state of the machine holds *definite* information, so when a branch occurs, there may be multiple successor states. Similarly, we choose to model memory cells as holding definite values, so when writing a set to memory, there is one successor state for each possible value that could be written. As an example of that interaction, consider the following example program, which was posted to the LLVMdev mailing list [5], that reads from an uninitialized memory location:

```
%buf = alloca i32
%val = load i32* %buf
store i32 10, i32* %buf
ret %val
```

The LLVM mem2reg pass optimizes this program to program (a) below; though according to the LLVM semantics, it would also be admissible to replace this program with option (b) (perhaps to expose yet more optimizations):

```
(a)  ret i32 10        (b)  ret i32 undef
```

### 6.4.2 Nondeterministic operational semantics of the SSA form

The LLVM$_{ND}$ semantics we have developed for Vellvm (and the others described below) is parameterized by a *configuration*, which is a triple of a module containing the code, a (partial) map $g$ that gives the values of global constants, and a function pointer table $\theta$ that is a (partial) map from values to function identifiers. The globals and function pointer maps are initialized from the module definition when the machine is started.

$$\text{Fun tables } \theta \quad ::= \quad v \mapsto id \qquad \text{Globals } g \quad ::= \quad id \mapsto v \qquad \text{Configurations } \mathit{config} \quad ::= \quad \mathit{mod}, g, \theta$$

The LLVM$_{ND}$ rules relate machine states to machine states, where a machine state takes the form of a memory $M$ (from Section 6.3) and a stack of evaluation frames. The frames keep track of the (sets of) values bound to locally-allocated temporaries and which instructions are currently being evaluated. Figure 6.6 shows a selection of evaluation rules from the development.

Most of the commands of the LLVM have straight-forward interpretation: the arithmetic, logic, and data manipulation instructions are all unsurprising—the **eval**$_{ND}$ function computes a set of flattened values from the global state, the local state, and an LLVM *val*, looking up the meanings of variables in the local state as needed; similarly, **evalbop**$_{ND}$ implements binary operations, computing the result set by combining all possible pairs drawn from its input sets. LLVM$_{ND}$'s **malloc** behaves as described in Section 6.3, while **load** uses the memory model's ability to detect ill-typed and uninitialized reads and, in the case of such errors, yields **undef** as the result. Function calls push a new stack frame whose initial local bindings are computed from the function parameters. The $\alpha$ component of the stack frame keeps track of which blocks of memory are created by the **alloca** instruction (see rule NDS_ALLOCA); these are freed when the function returns (rule NDS_RET). As discussed in Section 4.1, the $\textbf{computephinodes}_{ND}$ function in the operational semantics, as shown, for example, in rule NDS_BR_TRUE implements "parallel assignment".

### 6.4.3 Partiality, preservation, and progress

Throughout the rules the "lift" notation $f(x) = \lfloor v \rfloor$ indicates that a partial function $f$ is defined on $x$ with value $v$. As seen by the frequent uses of lifting, both the nondeterministic and deterministic semantics are *partial*—the program may get stuck.

Some of this partiality is related to well-formedness of the SSA program. For example, $\textbf{eval}_{ND}(g, \Delta, \%\texttt{x})$ is undefined if `%x` is not bound in $\Delta$. These kinds of errors are ruled out by the static well-formedness constraints imposed by the LLVM IR (Section 6.2).

In other cases, we have chosen to use partiality in the operational semantics to model certain failure modes for which the LLVM specification says that the behavior of the program is undefined. These include: (1) attempting to **free** memory via a pointer not returned from **malloc** or that has already been deallocated, (2) allocating a negative amount of memory, (3) calling **load** or **store** on a pointer with bad alignment or a deallocated address, (4) trying to call a non-function pointer, or (5) trying to execute the **unreachable** command. We model these events by stuck states because they correspond to fatal errors that will occur in *any* reasonable realization of the LLVM IR by translation to a target platform. Each of these errors is precisely characterized by a predicate over the machine state (*e.g.,* $\text{BadFree}(\mathit{config}, S)$), and the "allowed" stuck states are defined to be the disjunction of these predicates:

$$\begin{array}{rcl}\text{Stuck}(\mathit{config}, S) & = & \text{BadFree}(\mathit{config}, S) \\ & \vee & \text{BadLoad}(\mathit{config}, S) \\ & \vee & \ldots \\ & \vee & \text{Unreachable}(\mathit{config}, S)\end{array}$$

To see that the well-formedness properties of the static semantics rule out all but these known error configurations, we prove the usual *preservation* and *progress* theorems for the $\text{LLVM}_{ND}$ semantics.

**Theorem 18** (Preservation for $\text{LLVM}_{ND}$)**.** *If (config, S) is well formed and config* $\vdash S \twoheadrightarrow S'$*, then (config,* $S'$*) is well formed.*

Here, well-formedness includes the static scoping, typing properties, and SSA invariants from Section 6.2 for the LLVM code, but also requires that the local mappings $\Delta$ present in all frames of the call stack must be inhabited—each binding contains at least one value $v$—and that each defined variable that dominates the current continuation is in $\Delta$'s domain.

That defined variables dominate their uses in the current continuation follows Lemma 11 with considering the context of the full LLVM IR. To show that the $\Delta$ bindings are inhabited after the step, we prove that (1) non-**undef** values $V$ are singletons; (2) undefined values from constants *typ* **undef** contain all possible values of first class types *typ*; (3) undefined values from loading uninitialized memory or incompatible physical data contain at least paddings indicating errors; (4) evaluation of non-deterministic values by $\mathbf{evalbop}_{ND}$ returns non-empty sets of values given non-empty inputs.

**Theorem 19** (Progress for $\mathrm{LLVM}_{ND}$)**.** *If the pair (config, S) is well formed, then either S has terminated successfully or* $\mathrm{Stuck}(\mathit{config}, S)$ *or there exists S' such that* $\mathit{config} \vdash S \twoheadrightarrow S'$.

This theorem holds because in a well-formed machine state, $\mathbf{eval}_{ND}$ always returns a non-empty value set $V$; moreover jump targets and internal functions are always present.

### 6.4.4 Deterministic refinements

Although the $\mathrm{LLVM}_{ND}$ semantics is useful for reasoning about the validity of LLVM program transformations, Vellvm provides a $\mathrm{LLVM}_{D}$, a deterministic, small-step refinement, along with two large-step operational semantics $\mathrm{LLVM}^{*}_{DFn}$ and $\mathrm{LLVM}^{*}_{DB}$.

These different deterministic semantics are useful for several reasons: (1) they provide the basis for testing LLVM programs with a concrete implementation of memory (see the discussion about Vellvm's extracted interpreter in the next Section), (2) proving that $\mathrm{LLVM}_{D}$ is an instance of the $\mathrm{LLVM}_{ND}$ and relating the small-step rules to the large-step ones provides validation of *all* of the semantics (*i.e.,* we found bugs in Vellvm by formalizing multiple semantics and trying to prove that they are related), and (3) the small- and large-step semantics have different applications when reasoning about LLVM program transformations.

Unlike $\mathrm{LLVM}_{ND}$, the frames for these semantics map identifiers to single values, not sets, and the operational rules call deterministic variants of the nondeterministic counterparts (*e.g.,* **eval** instead of $\mathbf{eval}_{ND}$). To resolve the nondeterminism from **undef** and faulty memory operations, these semantics fix a concrete interpretation as follows:

- **undef** is treated as a **zeroinitializer**
- Reading uninitialized memory returns **zeroinitializer**

These choices yield unrealistic behaviors compared to what one might expect from running a LLVM program against a C-style runtime system, but the cases where this semantics differs correspond to *unsafe* programs. There are still many programs, namely those compiled to LLVM from type-safe languages, whose behaviors under this semantics should agree with their realizations on target platforms. Despite these differences from LLVM$_{ND}$, LLVM$_D$ also has the preservation and progress properties.

**Big-step semantics** Vellvm also provides big-step operational semantics LLVM$^*_{DFn}$, which evaluates a function call as one large step, and LLVM$^*_{DB}$, which evaluates each sub-block—*i.e.,* the code between two function calls—as one large step. Big-step semantics are useful because compiler optimizations often transform multiple instructions or blocks within a function in one pass. Such transformations do not preserve the small-step semantics, making it hard to create simulations that establish correctness properties.

As a simple application of the large-step semantics, consider trying to prove the correctness of a transformation that re-orders program statements that do not depend on one another. For example, the following two programs result in the same states if we consider their execution as one big-step, although their intermediate states do not match in terms of the small-step semantics.

```
(a) %x = add i32 %a, %b    (b) %y = load i32* %p
    %y = load i32* %p          %x = add i32 %a, %b
```

The proof of this claim in Vellvm uses the LLVM$^*_{DB}$ rules to hide the details about the intermediate states. To handle memory effects, we use a simulation relation that uses *symbolic evaluation* [52] to define the equivalence of two memory states. The memory contents are defined abstractly in terms of the program operations by recording the sequence of writes. Using this technique, we defined a simple translation validator to check whether the semantics of two programs are equivalent with respect to such re-orderings execution. For each pair of functions, the validator ensures that their control-flow graphs match, and that all corresponding sub-blocks are equivalent in terms of their symbolic evaluation. This approach is similar to the translation validation used in prior work for verifying instruction scheduling optimizations [68].

Although this is a simple application of Vellvm's large-step semantics, proving correctness of other program transformations such as dead expression elimination and constant propagation follow a similar pattern—the difference is that, rather than checking that two memories are syntactically equivalent according to the symbolic evaluation, we must check them with respect to a more semantic notion of equivalence [52].

**Relationships among the semantics** Figure 6.5 illustrates how these various operational semantics relate to one another. Vellvm provides proofs that LLVM$^*_{DB}$ simulates LLVM$^*_{DFn}$ and that LLVM$^*_{DFn}$ simulates LLVM$_D$. In these proofs, simulation is taken to mean that the machine states are syntactically identical at corresponding points during evaluation. For example, the state at a function call of a program running on the LLVM$^*_{DFn}$ semantics matches the corresponding state at the function call reached in LLVM$_D$. Note that in the deterministic setting, one-direction simulation implies bisimulation [42]. Moreover, LLVM$_D$ is a refinement instance of the nondeterministic LLVM$_{ND}$ semantics.

These relations are useful because the large-step semantics induce different proof styles than the small-step semantics: in particular, the induction principles obtained from the large step semantics allow one to gloss over insignificant details of the small step semantics.

**Omitted details** The operational semantics supports external function calls by assuming that their behavior is specified by axioms; the implementation applies these axioms to transition program states upon calling external functions.

## 6.5 Extracting an Interpreter

To test Vellvm's operational semantics for the LLVM IR, we used Coq's code extraction facilities to obtain an interpreter for executing the LLVM distribution's regression test suite. Extracting such an interpreter is one of the main motivations for developing a deterministic semantics, because the evaluation under the nondeterministic semantics cannot be directly compared against actual runs of LLVM IR programs.

Unfortunately, the small-step deterministic semantics LLVM$_D$ is defined relationally in the logical fragment of Coq, which is convenient for proofs, but can not be used to extract code. Therefore, Vellvm provides yet another operational semantics, LLVM$_{Interp}$, which is a deterministic functional interpreter implemented in the computational fragment of Coq. LLVM$_{Interp}$ is proved to be bisimilar to LLVM$_D$, so we can port results between the two semantics.

Although one could run this extracted interpreter directly, doing so is not efficient. First, integers with arbitrary bit-width are inductively defined in Coq. This yields easy proof principles, but does not give an efficient runtime representation; floating point operations are defined axiomatically. To remedy these problems, at extraction we realize Vellvm's integer and floating point values by efficient C++ libraries that are a standard part of the LLVM distribution. Second, the memory model implementation of Vellvm maintains memory blocks and their associated metadata as functional lists, and it converts between byte-list and value representations at each memory access. Using the extracted data-structures directly incurs tremendous performance overhead, so we replaced the memory operations of the memory model with native implementations from the C standard library. A value $v$ in local mappings $\delta$ is boxed, and it is represented by a reference to memory that stores its content.

Our implementation faithfully runs 134 out of the 145 tests from the LLVM regression suite that `lli`, the LLVM distribution interpreter, can run. The missing tests cover instructions (like variable arguments) that are not implemented in Vellvm.

Although replacing the Coq data-structures by native ones weakens the absolute correctness guarantees one would expect from an extracted interpreter, this exercise is still valuable. In the course of carrying out this experiment, we found one severe bug in the semantics: the **br** instruction inadvertently swapped the true and false branches.

# Chapter 7

# Verified SoftBound

To demonstrate the effectiveness of Vellvm, our first application of Vellvm is a verified instance of SoftBound [50, 51], a previously proposed program transformation that hardens C programs against spatial memory safety violations (*e.g.,* buffer overflows, array indexing errors, and pointer arithmetic errors). SoftBound works by first compiling C programs into the LLVM IR and then instrumenting the program with instructions that propagate and check per-pointer metadata. SoftBound maintains base and bound metadata with each pointer, shadowing loads and stores of pointer with parallel loads and stores of their associated metadata. This instrumentation ensures that each pointer dereferenced is within bounds and aborts the program otherwise.

The original SoftBound paper includes a mechanized proof that validates the correctness of this idea, but it is not complete. In particular, the proof is based on a subset of a C-like language with only straight-line commands and non-aggregate types, in contrast a SoftBound implementation needs to consider all of the LLVM IR shown in Figure 6.1 and Figure 6.2, the memory model, and the full operational semantics of the LLVM IR. Also the original proof ensures the correctness only with respect to a specification that the SoftBound instrumentation must implement, but it does not prove the correctness of the instrumentation pass itself. Moreover, the specification requires that every temporary must contain metadata, not just pointer temporaries.

**Using Vellvm to verify SoftBound** This chapter describes how we use Vellvm to formally verify the correctness of the SoftBound instrumentation pass with respect to the LLVM semantics, demonstrating that the promised spatial memory safety property is achieved. Moreover, Vellvm allows us to extract a verified

OCaml implementation of the transformation from Coq. The end result is a compiler pass that is formally verified to transform a program in the LLVM IR into a program augmented with sufficient checking code such that it will dynamically detect and prevent all spatial memory safety violations.

SoftBound is a good test case for the Vellvm framework. It is a non-trivial translation pass that nevertheless only inserts code, thereby making it easier to prove correct. SoftBound's intended use is to prevent security vulnerabilities, so bugs in its implementation can potentially have severe consequences. Also, the existing SoftBound implementation already uses the LLVM.

**Modifications to SoftBound since the original paper** As described in the original paper, SoftBound modifies function signatures to pass metadata associated with the pointer parameters or returned pointers. To improve the robustness of the tool, we transitioned to an implementation that instead passes all pointer metadata on a shadow stack [50]. This has two primary advantages. The first is that this design simplifies the implementation while simultaneously better supporting indirect function calls (via function pointers) and more robustly handling improperly declared function prototypes. The second is that it also simplifies the proofs.

## 7.1 Formalizing SoftBound for the LLVM IR

The SoftBound correctness proof has the following high-level structure:

1. We define a *nonstandard* operational semantics SBspec for the LLVM IR. This semantics "builds in" the safety properties that should be enforced by a correct implementation of SoftBound. It uses meta-level datastructures to implement the metadata and meta-level functions to define the semantics of the bounds checks.

2. We prove that an LLVM program P, when run on the SBspec semantics, has no spatial safety violations.

3. We define a translation pass SBtrans$(-)$ that instruments the LLVM code to propagate metadata.

4. We prove that if SBtrans$(P) = \lfloor P' \rfloor$ then P', when run on the LLVM$_D$, simulates P running on SBspec.

Nondeterministic rules:

$$\frac{\begin{array}{l}\textbf{eval}_{ND}(g,\Delta,val) = \lfloor V \rfloor \quad v \in V \quad c_0 = (id = \textbf{malloc}\, typ\, val\, align) \\ \textbf{malloc}\,(M,typ,v,align) = \lfloor M',blk \rfloor \quad \mu' = \mu\{id \leftarrow [blk.0, blk.(sizeof\ typ \times v))\}\end{array}}{mod,g,\theta \vdash M,MM,((fid,l,(c_0,\overline{c}),tmn,\Delta,\mu,\alpha),\overline{\hat{\Sigma}}) \twoheadrightarrow M',MM,((fid,l,\overline{c},tmn,\Delta\{id \leftarrow \{blk.0\}\},\mu',\alpha),\overline{\hat{\Sigma}})}\ \text{SB\_MALLOC}$$

$$\frac{\begin{array}{l}\textbf{eval}_{ND}(g,\Delta,val) = \lfloor V \rfloor \quad v \in V \quad c_0 = (id = \textbf{load}\,(typ*)val\, align) \\ \textbf{findbounds}(g,\mu,val) = \lfloor md \rfloor \quad \textbf{checkbounds}(typ,v,md) \quad \textbf{load}\,(M,typ,v,align) = \lfloor v' \rfloor \\ \text{if}\,\textbf{isPtrTyp}\, typ\, \text{then}\, \mu' = \mu\{id \leftarrow \textbf{findbounds}\,(MM,v)\}\, \text{else}\, \mu' = \mu\end{array}}{mod,g,\theta \vdash M,MM,((fid,l,(c_0,\overline{c}),tmn,\Delta,\mu,\alpha),\overline{\hat{\Sigma}}) \twoheadrightarrow M,MM,((fid,l,\overline{c},tmn,\Delta\{id \leftarrow \{|v'|\}\},\mu',\alpha),\overline{\hat{\Sigma}})}\ \text{SB\_LOAD}$$

$$\frac{\begin{array}{l}\textbf{eval}_{ND}(g,\Delta,val_1) = \lfloor V_1 \rfloor \quad v_1 \in V_1 \quad \textbf{eval}_{ND}(g,\Delta,val_2) = \lfloor V_2 \rfloor \quad v_2 \in V_2 \\ c_0 = (\textbf{store}\, typ\, val_1\, val_2\, align) \quad \textbf{findbounds}(g,\mu,val_2) = \lfloor md \rfloor \quad \textbf{checkbounds}(typ,v_2,md) \\ \textbf{store}\,(M,typ,v_1,v_2,align) = \lfloor M' \rfloor \quad \text{if}\,\textbf{isPtrTyp}\, typ\, \text{then}\, MM' = MM\{v_2 \leftarrow md\}\, \text{else}\, MM' = MM\end{array}}{mod,g,\theta \vdash M,MM,((fid,l,(c_0,\overline{c}),tmn,\Delta,\mu,\alpha),\overline{\hat{\Sigma}}) \twoheadrightarrow M',MM',((fid,l,\overline{c},tmn,\Delta,\mu,\alpha),\overline{\hat{\Sigma}})}\ \text{SB\_STORE}$$

Deterministic configurations:

$$\text{Frames}\,\hat{\sigma} \ ::= \ fid,l,\overline{c},tmn,\delta,\mu,\alpha \quad \text{Call stacks}\,\overline{\hat{\sigma}} \ ::= \ [] \mid \hat{\sigma},\overline{\hat{\sigma}} \quad \text{Program states}\,\hat{s} \ ::= \ M,MM,\overline{\hat{\sigma}}$$

Figure 7.1: SBspec: The specification semantics for SoftBound. Differences from the LLVM$_{ND}$ rules are highlighted.

**The SoftBound specification** Figure 7.1 gives the program configurations and representative rules for the SBspec semantics. SBspec behaves the same as the standard semantics except that it creates, propagates, and checks metadata of pointers in the appropriate instructions.

A program state $\hat{S}$ is an extension of the standard program state $S$ for maintaining metadata *md*, which is a pair defining the start and end address for a pointers: $\mu$ in each function frame $\hat{\Sigma}$ maps temporaries of pointer type to their metadata; *MM* is the shadow heap that stores metadata for pointers in memory. Note that although the specification is nondeterministic, the metadata is deterministic. Therefore, a pointer loaded from uninitialized memory space can be **undef**, but it cannot have arbitrary *md* (which might not be valid).

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Metadata | $md$ | $::=$ | $[v_1, v_2)$ | Memory metadata | $MM$ | $::=$ | $blk.ofs \mapsto md$ |
| Frames | $\hat{\Sigma}$ | $::=$ | $fid, l, \overline{c}, tmn, \Delta, \mu, \alpha$ | Call stacks | $\overline{\hat{\Sigma}}$ | $::=$ | $[] \mid \hat{\Sigma}, \overline{\hat{\Sigma}}$ |
| Local metadata | $\mu$ | $::=$ | $id \mapsto md$ | Program states | $\hat{S}$ | $::=$ | $M, MM, \overline{\hat{\Sigma}}$ |

SBspec is correct if a program *P* must either abort on detecting a spatial memory violation with respect to the SBspec, or preserve the LLVM semantics of the original program *P*; and, moreover, *P* is not stuck by any spatial memory violation in the SBspec (*i.e.,* SBspec must catch *all* spatial violations).

**Definition 6** (Spatial safety)**.** *Accessing a memory location at the offset ofs of a block blk is spatially safe if blk is less than the next fresh block N, and ofs is within the bounds of blk:*

$$blk < N \wedge (B(blk) = \lfloor size \rfloor \rightarrow 0 \leq ofs < size)$$

The legal stuck states of SoftBound—$\text{Stuck}_{SB}(config, \hat{S})$ include all legal stuck states of $\text{LLVM}_{ND}$ (recall Section 6.4.3) *except* the states that violate spatial safety. The case when *B* does not map *blk* to some size indicates that *blk* is not valid, and pointers into the *blk* are dangling—this indicates a temporal safety error that is not prevented by SoftBound and therefore it is included in the set of legal stuck states.

Because the program states of a program in the $\text{LLVM}_{ND}$ semantics are identical to the corresponding parts in the SBspec, it is easy to relate them: let $\hat{S} \supseteq^{\circ} S$ mean that common parts of the SoftBound state $\hat{S}$ and $S$ are identical. Because memory instructions in the SBspec may abort without accessing memory, the first part of correctness is by a straightforward simulation relation between states of the two semantics.

**Theorem 20** (SBspec simulates $\text{LLVM}_{ND}$)**.** *If the state $\hat{S} \supseteq^{\circ} S$, and $config \vdash \hat{S} \twoheadrightarrow \hat{S}'$, then there exists a state $S'$, such that $config \vdash S \twoheadrightarrow S'$, and $\hat{S}' \supseteq^{\circ} S'$.*

The second part of the correctness is proved by the following *preservation* and *progress* theorems.

**Theorem 21** (Preservation for SBspec)**.**

*If (config, $\hat{S}$) is well formed, and config $\vdash \hat{S} \twoheadrightarrow \hat{S}'$, then (config, $\hat{S}'$) is well formed.*

Here, SBspec well-formedness strengthens the invariants for $\text{LLVM}_{ND}$ by requiring that if any *id* defined in $\Delta$ is of pointer type, then $\mu$ contains its metadata and a *spatial safety invariant*: all bounds in $\mu$s of function frames and *MM* must be memory ranges within which all memory addresses are spatially safe.

The interesting part is proving that the spatial safety invariant is preserved. It holds initially, because a program's initial frame stack is empty, and we assume that *MM* is also empty. The other cases depend on the rules in Figure 7.1.

The rule SB_MALLOC, which allocates the number *v* of elements with *typ* at a memory block *blk*, updates the metadata of *id* with the start address that is the beginning of *blk*, and the end address that is at the offset $blk.(sizeof\, typ \times v)$ in the same block. LLVM's memory model ensures that the range of memory is valid.

The rule SB_LOAD reads from a pointer *val* with runtime data *v*, finds the *md* of the pointer, and ensures that *v* is within the *md* via **checkbounds**. If the *val* is an identifier, **findbounds** simply returns the identifier's metadata from $\mu$, which must be a spatial safe memory range. If *val* is a constant of pointer type, **findbounds** returns bounds as the following. For global pointers, **findbounds** returns bounds derived from their types because globals must be allocated before a program starts. For pointers converted from some constant integers by **inttoptr**, it conservatively returns the bounds $[\textbf{null}, \textbf{null})$ to indicate a potentially invalid memory range. For a pointer $cnst_1$ derived from an other constant pointer $cnst_2$ by **bitcase** or **getelementptr**, **findbounds** returns the same bound of $cnst_2$ for $cnst_1$. Note that $\{|v'|\}$ denotes conversion from a deterministic value to a nondeterministic value.

If the load reads a pointer-typed value *v* from memory, the rule finds its metadata in *MM* and updates the local metadata mapping $\mu$. If *MM* does not contain any metadata indexed by *v*, that means the pointer being loaded was not stored with valid bounds, so **findbounds** returns $[\textbf{null}, \textbf{null})$ to ensure the spatial safety invariant. Similarly, the rule SB_STORE checks whether the address to be stored to is in bounds and, if storing a pointer, updates *MM* accordingly. SoftBound disallows dereferencing a pointer that was converted from an integer, even if that integer was originally obtained from a valid pointer. Following the same design choice, **findbounds** returns $[\textbf{null}, \textbf{null})$ for pointers cast from integers. **checkbounds** fails when a program accesses such pointers.

**Theorem 22** (Progress for SBspec)**.** *If $\hat{S}_1$ is well-formed, then either $\hat{S}_1$ is a final state, or $\hat{S}_1$ is a legal stuck state, or there exists a $\hat{S}_2$ such that config $\vdash \hat{S}_1 \twoheadrightarrow \hat{S}_2$.*

This theorem holds because all the bounds in a well-formed SBspec state give memory ranges that are spatially safe, if **checkbounds** succeeds, the memory access must be spatially safe.

**The correctness of the SoftBound instrumentation** Given SBspec, we designed an instrumentation pass in Coq. For each function of an original program, the pass implements $\mu$ by generating two fresh temporaries for every temporary of pointer type to record its bounds. For manipulating metadata stored in *MM*, the pass axiomatizes a set of interfaces that manage a disjoint metadata space with specifications for their behaviors.

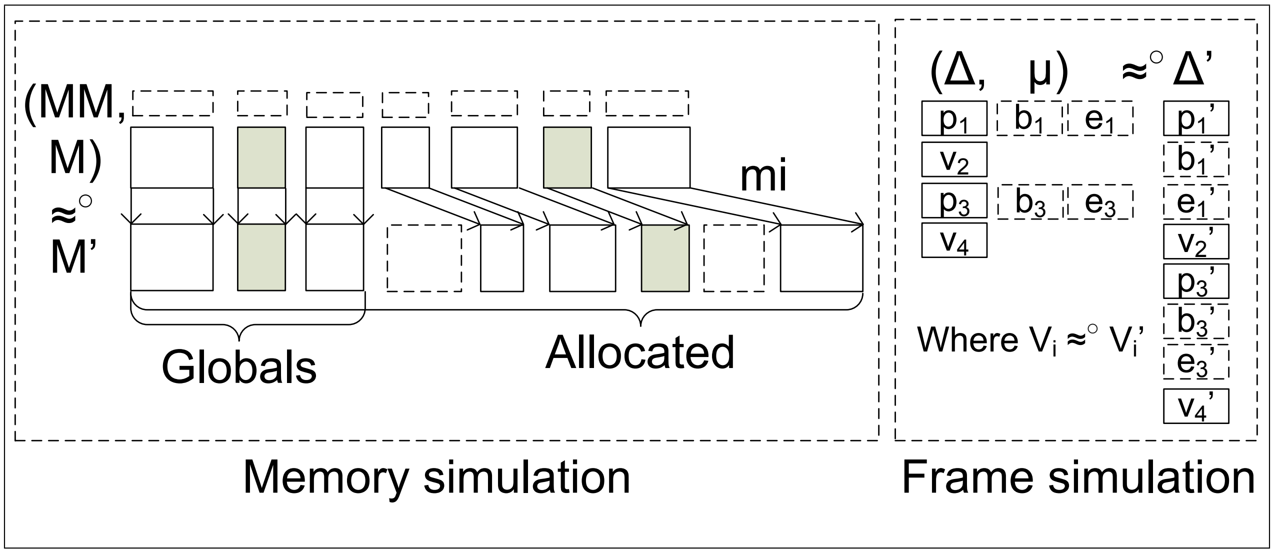


Figure 7.2: Simulation relations of the SoftBound pass

Figure 7.2 pictorially shows the simulation relations $\simeq^{\circ}$ between an original program *P* in the semantics of SBspec and its transformed program $P'$ in the LLVM semantics. First, because $P'$ needs additional memory space to store metadata, we need a mapping *mi* that maps each allocated memory block in *M* to a memory block in $M'$ without overlap, but allows $M'$ to have additional blocks for metadata, as shown in dashed boxes. Note that we assume the two programs initialize globals identically. Second, basic values are related in terms of the mapping between blocks: pointers are related if they refer to corresponding memory locations; other basic values are related if they are same. Two values are related if they are of the same length and the corresponding basic values are related.

Using the value simulations, $\simeq^{\circ}$ defines a simulation for memory and stack frames. Given two related memory locations *blk*.*ofs* and $blk'.ofs'$, their contents in *M* and $M'$ must be related; if *MM* maps *blk*.*ofs* to the bound $[v_1, v_2)$, then the additional metadata space in $M'$ must store $v_1'$ and $v_2'$ that relate to $v_1$ and $v_2$ for the location $blk'.ofs'$. For each pair of corresponding frames in the two stacks, $\Delta$ and $\Delta'$ must store related values for the same temporary; if $\mu$ maps a temporary *id* to the bound $[v_1, v_2)$, then $\Delta'$ must store the related bound in the fresh temporaries for the *id*.

**Theorem 23.** *Given a state $\hat{s}_1$ of $P$ with configuration config and a state $s'_1$ of $P'$ with configuration config$'$, if $\hat{s}_1 \simeq^\circ s'_1$, and config $\vdash \hat{s}_1 \longrightarrow \hat{s}_2$, then there exists a state $s'_2$, such that config$' \vdash s'_1 \longrightarrow^* s'_2$, $\hat{s}_2 \simeq^\circ s'_2$.*

Here, *config* $\vdash \hat{s}_1 \longrightarrow \hat{s}_2$ is a deterministic SBspec that, as in Section 6.4, is an instance of the non-deterministic SBspec.

**The correctness of SoftBound**

**Theorem 24** (SoftBound is correct)**.** *Let* SBtrans$(P) = \lfloor P' \rfloor$ *denote that the SoftBound pass instruments a well-formed program $P$ to be $P'$. A SoftBound instrumented program $P'$ either aborts on detecting spatial memory violations or preserves the LLVM semantics of the original program $P$. $P'$ is not stuck by any spatial memory violation.*

## 7.2 Extracted Verified Implementation of SoftBound

The above formalism not only shows that the SoftBound transformation enforces the promised safety properties, but the Vellvm framework allows us to extract a translator directly from the Coq code, resulting in a verified implementation of the SoftBound transformation. The extracted implementation uses the same underlying shadowspace implementation and wrapped external functions as the non-extracted SoftBound transformation written in C++. The only aspect not handled by the extracted transformation is initializing the metadata for pointers in the global segment that are non-NULL initialized (*i.e.,* they point to another variable in the global segment). Without initialization, valid programs can be incorrectly rejected as erroneous. Thus, we reuse the code from the C++ implementation of the SoftBound to properly initialize these variables.

**Effectiveness** To measure the effectiveness of the extracted implementation of SoftBound versus the C++ implementation, we tested both implementations on the same programs. To test whether the implementations detect spatial memory safety violations, we used 1809 test cases from the NIST Juliet test suite of C/C++ codes [53]. We chose the test cases which exercised the buffer overflows on both the heap and stack. Both implementations of SoftBound correctly detected all the buffer overflows without any false violations.

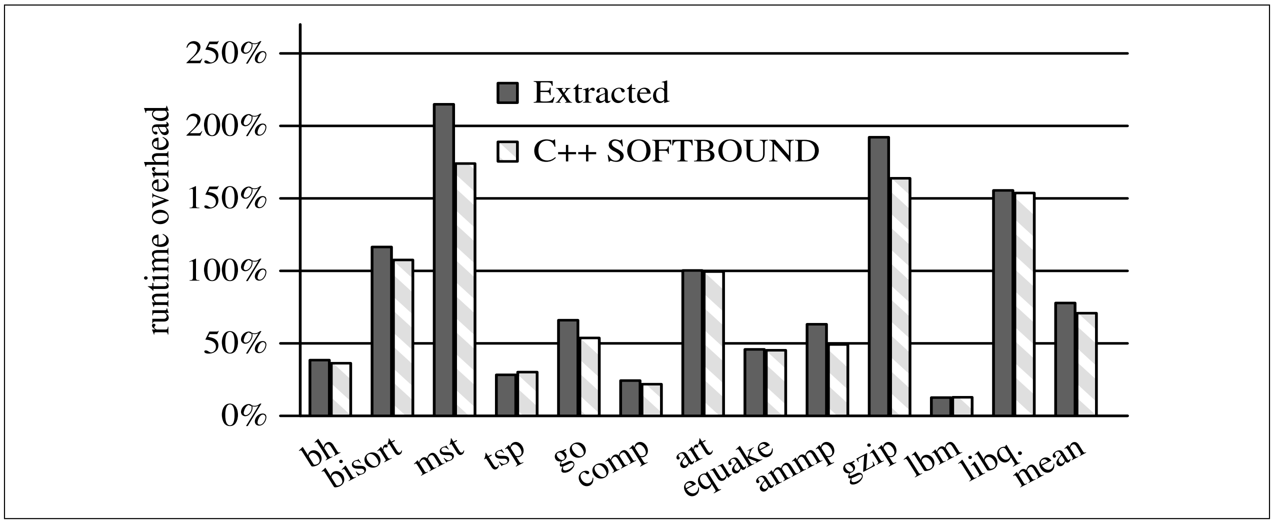


Figure 7.3: Execution time overhead of the extracted and the C++ version of SoftBound

We also confirmed that both implementations properly detected the buffer overflow in the go SPEC95 benchmark. Finally, the extracted implementation is robust enough to successfully transform and execute (without false violations) several applications selected from the SPEC95, SPEC2000, and SPEC2006 suites (around 110K lines of C code in total).

**Performance overheads** Unlike the C++ implementation of SoftBound that removes some obviously redundant checks, the extracted implementation of SoftBound performs no SoftBound-specific optimizations. In both cases, the same suite of standard LLVM optimizations are applied post-transformation to optimize the code to reduce the overhead of the instrumentation. To determine the performance impact on the resulting program, Figure **??** reports the execution time overheads (lower is better) of extracted SoftBound (leftmost bar of each benchmark) and the C++ implementation (rightmost bar of each benchmark) for various benchmarks from SPEC95, SPEC2000 and SPEC2006. Because of the check elimination optimization performed by the C++ implementation, the code is slightly faster, but overall the extracted implementation provides similar performance.

**Bugs found in the original SoftBound implementation** In the course of formalizing the SoftBound transformation, we discovered two implementation bugs in the original C++ implementation of SoftBound. First, when one of the incoming values of a $\phi$ node with pointer type is an **undef**, **undef** was propagated as its base and bound. Subsequent compiler transformations may instantiate the undefined base and bound with defined values that allow the **checkbounds** to succeed, which would lead to memory violation. Second, the base and bound of constant pointer $(typ*)\,\mathbf{null}$ was set to be $(typ*)\,\mathbf{null}$ and $(typ*)\,\mathbf{null} + sizeof(typ)$, allowing dereferences of **null** or pointers pointing to an offset from **null**. Either of these bugs could have resulted in faulty checking and thus expose the program to the spatial violations that SoftBound was designed to prevent. These bugs underscore the importance of a formally verified and extracted implementation to avoid such bugs.

# Chapter 8

# Verified SSA Construction for LLVM

Chapter 5 described the proof techniques we have developed for verifying SSA-based program transformations in the context of Vminus. This chapter demonstrates that these proof techniques can be used for practical compiler optimizations in Vellvm: verifying the most performance-critical optimization pass in LLVM's compilation strategy—the `mem2reg` pass.

## 8.1 The `mem2reg` Optimization Pass

LLVM provides a large suite of optimization passes, including aggressive dead code elimination (ADCE), global value numbering (GVN), partial redundancy elimination (PRE), and sparse conditional constant propagation (SCCP) among others. Figure 2.3 shows the tool chain of the LLVM compiler. Each transformation pass consumes and produces code in this SSA form, and they typically have the flavor of the code transformations described above in Chapter 5.

A critical piece of LLVM's compilation strategy is the `mem2reg` pass, which takes code that is "trivially" in SSA form and converts it into a *minimal, pruned* SSA program [62]. This strategy simplifies LLVM's many front ends by moving work in to `mem2reg`. An SSA form is "minimal" if each $\phi$ is placed only at the dominance frontier of the definitions of the $\phi$ node's incoming variables [28]. A minimal SSA form is "pruned" if it contains only live $\phi$ nodes [62]. This pass enables many subsequent optimizations (and, in particular, backend optimizations such as register allocation) to work effectively.

Figure 8.2 demonstrates the importance of the `mem2reg` pass for LLVM's generated code performance. In our experiments, running *only* the `mem2reg` pass yields a 81% speedup (on average) compared to LLVM

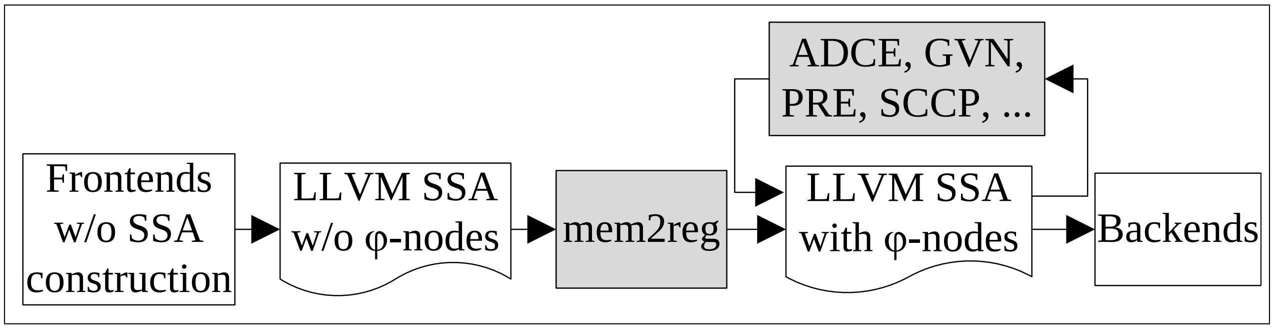


Figure 8.1: The tool chain of the LLVM compiler

without any optimizations; doing the full suite of `-O1` level optimizations (which includes `mem2reg`) yields a speedup of 102%, which means that `mem2reg` alone captures all but %12 of the benefit of the `-O1` level optimizations. Comparison with `-O3` optimizations yields similar results. These observations make `mem2reg` an obvious target for our verification efforts.

The "trivial" SSA form is generated directly by compiler front ends, and it uses the **alloca** instruction to allocate stack space for *every* source-program local variable and temporary needed. In this form, an LLVM SSA variable is used either only locally to access those stack slots, in which case the variable is never live across two basic blocks, or it is a reference to the stack slot, whose lifetime corresponds to the source-level variable's scope. These constraints mean that no $\phi$ instructions are needed—it is extremely straightforward for a front end to generate code in this form.

As an example, consider this C program (which is a running example through this chapter):

```
int i = 0;
while (i<=100) i++;
return i;
```

The "trivial" SSA form that might be produced by the frontend of a compiler is shown in the left-most column of Figure 8.4 and Figure 8.5. The $r_0 := \textbf{alloca}\,\textbf{int}$ instruction on the first line allocates space for the source variable `i`, and $r_0$ is a reference from which local **load** and **store** instructions access `i`'s contents.

The `mem2reg` pass converts *promotable* uses of stack-allocated variables to SSA temporaries.

**Definition 7** (Promotable allocations). *An allocation r is promotable in f, written* $\textbf{promotable}\,(f,r)$*, if* $r := \textbf{alloca}\,typ$ *is in the entry block of f, and r does not escape (r is not stored into memory;* $\forall insn \in f, insn\,\textbf{uses}\,r \Longrightarrow$ *insn is a* **store** *or* **load***).*

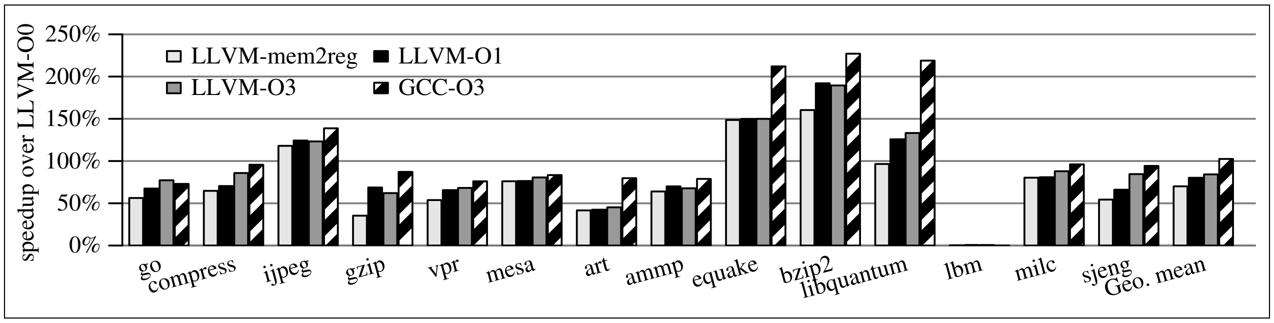


Figure 8.2: Normalized execution time improvement of the LLVM's mem2reg, LLVM's O1, and LLVM's O3 optimizations over the LLVM baseline with optimizations disabled. For comparison, GCC-O3's speedup over the same baseline is also shown.

An **alloca**'ed variable like $r_0$ is considered to be promotable if it is created in the entry block of function $f$ and it doesn't escape—*i.e.,* its value is never written to memory or passed as an argument to a function call. The `mem2reg` pass identifies promotable stack allocations and then replaces them by temporary variables in SSA form. It does this by placing $\phi$ nodes, substituting each variable defined by a **load** with the previous value stored into the stack slot, and then eliminating the memory operations (which are now dead). The right-most column of Figure 8.5 shows the resulting pruned SSA program for this example. The `mem2reg` algorithm can also be viewed as a restricted version of a transformation that considers a general register promotion problem by using sophisticated alias analysis and partial redundant elimination of loads and stores to make more locations promotable [44].

Algorithm 8.3 shows the algorithm that the LLVM `mem2reg` pass uses, and Figure 8.4 gives an example of the algorithm. The code on the left most of Figure 8.4 is the output of a front-end that compiles mutable variables of the non-SSA form to stack allocations, and is in the SSA form trivially. The first step of the `mem2reg` algorithm is to find all stack allocations (stored at *Allocas*) that can be promoted to temporaries by the function FINDPROMOTABLEALLOCAS that simply checks if the front-end follows the contract with LLVM—only the allocations in the entry block (returned by ENTRYOF) are candidates; stack allocations for mutable variables can only be used by **store** and **load**, and not written into memory. For example, $r_0$ is promotable. Note that promoting such allocations to temporaries is definitely safe for programs that do not have undefined behaviors, such as out-of-bound accessing, using dangling pointers, reading from uninitialized memory locations, *etc.*; on the other hand, the transformation is also correct for programs that violate these assumptions, because they can be of any behavior.

After finding all promotable allocations, the `mem2reg` algorithm applies the variant of the standard SSA construction. It first inserts minimal number of $\phi$ nodes by PHINODESPLACEMENT. The $\phi$-node placement

```
$A \leftarrow \emptyset$
function FINDPROMOTABLEALLOCAS($f$)
    for all $r :=$ alloca $typ \in$ ENTRYOF($f$) do
        if ISPROMOTABLE($f$, $r$) then
            $A \leftarrow A \cup \{r\}$
        end if
    end for
end function
function RENAME($f$, $l$, Vmap)
    $\lfloor l \overline{\phi} \overline{c}\, tmn \rfloor = f[l]$
    for all $\phi \in \overline{\phi}$ do
        if $\phi$ is placed for an $r \in A$ then
            Vmap[$r$] = GETID($\phi$)
        end if
    end for
    for all $c \in \overline{c}$ do
        if $c = r' :=$ load $(typ*)\, r$ and $r \in A$ then
            REPLACEALLUSES($f$, $r'$, Vmap[ $r$ ])
            REMOVE($f$, $c$)
        else if $c =$ store $typ\; val\; r$ and $r \in A$ then
            Vmap[$r$] = $val$
            REMOVE($f$, $c$)
        end if
    end for
    for all successor $l'$ of $l$ do
        $\lfloor l' \overline{\phi}' \overline{c}'\, tmn' \rfloor = f[l']$
        for all $\phi' \in \overline{\phi}'$ do
            if $\phi'$ is placed for promotion then
                SUBSTITUTION($f$, Vmap, $\phi'$, $l$)
            end if
        end for
    end for
    for all child $l'$ of $l$ do
        RENAME($f$, $l'$, Vmap)
    end for
end function
function MEM2REG($f$)
    FINDPROMOTABLEALLOCAS($f$)
    PHINODESPLACEMENT($f$)
    RENAME($f$, ENTRYOF($f$), INITVMAP())
    for all $r \in A$ and $r$ is not used do
        REMOVE($f$, $r$)
    end for
end function
```

Figure 8.3: The algorithm of `mem2reg`

algorithm avoids computing dominance frontiers explictly by using a data-structure called DJ-graphs [62], so is very fast in practice. We omitted its detail in the presentation. The second code in Figure 8.4 is the code after $\phi$ nodes placement. In this case, the algorithm only needs to place $r_6 = \textbf{phi}\,[r_0, l_1][r_0, l_3]$ at the beginning of block $l_2$. Note that after the replacement, the code is not well-formed because $r_6$ is expected to be of type **int**, while all its coming values are of type **int**$*$. The later pass RENAME will incrementally recover the well-formedness, and eventually makes the final program simulates the behavior of the original program.

The RENAME follows the structure of the classic renaming algorithm [8], but also does redundant memory operation eliminations, and constant propagation in the mean while. The algorithm follows dominator tree rooted by the entry block—not the flow graph, and also maintains a map *Vmap* in which for each promotable variable $r$, $Vmap[r]$ is the its most recently value with respect to the dominator tree of the function $f$. Initially, INITVMAP sets the most recently value to be the default value that **alloca** assigns for allocated memory; the depth-first-recursion starts from the entry block.

| Before `mem2reg` | $\phi$ nodes placement | Renamed $l_1$ | Renamed $l_1$ $l_2$ and $l_3$ | After `mem2reg` |
|---|---|---|---|---|
| $l_1 : r_0 := \mathbf{alloca\,int}$ | $l_1 : r_0 := \mathbf{alloca\,int}$ | $l_1 : r_0 := \mathbf{alloca\,int}$ | $l_1 : r_0 := \mathbf{alloca\,int}$ | $l_1 :$ |
| $\quad\mathbf{store\,int}\,0\,r_0$ | $\quad\mathbf{store\,int}\,0\,r_0$ | | | |
| $\quad\mathbf{br}\,l_2$ | $\quad\mathbf{br}\,l_2$ | $\quad\mathbf{br}\,l_2$ | $\quad\mathbf{br}\,l_2$ | $\quad\mathbf{br}\,l_2$ |
| $l_2 :$ | $l_2 : r_6 = \mathbf{phi}\,[r_0, l_1][r_0, l_3]$ | $l_2 : r_6 = \mathbf{phi}\,[0, l_1][r_0, l_3]$ | $l_2 : r_6 = \mathbf{phi}\,[0, l_1][r_4, l_3]$ | $l_2 : r_6 = \mathbf{phi}\,[0, l_1][r_4, l_3]$ |
| $\quad r_1 := \mathbf{load}\,(\,\mathbf{int}*\,)\,r_0$ | $\quad r_1 := \mathbf{load}\,(\,\mathbf{int}*\,)\,r_0$ | $\quad r_1 := \mathbf{load}\,(\,\mathbf{int}*\,)\,r_0$ | | |
| $\quad r_2 := r_1 \leq 100$ | $\quad r_2 := r_1 \leq 100$ | $\quad r_2 := r_1 \leq 100$ | $\quad r_2 := r_6 \leq 100$ | $\quad r_2 := r_6 \leq 100$ |
| $\quad\mathbf{br}\,r_2\,l_3\,l_4$ | $\quad\mathbf{br}\,r_2\,l_3\,l_4$ | $\quad\mathbf{br}\,r_2\,l_3\,l_4$ | $\quad\mathbf{br}\,r_2\,l_3\,l_4$ | $\quad\mathbf{br}\,r_2\,l_3\,l_4$ |
| $l_3 : r_3 := \mathbf{load}\,(\,\mathbf{int}*\,)\,r_0$ | $l_3 : r_3 := \mathbf{load}\,(\,\mathbf{int}*\,)\,r_0$ | $l_3 : r_3 := \mathbf{load}\,(\,\mathbf{int}*\,)\,r_0$ | $l_3 :$ | $l_3 :$ |
| $\quad r_4 := r_3 + 1$ | $\quad r_4 := r_3 + 1$ | $\quad r_4 := r_3 + 1$ | $\quad r_4 := r_6 + 1$ | $\quad r_4 := r_6 + 1$ |
| $\quad\mathbf{store\,int}\,r_4\,r_0$ | $\quad\mathbf{store\,int}\,r_4\,r_0$ | $\quad\mathbf{store\,int}\,r_4\,r_0$ | | |
| $\quad\mathbf{br}\,l_2$ | $\quad\mathbf{br}\,l_2$ | $\quad\mathbf{br}\,l_2$ | $\quad\mathbf{br}\,l_2$ | $\quad\mathbf{br}\,l_2$ |
| $l_4 : r_5 := \mathbf{load}\,(\,\mathbf{int}*\,)\,r_0$ | $l_4 : r_5 := \mathbf{load}\,(\,\mathbf{int}*\,)\,r_0$ | $l_4 : r_5 := \mathbf{load}\,(\,\mathbf{int}*\,)\,r_0$ | $l_4 : r_5 := \mathbf{load}\,(\,\mathbf{int}*\,)\,r_4$ | $l_4 :$ |
| $\quad\mathbf{ret\,int}\,r_5$ | $\quad\mathbf{ret\,int}\,r_5$ | $\quad\mathbf{ret\,int}\,r_5$ | $\quad\mathbf{ret\,int}\,r_5$ | $\quad\mathbf{ret\,int}\,r_4$ |

Figure 8.4: The SSA construction by the `mem2reg` pass

At each visited block $l\overline{\phi}\overline{c}tmn$, the algorithm first checks if there is any $\phi$ placed for a promotable temporary $r$. If so, the algorithm takes the temporary defined by the $\phi$ as the most recent value for $r$ in the map $Vmap$. Then, for each command $c$, if $c$ is a load from a promotable temporary $r$ to $r'$, then the algorithm replaces all the uses of $r'$ by the most recent value of $r$ stored in $Vmap$, then remove the $c$; if $c$ is a store to a promotable temporary $r$ with a value $val$, then the algorithm sets $val$ to be the most recent value for $r$, then removes the $c$; otherwise, the algorithm does nothing. At the end, it examines all the successors (in term of the control-flow graph) of $l$ to see if there are any $\phi$ nodes whose operands need to be properly renamed, and then recursively renames all children blocks (in term of the dominator tree) of $l$.

After the renaming of block $l_1$, the store **store int** $0\, r_0$ in block $l_1$ was removed; because at the end of block $l_1$ the recent value of $r_1$ is 0 that is from the removed store, in the $\phi$ of $l_2$ that is the successor of $l_1$, the algorithm replaced the $r_0$ corresponding to $l_1$ by 0. The next code in Figure 8.4 shows the depth-first-search-based renaming up to one leaf of the dominator tree when all the blocks $l_1$, $l_2$ and $l_3$ were renamed. Note that the algorithm does not change the incoming value of the $\phi$ node in block $l_2$ when RENAME visited $l_2$, but changed the $r_0$ of the incoming block $l_3$ to be $r_4$ when RENAME visited the end of the block $l_3$ whose successor is $l_2$. The other observation is that although the code is well-formed, it does not preserve the meaning of its original program because the value of $r_5$ is read from the uninitialized location $r_0$, while in the original program $r_5$ should be 100 at the return of the program.

After renaming, the last step of the `mem2reg` pass is checking if there is any promotable temporaries $r$ which is not used at all, and, therefore, can be safely removed. As shown in the right most code of Figure 8.4, renaming the block $l_4$ removed the load in block $l4$, and then the $l_0$ is not used any more, and was removed. At this point, the code is not only well-formed, but also preserves the semantics of the original code by returning the same final result 100.

Proving that `mem2reg` is correct is nontrivial because it makes significant, non-local changes to the use of memory locations and temporary variables. Furthermore, the specific `mem2reg` algorithm used by LLVM is not directly amenable to the proof techniques developed in Chapter 5—it was not designed with verification in mind, so it produces intermediate stages that break the SSA invariants or do not preserve semantics. The next section therefore describes an alternate algorithm that is more suitable to formalization.

| Before `vmem2reg` | Maximal $\phi$ nodes placement | After LAS/LAA/SAS | After DSE/DAE | After $\phi$ nodes elimination |
|---|---|---|---|---|
| $l_1 : r_0 :=$ **alloca int** | $l_1 : r_0 :=$ **alloca int** | $l_1 : r_0 :=$ **alloca int** | $l_1 :$ | $l_1 :$ |
| **store int** $0\, r_0$ | **store int** $0\, r_0$ | **store int** $0\, r_0$ | | |
| | $r_7 :=$ **load** ( **int** $*$ ) $r_0$ | | | |
| **br** $l_2$ | **br** $l_2$ | **br** $l_2$ | **br** $l_2$ | **br** $l_2$ |
| $l_2 :$ | $l_2 : r_6 =$ **phi** $[r_7, l_1][r_9, l_3]$ | $l_2 : r_6 =$ **phi** $[0, l_1][r_9, l_3]$ | $l_2 : r_6 =$ **phi** $[0, l_1][r_4, l_3]$ | $l_2 : r_6 =$ **phi** $[0, l_1][r_4, l_3]$ |
| | **store int** $r_6\, r_0$ | **store int** $r_6\, r_0$ | | |
| $r_1 :=$ **load** ( **int** $*$ ) $r_0$ | $r_1 :=$ **load** ( **int** $*$ ) $r_0$ | | | |
| $r_2 := r_1 \leq 100$ | $r_2 := r_1 \leq 100$ | $r_2 := r_6 \leq 100$ | $r_2 := r_6 \leq 100$ | $r_2 := r_6 \leq 100$ |
| | $r_8 :=$ **load** ( **int** $*$ ) $r_0$ | | | |
| **br** $r_2\, l_3\, l_4$ | **br** $r_2\, l_3\, l_4$ | **br** $r_2\, l_3\, l_4$ | **br** $r_2\, l_3\, l_4$ | **br** $r_2\, l_3\, l_4$ |
| $l_3 :$ | $l_3 : r_{10} =$ **phi** $[r_8, l_2]$ | $l_3 : r_{10} =$ **phi** $[r_6, l_2]$ | $l_3 : r_{10} =$ **phi** $[r_6, l_2]$ | $l_3 :$ |
| | **store int** $r_{10}\, r_0$ | | | |
| $r_3 :=$ **load** ( **int** $*$ ) $r_0$ | $r_3 :=$ **load** ( **int** $*$ ) $r_0$ | | | |
| $r_4 := r_3 + 1$ | $r_4 := r_3 + 1$ | $r_4 := r_{10} + 1$ | $r_4 := r_{10} + 1$ | $r_4 := r_6 + 1$ |
| **store int** $r_4\, r_0$ | **store int** $r_4\, r_0$ | **store int** $r_4\, r_0$ | | |
| | $r_9 :=$ **load** ( **int** $*$ ) $r_0$ | | | |
| **br** $l_2$ | **br** $l_2$ | **br** $l_2$ | **br** $l_2$ | **br** $l_2$ |
| $l_4 :$ | $l_4 : r_{11} =$ **phi** $[r_8, l_2]$ | $l_4 : r_{11} =$ **phi** $[r_6, l_2]$ | $l_4 : r_{11} =$ **phi** $[r_6, l_2]$ | $l_4 :$ |
| | **store int** $r_{11}\, r_0$ | **store int** $r_{11}\, r_0$ | | |
| $r_5 :=$ **load** ( **int** $*$ ) $r_0$ | $r_5 :=$ **load** ( **int** $*$ ) $r_0$ | | | |
| **ret int** $r_5$ | **ret int** $r_5$ | **ret int** $r_{11}$ | **ret int** $r_{11}$ | **ret int** $r_6$ |

Figure 8.5: The SSA construction by the `vmem2reg` pass

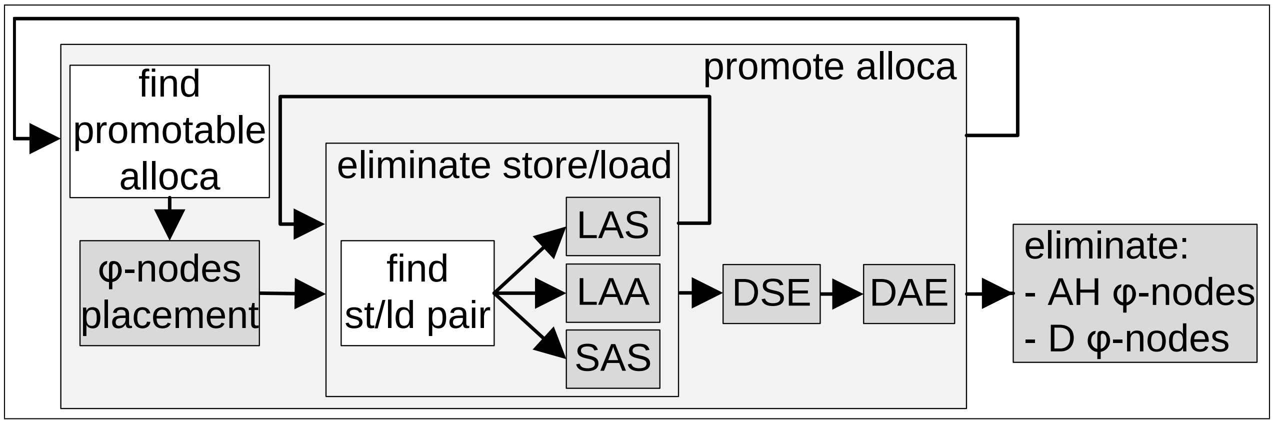


Figure 8.6: Basic structure of `vmem2reg_fn`

## 8.2 The `vmem2reg` Algorithm

This section presents `vmem2reg`, an SSA algorithm that is structured to lead to a clean formalism and yet still produce programs with effectiveness similar to the LLVM `mem2reg` pass. To demonstrate the main ideas of `vmem2reg`, this section describes an algorithm that uses straightforward micro-pass pipelining. Section 8.5 presents a smarter way to "fuse" the micro passes, thereby reducing compilation time. Proving pipeline fusion correct is (by design) independent of the proofs for the `vmem2reg` algorithm shown in the section.

At a high level, `vmem2reg` (whose code is shown in Figure 8.7) traverses all functions of the program, applying the transformation `vmem2reg_fn` to each. Figure 8.6 depicts the main loop, which is an extension of Aycock and Horspool's SSA construction algorithm [12]. `vmem2reg_fn` first iteratively promotes each promotable **alloca** by adding $\phi$ nodes at the beginning of every block. After processing all promotable **alloca**s, `vmem2reg_fn` removes redundant $\phi$ nodes, and eventually will produce a program almost in pruned SSA form,[1] in a manner similar to previous algorithms [62].

The transformation that `vmem2reg_fn` applies to each function is a composition of a series of micro transformations (LAS, LAA, SAS, DSE, and DAE, shown in Figure 8.6). Each of these transformations preserves the well-formedness and semantics of its input program; moreover, these transformations are relatively small and local, and can therefore be reasoned about more easily.

At each iteration of **alloca** promotion, `vmem2reg_fn` finds a promotable allocation $r$. Then $\phi$-`nodes_placement` (code shown in Figure 8.7) adds $\phi$ nodes for $r$ at the beginning of every block. To preserve both well-formedness and the original program's semantics, $\phi$-`nodes_placement` also adds

[1]Technically, fully pruned SSA requires a more aggressive dead-$\phi$-elimination pass that we omit for the sake of simplicity. Section 8.4 shows that this omission has negligible impact on performance.

additional **load**s and **store**s around each inserted $\phi$ node. At the end of every block that has successors, `ϕ-nodes_placement` introduces a **load** from $r$, and stores the result in a fresh temporary; at the beginning of every block that has predecessor, `ϕ-nodes_placement` first inserts a fresh $\phi$ node whose incoming value from a predecessor $l$ is the value of the additional **load** we added at the end of $l$, then inserts a **store** to $r$ with the value of the inserted $\phi$ node.

The second column in Figure 8.5 shows the result of running the $\phi$-node placement pass starting from the example program in its trivial SSA form. It is not difficult to check that this code is in SSA form. Moreover, the output program also preserves the meaning of the original program. For example, at the end of block $l_1$, the program loads the value stored at $r_0$ into $r_7$. After jumping to block $l_2$, the value of $r_7$ is stored into the location $r_0$, which should contain the same values as $r_7$. Therefore, the additional store does not change the status of memory. Although the output program contains more temporaries than the original program, these temporaries are used only to connect inserted **load**s and **store**s, and so they do not interfere with the original temporaries.

To remove the additional **load**s and **store**s introduced by the $\phi$-node placement pass and eventually promote **alloca**s to registers, `vmem2reg_fn` next applies a series of micro program transformations until no more optimizations can be applied.

First, `vmem2reg_fn` iteratively does the following transformations (implemented by `eliminate_stld` shown in Figure 8.7):

1. `LAS` $(r_1, pc_2, val_2)$ "Load After Store": $r_1$ is **load**ed from $r$ after a store of $val_2$ to $r$ at program counter $pc_2$, and there are no other **store**s of $r$ in any path (on the control-flow graph) from $pc_2$ to $r_1$. In this case, all uses of $r_2$ can be replaced by $val_2$, and the **load** can be removed.

2. `LAA` $r_1$ "Load After Alloca": As above, but the load is from an uninitialized memory location at $r$. $r_1$ can be replaced by LLVM's default memory value, and the **load** can be removed.

3. `SAS` $(pc_1, pc_2)$: The **store** at program counter $pc_2$ is a store after the store at program counter $pc_1$. If both of them access $r$, and there is no **load** of $r$ in any path (on the control-flow graph) from $pc_1$ to $pc_2$, then the **store** at $pc_1$ can be removed.

At each iteration step of `eliminate_stld`, the algorithm uses the function `find_stld_pair` to identify each of the above cases. Because the $\phi$-node placement pass only adds a **store** and a **load** as the first and the last commands at each block respectively, `find_stld_pair` only needs to search for the above cases within

```
let vmem2reg $prog$ =
  map (function $f \rightarrow$ vmem2reg_fn $f$
             | $prod \rightarrow$  $prod$) $prog$
let rec eliminate_stld $f$ $r$ =
  match find_stld_pair $f$ $r$ with
  | LAS ($pc_2$, $val_2$, $r_1$) $\rightarrow$ eliminate_stld ($f\{val_2/r_1\} - r_1$) $r$
  | LAA $r_1$ $\rightarrow$ eliminate_stld ($f\{0/r_1\} - r_1$) $r$
  | SAS ($pc_1$, $pc_2$) $\rightarrow$ eliminate_stld ($f - pc_1$) $r$
  | NONE $\rightarrow$  $f$
  end
let $\phi$-nodes_placement $f$ $r$ =
  let $\textbf{define}\, typ\, fid(\overline{arg})\,\{\overline{b}\}$ = $f$ in
  let (ldnms, phinms) = gen_fresh_names $\overline{b}$ in
  $\textbf{define}\, typ\, fid(\overline{arg})\,\{$(map
    (function $l\overline{\phi}\,\overline{c}\,tmn \rightarrow$
     let $r := \textbf{alloca}\, typ \in f$ in
     let ($\overline{\phi}'$, $\overline{c}_1$) = match predecessors_of $f$ $l$ with
                   | [] $\rightarrow$ ($\overline{\phi}$, $\overline{c}$)
                   | $\overline{l_j}^j$ $\rightarrow$ let $\overline{r_j}^j$ = map (find ldnms) $\overline{l_j}^j$ in
                           let $r'$ = find phinms $l$ in
                           ($r' = \textbf{phi}\, typ\, \overline{[r_j, l_j]}^j :: \overline{\phi}$, $\textbf{store}\, typ\, r'\, r :: \overline{c}$)
                   end in
     let $\overline{c}'$ = match successors_of $f$ $l$ with
             | [] $\rightarrow \overline{c}_1$
             | _ $\rightarrow$ let $r'$ = find ldnms $l$ in $\overline{c}_1$ ++ [$r' := \textbf{load}\,(typ*)\,r$]
             end in
    $l\overline{\phi}'\,\overline{c}'\,tmn$) $\overline{b}$)$\}$
```

Figure 8.7: The algorithm of vmem2reg

blocks. This simplifies both the implementation and proofs. Moreover, eliminate_stld must terminate because each of its transformations removes one command. The third column in Figure 8.5 shows the code after eliminate_stld.

Next, the algorithm uses DSE (Dead Store Elimination) and DAE (Dead Alloca Elimination) to remove the remaining unnecessary **store**s and **alloca**s.

1. DSE "Dead Store Elimination": The **store** of $r$ at program counter $pc_1$ is dead—there is no **load** of $r$, so the **store** at $pc_1$ can be removed.

2. DAE "Dead Alloca Elimination": The allocation of $r$ is dead—there is no use of $r$, so the **alloca** can be removed.

The fourth column in Figure 8.5 shows the code after DSE and DAE.

Finally, `vmem2reg_fn` eliminates unnecessary and dead $\phi$ nodes [12]:

1. AH $\phi$-nodes [12]: if any $\phi$ is of the form $r = \textbf{phi}\, typ\, \overline{[val_j, l_j]}^j$ where all $val_j$ are either equal to $r$ or $val$, then all uses of $r$ can be replaced by $val$, and the $\phi$ can be removed. Aycock and Horspool [12] proved that when there is no such $\phi$ node in a reducible program, the program is of the minimal SSA form.

2. D $\phi$-nodes: if there is no any use of the $\phi$ node. Removing D $\phi$-nodes produces programs in nearly pruned SSA form.

The right-most column in Figure 8.5 shows the final output of the algorithm.

## 8.3 Correctness of `vmem2reg`

We prove the correctness of `vmem2reg` using the techniques developed in Chapter 5. At a high level, the correctness of `vmem2reg` is the composition of the correctness of each micro transformation of `vmem2reg` shown in Figure 8.7. Given a well-formed input program, each shaded box must produce a well-formed program that preserves the semantics of the input program. Moreover, the micro transformations except `DAE` and $\phi$-nodes elimination must preserve the **promotable** predicate (Definition 7), because the correctness of subsequent transformations relies on fact that promotable allocations aren't aliased.

Formally, let $prog\{f'/f\}$ be the substitution of $f$ by $f'$ in $prog$, and let $(\!|f|\!)$ be a micro transformation of $f$ applied by `vmem2reg`. $(\!|\_|\!)$ must satisfy:

1. Preserving **promotable**: when $(\!|\_|\!)$ is not `DAE` or $\phi$-nodes elimination, if **promotable** $(f, r)$, then **promotable** $((\!|f|\!), r)$.

2. Preserving well-formedness: if **promotable** $(f, r)$ when $(\!|\_|\!)$ is $\phi$-nodes placement, and $\vdash prog$, then $\vdash prog\{(\!|f|\!)/f\}$.

3. Program refinement: if **promotable** $(f, r)$ when $(\!|\_|\!)$ is not $\phi$-nodes elimination, and $\vdash prog$, then $prog \supseteq prog\{(\!|f|\!)/f\}$.

### 8.3.1 Preserving promotability

At the beginning of each iteration for promoting **alloca**s, the algorithm indeed finds promotable allocations.

**Lemma 25.** *If $prog \vdash f$, and* `vmem2reg_fn` *finds a promotable allocation $r$ in $f$, then* **promotable** $(f, r)$.

We next show that $\phi$-nodes placement preserves **promotable**:

**Lemma 26.** *If* **promotable** $(f, r)$,

*then* **promotable** $(\phi\text{–}nodes\_placement\ f\, r, r)$.

**P**roof (sketch): The $\phi$-nodes placement pass only inserts instructions. Therefore, if $r$ is in the entry block of the original function, $r$ is still in the entry block of the transformed one. Moreover, in the transformed function, the instructions copied from the original function use $r$ in the same way, the inserted **store**s only write fresh definitions into memory, and the $\phi$-nodes only use fresh definitions. Therefore, $r$ is still promotable after $\phi$-nodes placement. □

Each of the other micro transformations is composed of one or two more basic transformations: variable substitution, denoted by $f\{val/r\}$, and instruction removal, denoted by **filter check** $f$ where **filter** removes an instruction *insn* from $f$ if **check** $insn = \textbf{false}$. For example, $f\{val_2/r_1\} - r_1$ (`LAS`) is a substitution followed by a removal in which **check** $insn = \textbf{false}$ iff *insn* defines $r_1$; `DSE` of a promotable **alloca** $r$ is a removal in which **check** $insn = \textbf{false}$ iff *insn* is a store to $r$. We first establish that substitution and removal preserve **promotable**.

**Lemma 27.** *Suppose* **promotable** $(f, r)$,

1. *If* $\neg(val_1\ \textbf{uses}\ r)$, *then* **promotable** $(f\{val_1/r_1\}, r)$.

2. *If* **check** $insn = \textbf{false} \Rightarrow$ *insn does not define r, then* **promotable** $(\textbf{filter check}\ f, r)$.

We can show that the other micro transformations preserve **promotable** by checking the preconditions of Lemma 27.

**Lemma 28.** *Suppose* **promotable** $(f, r)$, *r is still* **promotable** *after* `LAS`, `LAA`, `SAS` *or* `DSE`.

The substituted value of `LAS` is written to memory by a **store** in $f$, which cannot use $r$ because $r$ is promotable in $f$. The substituted value of `LAA` is a constant that cannot use $r$ trivially. Moreover, `LAS`, `LAA`, `SAS` and `DSE` remove only **load**s or **store**s.

### 8.3.2 Preserving well-formedness

It is sufficient to check the following conditions to show that a function-level transformation preserves well-formedness:

**Lemma 29.** *Suppose*

1. $(\!|f|\!)$ *and* $f$ *have the same signature.*
2. *if* $prog \vdash f$*, then* $prog\{(\!|f|\!)/f\} \vdash (\!|f|\!)$.

*If* $\vdash prog$*, then* $\vdash prog\{(\!|f|\!)/f\}$.

It is easy to see that all transformations `vmem2reg` applies satisfy the first condition. We first prove that $\phi$-nodes placement preserves the second condition:

**Lemma 30.** *If* **promotable** $(f,r)$*,* $prog \vdash f$ *and let* $f'$ *be* $\phi$*–nodes_placement* $f\,r$*, then* $prog\{f'/f\} \vdash f'$.

**P**roof (sketch): Because $\phi$-nodes placement only inserts fresh definitions, and does not change control-flow graphs, dominance relations are preserved, and all the instructions from the original program are still well-formed after the transformation.

To show the well-formedness of the inserted instructions, we need to check that they satisfy the use/def properties of SSA. The inserted instructions only use $r$ or fresh definitions introduced by the pass. The well-formedness of $f$ ensures that 1) because $r$ is defined at the entry block, it must dominate the end of all blocks, and the beginning of all non-entry block; 2) the entry block has not predecessors. Therefore, the definition of $r$ must strictly dominate all its uses in the inserted **load**'s and **store**'s. The fresh variable used by each inserted **store** is well-formed because its definition is by an inserted $\phi$-node in the same block of the **store**, and must strictly dominate its use in the **store**. The incoming variables used by each $\phi$-node is well-formed because they are all defined at the end of the corresponding incoming blocks. □

Similarly, to reason about other transformations, we first establish that substitution and removal preserve well-formedness.

**Lemma 31.** *Suppose* $prog \vdash f$*,*

1. *If* $f \vdash val_1 \gg r_2$*,* $f' = f\{val_1/r_2\}$*, then* $prog\{f'/f\} \vdash f'$.
2. *If* **check** $insn =$ **false** $\Rightarrow$ $f$ *does not use insn, and let* $f'$ *be* **filter check** $f$*, then* $prog\{f'/f\} \vdash f'$.

Here, $f \vdash val_1 \gg r_2$ if $f \vdash r_1 \gg r_2$ when $val_1$ **uses** $r_1$. Note that the first part of Lemma 31 is an extension of Lemma 15 that only allows substitution on commands. In `vmem2reg`, `LAS` and $\phi$-nodes elimination may transform $\phi$-nodes.

LAS, LAA and $\phi$-nodes elimination remove instructions after substitution. The following auxiliary lemma shows that the substituted definition is removable after substitution:

**Lemma 32.** *If $f \vdash val_1 \gg r_2$, then $f\{val_1/r_2\}$ does not use $r_2$.*

This lemma holds because $val_1$ cannot use $r_2$ by Lemma 7.

**Lemma 33.** LAS*,* LAA*,* SAS*,* DSE*,* DAE *and $\phi$-nodes elimination preserve well-formedness.*

**P**roof (sketch): Most of the proofs follow Lemma 31 and Lemma 32. The interesting case is showing that if a $\phi$-node in $f$ is of the form $r = \mathbf{phi}\, typ\, \overline{[val_j, l_j]}^{j}$ where all $val_j$ are either equal to $r$ or $val'$ (which is an AH $\phi$-node [12]), then $f \vdash val' \gg r$.

It is trivial if $val'$ is a constant. Suppose $val'$ **uses** $r'$, $r$ and $r'$ are defined in $l$ and $l'$ respectively. We first have that $r = \mathbf{phi}\, typ\, \overline{[r_j, l]}^{j}$ is not well-formed. Suppose such a $\phi$-node is well-formed. The well-formedness of the $\phi$-node ensures that the definition of $r_j$ dominates the end of all $l$'s predecessors. Therefore, $l$ strictly dominates itself. This is a contradiction by Lemma 7.

By the above result, $r'$ cannot be $r$, and $l'$ cannot be $l$. Suppose $\neg f \vdash r' \gg r$. There must exist a simple path (which has no cycles) from the entry to $l$ that bypasses $l'$. The simple path must visit one of $l$'s predecessors. The predecessor can be neither the one for $r$ because the path is simple, nor the one for $r'$ because the path bypasses $l'$. This is a contradiction. $\square$

### 8.3.3 Program refinement

The proofs of program refinement use the simulation diagrams in Chapter 2 and different instantiations of the GWF_FR rule we developed in Chapter 5, where instead of just a function $f$ and frame $\sigma$, we now have a configuration *config* that also includes the program memory.

$$config, P \vdash S \quad \triangleq \quad S \in config.prog \wedge P\, config\, (S|_{sdom})$$

Let $\sigma|_{sdom}$ be $(\sigma.f,\ \sigma.pc,\ (\sigma.\delta)|_{(\mathbf{sdom}_{(\sigma.f)}(\sigma.pc))},\ \sigma.\alpha)$. $S|_{sdom}$ is $(S.M, S.\overline{\sigma|_{sdom}})$. $S \in prog$ ensures that all $f$ and $pc$ in each frame of $S$ are defined in $prog$.

**Promotability** As we discussed above, the micro transformations (except $\phi$-nodes elimination) rely on the **promotable** property. We start by establishing the invariants related to promotability, namely that promotable allocations aren't aliased. This proof is itself an application of GWF_FR.

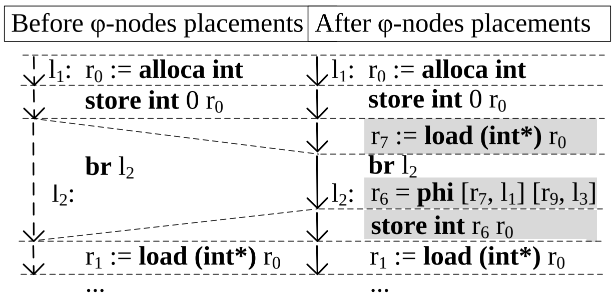


Figure 8.8: The simulation relation for the correctness of ϕ-node placement

The **promotable** property ensures that a promotable **alloca** of a function does not escape—the function can access the data stored at the allocation, but cannot pass the address of the allocation to other contexts. Therefore, in the program, the promotable **alloca** and all other pointers (in memory, local temporaries and temporaries on the stack) must not alias. Formally, given a promotable allocation $r$ with type $typ*$ in $f$, we define $P_{\text{noalias}}(f, r, typ)$:

$$\begin{aligned}
&\lambda config.\lambda S.\\
&\quad \forall \overline{\sigma}_1 {+\!+} \sigma :: \overline{\sigma}_2 = S.\overline{\sigma}.\ f = \sigma.f \wedge [\![r]\!]_{\sigma.\delta} = \lfloor blk \rfloor \Longrightarrow\\
&\qquad \exists v.\mathbf{load}\,(S.M, typ, blk) = \lfloor v \rfloor\\
&\quad \wedge \quad \forall blk'.\forall typ'.\neg\mathbf{load}\,(S.M, typ', blk') = \lfloor blk \rfloor\\
&\quad \wedge \quad \forall r' \neq r \Longrightarrow \neg [\![r']\!]_{\sigma.\delta} = \lfloor blk \rfloor\\
&\quad \wedge \quad \forall \sigma' \in \overline{\sigma}_1.\forall r'.\neg [\![r']\!]_{\sigma'.\delta} = \lfloor blk \rfloor
\end{aligned}$$

The last clause ensures that the **alloca** and the variables in the callees reachable from $f$ do no alias. In CompCert, the translation from C#minor to Cminor uses properties (in non-SSA form) similar to $P_{\text{noalias}}(f, r, typ)$ to allocate local variables on stack.

**Lemma 34** (Promotable alloca is not aliased)**.** *At any reachable program state S, config*$, P_{\text{noalias}}(f, r, typ) \vdash S$ *holds.*

The invariant holds initially. At all reachable states, the invariant holds because a promotable allocation cannot be copied to other temporaries, stored to memory, passed into a function, or returned. Therefore, in a well-defined program no external code can get its location by accessing other temporaries and memory locations. Importantly, the memory model ensures that from a consistent initial memory state, all memory blocks in temporaries and memory are allocated—it is impossible to forge a fresh pointer from an integer.

**$\phi$-node placement** Figure 8.8 pictorially shows an example (which is the code fragment from Figure 8.5) of the simulation relation $\sim$ for proving that the $\phi$-node placement preserves semantics. It follows left "option" simulation, because $\phi$-node placement only inserts instructions. We use the number of unexecuted instructions in the current block as the measure function.

The dashed lines indicate where the two program counters must be synchronized. Although the pass defines new variables and **store**s (shaded in Figure 8.8), the variables are only passed to the new $\phi$ nodes, or stored into the promotable allocation; additional **store**s only update the promotable allocation with the same value. Therefore, by Lemma 34, $\sim$ requires that two programs have the same memory states and the original temporaries match.

**Lemma 35.**

*If* $f' = \phi$*–nodes_placement* $f\ r$, *and* **promotable** $(f,r)$, *and* $\vdash prog$, *then* $prog \supseteq prog\{f'/f\}$.

The interesting case is to show that $\sim$ implies a correspondence between stuck states. Lemma 34 ensures that the promotable allocation cannot be dereferenced by operations on other pointers. Therefore, the inserted memory accesses are always safe.

`LAS`/`LAA` We present the proofs for the correctness of `LAS`. The proofs for the correctness of `LAA` is similar. In the code after $\phi$-node placement of Figure 8.5, $r_7 := \textbf{load}\,(\textbf{int}*)\,r_0$ is an `LAS` of $\textbf{store}\,\textbf{int}\,0\,r_0$. We observe that at any program counter $pc$ between the **store** and **load**, the value stored at $r_0$ must be 0 because $\textbf{alive}\,(pc_1,pc_2)$ holds—the **store** defined at $pc_1$ is not overwritten by other writes until $pc$.

To formalize the observation, consider a promotable $r$ with type $typ*$ in $f$. Suppose `find_stld_pair` $f$ $r$ `= LAS (`$pc_2$`,` $val_2$`,` $r_1$`)`. Consider the invariant $P_{\text{las}}(f,r,typ,pc_2,val_2)$:

$$\begin{aligned}&\lambda config.\,\lambda S.\,\forall\sigma \in S.\overline{\sigma}.\\ &\ (f = \sigma.f \wedge [\![val_2]\!]_{\sigma.\delta} = \lfloor v_2 \rfloor \wedge [\![r]\!]_{\sigma.\delta} = \lfloor blk \rfloor \wedge\\ &\ \ \textbf{alive}\,(pc_2,\sigma.pc)) \Longrightarrow \textbf{load}\,(S.M,typ,blk) = \lfloor v_2 \rfloor\end{aligned}$$

Using Lemma 34, we have that:

**Lemma 36.** *If* **promotable** $(f,r)$, *then* **alive** $(pc_2,r_1)$ *and at any reachable state S, config*, $P_{\text{las}}(f,r,typ,pc_2,val_2) \vdash$ *S holds.*

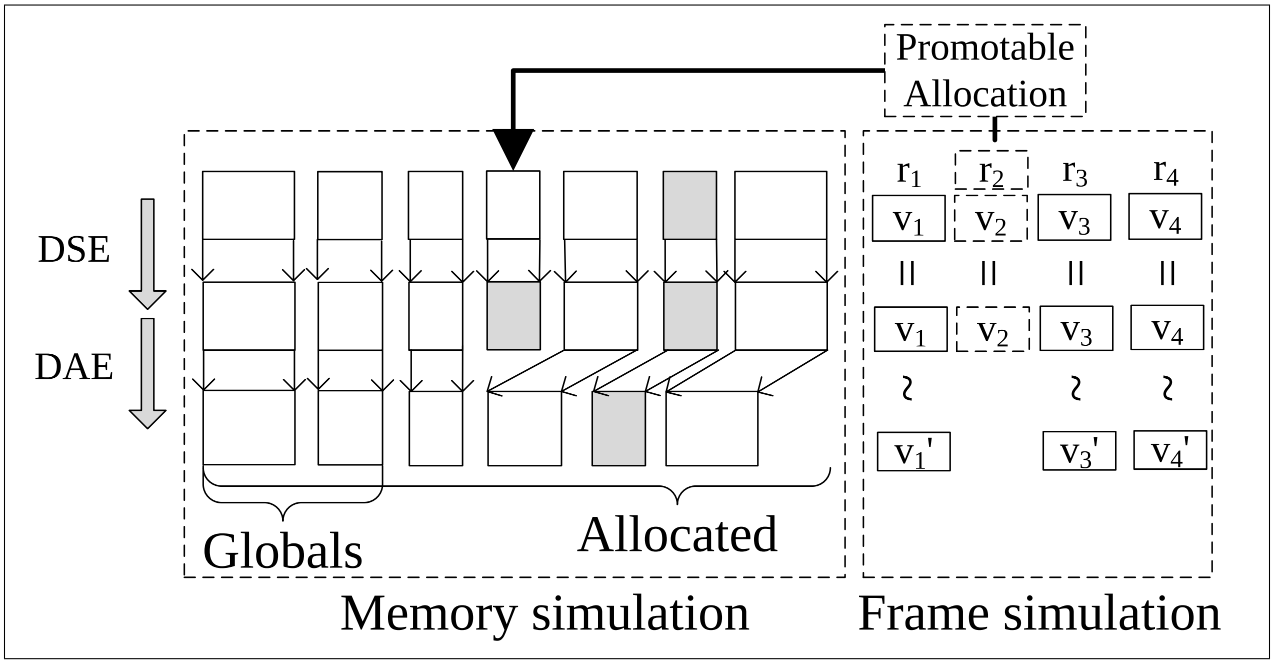


Figure 8.9: The simulation relation for DSE and DAE

Let two programs relate to each other if they have the same program states. Lemma 36 establishes that the substitution in LAS is correct. The following lemma shows that removal of unused instructions preserves semantics in general.

**Lemma 37.** *If* **check** *insn* $=$ **false** $\Rightarrow$ *f does not use insn, and* $\vdash$ *prog, then prog* $\supseteq$ *prog*$\{$**filter check** $f/f\}$.

Lemma 32 shows that the precondition of Lemma 37 holds after the substitution in LAS. Finally, we have that:

**Lemma 38.** LAS *preserves semantics.*

SAS/DSE/DAE Here we discuss only the simulation relations used by the proofs. SAS removes a **store** to a promotable allocation overwritten by a following memory write. We consider a memory simulation that is the identity when the program counter is outside the SAS pair, but ignores the promotable **alloca** when the program counter is between the pair. Due to Lemma 34 and the fact that there is no **load** between a SAS pair, no temporaries or other memory locations can observe the value stored at the promotable **alloca** between the pair.

Figure 8.9 pictorially shows the simulation relations between the program states before and after DSE or DAE. Shaded memory blocks contain uninitialized values. The program states on the top are before DSE, where $r_2$ is a temporary that holds the promotable stack allocation and is not used by any loads. After DSE, the memory values for the promotable allocation may not match the original program's corresponding block. However, values in temporaries and all other memory locations must be unchanged (by Lemma 34). Note that unmatched memory states only occur after the promotable allocation; before the allocation, the two memory states should be the same.

The bottom part of Figure 8.9 illustrates the relations between programs before and after DAE. After DAE, the correspondence between memory blocks of the two programs is not bijective, due to the removal of the promotable **alloca**. However, there must exist a mapping $\sim$ from the output program's memory blocks to the original program's memory blocks. The simulation requires that all values stored in memory and temporaries (except the promotable allocation) are equal modulo the mapping $\sim$.

**$\phi$-nodes elimination** Consider $r = \textbf{phi}\, typ\, \overline{[val_j, l_j]}^{j}$ (an AH $\phi$-node) where all the $val_j$'s are either equal to $r$ or some $val'$. Lemma 33 showed that $f \vdash val' \gg r$. Intuitively, at any $pc$ that both $val'$ and $r$ strictly dominate, the values of $val'$ and $r$ must be the same. Consider the invariant $P_{\text{ah}}(f, r, val')$:

$$\begin{aligned}&\lambda config.\lambda S.\forall\sigma \in S.\overline{\sigma}.\\&\quad f = \sigma.f \wedge [\![r]\!]_{\sigma.\delta} = \lfloor v_1 \rfloor \wedge [\![val']\!]_{\sigma.\delta} = \lfloor v_2 \rfloor \Longrightarrow v_1 = v_2\end{aligned}$$

**Lemma 39.** *$config, P_{\text{ah}}(f, r, val') \vdash S$ holds for any reachable program state $S$.*

Lemma 39 establishes that the substitution in $\phi$-nodes elimination is correct by using the identity relation. Lemma 32 and Lemma 37 show that removing dead $\phi$-nodes is correct.

### 8.3.4 The correctness of `vmem2reg`

Our main result, fully verified in Coq, is the composition of the correctness proofs for all the micro program transformations:

**Theorem 40** (`vmem2reg` is correct)**.** *If $f'$ = `vmem2reg` $f$ and $\vdash prog$, then $\vdash prog\{f'/f\}$ and $prog \supseteq prog\{f'/f\}$.*

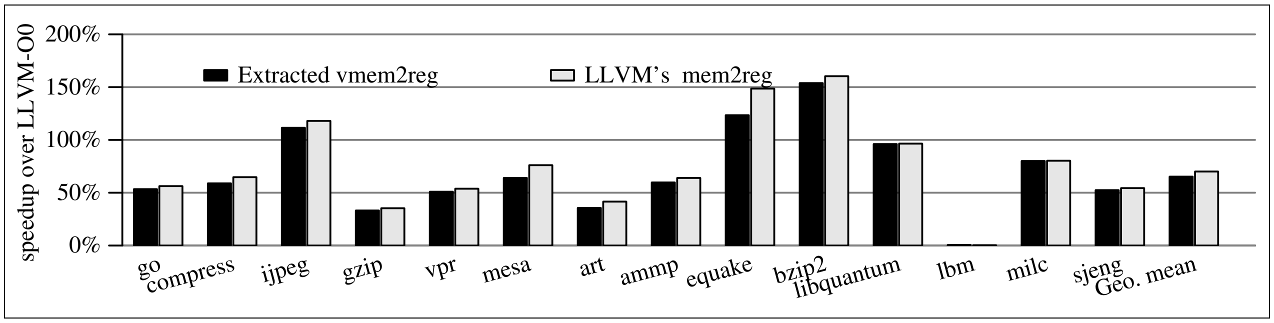


Figure 8.10: Execution speedup over LLVM `-O0` for both the extracted `vmem2reg` and the original `mem2reg`.

## 8.4 Extraction and Performance Evaluation

This section shows that (1) an implementation of `vmem2reg` extracted directly from the Coq code can successfully transform actual programs and (2) `vmem2reg` is almost as effective at optimizing code as LLVM's existing unverified implementation in C++.

**Extracted `vmem2reg` and experimental methodology** We used the Coq extraction mechanism to obtain a certified implementation of the `vmem2reg` optimization directly from the Coq sources (which are 838 lines to specify the algorithm). `mem2reg` is the first optimization pass applied by LLVM[2], so we tested the efficacy of the extracted implementation on LLVM IR bitcode generated directly from C source code using the clang compiler. At this stage, the LLVM bitcode is unoptimized and in "trivial" SSA form (as was discussed earlier). To prevent the impact of this optimization pass from being masked by subsequent optimizations, we apply either LLVM's `mem2reg` or the extracted `vmem2reg` to the unoptimized LLVM bitcode and then immediately invoke the back-end code generator. We evaluate the performance of the resultant code on a 2.66 GHz Intel Core 2 processor running benchmarks selected from the SPEC CPU benchmark suite that consist of over 336k lines of C source code in total.

Figure 8.10 reports the execution time speedups (larger is better) over a LLVM's-O0 compilation baseline for various benchmarks. The left bar of each group shows the speedup of the extracted `vmem2reg`, which provides an average speedup of 77% over the baseline. The right bar of each group is the benefit provided by LLVM's `mem2reg`, which provides 81% on average; `vmem2reg` captures much of the benefit of the LLVM's `mem2reg`.

[2]All results reported are for LLVM version 3.0.

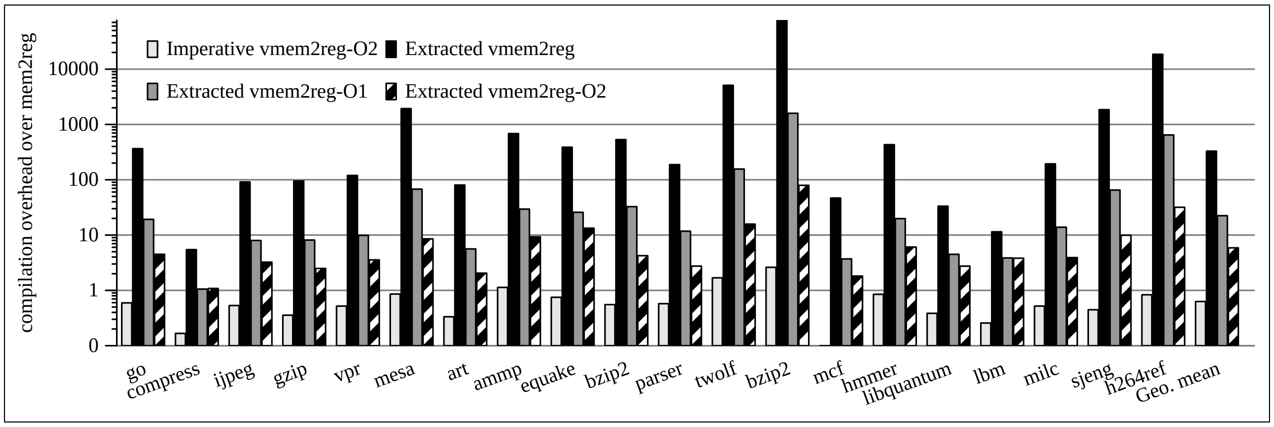


Figure 8.11: Compilation overhead over LLVM's original `mem2reg`.

**Comparing** `vmem2reg` **and** `mem2reg` The `vmem2reg` pass differs from LLVM's `mem2reg` in a few ways. First, `mem2reg` promotes **alloca**s used by LLVM's intrinsics, while `vmem2reg` conservatively considers such **alloca**s to potentially escape, and so does not promote them. We determined that such intrinsics (used by LLVM to annotate the liveness of variable definitions) lead to almost all the difference in performance in the *equake* benchmark. Second, although `vmem2reg` deletes most unused $\phi$-nodes, it does not aggressively remove them and, therefore, does not generate fully pruned SSA as `mem2reg` does. However, our results show that this does not impose a significant difference in performance.

## 8.5 Optimized `vmem2reg`

The algorithm of `vmem2reg` is designed with verification in mind, but it is not efficient in practice: Figure 8.11 shows that on average `vmem2reg` is 329 times slower than `mem2reg` in terms of compile-time. Such an inefficient design is aimed at streamlining the presentation of the proof techniques we developed for SSA, such that our research can focus on the crucial part of the problem—understanding how the proofs should go. This section shows how to design an efficient algorithm based on `vmem2reg`, and verify its correctness by extending the proofs for `vmem2reg`.

The costs of `vmem2reg` include (1) the pessimistic $\phi$-node insertion algorithm, which introduces unnecessary $\phi$ nodes that lead to more inserted **load**s and **store**s to remove; and (2) the pipelined strategy that requires much more passes than necessary. Given a CFG with $N$ nodes and $I$ instructions and a promotable **alloca**, `vmem2reg`, in the worst case, first inserts $N$ $\phi$ nodes and $N$ "Load After Store" or "Load After Alloca"

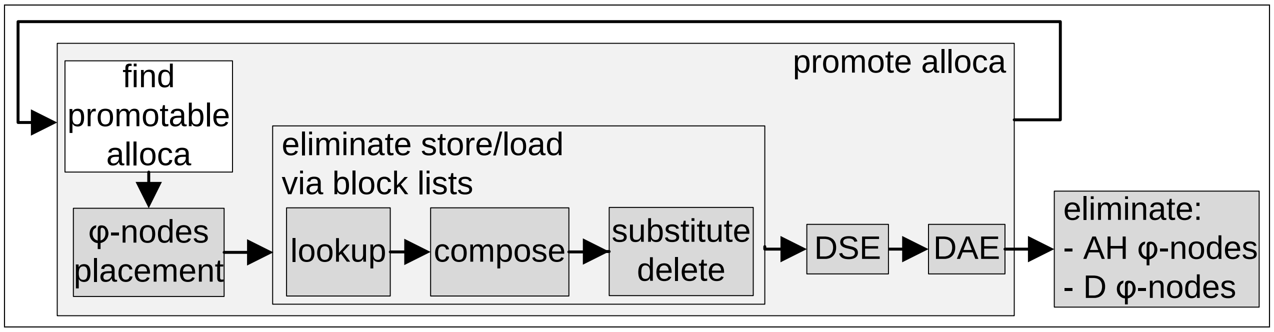


Figure 8.12: Basic structure of `vmem2reg-O1`

pairs, then takes $N$ passes to promote the **load**s and **store**s, and finally takes at most $N$ passes to remove AH $\phi$-nodes. Therefore, the complexity of `vmem2reg` is $O(N*I)$.

To address the compilation overhead, we implemented two improved algorithms: `vmem2reg-O1` and `vmem2reg-O2` in terms of the difficulty for reasoning about their correctness. Section 8.5.1 shows `vmem2reg-O1` that composes the pipelined elimination passes into a single pass. Section 8.5.2 shows `vmem2reg-O2` that improves `vmem2reg-O1` by placing the minimal number of $\phi$ nodes at domination frontier, and does not need the AH $\phi$-node elimination pass. Note that `vmem2reg-O1` is verified in Coq, and `vmem2reg-O2` is not fully verified in Coq.

### 8.5.1 O1 Level—Pipeline fusion

Figure 8.12 gives the structure of `vmem2reg-O1`, which takes one pass to collect all `LAS`/`LAA` pairs and then uses one more pass to remove them. Figure 8.13 presents the composed elimination algorithm (`eliminate_stld`). We denote each micro elimination by actions $ac$.

$$\text{Actions } ac \ ::= \ r \mapsto val \qquad \text{Lists of Actions } AC \ ::= \ \emptyset \mid ac, AC$$

Here, $r \mapsto val$ denotes `LAS` ($r$, $pc$, $val$) or `LAA` $r$ with the default memory value $val$. Note that unlike `vmem2reg` the optimized version does not consider `SAS` because (1) the later `DSE` removes all dead **store**s in one pass (2) `vmem2reg-O2` (in Section 8.5.2) needs to traverse all subtrees to find `SAS`, which does not lead to a simple algorithm.

To find all initial elimination pairs $AC$, `eliminate_stld` traverses the list of blocks of a function, finds elimination pairs for each block (by `find_stld_pairs_block`), and then concatenates them. At each block, we use `stld_state` to keep track of the search state (by `find_stld_pairs_cmd`): `STLD_INIT` is the initial state; `STLD_AL` *typ* records the element type of the memory value stored at the latest promotable

```
let find_stld_pair_cmd $r$ (acc:stld_state * Action list) $c$: stld_state * Action list =
  let (st, $AC$) = acc in
  match $c$ with
  | $r_0 := \textbf{alloca}\, typ$ $\rightarrow$ if $r$ = $r_0$ then (STLD_AL $typ$, $AC$) else acc
  | $\textbf{store}\, typ\, val_1\, r_2$ $\rightarrow$ if $val_1\, \textbf{uses}\, r_0$ then (STLD_ST $val_1$, $AC$) else acc
  | $r_0 := \textbf{load}\,(typ*)\, r_1$ $\rightarrow$
      if $r$ = $r_1$ then
         match st with
         | STLD_ST $val$ $\rightarrow$ (st, $(r_0 \mapsto val, AC)$)
         | STLD_AL $typ$ $\rightarrow$ (st, $(r_0 \mapsto \textbf{undef}\, typ, AC)$)
         | _ $\rightarrow$ acc
         end
      else acc
  | _ $\rightarrow$ acc
  end

let find_stld_pairs_block $r$ (acc:stld_state * Action list) $b$: stld_state * Action list =
  let (_ _ $\overline{c}$ _) = $b$ in
  fold_left (find_stld_pair_cmd $r$) $\overline{c}$ acc

let eliminate_stld $r$ $f$ =
  let $fheader\{\overline{b}\}$ = $f$ in
  let $AC$ = flat_map (rev (snd (find_stld_pairs_block $r$ (STLD_INIT, $\emptyset$)))) $\overline{b}$ in
  $\overline{AC}(f)$
```

Figure 8.13: **eliminate_stld** of `vmem2reg-O1`

allocation; `STLD_ST` *val* records the the value stored by the latest **store** to the promotable allocation. When `find_stld_pairs_cmd` meets a **load**, it generates an action in terms of the current state.

Consider the following code in Figure 8.14 with entry $l_1$. The algorithm finds a list of actions: $r_4 \mapsto r_3, r_5 \mapsto r_4, r_2 \mapsto r_1, r_3 \mapsto r_2, r_6 \mapsto r_3, \emptyset$, which forms a tree because SSA ensures acyclicity of def/use chains. However, we cannot simply take a pass that, for each $r \mapsto val$, replaces all uses of $r$ by $val$, and then deletes the definition of $r$, because the later actions may depend on the former ones—for example, after applying $r_4 \mapsto r_3$, the action $r_5 \mapsto r_4$ should update to $r_5 \mapsto r_3$; and the later actions can also affect the former ones—the action $r_3 \mapsto r_2$ will change the first action to be $r_4 \mapsto r_2$.

To address the problem, we first define the basic operations for actions:

$$\begin{array}{rcll} AC[r] & = & \lfloor val \rfloor & when\, r \mapsto val \in AC \\ AC[val] & = & \cdot & otherwise \end{array} \qquad \begin{array}{rcll} AC\{val\} & = & val' & when\, AC[val] = \lfloor val' \rfloor \\ & = & val & otherwise \end{array}$$

```
l2 : ···
    store int r3 r0
    r4 := load (int∗) r0
    store int r4 r0
    r5 := load (int∗) r0
    ret int r5
l1 : r0 := alloca int
    ···
    store int r1 r0
    r2 := load (int∗) r0
    store int r2 r0
    r3 := load (int∗) r0
    ···
    br r7 l2 l3
l3 : ···
    store int r3 r0
    r6 := load (int∗) r0
    ret int r6
```

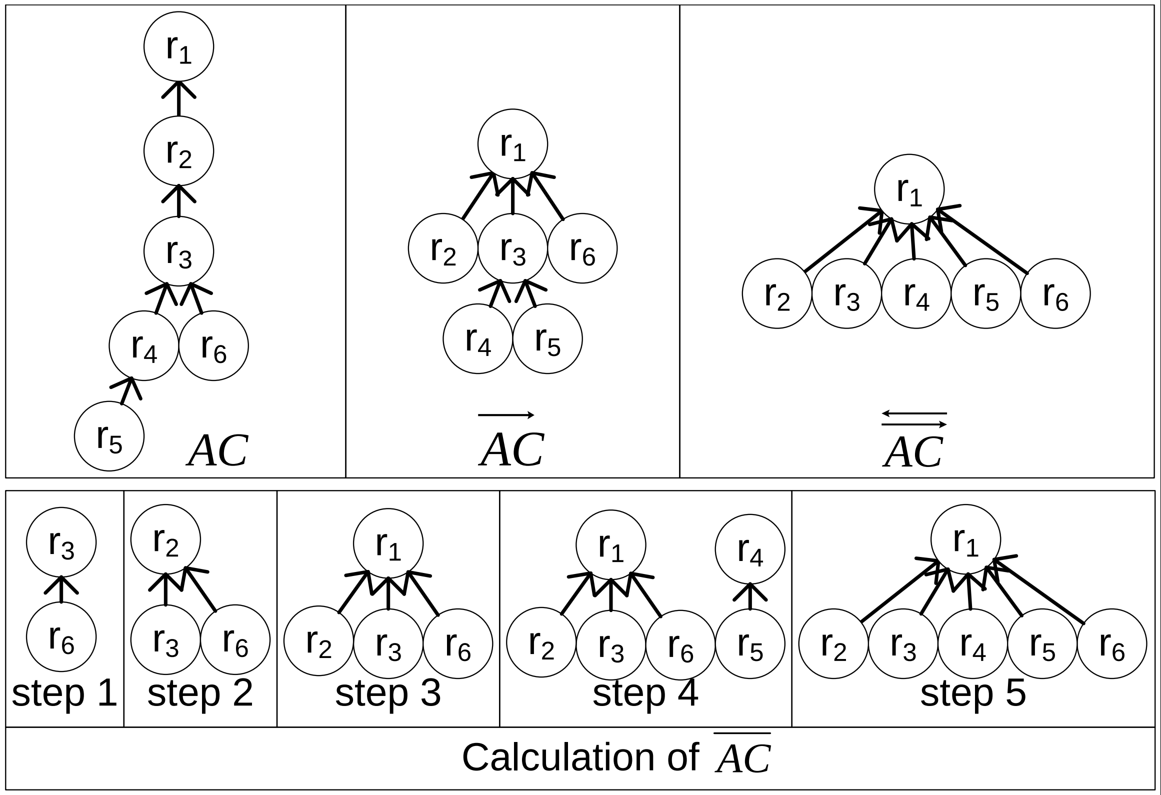


Figure 8.14: The operations for elimination actions

$$
\begin{aligned}
AC\{val/r\} \triangleq \quad & \emptyset\{val/r\} = \emptyset \\
& (r_0 \mapsto val_0, AC)\{val/r\} = r_0 \mapsto val_0\{val/r\}, AC\{val/r\}
\end{aligned}
$$

$$
\begin{aligned}
AC(val) \triangleq \quad & \emptyset(val) = val \\
& (r_0 \mapsto val_0, AC)(val) = AC(val\{val_0/r_0\})
\end{aligned}
$$

where $AC[val]$ finds the value mapped from $val$; $AC\{val\}$ returns $AC[val]$ if $val$ is mapped to some value, otherwise returns $val$; $AC\{val/r\}$ substitutes $r$ in all substitutees of $AC$ by $val$; $AC(val)$ applies $AC$ to $val$. Given the basic operations, we define

$$
\begin{aligned}
\overrightarrow{AC} \triangleq \quad & \overrightarrow{\emptyset} = \emptyset \\
& \overrightarrow{(r \mapsto val, AC)} = r \mapsto val, \overrightarrow{(AC\{val/r\})}
\end{aligned}
\qquad
\begin{aligned}
\overleftarrow{AC} \triangleq \quad & \overleftarrow{\emptyset} = \emptyset \\
& \overleftarrow{(r \mapsto val, AC)} = r \mapsto AC(val), \overleftarrow{AC}
\end{aligned}
$$

$$
\overleftrightarrow{AC} \triangleq \quad \overleftarrow{\overrightarrow{AC}}
\qquad
\begin{aligned}
\overline{AC} \triangleq \quad & \widetilde{\emptyset} = \emptyset \\
& \overline{(r \mapsto val, AC)} = r \mapsto \overline{AC}\{val\}, (\overline{AC})\{\overline{AC}\{val\}/r\}
\end{aligned}
$$

Here, $\overrightarrow{AC}$ applies all the former substitutions to the later actions; $\overleftarrow{AC}$ applies all the later substitutions to the former actions; $\overleftrightarrow{AC}$ composes $\overrightarrow{AC}$ and $\overleftarrow{AC}$, actually equals to the actions that `vmem2reg` finds in the pipelined transformation. Figure 8.14 gives the calculation of $\overleftrightarrow{AC}$ whose result is a flattened tree with height one. The complexity of $\overrightarrow{AC}$ and $\overleftarrow{AC}$ are $O((log(N) * N^2)$ where the $log(N)$ is from the absence of efficient, purely functional hash tables. Applying actions to a function costs $O(log(N) * I)$. Note that in practice $I$ is much larger than $N$.

In fact, we can compute $\overleftrightarrow{AC}$ with a faster algorithm $\overline{AC}$ that processes the initial actions from right to left, and has the invariant that the trees of its intermediate forest are flattened. Figure 8.14 gives the calculation of $\overline{AC}$. The complexity of $\overline{AC}$ is $O((log(N) * N^2)$, which is the half of $\overleftrightarrow{AC}$'s.

Figure 8.11 shows that on average `vmem2reg-O1` is 22 times slower than `mem2reg` in terms of compile-time. Appendix A discusses the correctness of `vmem2reg-O1` (which is fully verified in Coq).

### 8.5.2 O2 Level—Minimal $\phi$-nodes Placement

`vmem2reg-O1` addresses one kind of compile-time cost by "fusing" micro passes. To address the other cost—the number of $\phi$-nodes, we implemented `vmem2reg-O2` based on `vmem2reg-O1`, which is shown in Figure 8.15.

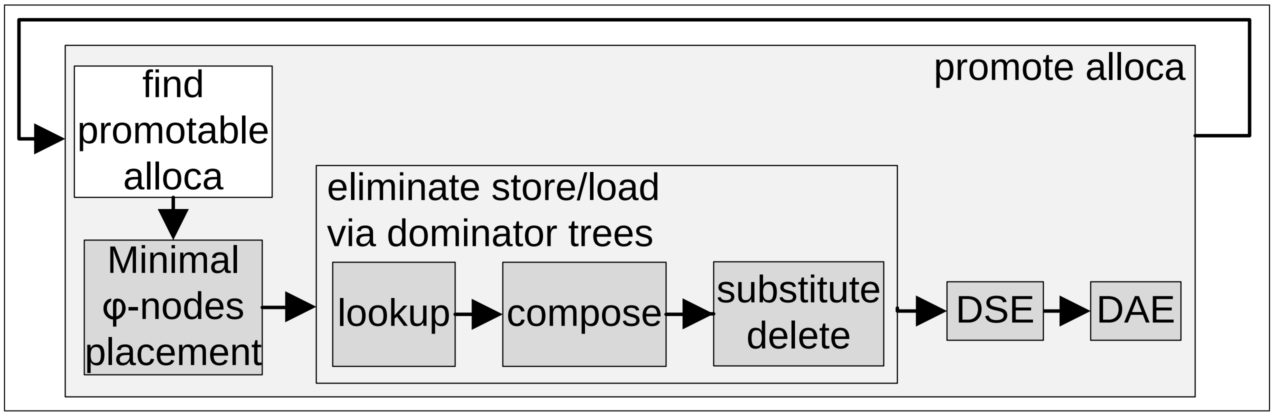


Figure 8.15: Basic structure of vmem2reg-O2

```
let find_stld_pairs_dtree r (acc:stld_state * Action list) (dt:DTree)
  : stld_state * Action list =
  match dt with
  | DT_node b dts → find_stld_pairs_dtrees r (find_stld_pairs_block r acc b) dts
  end
with find_stld_pairs_dtrees r (acc:stld_state * Action list) (dts:DTrees)
  : stld_state * Action list =
  match dts with
  | DT_nil → acc
  | DT_cons dt dts' →
      let (_, AC) = find_stld_pairs_dtree r acc dt in
      find_stld_pairs_dtrees r (fst acc, AC) dts'
  end

let eliminate_stld r f =
  let dt = construct_dtree f in
  let AC = rev (snd (find_stld_pairs_dtree r (STLD_INIT, ∅) dt)) in
  $\overline{AC}(f)$
```

Figure 8.16: **eliminate_stld** of vmem2reg-O2

vmem2reg-O2 places the minimal number of ϕ-nodes by the dominance-frontier algorithm implemented in Section 3.5. Our experiments show that on average, the algorithm only introduces 1/8 of the ϕ-nodes of the pessimistic one and does not need the additional AH ϕ-node elimination pass.

vmem2reg-O2 does not insert ϕ-nodes at every block, so LAS/LAA pairs may appear across blocks. To find them, Figure 8.16 extends the algorithm in Figure 8.13 by depth-first-searching functions' dominator trees (which are computed by the algorithm in Section 3.4).

Although vmem2reg-O2 has the same complexity as vmem2reg-O1, Figure 8.11 shows that on average vmem2reg-O2 is 5.9 times slower than mem2reg in terms of compile-time. To study the overhead cause by the purely functional programming, we also implemented the C++ version of vmem2reg-O2. Because it uses constant-time hashtables and does alias-based substitution, the C++ version's complexity is $O(I)$. In practice, Figure 8.11 shows that its compile-time is 0.63 time of mem2reg's because we use a slightly more efficient dominance-frontier calculation [24] and do not allow intrinsics to use promotable allocations.

The correctness of vmem2reg-O2 is composed of two parts. The first part needs to generalize the proofs of vmem2reg that assume that LAS/LAA pairs must be in the same block to allow LAS/LAA pairs in terms of arbitrary domination relations. The second part can reuse the proofs of vmem2reg-O1 for reasoning about composing micro transformations. Appendix B discusses the correctness of vmem2reg-O2 (which have not fully been verified in Coq).

# Chapter 9

# The Coq Development

This chapter summarizes our Coq development.

## 9.1 Definitions

Table 9.1 shows the size of our development. Note that the size of the formalism of `vmem2reg-O1` does not include the development of `vmem2reg`. Vellvm encodes the abstract syntax from Chapter 6 in an entirely straightforward way using Coq's inductive datatypes (generated in a preprocessing step via the Ott [60] tool). The implementation uses Penn's Metatheory library [13], which was originally designed for the locally nameless representation, to represent identifiers of the LLVM, and to reason about their freshness.

The Coq representation deviates from the full LLVM language in only a few (mostly minor) ways. In particular, the Coq representation requires that some type annotations be in normal form (*e.g.,* the type annotation on **load** must be a pointer; named types must be sorted in terms of their dependency), which simplifies type checking at the IR level. The Vellvm tool that imports LLVM bitcode into Coq provides such normalization, which simply expands definitions to reach the normal form.

Vellvm's type system is also represented via Ott [60], and refers to the imperative LLVM verification pass that checks the well-formedness of LLVM bitcode. The current type system is formalized by predicates that is not extractable. We leave the extraction as our future work, *i.e.,* a verified LLVM type checker.

Vellvm's memory model implementation extends CompCert's with 8,889 lines of code to support integers with arbitrary precision, padding, and an experimental treatment of casts that has not yet needed for any

|  |  |  | Definition | Metatheory | Total |
|---|---|---|---|---|---|
| Coq | Core | Syntax | 652 | 6,443 | 7,095 |
|  |  | Computing dominators | 1,658 | 14,437 | 16,095 |
|  |  | Type system | 1,225 | 6,308 | 7,533 |
|  |  | Memory model (extension) | 1,045 | 7,844 | 8,889 |
|  |  | Operational semantics | 1,960 | 6,443 | 8,403 |
|  |  | Interpreter | 228 | 279 | 507 |
|  |  | Total | 5,110 | 27,317 | 32,427 |
|  | App. | SoftBound | 762 | 17,420 | 18,182 |
|  |  | Translation validators | 127 | 9,768 | 9,895 |
|  |  | `vmem2reg` | 2,358 | 52,138 | 54,496 |
|  |  | `vmem2reg-O1` | 665 | 10,318 | 10,983 |
|  |  | Total | 3,912 | 89,644 | 92,556 |
|  | Vminus |  | 806 | 21,541 | 22,347 |
|  | Total |  | 9,828 | 138,502 | 148,330 |

|  |  | Total |
|---|---|---|
| OCaml | Parser & Printer | 2,031 |
|  | LLVM bindings (extension) | 6,369 |

Table 9.1: Size of the development (approx. lines of code)

of our proofs. On top of this extended memory model, all of the operational semantics and their metatheory have been proved in Coq.

## 9.2 Proofs

Checking the entire Vellvm implementation using `coqc` in a single processor takes about 105 minutes on a 1.73 GHz Intel Core i7 processor with 4 GB RAM. We expect that this codebase could be significantly reduced in size by refactoring the proof structure and making it more modular.

Our formalism uses two logical axioms: functional extensionality and proof irrelevance [1]. We also use axioms to specify the specification of external functions and intrinsics, and the behavior of program initialization. The verification of `mem2reg` relies on about a dozen axioms, almost all of which define either the initial state of the machine (*i.e.,* where in memory functions and globals are stored) or the behavior of external function calls. One axiom asserts that memory alignment is a power of two, which is not necessary for LLVM programs in general, but is true of almost all real-world platforms.

## 9.3 OCaml Bindings and Coq Extraction

The LLVM distribution includes primitive OCaml bindings that are sufficient to generate LLVM IR code ("bitcode" in LLVM jargon) from OCaml. To convert between the LLVM bitcode representation and the extracted OCaml representation, we implemented a library consisting of about 8,400 lines of OCaml-LLVM bindings. This library also supports pretty-printing of the abstract syntax tree of the LLVM IR; this code was also useful in the extracted interpreter.

# Chapter 10

# Related Work

**Verified compilers** Compiler verification has a considerable history; see the bibliography of Leroy [42] for a comprehensive overview. Vellvm is closest in spirit to CompCert [42], which was the first fully-verified compiler to generate compact and efficient assembly code for a large fragment of the C language. CompCert also uses Coq. It formalizes the operational semantics of CompCert C, several intermediate languages used in the compilation, and assembly languages including PowerPC, ARM and x86. The latest version of CompCert also provides an executable reference interpreter for the semantics of CompCert C. Based on the formalized semantics, the CompCert project fully proves that all compiler phases produce programs that preserve the semantics of the original program. Optimization passes include local value numbering, constant propagation, coalescing graph coloring register allocation [18], and other back-end transformations. It uses translation validators for certifying advanced compiler optimizations, such as instruction scheduling [68], lazy code motion [69], and software pipelining [70]. The XCERT project [64, 66] extends the CompCert compiler by a generic translation validator based on SMT solvers.

Other research has also used Coq for compiler verification tasks, including much recent work on compiling functional source languages to assembly [15, 21, 22].

**Formalization for computing dominators** The CompCertSSA project [14] improves the CompCert compiler by creating a verified SSA-based middle-end and a GVN optimization pass. They also formalize the AC algorithm to validate SSA construction and GVN passes, and prove the soundness of AC. We implement both AC and CHK—an extension of AC in a generic way, and prove they are both sound and complete. We also provide the corresponding dominator tree constructions, and evaluate performance.

There are also informal formalizations for computing dominators. Georgiadis and Tarjan [30] propose an almost linear-time algorithm that validates if a tree is a dominator tree of a CFG. Although the algorithm is fast, it is nearly as complicated as the LT algorithm, and it requires a substantial amount of graph theory. Ramalingam [4] proposes another dominator tree validation algorithm by reducing validating dominator trees to validating loop structures. However, in practice, most of modern loop identification algorithms used in LLVM and GCC are based on dominance analysis to find loop headers and bodies.

**Formalization for SSA and SSA-based optimizations** Verifying the correctness of compiler transformations is an active research area with a sizable amount of literature. We focus on the work relevant to SSA-based optimizations.

CompCertSSA verified a *translation validator* for an SSA construction algorithm that takes imperative variables to variables in a pruned SSA form. In contrast, our work fully verifies the SSA construction pass `vmem2reg` for LLVM directly. A bug in the CompCertSSA compiler will cause the validator to abort the compilation, whereas verifying the compiler rules out such a possibility. More pragmatically, translation validation is harder to apply in the context of LLVM, because the compiler infrastructure was not created with validation in mind. For example, the CompCertSSA translations maintain a close mapping between source and target variable names so that simulation can be checked by simple erasure; this is not feasible in the LLVM framework. The CompCertSSA project reports performance measurements of only small benchmarks totaling about 6k lines, whereas we have tested our pass on 336k lines, including larger programs.

Unsurprisingly, the CompCertSSA and Vellvm proofs share some similarities. For example, CompCert-SSA's GVN proof uses an invariant similar to the one in our Theorem 13 and Lemma 17. However, the LLVM's strategy of promoting **alloca**s means that our proofs need a combination of both SSA and aliasing properties to prove correctness. Moreover, our proof technique of pipelining "micro" transformations is novel, and it should be broadly applicable.

To fully prove GVN, we would need additional properties about congruence-based term equivalence. Although this fits naturally into our framework, Figure 8.2 shows that the combination of GVN with all other optimizations (except `mem2reg`) does not provide significant speedup—the full suite of `-O2` and `-O3` level optimizations only yields a 11% speedup (on average).

The validation algorithm of CompCertSSA is proven to be complete to certificate the classic SSA construction [28] (which computes dominators by the Lengauer-Tarjan algorithm [40]). Although `vmem2reg` is based on the Aycock-Horspool algorithm [12], Section 8.5 shows that the correctness of the classic algorithm is independent to the proofs for `vmem2reg`, and that the performance of the optimized `vmem2reg` is compatible with the classic algorithm.

Mansky *et al.* designed an Isabelle/HOL framework that uses control-flow graph rewrites to transform programs and uses temporal logic and model-checking to specify and prove the correctness of program transformations [45]. They verified an SSA construction algorithm in the framework. Other researchers have formalized specific SSA-based optimizations by using SSA forms with different styles of semantics: an informal semantics that describes the intuitive idea of the SSA form [28]; an operational semantics based on a matrix representation of $\phi$ nodes [72]; a data-flow semantics based term graphs using the Isabelle/HOL proof assistant [19]. Matsuno *et al.* defined a type system equivalent to the SSA form and proved that dead code elimination and common subexpression elimination preserve types [47]. There are also conversions between the programs in SSA form and functional programs [9, 34].

**Validating LLVM optimizations** The CoVac project [74] develops a methodology that adapts existing program analysis techniques to the setting of translation validation, and it reports on a prototype tool that applies their methodology to verification of the LLVM compiler. The LLVM-MD project [67] validates LLVM optimizations by symbolic evaluation. The Peggy tool performs translation validation for the LLVM compiler using a technique called equality saturation [63]. These applications are not fully certified.

**Mechanized language semantics** There is a large literature on formalizing language semantics and reasoning about the correctness of language implementations. Prominent examples include: Foundational Proof Carrying Code [10], Foundational Typed Assembly Language [26], Standard ML [27, 65], and (a substantial subset of) Java [37].

**Other mechanization efforts** The verified software tool-chain project [11] assures that the machine-checked proofs claimed at the top of the tool-chain hold in the machine language program. Typed assembly languages [20] provide a platform for proving back-end optimizations. Similarly, The Verisoft project [6] also attempts to mathematically prove the correct functionality of systems in automotive engineering and security technology. ARMor [78] guarantees control flow integrity for application code running on embedded processors. The Rhodium project [41] uses a domain specific language to express optimizations via local rewrite rules and provides a soundness checker for optimizations

# Chapter 11

# Conclusions and Future Work

This dissertation presents Vellvm in which we fully mechanized the semantics of LLVM and the proof techniques for reasoning about the properties of the SSA form and the correctness of transformations in LLVM using the Coq proof assistant. To demonstrate the effectiveness of Vellvm, we verified SoftBound—a program transformation that hardens C programs against spatial memory safety violations (*e.g.,* buffer overflows, array indexing errors, and pointer arithmetic errors) and the most performance-critical optimization pass in LLVM's compilation strategy—the `mem2reg` pass. We have showed that the formal models of SSA-based compiler intermediate representations can be used to verify low-level program transformations, thereby enabling the construction of high-assurance compiler passes.

This dissertation focused on formalizing and reasoning about general-purpose intermediate representation and the SSA form. In the following we show some of future research directions for developing compilers effectively.

**Memory-aware optimizations** Like `mem2reg`, most of the SSA-based passes in LLVM transform code are based on not only SSA invariants but also on aliasing information that is crucial for compilers to produce output with higher performance: in the absence of alias analysis, the global value numbering (GVN) and loop invariant code motion (LICM) passes in LLVM can get only insignificant speed-up [39].

The GVN of LLVM optimizes both pure instructions and instructions with memory-effects (such as loads, stores, and calls), and is the most performance-critical -O2 optimizations in LLVM. Figure 11.1 experimentally shows the effectiveness of GVN in the LLVM's `-O2` level optimizations. In our experiments, doing the full suite of `-O1` level optimizations with GVN yields a speedup of 3.3% (on average) compared

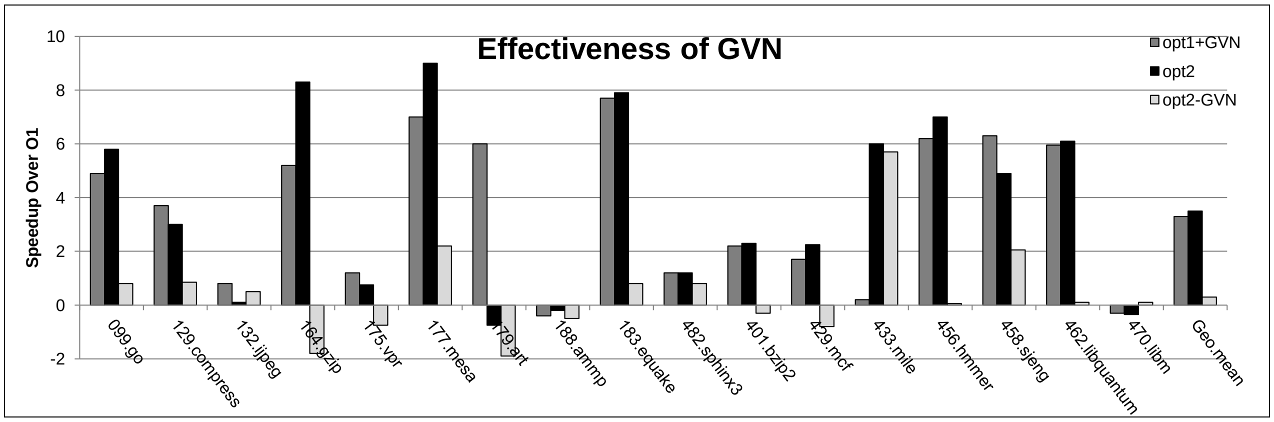


Figure 11.1: The effectiveness of GVN

to only `-O1` level optimizations of LLVM; doing the full suite of `-O2` level optimizations (which includes GVN) yields a speedup of 3.5%; doing the full suite of `-O2` level optimizations without GVN yields a speedup of 0.3%. Therefore, GVN is another good application for verification. Figure 11.2 experimentally

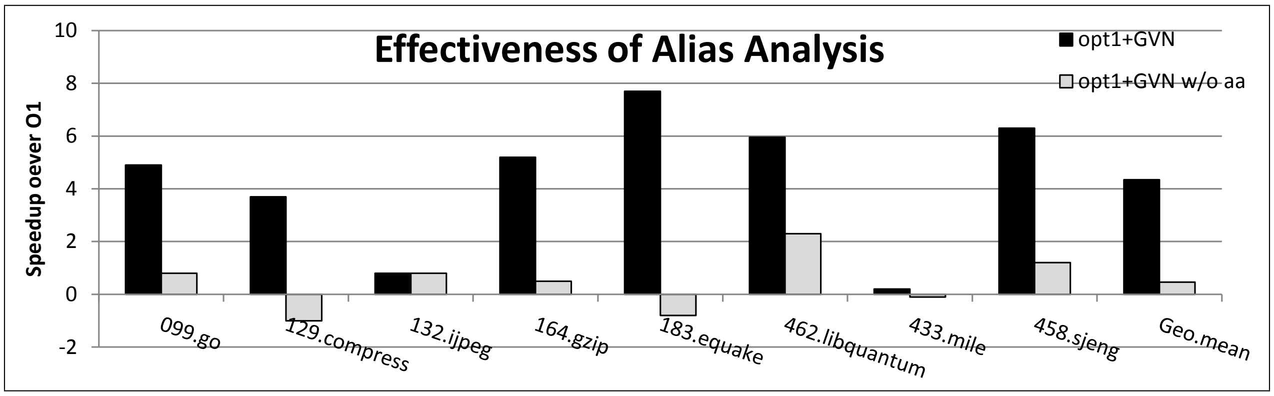


Figure 11.2: The effectiveness of Alias Analysis

shows that the alias analysis in LLVM has a significant impact on performance of GVN-optimized code. In our experiments, doing the full suite of `-O1` level optimizations with GVN yields a speedup of 4.3% (on average) compared to only `-O1` level optimizations of LLVM; doing the full suite of `-O1` level optimizations with GVN that does not use the alias analysis pass yields a speedup of 0.5%.

Given the performance impact of aliasing information, the correctness of alias analysis serves as a formal foundation for the memory-aware optimizations. Because LLVM does not represent memory in SSA, we need new metatheory for reasoning about memory aliasing. Based on the verified alias analysis, we can verify GVN by using the micro code transformations and pipeline fusion described in the dissertation.

**Loop analysis and transformations** Transformations for loops form the other kind of intra-procedural optimizations in LLVM, which all depend on the `loops` analysis that identifies natural loops in a CFG. Because the code in loops executes more frequently than other code, optimizing loops is crucial for improving performance.

In the loop optimizations in LLVM, LICM (which performs loop invariant code motion, attempting to remove as much code from the body of a loop as possible) is a good candidate to verify. First, LICM does not arbitrarily transform CFGs like what other loop optimizations (loop-deletion, loop-unrolling, loop-unswitch, loop-rotation, *etc.*) do. Therefore, we can be focused on the correctness of the `loops` analysis. Second, moving memory operations out of loop can potentially lead to relatively large speedup [32, 39]. Third, the recent work [46] shows that the LLVM's LICM is a problematic pass in terms of the sequentially consistent memory model because it speculatively hosts or sinks stores out of loops, which potentially causes additional data races in the transformed program. Formalizing the LICM in the sequential setting may lead to a straight-forward extension for studying the LICM in the sequential consistent memory model. Fourth, although the CompCert project verified lazy code motion [69], it only hoists instructions in the absence of alias information and SSA. Therefore, formalizing the LLVM LICM could lead to more interesting results.

**Efficiency versus verifiability** Industrial-strength compilers should not only be correct, but also be efficient in compile-time. Therefore, most of the main-stream production compilers are implemented in imperative languages, and use imperative data structures and sophisticated algorithms. On the other hand, Coq is a pure functional language that does not follow the imperative design pattern. For example, in-place update of data structures (which are frequently used for transforming programs imperatively) and hashtables are not allowed. Moreover, imperative algorithms used by practical compilers complicate reasoning about termination and invariant preservation. The verification of `mem2reg` illustrates the trade-off we made for achieving both efficiency and verifiability.

There is still much design space to explore. First, we can design verifiable functional data structures and algorithms. Designing efficient functional algorithms has a long history and many results [23, 29, 56]. The challenge is how to adopt the results in Coq that only allows recursions proven to terminate, and in which a good formalization pattern can dramatically reduce proof costs. Second, we may add selective imperative features to Coq, which should enable common imperative design, and also work with the existent features in Coq, such as dependent types, polymorphism, module systems and *etc.* Moreover, we need to check termination more carefully, because recursion can be encoded by using reference types.

# Bibliography

[1] Axioms in Coq. `http://coq.inria.fr/V8.1/faq.html#htoc36`.

[2] The GNU Compiler Collection. `http://gcc/gnu.org`.

[3] MathWorks. http://www.mathworks.com.

[4] On loops, dominators, and dominance frontiers. *ACM Trans. Program. Lang. Syst.*, 24(5):455–490, Sept. 2002.

[5] LLVM Developers Mailing List, 2011. `http://lists.cs.uiuc.edu/pipermail/llvmdev/2011-May/040387.html`.

[6] E. Alkassar and M. A. Hillebrand. Formal Functional Verification of Device Drivers. In *VSTTE '08: Proceedings of the 2nd International Conference on Verified Software: Theories, Tools, Experiments*, 2008.

[7] F. E. Allen and J. Cocke. Graph Theoretic Constructs For Program Control Flow Analysis. Technical report, IBM T.J. Watson Research Center, 1972.

[8] A. W. Appel. *Modern Compiler Implementation in C: Basic Techniques*. Cambridge University Press, New York, NY, USA, 1997.

[9] A. W. Appel. SSA is Functional Programming. *SIGPLAN Not.*, 33(4):17–20, April 1998.

[10] A. W. Appel. Foundational Proof-Carrying Code. In *LICS '01: Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, 2001.

[11] A. W. Appel. Verified software toolchain. In *ESOP '11: Proceedings of the 20th European Conference on Programming Languages and Systems*, 2011.

[12] J. Aycock and N. Horspool. Simple Generation of Static Single Assignment Form. In *Compiler Construction (CC)*, 2000.

[13] B. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. In *POPL '08: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2008.

[14] G. Barthe, D. Demange, and D. Pichardie. A formally verified SSA-based middle-end - Static Single Assignment meets CompCert. In *ESOP '12: Proceedings of the 21th European Conference on Programming Languages and Systems*, 2012.

[15] N. Benton and N. Tabareau. Compiling Functional Types to Relational Specifications for Low Level Imperative Code. In *TLDI '09: Proceedings of the 4th International Workshop on Types in Language design and Implementation*, 2009.

[16] L. Beringer, P. Brisk, F. Chow, D. Das, A. Ertl, S. Hack, U. Ramakrishna, F. Rastello, J. Singer, , V. Sreedhar, *et al. Static Single Assignment Book.* 2012. Working draft available at `http://ssabook.gforge.inria.fr/latest/book.pdf`.

[17] Y. Bertot and P. Castéran. Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions, 2004.

[18] S. Blazy, B. Robillard, and A. W. Appel. Formal Verification of Coalescing Graph-Coloring Register Allocation. In *ESOP '10: Proceedings of the 19th European Conference on Programming Languages and Systems*, 2010.

[19] J. O. Blech, S. Glesner, J. Leitner, and S. Mülling. Optimizing Code Generation from SSA Form: A Comparison Between Two Formal Correctness Proofs in Isabelle/HOL. *Electron. Notes Theor. Comput. Sci.*, 141(2):33–51, 2005.

[20] J. Chen, D. Wu, A. W. Appel, and H. Fang. A Provably Sound TAL for Back-end Optimization. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, 2003.

[21] A. Chlipala. A Certified Type-preserving Compiler from Lambda Calculus to Assembly Language. In *PLDI '07: Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, 2007.

[22] A. Chlipala. A Verified Compiler for an Impure Functional Language. In *POPL '10: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2010.

[23] S. Conchon, J.-C. Filliâtre, and J. Signoles. Designing a Generic Graph Library using ML Functors. In *Trends in Functional Programming (TFP'07)*, New York City, USA, Apr. 2007.

[24] K. D. Cooper, T. J. Harvey, and K. Kennedy. A Simple, Fast Dominance Algorithm. Available online at `www.cs.rice.edu/~keith/Embed/dom.pdf`, 2000.

[25] The Coq Development Team. *The Coq Proof Assistant Reference Manual (Version 8.3pl1)*, 2011.

[26] K. Crary. Toward a Foundational Typed Assembly Language. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2003.

[27] K. Crary and R. Harper. Mechanized Definition of Standard ML (alpha release), 2009. `http://www.cs.cmu.edu/~crary/papers/2009/mldef-alpha.tar.gz`.

[28] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13:451–490, 1991.

[29] M. Erwig. Inductive graphs and functional graph algorithms. *J. Funct. Program.*, 11(5):467–492, Sept. 2001.

[30] L. Georgiadis and R. E. Tarjan. Dominator tree verification and vertex-disjoint paths. In *SODA '05: Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2005.

[31] L. Georgiadis, R. F. Werneck, R. E. Tarjan, and D. I. August. Finding dominators in practice. In *Proceedings of the 12th Annual European Symposium on Algorithms*, 2004.

[32] R. Ghiya and L. J. Hendren. Putting Pointer Analysis to Work. In *POPL '98: Proceedings of the 25th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1998.

[33] J. B. Kam and J. D. Ullman. Global Data Flow Analysis and Iterative Algorithms. *Journal of the ACM*, 23(1):158–171, Jan. 1976.

[34] R. A. Kelsey. A Correspondence between Continuation Passing Style and Static Single Assignment Form. In *IR '95: Papers from the 1995 ACM SIGPLAN workshop on Intermediate representations*, 1995.

[35] G. A. Kildall. A unified approach to global program optimization. In *POPL '73: Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1973.

[36] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, 2009.

[37] G. Klein, T. Nipkow, and T. U. München. A Machine-checked Model for a Java-like Language, Virtual Machine and Compiler. *ACM Trans. Program. Lang. Syst.*, 28:619–695, 2006.

[38] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, 2004.

[39] C. A. Lattner. *Macroscopic Data Structure Analysis and Optimization*. PhD thesis, Champaign, IL, USA, 2005.

[40] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1:121–141, 1979.

[41] S. Lerner, T. Millstein, E. Rice, and C. Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2005.

[42] X. Leroy. A Formally Verified Compiler Back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[43] The LLVM Development Team. *The LLVM Reference Manual (Version 2.6)*, 2010. http://llvm.org/releases/2.6/docs/LangRef.html.

[44] R. Lo, F. Chow, R. Kennedy, S.-M. Liu, and P. Tu. Register promotion by sparse partial redundancy elimination of loads and stores. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, 1998.

[45] W. Mansky and E. L. Gunter. A Framework for Formal Verification of Compiler Optimizations. In *ITP '10: Interactive Theorem Proving 2010*, 2010.

[46] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. A Case for an SC-Preserving Compiler. *SIGPLAN Not.*, 46:199–210, June 2011.

[47] Y. Matsuno and A. Ohori. A Type System Equivalent to Static Single Assignment. In *PPDP '06: Proceedings of the 8th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, 2006.

[48] V. S. Menon, N. Glew, B. R. Murphy, A. McCreight, T. Shpeisman, A.-R. Adl-Tabatabai, and L. Petersen. A Verifiable SSA Program Representation for Aggressive Compiler Optimization. In *POPL '06: Proceedings of the 33th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2006.

[49] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[50] S. Nagarakatte. *Practical Low-Overhead Enforcement of Memory Safety for C Programs*. PhD thesis, Philadelphia, PA, USA, 2012.

[51] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *PLDI '09: Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation*, 2009.

[52] G. C. Necula. Translation validation for an optimizing compiler. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, 2000.

[53] NIST. *NIST Juliet Test Suite for C/C++*, 2010. http://samate.nist.gov/SRD/testCases/suites/Juliet-2010-12.c.cpp.zip.

[54] M. Nita and D. Grossman. Automatic Transformation of Bit-level C Code to Support Multiple Equivalent Data Layouts. In *CC'08: Proceedings of the 17th International Conference on Compiler Construction*, 2008.

[55] M. Nita, D. Grossman, and C. Chambers. A Theory of Platform-dependent Low-level Software. In *POPL '08: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2008.

[56] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, June 1999.

[57] M. Paleczny, C. Vick, and C. Click. The Java HotSpot(TM) Server Compiler. In *Proceedings of the 2001 Symposium on Java(TM) Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, Berkeley, CA, USA, 2001. USENIX Association.

[58] R. V. Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '99. IBM Press, 1999.

[59] D. Schouten, X. Tian, A. Bik, and M. Girkar. Inside the Intel compiler. *Linux J.*, 2003, Feb. 2003.

[60] P. Sewell, F. Zappa Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strniša. Ott: Effective Tool Support for the Working Semanticist. *J. Funct. Program.*, 20(1):71–122, 2010.

[61] J. Souyris, V. Wiels, D. Delmas, and H. Delseny. Formal Verification of Avionics Software Products. In *FM '09: Proceedings of the 2nd World Congress on Formal Methods*, volume 5850, Berlin, Heidelberg, 2009. Springer-Verlag.

[62] V. C. Sreedhar and G. R. Gao. A Linear Time Algorithm for Placing $\phi$-nodes. In *POPL '95: Proceedings of the 22nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1995.

[63] M. Stepp, R. Tate, and S. Lerner. Equality-Based Translation Validator for LLVM. In *CAV '11: Proceedings of the 23rd International Conference on Computer Aided Verification*, 2011.

[64] Z. T. Sudipta Kundu and S. Lerner. Proving Optimizations Correct Using Parameterized Program Equivalence. In *PLDI '09: Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation*, 2009.

[65] D. Syme. Reasoning with the Formal Definition of Standard ML in HOL. In *Sixth International Workshop on Higher Order Logic Theorem Proving and its Applications*, 1993.

[66] Z. Tatlock and S. Lerner. Bringing Extensibility to Verified Compilers. In *PLDI '10: Proceedings of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation*, 2010.

[67] J.-B. Tristan, P. Govereau, and G. Morrisett. Evaluating Value-graph Translation Validation for LLVM. In *PLDI '11: Proceedings of the ACM SIGPLAN 2011 Conference on Programming Language Design and Implementation*, 2011.

[68] J.-B. Tristan and X. Leroy. Formal Verification of Translation Validators: a Case Study on Instruction Scheduling Optimizations. In *POPL '08: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2008.

[69] J.-B. Tristan and X. Leroy. Verified Validation of Lazy Code Motion. In *PLDI '09: Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation*, 2009.

[70] J. B. Tristan and X. Leroy. A Simple, Verified Validator for Software Pipelining. In *POPL '10: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2010.

[71] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[72] B. Yakobowski. Étude sémantique d'un langage intermédiaire de type Static Single Assignment. Rapport de dea (Master's thesis), ENS Cachan and INRIA Rocquencourt, Sept. 2004.

[73] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *PLDI '11: Proceedings of the ACM SIGPLAN 2011 Conference on Programming Language Design and Implementation*, 2011.

[74] A. Zaks and A. Pnueli. Program Analysis for Compiler Validation. In *PASTE '08: Proceedings of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2008.

[75] J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *POPL '12: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2012.

[76] J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formal verification of SSA-based optimizations for LLVM. In *PLDI '13: Proceedings of the ACM SIGPLAN 2012 Conference on Programming Language Design and Implementation*, 2013.

[77] J. Zhao and S. Zdancewic. Mechanized Verification of Computing Dominators for Formalizing Compilers. In *CPP '12: The Second International Conference on Certified Programs and Proofs*, 2012.

[78] L. Zhao, G. Li, B. De Sutter, and J. Regehr. ARMor: Fully Verified Software Fault Isolation. In *EMSOFT '11: Proceedings of the 9th ACM International Conference on Embedded Software*, 2011.

# Appendix A: The Correctness of `vmem2reg-O1`

This appendix presents the correctness of `vmem2reg-O1` (which are fully verified in Coq). The following diagram shows the proof structure for the correctness of `vmem2reg-O1`.

$$\begin{array}{lcr}
prog_0 \sqsupseteq prog_1 = prog_0\{ac_0(f_0)/f_0\} \sqsupseteq prog_2 = prog_1\{ac_1(f_1)/f_1\} & \sqsupseteq \cdots \sqsupseteq & prog_n = prog_{n-1}\{ac_{n-1}(f_{n-1})/f_{n-1}\} \\
 & & \shortparallel ? \\
prog_0 & \sqsupseteq ? & prog' = prog_0\{\overleftrightarrow{AC}(f_0)/f_0\} \\
 & & \shortparallel ? \\
prog_0 & \sqsupseteq ? & prog'' = prog_0\{\overline{AC}(f_0)/f_0\}
\end{array}$$

Suppose that we optimize the function $f_0$ in a program $prog_0$. Let $ac_i$ be the elimination action applied in the $i$-th step of `vmem2reg`, $f_i$ be the function after the $i$-th step from $f_0$, and $prog_i$ be the function after the $i$-th step from $prog_0$. By composing Theorem 40, we can prove that $prog_n$ refines $prog_0$:

**Theorem 41** (Composition of `vmem2reg`). *If $\vdash prog_0$, then $\vdash prog_n$ and $prog_0 \sqsupseteq prog_n$.*

To show that `vmem2reg-O1` is correct, we only need to show that $prog_0\{\overline{AC}(f_0)/f_0\}$ equals to $prog_n$. To simplify reasoning, we prove that both of them equal to $prog_0\{\overleftrightarrow{AC}(f_0)/f_0\}$.

## 1. The equivalence of $prog_0\{\overleftrightarrow{AC}(f_0)/f_0\}$ and $prog_n$

**Theorem 42.** *If $prog \vdash f_0$, then $prog_0\{\overleftrightarrow{AC}(f_0)/f_0\} = prog_n$.* [1]

## 2. The equivalence of $prog_0\{\overline{AC}(f_0)/f_0\}$ and $prog_0\{\overleftrightarrow{AC}(f_0)/f_0\}$

[1] Here, we omit the proofs. See our Coq development.

Figure 8.14 gives the following observations: 1) the SSA form ensures that the original $AC$ is acyclic, and forms a tree; 2) $\overleftrightarrow{AC}$ and $\overline{AC}$ computed from an acyclic $AC$ form the same "flattened" tree. To formalize the observations, we first define the following functions and predicates:

1. Paths $\rho$: connected definitions. For example, $< r_3, r_2, r_1, r_0 >$ denotes

$$r_0 \to r_1 \to r_2 \to r_3$$

2. $(r, val) \in \rho$: an edge from $r$ to $val$ is in a path $\rho$.
3. $< r, \rho >$: extend the path $\rho$ at head with $r$.
4. $< \rho, val >$: extend the path $\rho$ at tail with $val$.
5. $\rho;\rho'$: connect two paths $\rho$ and $\rho'$.
6. $(r, val) \in AC$: $AC$ maps $r$ to $val$.
7. $\rho \subseteq AC$: $\forall r$, if $(r, val) \in \rho$, then $(r, val) \in AC$.
8. $AC \vdash val_1 \overset{\rho}{\longrightarrow}{}^{*} val_2$: a path $< val_2, \rho >$ from $val_1$ to $val_2$ defined in terms of $AC$—$< val_2, \rho > \subseteq AC$.
9. $AC \vdash val_1 \overset{\rho}{\multimap}{}^{*} val_2$: $AC \vdash val_1 \overset{\rho}{\longrightarrow}{}^{*} val_2$ and $val_2$ is a root of $AC$—$AC[val_2] = \cdot$. We also define an algorithm for finding roots:

$$\begin{aligned} AC \Uparrow r:\ (AC_1; r \mapsto r_1, AC_2) \Uparrow r &= (AC_1; AC_2) \Uparrow r_1 \\ \_ &= r \end{aligned}$$

10. $AC \Rightarrow AC'$: $\forall r\, val$, if $AC \vdash r \overset{\rho}{\longrightarrow}{}^{*} val$, then $\exists \rho', AC' \vdash r \overset{\rho'}{\longrightarrow}{}^{*} val$.
11. $AC =] AC'$: $\forall r\, val$, if $AC \vdash r \overset{\rho}{\multimap}{}^{*} val$, then $\exists \rho', AC' \vdash r \overset{\rho'}{\multimap}{}^{*} val$.
12. $AC [=] AC'$: $AC =] AC'$ and $AC' =] AC$.
13. $\neg @ AC$: $\forall \rho \subseteq AC$, $\rho$ is acyclic.
14. $\Uparrow AC$: if $AC = AC_1; r \mapsto val, AC_2$, then $r \notin$ **codom** $(AC_2)$.
15. **uniq**$AC$: the domain of $AC$ is unique.

16. $\square AC$: $\forall (r_1, r_2) \in AC, \neg\exists val.(r_2, val) \in AC$.

## 2.1 *AC* is well-formed

**Lemma 43.** *If $prog \vdash f$, $fheader\{\overline{b}\} = f$, and $AC$ = flat_map (rev (snd (find_stld_pairs_block r (STLD_INIT, $\emptyset$)))) $\overline{b}$, then* **uniq***AC and* $\neg @AC$.

## 2.2 The equivalence of *AC* and $\overrightarrow{AC}$

We first prove the facts about substituting codomains of $AC$—$AC\{val/r\}$, which are useful for reasoning about $\overrightarrow{AC}$.

**Lemma 44.** *If $(r, val) \in AC$ and $\neg val$* **uses** *$r'$, then $(r, val) \in AC\{val'/r'\}$.*

**Lemma 45.** *If $AC \vdash r \overset{\rho}{\longrightarrow}{}^{*} val$ and $r' \notin$* **rl** *$< val, \rho >$, then $AC\{val'/r'\} \vdash r \overset{\rho}{\longrightarrow}{}^{*} val$ (Here, rl denotes removelast.)*

**P**roof (sketch): Because $r' \notin$ **rl** $< val, \rho >$, all targets of the edges in $< val, \rho >$ do not use $r'$. By Lemma 44, we prove that $AC\{val'/r'\}$ has the same path from $r$ to $val$. $\square$

**Lemma 46.** *If $(r', val') \notin AC$ and $(r', val') \in AC\{val/r\}$, then $(r', r) \in AC$ and $val = val'$.*

**Lemma 47.** *If $(r', r) \in AC$, then $(r', val) \in AC\{val/r\}$.*

**Lemma 48.** *If* **uniq***AC, $\neg @AC$ and $(r, val) \in AC$, then $AC =] AC\{val/r\}$.*

**P**roof (sketch): Consider $AC \vdash r_0 \overset{\rho}{\multimap}{}^{*} val_0$. If $r \notin$ **rl** $< val_0, \rho >$, Lemma 45 concludes. If $r \in$ **rl** $< val_0, \rho >$, by **uniq**$AC$, $\neg @AC$ and that $val_0$ is a root, we can partition $< val_0, \rho >$ as below:

$$r_0 \overset{\rho_1}{\longrightarrow}{}^{*} r' \rightarrow r \rightarrow val \overset{\rho_2}{\multimap}{}^{*} val_0$$

Here, $r \notin$ **rl** $< r', \rho_1 >$ and $r \notin < val_0, \rho_2 >$.

Consider the path $\rho'$:

$$r_0 \overset{\rho_1}{\longrightarrow}{}^{*} r' \rightarrow val \overset{\rho_2}{\multimap}{}^{*} val_0$$

By Lemma 47, $(r', val) \in AC\{val/r\}$. By Lemma 45, $AC\{val/r\} \vdash r_0 \overset{\rho_1}{\longrightarrow}{}^{*} r'$ and $AC\{val/r\} \vdash val \overset{\rho_2}{\longrightarrow}{}^{*} val_0$. This concludes the proofs. $\square$

**Lemma 49.** *If* **uniq**$AC$ *and* $(r, val) \in AC$*, then* $AC\{val/r\} \Rightarrow AC$*.*

**P**roof (sketch): Consider $AC\{val/r\} \vdash r_0 \overset{\rho'}{\longrightarrow^*} val_0$. We can partition $\rho'$ as below:

$$r_0 \overset{\rho_0}{\longrightarrow^*} r'_0 \to val'_0 \overset{\rho_1}{\longrightarrow^*} r'_1 \to val'_1 \cdots r'_n \to val'_n \overset{\rho_n}{\longrightarrow^*} val_0$$

Here, $(r'_i, val'_i) \notin AC$, and $< r'_i, \rho_i > \subseteq AC$ when $i < n$, and $< val_0, \rho_n > \subseteq AC$.

We construct the path $\rho$:

$$r_0 \overset{\rho_0}{\longrightarrow^*} r'_0 \to r \to val \overset{\rho_1}{\longrightarrow^*} r'_1 \to r \to val \cdots r'_n \to r \to val \overset{\rho_n}{\longrightarrow^*} val_0$$

Lemma 46 shows that $(r'_i, r) \in AC$ and $val'_i = val$. Therefore, $AC \vdash r_0 \overset{\rho}{\longrightarrow^*} val_0$. □

By Lemma 48 and Lemma 49, we have that:

**Lemma 50.** *If* **uniq**$AC$*,* $\neg @AC$ *and* $(r, val) \in AC$*, then* $AC\{val/r\}[=]AC$*.*

By Lemma 49, we have that:

**Lemma 51.** *If* **uniq**$AC$*,* $(r, val) \in AC$*, and* $\neg @AC$*, then* $\neg @AC\{val/r\}$*.*

**Lemma 52.** *If* $\neg val$ **uses** $r$*, then* $r \notin$ **codom** $(AC\{val/r\})$*.*

**Lemma 53.** *If* **uniq**$AC$*, then* **uniq** $(AC\{val/r\})$*.*

We also need the following properties about weakening:

**Lemma 54.** *If* $AC_1 \Rightarrow AC_2$*, then* $AC;AC_1 \Rightarrow AC;AC_2$*.*

*Proof.* By induction of $AC$. Consider the inductive case $AC = r_0 \mapsto val_0, AC'$. Consider $AC;AC_1 \vdash r \overset{\rho}{\longrightarrow^*} val$. Partition $< r, \rho >$ into

$$r \overset{\rho_0}{\longrightarrow^*} r_0 \to val_0 \overset{\rho_1}{\longrightarrow^*} r_0 \to val_0 \cdots r_0 \to val_0 \overset{\rho_n}{\longrightarrow^*} val$$

where $(r_0, val_0) \notin < r, \rho_0 >$ and $(r_0, val_0) \notin < val_0, \rho_i >$ where $i > 0$.

Consider each $AC;AC_1 \vdash val_i \overset{\rho_i}{\longrightarrow^*} val'_i$. Because $(r_0, val_0) \notin < val_i, \rho_i >$ and IH, $AC';AC_2 \vdash val_i \overset{\rho'_i}{\longrightarrow^*} val'_i$. So, $AC;AC_2 \vdash val_i \overset{\rho'_i}{\longrightarrow^*} val'_i$. The proof concludes by $\rho'$ :

$$r \overset{\rho'_0}{\longrightarrow^*} r_0 \to val_0 \overset{\rho'_1}{\longrightarrow^*} r_0 \to val_0 \cdots r_0 \to val_0 \overset{\rho'_n}{\longrightarrow^*} val$$

□

**Lemma 55.** *If $AC_1 =]AC_2$,* $\mathbf{uniq}(AC;AC_1)$ *and* $\mathbf{dom}AC_1 = \mathbf{dom}AC_2$*, then* $AC;AC_1 =]AC;AC_2$.

*Proof.* By induction of $AC$. Consider the inductive case $AC = r_0 \mapsto val_0, AC'$. Consider $AC;AC_1 \vdash r \overset{\rho}{\multimap}{}^* val$. Partition $< r, \rho >$ into

$$r \overset{\rho_0}{\longrightarrow}{}^* r_0 \to val_0 \overset{\rho_1}{\longrightarrow}{}^* r_0 \to val_0 \cdots r_0 \to val_0 \overset{\rho_n}{\multimap}{}^* val$$

where $(r_0, val_0) \notin < r, \rho_0 >$ and $(r_0, val_0) \notin < val_0, \rho_i >$ where $i > 0$.

Consider each $AC;AC_1 \vdash val_i \overset{\rho_i}{\longrightarrow}{}^* val_i'$. Because $(r_0, val_0) \notin < val_i, \rho_i >$, $AC';AC_1 \vdash val_i \overset{\rho_i}{\longrightarrow}{}^* val_i'$. By $\mathbf{uniq}(AC;AC_1)$, $AC';AC_1 \vdash val_i \overset{\rho_i}{\multimap}{}^* val_i'$. By IH, $AC';AC_2 \vdash val_i \overset{\rho_i'}{\multimap}{}^* val_i'$. So, $AC;AC_2 \vdash val_i \overset{\rho_i'}{\longrightarrow}{}^* val_i'$.

Because $val_0$ is the root of $AC;AC_1$ and $\mathbf{dom}AC_1 = \mathbf{dom}AC_2$, $val_0$ must also be the root of $AC;AC_2$.

The proof concludes by $\rho'$:

$$r \overset{\rho_0'}{\longrightarrow}{}^* r_0 \to val_0 \overset{\rho_1'}{\longrightarrow}{}^* r_0 \to val_0 \cdots r_0 \to val_0 \overset{\rho_n'}{\multimap}{}^* val$$

□

By Lemma 55, we have:

**Lemma 56.** *If* $AC_1[=]AC_2$, $\mathbf{uniq}(AC;AC_1)$, $\mathbf{uniq}(AC;AC_2)$ *and* $\mathbf{dom}AC_1 = \mathbf{dom}AC_2$*, then* $AC;AC_1[=]AC;AC_2$.

By Lemma 54, we have:

**Lemma 57.** *If* $AC_2 \Rightarrow AC_1$*, then* $\neg @AC;AC_1 \Rightarrow \neg @AC;AC_2$.

With the above properties, we prove that $AC$ and $\overrightarrow{AC}$ are equivalent.

**Lemma 58.** *If* $\mathbf{uniq}AC$ *and* $\neg @AC$*, then* $\overrightarrow{AC}[=]AC$.

*Proof.* By induction on the length of $AC$. The base case is trivial. Consider the case $AC = r \mapsto val, AC'$. We have $\overrightarrow{AC} = r \mapsto val, \overrightarrow{(AC'\{val/r\})}$. By Lemma 50, $AC[=]r \mapsto val\{val/r\}, AC'\{val/r\}$. Because of $\neg @AC$, $val\{val/r\} = val$. We conclude by IH and Lemma 56. □

## 2.3 The equivalence of $AC$ and $\overleftarrow{AC}$

**Lemma 59.**

1. *If* $\neg @AC$, $r \in \mathbf{dom}(AC) \vee r \in \mathbf{codom}(AC)$ *and* $AC \Uparrow r = val$, *then* $AC \vdash r \overset{\rho}{\multimap}{}^* val$.

2. *If* $\mathbf{uniq} AC$, $\neg @AC$ *and* $AC \vdash r \overset{\rho}{\multimap}{}^* val$, *then* $AC \Uparrow r = val$.

**Lemma 60.**

1. $AC[r] = \lfloor cnst \rfloor$ *iff* $\overrightarrow{AC}[r] = \lfloor cnst \rfloor$.

2. $AC[r] = \cdot$ *iff* $\overrightarrow{AC}[r] = \cdot$.

**Lemma 61.** *If* $\mathbf{uniq} AC$ *and* $\neg @AC$, *then* $\overrightarrow{AC} \Rightarrow AC$ *and* $\neg @\overrightarrow{AC}$.

By Lemma 58, Lemma 60, Lemma 59 and Lemma 61,

**Theorem 62.** *If* $\mathbf{uniq} AC$ *and* $\neg @AC$, *then* $AC \Uparrow r = \overrightarrow{AC} \Uparrow r$.

**Lemma 63.** *If* $r \notin \mathbf{dom}(AC)$ *and* $r \notin \mathbf{codom}(AC)$, *then* $r \notin \mathbf{codom}(\overrightarrow{AC})$.

All elements in $\overrightarrow{AC}$ are sorted in terms of $AC$—$\Uparrow \overrightarrow{AC}$.

**Lemma 64.** *If* $\mathbf{uniq} AC$ *and* $\neg @AC$, *then* $\Uparrow \overrightarrow{AC}$.

**P**roof (sketch): By induction on the length of $AC$. Consider the case $AC = r \mapsto val, AC'$ and $\overrightarrow{AC} = r \mapsto val, \overrightarrow{(AC'\{val/r\})}$. By Lemma 51 and Lemma 53, $\neg @AC'\{val/r\}$ and $\mathbf{uniq} AC'\{val/r\}$.

Let $\overrightarrow{AC} = AC_1; r_1 \mapsto val_1, AC_2$. If $(r_1, val_1) \in AC'\{val/r\}$, the proof is by IH—$\Uparrow AC'\{val/r\}$. Otherwise, if $r_1 = r$ and $val_1 = val$, the proof is by Lemma 52 and Lemma 63. □

**Lemma 65.** *If* $\mathbf{uniq}(AC_1; r_1 \mapsto r_2, AC_2)$, *then* $(AC_1; r_1 \mapsto r_2, AC_2)(r_1) = AC_2(r_2)$.

**Lemma 66.** *If* $\mathbf{uniq}(AC_1; r_1 \mapsto r_2, AC_2)$ *and* $\Uparrow (AC_1; r_1 \mapsto r_2, AC_2)$, *then* $(AC_1; AC_2) \Uparrow r_2 = AC_2 \Uparrow r_2$.

**Lemma 67.** *If* $\mathbf{uniq} AC$ *and* $\Uparrow AC$, *then* $AC(r) = AC \Uparrow r$.

**P**roof (sketch): By induction on the length of $AC$. It is trivial if $AC$ does not map $r$. Consider the case $AC = AC_1; r \mapsto r', AC_2$.

$$\begin{array}{rcll} AC \Uparrow r & = & (AC_1; AC_2) \Uparrow r' & \text{definition} \\ & = & AC_2 \Uparrow r' & \text{By Lemma 66} \\ & = & AC_2(r') & \text{By IH} \\ & = & AC(r') & \text{By Lemma 65} \end{array}$$

□

**Theorem 68.** *If* **uniq**$AC$ *and* $\Uparrow AC$*, then* $\overleftarrow{AC}[r] = AC \Uparrow r$.

**P**roof (sketch): It is trivial if $AC$ does not map $r$. Consider the case $AC = AC_1; r \mapsto r', AC_2$.

$$\begin{array}{rcll} \overleftarrow{AC}[r] & = & AC_1'; r \mapsto (\overleftarrow{AC_2}(r')), \overleftarrow{AC_2}[r] & \text{definition} \\ & = & \overleftarrow{AC_2}(r') & \text{definition} \\ & = & AC_2 \Uparrow r' & \text{By Lemma 67} \\ & = & (AC_1; AC_2) \Uparrow r' & \text{By Lemma 66} \\ & = & AC \Uparrow r & \text{definition} \end{array}$$

□

## 2.4 The equivalence of $AC$ and $\overleftrightarrow{AC}$

**Theorem 69.** *If* **uniq**$AC$ *and* $\neg @AC$*, then* $\overleftrightarrow{AC}[r] = AC \Uparrow r$.

**P**roof (sketch):

$$\begin{array}{rcll} AC \Uparrow r & = & \overrightarrow{AC} \Uparrow r & \text{By Theorem 62} \\ & = & \overleftrightarrow{AC}[r] & \text{By Theorem 68 and Lemma 64} \end{array}$$

□

## 2.5 The equivalence of $AC$ and $\overline{AC}$

**Lemma 70.** *If* **uniq** $(AC_1; AC)$ *and* $\neg @(AC_1; AC)$*, then* $\neg @(AC_1; \overline{AC})$ *and* $\Box \overline{AC}$.

**P**roof (sketch): To streamline the presentation, we show the proofs separately in the following. We first show $\neg @(AC_1; \overline{AC})$.

1. By induction of $AC$. Consider the case $AC = r \mapsto val, AC'$. By IH, $\neg @(AC_1; r \mapsto val, \overline{AC'})$. By Lemma 49 and Lemma 57, $\neg @(AC_1; r \mapsto val\{val/r\}, (\overline{AC'})\{val/r\})$. By $\neg @(AC_1; AC)$, $val\{val/r\} = val$, so $\neg @(AC_1; r \mapsto val, (\overline{AC'})\{val/r\})$.

   It is trivial if $val$ is a constant. Suppose $val = r'$. If $(\overline{AC'})\{r'\} = r'$, it is trivial. If $(\overline{AC'})\{r'\} = val'$, then $(r', val') \in \overline{AC'}$. By acyclicity, $\neg val'$ **uses** $r$. By Lemma 44, $(r', val') \in \overline{AC'}\{r'/r\}$. By Lemma 49 and Lemma 57, $\neg @(AC_1; r \mapsto r'\{val'/r'\}, (\overline{AC'})\{r'/r\}\{val'/r'\})$. Because $\Box \overline{AC'}$ (by IH), $(\overline{AC'})\{r'/r\}\{val'/r'\} = (\overline{AC'})\{val'/r\}$. Therefore, $\neg @(AC_1; r \mapsto val', (\overline{AC'})\{val'/r\})$.

2. Proving $\Box\overline{AC}$ is equivalent to prove that if $\neg @AC$ and $\overline{AC}[r] = \lfloor val \rfloor$, then $\overline{AC}[val] = \cdot$.

   By induction on $AC$. Consider the case $AC = r \mapsto r', AC'$, and $\overline{AC} = r \mapsto val', (\overline{AC'})\{val'/r\}$ where $val' = (\overline{AC'})\{r'\}$ and $(r', val') \in \overline{AC'}$. By the first part of the proof, $\neg @\overline{AC}$.

   Suppose $\overline{AC}[r_1] = \lfloor r_2 \rfloor$. Case $r_1 = r$ and $r_2 = val'$. By acyclicity, $\neg val'\ \mathbf{uses}\, r$. By IH, $\overline{AC}[r_2] = \cdot$.

   Case $r_1 \neq r$. $\overline{AC}[r_1] = (\overline{AC'})\{val'/r\}[r_1] = \lfloor r_2 \rfloor$. Therefore, $\overline{AC'}[r_1] = \lfloor r_2' \rfloor$ where $r_2 = r_2'\{val'/r\}$. By IH, $\overline{AC'}[r_2'] = \cdot$, so $(\overline{AC'})\{val'/r\}[r_2'] = \cdot$.

   If $r_2' \neq r$, then $r_2 = r_2'$ and it is trivial. If $r_2' = r$, then $r_2 = val'$ and the proof is by IH.

□

**Lemma 71.** *If $\neg @AC$, then $\overline{AC}[=]AC$.*

**P**roof (sketch): By induction on $AC$. Consider the case $AC = r \mapsto r', AC'$. Let $val' = (\overline{AC'})\{r'\}$. By Lemma 70, $\neg @(r \mapsto r', \overline{AC'})$. So, $(r, val') \in (\overline{AC'})\{r'/r\}$.

$$\begin{array}{lll}
\overline{AC} & = & r \mapsto val', \overline{AC'}\{val'/r\} & \\
 & = & r \mapsto r'\{val'/r'\}, \overline{AC'}\{r'/r\}\{val'/r'\} & \\
 & = & (r \mapsto r', \overline{AC'}\{r'/r\})\{val'/r'\} & \\
 & [=] & (r \mapsto r', \overline{AC'}\{r'/r\}) & \text{By Lemma 50} \\
 & = & (r \mapsto r', \overline{AC'})\{r'/r\} & \text{By acyclicity} \\
 & [=] & (r \mapsto r', \overline{AC'}) & \text{By Lemma 50} \\
 & [=] & (r \mapsto r', AC') & \text{By Lemma 56 and IH}
\end{array}$$

□

By Lemma 71, Lemma 70 and Lemma 59,

**Theorem 72.** *If* **uniq**$AC$ *and* $\neg @AC$, *then* $\overline{AC}[r] = AC \Uparrow r$.

## 2.6 The equivalence of $\overline{AC}$ and $\overleftrightarrow{AC}$

By Theorem 72 and Theorem 69,

**Theorem 73.** *If* **uniq**$AC$ *and* $\neg @AC$, *then* $\overline{AC}[r] = \overleftrightarrow{AC}[r]$.

# 3. The correctness of `vmem2reg-O1`

By Theorem 41, Theorem 73, Lemma 43 and Theorem 42,

**Theorem 74** (`vmem2reg-O1` is correct)**.** *If* $f'$ = `vmem2reg-O1` $f$ *and* $\vdash prog$*, then* $\vdash prog\{f'/f\}$ *and* $prog \supseteq prog\{f'/f\}$.

# Appendix B: The Correctness of `vmem2reg-O2`

```
Record IDFstate := mkIDFst {
  IDFwrk : list l;
  IDFphi : AVLMap.t unit
}.

Definition IDFstep D DF (st : IDFstate) : AVLMap.t unit + IDFstate :=
let '(W, Φ) := st in
match W with
| nil => inl Φ
| l0::W' => inr (W'∪(DF[l0]−D−Φ), Φ∪DF[l0])
end.

Definition IDF D DF :=
  PrimIter.iterate _ _ (IDFstep D DF) (D, ∅).
```

Figure 1: The algorithm of inserting $\phi$-nodes

This appendix discusses the correctness of `vmem2reg-O2`. Note that the proofs are not fully verified in Coq yet).

We first study the algorithms used in `vmem2reg-O2` that are omitted by the main part of the dissertation.

**Lemma 75.** *The dominance frontier computation algorithm in Section 3.5 is correct: the set of blocks the algorithm calculates for a block $l_0$ equals to $l_0$'s dominance frontier.*

*Proof.* This is equivalent to show that $l_1$ is $l_0$'s dominance frontier iff $l_1$ has a predecessor $l_2$, $l_0$ dominates $l_2$, and $l_1$'s immediate dominator $l_4$ strictly dominates $l_0$. The "if" part is straight-forward. We present the "only-if" part.

Suppose $l_2$ is $l_1$'s predecessor, $l_0$ dominates $l_2$ and does not strictly dominates $l_1$. Because dominance relations form a tree, the tree path to $l_1$ and the tree path to $l_2$ must have the same prefix.

Suppose the path of $l_2$ joins $l_1$'s at $l_3$ that strictly dominates $l_1$'s immediate dominator $l_4$. Then, there must exist a path $\rho$ to $l_2$ that does not go through $l_4$. Otherwise, $l_4$ must strictly dominate $l_2$, and the tree paths of $l_1$ and $l_2$ must join at $l_4$. However, $\rho$ also reaches $l_1$. This is contradictory to that $l_4$ strictly dominates $l_1$. Therefore, $l_4$ must be in the same prefix of the two tree paths.

$l_0$ cannot dominate $l_4$. Otherwise $l_0$ strictly dominate both $l_1$ and $l_2$. Therefore, $l_0$ must be in the set of blocks calculated by the algorithm. □

Figure 1 shows the algorithm that calculates where to insert $\phi$-node [8]: given a promotable location, all the dominance frontiers of the definitions at the location need $\phi$-nodes. The definitions of a promotable location include **alloca**'s of the location, **store**'s to the location and inserted $\phi$-nodes for the location. Therefore, the algorithm needs to iteratively insert $\phi$-nodes until all the inserted $\phi$-nodes also satisfy the above requirement.

The algorithm is implemented by a primitive recursion (`PrimIter.iterate`) based on a worklist. `IDFstate` defines calculation states of each recursion step: `IDFwrk` is the worklist that records blocks to process; `IDFphi` is the blocks that need to insert $\phi$-nodes. Initially, the worklist includes blocks all with original definitions (which are denoted by $D$, and only contain **alloca**'s and **store**'s) of a promotable locations. `IDFstep`, given $D$ and dominance frontiers $DF$, implements each recursion step. If the current worklist is empty, `IDFstep` returns the inserted $\phi$-nodes, and stops the entire recursion. Otherwise, `IDFstep` picks a block from the worklist, adds the dominance frontiers that do not have the original and inserted definitions to the worklist, and inserts $\phi$-nodes for the dominance frontiers.

**Lemma 76.** `IDF` *(in Figure 1) terminates.*

*Proof.* Consider the following measure function:

$$M(W,\Phi) = |W| + N * (N - |\Phi|)$$

Here, $||$ computes the size of a set; $N$ is the number of blocks in the function `IDF` computes. It is sufficient to show that

1. $M(W,\Phi) \geq 0$.

2. If `IDF` $D\ DF\ (W,\Phi) = \mathtt{inr}(W',\Phi')$, then $M(W,\Phi) > M(W',\Phi')$.

The first fact is true because the number of inserted $\phi$-nodes cannot be greater than the number of all blocks.

Suppose $W = l_0 :: W''$, $W' = W'' \cup (DF[l_0] - D - \Phi)$ and $\Phi' = \Phi \cup DF[l_0]$.

$$\begin{aligned} M(W',\Phi') - M(W,\Phi) &= N * (|\Phi| - |\Phi'|) + |W'| - |W| \\ &= N * (|\Phi| - |\Phi \cup DF[l_0]|) + |W'' \cup (DF[l_0] - D - \Phi| - 1 - |W''| \end{aligned}$$

Consider two cases. The first case is when $DF[l_0] \not\subset \Phi$.

$$\begin{aligned} M(W',\Phi') - M(W,\Phi) &\leq N * (|\Phi| - (|\Phi| + 1)) + |W'' \cup \Phi| - 1 - |W''| \\ &\leq N * (|\Phi| - (|\Phi| + 1)) + |W''| + |\Phi| - 1 - |W''| \\ &= -N + |\Phi| - 1 \\ &< 0 \end{aligned}$$

The second case is when $DF[l_0] \subset \Phi$.

$$\begin{aligned} M(W',\Phi') - M(W,\Phi) &= N * (|\Phi| - |\Phi|) + |W''| - 1 - |W''| \\ &< 0 \end{aligned}$$

□

**Lemma 77.** `IDF` *is correct: if* `IDF` $D\,DF\ (D,\emptyset) =$ `inl`$\Phi$*, then* $\forall l_0 \in D \cup \Phi$*,* $DF[l_0] \subset D \cup \Phi$*.*

*Proof.* In general, consider the following invariant:

$$INV\,D\,DF\ (W,\Phi) = \forall l_0 \in D \cup \Phi, l_0 \in W \vee DF[l_0] \subset D \cup \Phi$$

It is sufficient to show that

If `IDF` $D\,DF\ (W,\Phi) =$ `inr`$(W',\Phi')$ and $INV\,D\,DF\ (W,\Phi)$, then $INV\,D\,DF\ (W',\Phi')$.

It is trivial if $W$ is empty. Consider $W = l_1 :: W''$, $W' = W'' \cup (DF[l_1] - D - \Phi)$ and $\Phi' = \Phi \cup DF[l_1]$. Suppose $l_0 \in D \cup \Phi'$.

1. $l_0 \in D \cup \Phi$: By assumption, $l_0 \in W \vee DF[l_0] \subseteq D \cup \Phi$.

   a) $l_0 \in W = l_1 :: W''$:

      i. $l_0 = l_1$: $DF[l_1] \subseteq \Phi \cup DF[l_1] = \Phi' \subseteq D \cup \Phi'$

      ii. $l_0 \in W''$: $l_0 \in W' = W'' \cup (DF[l_1] - D - \Phi)$

   b) $DF[l_0] \subseteq D \cup \Phi$: $DF[l_0] \subseteq D \cup (\Phi \cup DF[l_1]) = D \cup \Phi'$:

2. $l_0 \in DF[l_1] \wedge l_0 \notin D \cup \Phi$: $l_0 \in (DF[l_1] - D - \Phi) \subseteq W' = W'' \cup (DF[l_1] - D - \Phi)$.

□

By Lemma 75 and the proofs in [28], we have that

**Lemma 78.** *Given the dominance frontier calculated by the algorithm in Section 3.5,* `IDF` *and the iterated path-convergence criterion [8] specify exactly the same set of nodes at which to put* $\phi$*-nodes.*

By Lemma 75 and Lemma 77, we prove that

**Lemma 79.** *After the* $\phi$*-node insertion of* `vmem2reg-O2`*, given a* **load** *to* $r_1$ *from a promotable location,*

1. *If there exists a* **store** *with value* $val_2$ *to the promotable location at program counter* $pc_2$ *and the* **store** *is the closest one that dominates the* **load***, we have* `LAS` *(*$r_1$*,* $pc_2$*,* $val_2$*): in other words, there are no other* **store***'s to the location between the* **load** *and the* **store***.*

2. *Otherwise, we have* `LAA` $r_1$*: in other words, there are no other* **store***'s to the location between the* **load** *and the* **alloca***.*

*Proof.* We present the proofs of the first fact. Suppose between $pc_2$ and $r_1$ there exists a simple path $\rho$ that goes through another **store** to the location. Consider the closest **store** at $pc_3$ to $r_1$ on $\rho$. Because $pc_2$ is the closest **store** that dominates $r_1$, there must exist a path $\rho'$ from $pc_2$ to $r_1$ that bypasses $pc_3$, and $\rho$ and $\rho'$ join between $pc_3$ and $r_1$. In terms of the iterated path-convergence criterion and Lemma 78, a $\phi$-node and a corresponding **store** must be inserted at the joint point. Therefore, $pc_3$ is not the closest **store** to $r_1$ on $\rho$. □

Finally, by Lemma 79, we need the following extended lemma for reasoning about `vmem2reg-O2`.

**Lemma 80.** `LAS`*/*`LAA` *are correct with respect to arbitrary domination relations (Section 8.3 requires that domination relations must be in the same block).*