# Formalizing the LLVM Intermediate Representation for Verified Program Transformations *

Jianzhou Zhao Santosh Nagarakatte Milo M. K. Martin Steve Zdancewic
Computer and Information Science Department, University of Pennsylvania
jianzhou@cis.upenn.edu santoshn@cis.upenn.edu milom@cis.upenn.edu stevez@cis.upenn.edu

## Abstract

This paper presents Vellvm (*verified LLVM*), a framework for reasoning about programs expressed in LLVM's intermediate representation and transformations that operate on it. Vellvm provides a mechanized formal semantics of LLVM's intermediate representation, its type system, and properties of its SSA form. The framework is built using the Coq interactive theorem prover. It includes multiple operational semantics and proves relations among them to facilitate different reasoning styles and proof techniques.

To validate Vellvm's design, we extract an interpreter from the Coq formal semantics that can execute programs from LLVM test suite and thus be compared against LLVM reference implementations. To demonstrate Vellvm's practicality, we formalize and verify a previously proposed transformation that hardens C programs against spatial memory safety violations. Vellvm's tools allow us to extract a new, verified implementation of the transformation pass that plugs into the real LLVM infrastructure; its performance is competitive with the non-verified, ad-hoc original.

***Categories and Subject Descriptors*** D.2.4 [*Software Engineering*]: Software/Program Verification - Correctness Proofs; F.3.1 [*Logics and Meanings of Programs*]: Speficying and Verifying and Reasoning about Programs - Mechanical verification; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages - Operational semantics

***General Terms*** Languages, Verification, Reliability

***Keywords*** LLVM, Coq, memory safety

## 1. Introduction

Compilers perform their optimizations and transformations over an *intermediate representation* (IR) that hides details about the target execution platform. Rigorously proving properties about these IR transformations requires that the IR itself have a well-defined formal semantics. Unfortunately, the IRs used in main-stream production compilers generally do not. To address this deficiency, this paper formalizes both the static and dynamic semantics of the IR that forms the heart of the LLVM compiler infrastructure.

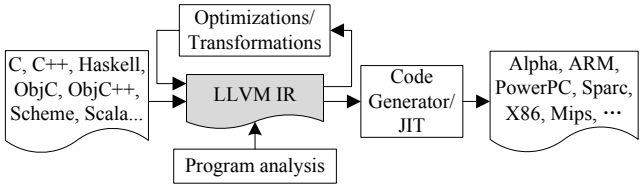


**Figure 1.** The LLVM compiler infrastructure

LLVM [19] (*Low-Level Virtual Machine*) uses a platform-independent SSA-based IR originally developed as a research tool for studying optimizations and modern compilation techniques [16]. 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 [16]. As a consequence, it has been widely used in both academia and industry.

An LLVM-based compiler is structured as a translation from a high-level source language to the LLVM IR (see Figure 1). 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.

This paper introduces Vellvm—for *verified* LLVM—a framework that includes a formal semantics and associated tools for mechanized verification of LLVM IR code, IR to IR transformations, and analyses. The description of this framework in this paper is organized into two parts.

The first part formalizes the LLVM IR. It presents the LLVM syntax and static properties (Section 2), including a variety of well-formedness and structural properties about LLVM's static single assignment (SSA) representation that are useful in proofs about LLVM code and transformation passes. Vellvm's memory model (Section 3) is based on CompCert's [18], extended to handle LLVM's arbitrary bit-width integers, padding, and alignment issues. In developing the operational semantics (Section 4), a significant challenge is adequately capturing the nondeterminism that arises due to LLVM's explicit **undef** value and its intentional underspecification of certain erroneous behaviors such as reading from uninitialized memory; this underspecification is needed to justify the correctness of aggressive optimizations. Vellvm therefore implements several related operational semantics, including a nondeterministic semantics and several deterministic refinements to facilitate different proof techniques and reasoning styles.

* This research was funded in part by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.



*POPL'12,* January 25–27, 2012, Philadelphia, PA, USA.

The second part of the paper focuses on the utility of formalizing the LLVM IR. We describe Vellvm's implementation in Coq [10] and validate its LLVM IR semantics by extracting an executable interpreter and comparing its behavior to that of the LLVM reference interpreter and compiled code (Section 5). The Vellvm framework provides support for moving code between LLVM's IR representation and its Coq representation. This 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. We demonstrate the effectiveness of this technique by using Vellvm to implement a verified instance of SoftBound [21], an LLVM-pass that hardens C programs against buffer overflows and other memory safety violations (Section 6).

To summarize, this paper and the Vellvm framework provide:

- A formalization of the LLVM IR, its static semantics, memory model, and several operational semantics;
- Metatheoretic results (preservation and progress theorems) relating the static and dynamic semantics;
- Coq infrastructure implementing the above, along with tools for interacting with the LLVM compiler;
- Validation of the semantics in the form of an extracted LLVM interpreter; and
- Demonstration of applying this framework to extract a verified transformation pass for enforcing spatial memory-safety.

## 2. Static Properties of the LLVM IR

The LLVM IR is a typed, static single assignment (SSA) [13] language that is a suitable representation for expressing many compiler transformations and optimizations. This section describes the syntax and basic static properties, emphasizing those features that are either unique to the LLVM or have non-trivial implications for the formalization. Vellvm's formalization is based on the LLVM release version 2.6, 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.

### 2.1 Language syntax

Figure 3 shows (a fragment of) 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 $mod$ 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 be function declarations, function definitions, and global variables. Figure 2 shows a small example of LLVM syntax (its meaning is described in more detail in Section 3).

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 well-formed program is compatible with the type of the value—a well-formed program can still be stuck (see Section 4.3).

[1] See `http://llvm.org/releases/2.6/docs/LangRef.html`

```
%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
}
```

**Figure 2.** An example use of LLVM's memory operations. 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.

Types $typ$ include arbitrary bit-width integers $\mathbf{i}8$, $\mathbf{i}16$, $\mathbf{i}32$, *etc.*, or, more generally, $\mathbf{i}sz$ 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. **int** $sz$ $align_0$ $align_1$ dictates that values with type $\mathbf{i}sz$ 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 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**).

| | | | |
|---|---|---|---|
| Modules | $mod, P$ | $::=$ | $\overline{layout}\ \overline{namedt}\ \overline{prod}$ |
| Layouts | $layout$ | $::=$ | $\mathbf{bigendian} \mid \mathbf{littleendian} \mid \mathbf{ptr}\ sz\ align_0\ align_1 \mid \mathbf{int}\ sz\ align_0\ align_1$ |
| | | $\mid$ | $\mathbf{float}\ sz\ align_0\ align_1 \mid \mathbf{aggr}\ sz\ align_0\ align_1 \mid \mathbf{stack}\ sz\ align_0\ align_1$ |
| Products | $prod$ | $::=$ | $id = \mathbf{global}\ typ\ const\ align \mid \mathbf{define}\ typ\ id(\overline{arg})\{\overline{b}\} \mid \mathbf{declare}\ typ\ id(\overline{arg})$ |
| Floats | $fp$ | $::=$ | $\mathbf{float} \mid \mathbf{double}$ |
| Types | $typ$ | $::=$ | $\mathbf{i}sz \mid fp \mid \mathbf{void} \mid typ* \mid [\,sz \times typ\,] \mid \{\,\overline{typ_j}^{\,j}\,\} \mid typ\ \overline{typ_j}^{\,j} \mid id$ |
| Values | $val$ | $::=$ | $id \mid cnst$ |
| Binops | $bop$ | $::=$ | $\mathbf{add} \mid \mathbf{sub} \mid \mathbf{mul} \mid \mathbf{udiv} \mid \mathbf{sdiv} \mid \mathbf{urem} \mid \mathbf{srem} \mid \mathbf{shl} \mid \mathbf{lshr} \mid \mathbf{ashr} \mid \mathbf{and} \mid \mathbf{or} \mid \mathbf{xor}$ |
| Float ops | $fbop$ | $::=$ | $\mathbf{fadd} \mid \mathbf{fsub} \mid \mathbf{fmul} \mid \mathbf{fdiv} \mid \mathbf{frem}$ |
| Extension | $eop$ | $::=$ | $\mathbf{zext} \mid \mathbf{sext} \mid \mathbf{fpext}$ |
| Cast op | $cop$ | $::=$ | $\mathbf{fptoui} \mid \mathbf{ptrtoint} \mid \mathbf{inttoptr} \mid \mathbf{bitcast}$ |
| Trunc op | $trop$ | $::=$ | $\mathbf{trunc}_{int} \mid \mathbf{trunc}_{fp}$ |
| Constants | $cnst$ | $::=$ | $\mathbf{i}sz\ Int \mid fp\ Float \mid typ * id \mid (typ*)\,\mathbf{null} \mid typ\,\mathbf{zeroinitializer} \mid typ[\,\overline{cnst_j}^{\,j}\,] \mid \{\,\overline{cnst_j}^{\,j}\,\}$ |
| | | $\mid$ | $typ\,\mathbf{undef} \mid bop\ cnst_1\ cnst_2 \mid fbop\ cnst_1\ cnst_2 \mid trop\ cnst\ \mathbf{to}\ typ \mid eop\ cnst\ \mathbf{to}\ typ$ |
| | | $\mid$ | $cop\ cnst\ \mathbf{to}\ typ \mid \mathbf{getelementptr}\ cnst\ \overline{cst_j}^{\,j} \mid \mathbf{select}\ cnst_0\ cnst_1\ cnst_2 \mid \mathbf{icmp}\ cond\ cnst_1\ cnst_2$ |
| | | $\mid$ | $\mathbf{fcmp}\ fcond\ cnst_1\ cnst_2$ |
| Blocks | $b$ | $::=$ | $l\ \overline{\phi}\ \overline{c}\ tmn$ |
| $\phi$ nodes | $\phi$ | $::=$ | $id = \mathbf{phi}\ typ\ \overline{[val_j, l_j]}^{\,j}$ |
| Tmns | $tmn$ | $::=$ | $\mathbf{br}\ val\ l_1\ l_2 \mid \mathbf{br}\ l \mid \mathbf{ret}\ typ\ val \mid \mathbf{ret\ void} \mid \mathbf{unreachable}$ |
| Commands | $c$ | $::=$ | $id = bop(\,\mathbf{int}\ sz)val_1\ val_2 \mid id = fbop\ fp\ val_1\ val_2 \mid id = \mathbf{load}\,(typ*)val_1\ align$ |
| | | $\mid$ | $\mathbf{store}\ typ\ val_1\ val_2\ align \mid id = \mathbf{malloc}\ typ\ val\ align \mid \mathbf{free}\,(\,typ*\,)\ val$ |
| | | $\mid$ | $id = \mathbf{alloca}\ typ\ val\ align \mid id = trop\ typ_1\ val\ \mathbf{to}\ typ_2 \mid id = eop\ typ_1\ val\ \mathbf{to}\ typ_2$ |
| | | $\mid$ | $id = cop\ typ_1\ val\ \mathbf{to}\ typ_2 \mid id = \mathbf{icmp}\ cond\ typ\ val_1\ val_2 \mid id = \mathbf{select}\ val_0\ typ\ val_1\ val_2$ |
| | | $\mid$ | $id = \mathbf{fcmp}\ fcond\ fp\ val_1\ val_2 \mid option\ id = \mathbf{call}\ typ_0\ val_0\ \overline{param}$ |
| | | $\mid$ | $id = \mathbf{getelementptr}\,(\,typ*\,)\ val\ \overline{val_j}^{\,j}$ |

**Figure 3.** Syntax for LLVM. Note that this figure omits some syntax definitions (*e.g., cond*—the comparison operators) for the sake of space; they are, of course, present in Vellvm's implementation. Some other parts of the LLVM have been omitted from the Vellvm development; these are discussed in Section 5.

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.

### 2.2 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 [13].

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 ensures that the physical sizes of such *typ*s are positive, 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. Within a block $insn_1$ dominates $insn_2$ if $insn_1$ appears before $insn_2$ in a program order. A block labeled $l_1$ dominates a block labeled $l_2$ if every execution path from the program entry to $l_2$ must go through $l_1$.

The Vellvm formalization provides an implementation of this dominator analysis using a standard dataflow fixpoint computation [14]. It also proves that the implementation is correct, as stated in the following lemma, which is needed to establish preservation of the well-formedness invariants by the operational semantics (see Section 4).

LEMMA 1 (Dominator Analysis Correctness).

- *The entry block of a function dominates itself.*
- *Given a block $b_2$ that is an immediate successor of $b_1$, all the strict dominators of $b_2$ also dominate $b_1$*

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.

## 3. A Memory Model for Vellvm

### 3.1 Rationale

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 mem-

ory 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 paper we adopt a memory model that is based on the one implemented for CompCert [18]. 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 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 reuse 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.

### 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 2 gives a small example program that uses these operations. Importantly, the type annotations on these operations can

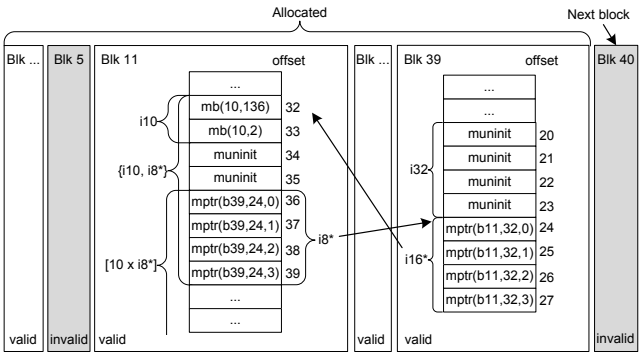


**Figure 4. Vellvm's byte-oriented memory model.** This figure shows (part of) a memory state that might be reached by calling the function `foo` from Figure 2. Blocks less than 40 were allocated; the next fresh block to allocate is 40. Block 5 is deallocated, and thus marked invalid to access; fresh 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 $24^{\text{th}}$ 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 $32^{\text{nd}}$ memory cell of block 11.

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.

### 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:

$$\begin{array}{lrl} \text{Memory cells } mc & ::= & \mathsf{mb}(sz, byte) \mid \mathsf{mptr}(blk, ofs, idx) \\ & \mid & \mathsf{muninit} \end{array}$$

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 10 (see Figure 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 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 4 gives a concrete example of such a memory state for the program in Figure 2.

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.

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

**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.

### 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.

$$\begin{array}{ccccccc} & & \mathrm{LLVM}_{ND} & & & & \\ & & \cup & & & & \\ \mathrm{LLVM}_{Interp} & \approx & \mathrm{LLVM}_{D} & \gtrsim & \mathrm{LLVM}^{*}_{DFn} & \gtrsim & \mathrm{LLVM}^{*}_{DB} \end{array}$$

**Figure 5.** Relations between different operational semantics. Each equivalence or inclusion is justified by a proof in Vellvm.

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 [24, 25].

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.

## 4. Operational Semantics

Vellvm provides several related operational semantics for the LLVM IR, as summarized in Figure 5. The most general is $\mathrm{LLVM}_{ND}$, a small-step, nondeterministic evaluation relation given by rules of the form $\mathit{config} \vdash S \twoheadrightarrow S'$ (see Figure 6). This section first motivates the need for nondeterminism in understanding the LLVM semantics and then illustrates $\mathrm{LLVM}_{ND}$ by explaining some of its rules. Next, we introduce several equivalent deterministic refinements of $\mathrm{LLVM}_{ND}$—$\mathrm{LLVM}_{D}$, $\mathrm{LLVM}^{*}_{DB}$, and $\mathrm{LLVM}^{*}_{DFn}$—each of which has different uses, as described in Section 4.4. All of these operational semantics must handle various error conditions, which manifest as partiality in the rules. Section 4.3 describes these error conditions, and relates them to the static semantics of Section 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.

### 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 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.

Configurations:

Fun tables $\theta$ $::=$ $v \mapsto id$ Globals $g$ $::=$ $id \mapsto v$ Configurations *config* $::=$ $mod, g, \theta$

Nondeterministic Machine States:

Value sets $V$ $::=$ $\{v \mid \Phi(v)\}$ Locals $\Delta$ $::=$ $id \mapsto V$ Allocas $\alpha$ $::=$ $[] \mid blk, \alpha$

Frames $\Sigma$ $::=$ $fid, l, \overline{c}, tmn, \Delta, \alpha$ Call stacks $\overline{\Sigma}$ $::=$ $[] \mid \Sigma, \overline{\Sigma}$ Program states $S$ $::=$ $M, \overline{\Sigma}$

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

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

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

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

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

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

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

**Figure 6.** $\text{LLVM}_{ND}$: Small-step, nondeterministic semantics of the LLVM IR (selected rules).

Nondeterminism shows up in two ways in the $\text{LLVM}_{ND}$ 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.

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, $\text{LLVM}_{ND}$ 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, 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
```

### 4.2 Nondeterministic operational semantics of the SSA form

The $\text{LLVM}_{ND}$ 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 (see the top of Figure 6). The globals and function pointer maps are initialized from the module definition when the machine is started.

The $\text{LLVM}_{ND}$ rules relate machine states to machine states, where a machine state takes the form of a memory $M$ (from Section 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 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 $\mathbf{eval}_{ND}$ 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, $\mathbf{evalbop}_{ND}$ implements binary operations, computing the result set by combining all possible pairs drawn from its input sets. $\text{LLVM}_{ND}$'s **malloc** behaves as described in Section 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).

There is one other 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 possibility of self loops and dependencies among the $\phi$ nodes. For example the well-formed code fragment below has a circular dependency between `%x` and `%z`

```
blk:
  %x = phi i32 [ %z, %blk ], [ 0, %pred ]
  %z = phi i32 [ %x, %blk ], [ 1, %pred ]
  %b = icmp leq %x %z
  br %b %blk %succ
```

If control enters this block from `%pred`, `%x` will map to `0` and `%z` to `1`, which causes the conditional branch to succeed, jumping back to the label `%blk`. The new values of `%x` and `%z` should be `1` and `0`, and not, `1` and `1` as might be computed if they were handled sequentially. This update of the local state is handled by the $\mathbf{computephinodes}_{ND}$ function in the operational semantics, as shown, for example, in rule NDS_BR_TRUE.

### 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, $\mathbf{eval}_{ND}(g, \Delta, \texttt{\%x})$ 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 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 2 (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 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.

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.

The difficult part of showing that defined variables dominate their uses in the current continuation is proving that control-transfers maintain the dominance property [20]. If a program branches from a block $b_1$ to $b_2$, the first command in $b_2$ can use either the falling-through variables from $b_1$, which must be defined in $\Delta$ by Lemma 1, or the variables updated by the $\phi$s at the beginning of $b_2$. This latter property requires a lemma showing that $\mathbf{computephinode}_{ND}$ behaves as expected.

THEOREM 3 (Progress for $\text{LLVM}_{ND}$). *If the pair (config, S) is well formed, then either S has terminated successfully or* $\text{Stuck}(\mathit{config}, S)$ *or there exists S' such that 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.

### 4.4 Deterministic refinements

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

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 $\text{LLVM}_{D}$ is an instance of the $\text{LLVM}_{ND}$ 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 $\text{LLVM}_{ND}$, 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 $\textbf{eval}_{ND}$). 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 $\text{LLVM}_{ND}$, $\text{LLVM}_{D}$ also has the preservation and progress properties.

***Big-step semantics*** Vellvm also provides big-step operational semantics $\text{LLVM}^{*}_{DFn}$, which evaluates a function call as one large step, and $\text{LLVM}^{*}_{DB}$, 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 $\text{LLVM}^{*}_{DB}$ rules to hide the details about the intermediate states. To handle memory effects, we use a simulation relation that uses *symbolic evaluation* [22] 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 [32].

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 [22].

***Relationships among the semantics*** Figure 5 illustrates how these various operational semantics relate to one another. Vellvm provides proofs that $\text{LLVM}^{*}_{DB}$ simulates $\text{LLVM}^{*}_{DFn}$ and that $\text{LLVM}^{*}_{DFn}$ simulates $\text{LLVM}_{D}$. 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 $\text{LLVM}^{*}_{DFn}$ semantics matches the corresponding state at the function call reached in $\text{LLVM}_{D}$. Note that in the deterministic setting, one-direction simulation implies bisimulation [18]. Moreover, $\text{LLVM}_{D}$ is a refinement instance of the nondeterministic $\text{LLVM}_{ND}$ 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.

## 5. Vellvm Infrastructure and Validation

This section briefly describes the Coq implementation of Vellvm and its related tools for interacting with the LLVM infrastructure. It also describes how we validate the Vellvm semantics by extracting an executable interpreter and comparing its behavior to the LLVM reference interpreter.

### 5.1 The Coq development

Vellvm encodes the abstract syntax from Section 2 in an entirely straightforward way using Coq's inductive datatypes (generated in a preprocessing step via the Ott [27] tool). The implementation uses Penn's Metatheory library [4], 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), 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. In total, the syntax and static semantics constitute about 2500 lines of Coq definitions and proof scripts.

Vellvm's memory model implementation extends CompCert's with approximately 5000 lines of code to support integers with arbitrary precision, padding, and an experimental treatment of casts that has not yet been needed for any of our proofs. On top of this extended memory model, all of the operational semantics and their metatheory have been proved in Coq. In total, the development represents approximately 32,000 lines of Coq code. Checking the entire Vellvm implementation using `coqc` takes about 13.5 minutes on a 1.73 GHz Intel Core i7 processor with 8 GB RAM. We expect that this codebase could be significantly reduced in size by refactoring the proof structure and making it more modular.

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 5200 lines of OCaml-LLVM bindings. This library also supports pretty-printing of the AST's; this code was also useful in the extracted the interpreter.

***Omitted details*** This paper 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 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; (3) 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 extend-

ing 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, though 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.

### 5.2 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 $\mathrm{LLVM}_D$ 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, $\mathrm{LLVM}_{Interp}$, which is a deterministic functional interpreter implemented in the computational fragment of Coq. $\mathrm{LLVM}_{Interp}$ is proved to be bisimilar to $\mathrm{LLVM}_D$, 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 yet implemented in Vellvm.

Although replacing the Coq data-structures by native ones invalidates 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.

## 6. Verified SoftBound

SoftBound [21] is 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, while a real SoftBound implementation needs to consider all of the LLVM IR shown in Figure 3, the memory model, and the operational semantics of the LLVM. Also the original proof ensures the correctness only with respect to a specification that the SoftBound instrumentation must implement, but 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 section 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. 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.

### 6.1 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 $\mathrm{SBtrans}(-)$ that instruments the LLVM code to propagate metadata.
4. We prove that a program if $\mathrm{SBtrans}(P) = \lfloor P' \rfloor$ then P', when run on the $\mathrm{LLVM}_D$, simulates P running on SBspec.

***The SoftBound specification*** Figure 7 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.

Nondeterministic rules:

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 $\hat{S} ::= M, MM, \overline{\hat{\Sigma}}$

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

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

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

Deterministic configurations:

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

**Figure 7.** SBspec: The specification semantics for SoftBound. Differences from the $\text{LLVM}_{ND}$ rules are highlighted.

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).

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 1 (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 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 4 (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 5 (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.

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 $[\mathbf{null}, \mathbf{null})$ 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

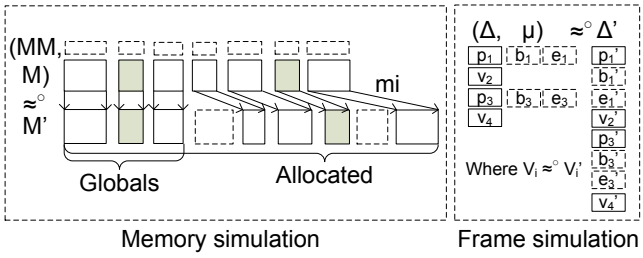


**Figure 8.** Simulation relations of the SoftBound pass

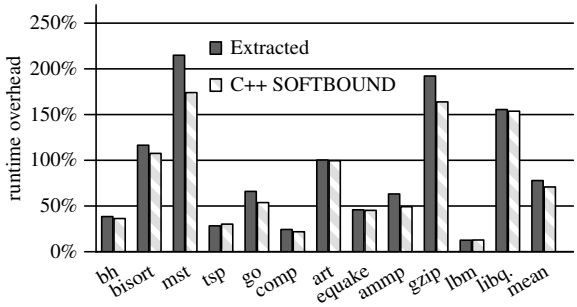


**Figure 9.** Execution time overhead of the extracted and the C++ version of SoftBound

$v$, that means the pointer being loaded was not stored with valid bounds, so **findbounds** returns $[\mathbf{null}, \mathbf{null})$ 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 interger, even if that integer was originally obtained from a valid pointer. Following the same design choice, **findbounds** returns $[\mathbf{null}, \mathbf{null})$ for pointers cast from integers. **checkbounds** fails when a program accesses such pointers.

THEOREM 6 (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 8 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 7. *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 4, is an instance of the non-deterministic SBspec.

***The correctness of SoftBound***

THEOREM 8 (SoftBound is correct). *Let $\mathrm{SBtrans}(P) = \lfloor P' \rfloor$ 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.*

### 6.2 Extracted verified implemention 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 [23]. 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. 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 9 reports the execution time overheads (lower is better) of extracted SoftBound (leftmost bar of each benchmark) and the C++ imple-

mentation (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 $\mathbf{undef}$, $\mathbf{undef}$ was propagated as its base and bound. Subsequent compiler transformations may instantiate the undefined base and bound with defined values that allow the $\mathbf{checkbounds}$ 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 $\mathbf{null}$ or pointers pointing to an offset from $\mathbf{null}$. 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.

## 7. Related Work

***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 [2], Foundational Typed Assembly Language [11], Standard ML [12, 30], and (a substantial subset of) Java [15].

***Verified compilers*** Compiler verification has a considerable history; see the bibliography [18] for a comprehensive overview. Other research has also used Coq for compiler verification tasks, including much recent work on compiling functional source languages to assembly [5, 8, 9].

Vellvm is closer in spirit to CompCert [18], 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 [6], and other back-end transformations. CompCert has also certified some advanced compiler optimizations [32–34] using translation validation [22, 26]. The XCERT project [29, 31] extends the CompCert compiler by a generic translation validator based on SMT solvers.

***Other mechanization efforts*** The verified software tool-chain project [3] assures that the machine-checked proofs claimed at the top of the tool-chain hold in the machine language program. Typed assembly languages [7] provide a platform for proving back-end optimizations. Similarly, The Verisoft project [1] also attempts to mathematically prove the correct functionality of systems in automotive engineering and security technology. ARMor [37] guarantees control flow integrity for application code running on embedded processors. The Rhodium project [17] uses a domain specific language to express optimizations via local rewrite rules and provides a soundness checker for optimizations

***Validating LLVM optimizations*** The CoVac project [36] develops a methodology that adapts existing program analysis techniques to the setting of translation validation, and reports on a prototype tool that applies their methodology to verification of the LLVM compiler. The LLVM-MD project [35] validates LLVM optimizations by symbolic evaluation. The Peggy tool performs translation validation for the LLVM compiler using a technique called equality saturation [28]. These applications are not fully certified.

## 8. Conclusion

Although we do not consider it in this paper, our intention is that the Vellvm framework will serve as a first step toward a fully-verified LLVM compiler, similar to that of Leroy *et al.*'s CompCert [18]. Our Coq development extends some of CompCert's libraries and our LLVM memory model is based on CompCert's memory model. The focus of this paper is the LLVM IR semantics itself, the formalization of which is a necessary step toward a fully-verified LLVM compiler. Because much of the complexity of an LLVM-based compiler lies in the IR to IR transformation passes, formalizing correctness properties at this level stands to yield a significant payoff, as demonstrated by our SoftBound case study, even without fully verifying a compiler.

## Acknowledgments

This research was funded in part by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government. This research was funded in part by DARPA contract HR0011-10-9-0008 and ONR award N000141110596.

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1116682, CCF-1065166, and CCF-0810947. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

## References

[1] 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.

[2] A. W. Appel. Foundational proof-carrying code. In *LICS '01: Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, 2001.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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`.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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 32th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2005.

[18] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

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

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

[26] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS '98: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 1998.

[27] P. Sewell, F. Zappa Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strniša. Ott: Effective tool support for the working semanticist. In *ICFP '07: Proceedings of the 9th ACM SIGPLAN International Conference on Functional Programming*, 2007.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.