Ph.D. DISSERTATION

A Validated Semantics for LLVM IR

LLVM 컴파일러의 중간언어를 위한 의미를
정의하고 검증하기

August 2021

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Juneyoung Lee

Abstract

Intermediate representation (IR) is a language that is used internally by a compiler to represent programs. Translation to an IR should preserve guarantees from the source language's specification because they enable various optimizations. This naturally makes an IR a language that is rich with high-level information.

In LLVM, the semantics of important high-level features in IR was not rigorously defined. It caused compiler optimizations in LLVM to use different interpretations, and bad interactions between the optimizations resulted in miscompilation bugs that are hard to fix. To solve this problem, the IR's semantics must be defined precisely. Then, optimizations that are incorrect with respect to the chosen semantics must be fixed. Both processes are challenging because LLVM is a large, fastly evolving software.

This thesis proposes (1) formal semantics of LLVM IR that resolves critical problems that we have found in the old IR semantics, making it consistent (2) a translation validation framework for LLVM's optimizations to validate the new semantics. We show that the old semantics of undefined behavior and memory model in the IR cannot explain important optimizations in LLVM. We propose new semantics that solves this problem. Next, we present Alive2, a translation validation framework for LLVM based on the new semantics. Alive2 relies on an automatic theorem prover to validate optimizations without any hints from LLVM. It supports most of integer and float operations, memory operations, function calls, and branches. To make validation practical, resources used by the tool is bounded.

The new formal semantics of undefined behavior has been adopted by LLVM. The ‘freeze’ instruction that is proposed by us is officially added into LLVM 10.0, and the official document is updated to use our semantics. Also, critical problems in the old memory model we have found were shared with compiler developers, and patches have landed in LLVM to fix it. Alive2 has found more than 50 miscompilation bugs in LLVM so far and is used daily by LLVM developers.

Keywords: Compiler, programming language, compiler intermediate representation, formal language semantics, formal memory model, translation validation, automatic verification, SMT solver

Student Number: 2016-21227

Acknowledgements

I am fortunate to have Nuno P. Lopes and Chung-Kil Hur as my advisors. They taught me how to attack intricate problems and find concise and straightforward solutions. Also, I could learn what is being a great researcher like from their enthusiasm and immersion in research. I was impressed with the infinite amount of trust and opportunities they have given to me as well.

It was a great pleasure to be with my brilliant colleagues in Software Foundations Lab and ROPAS Lab. I will never forget the moments when discussing interesting research issues as well as non-research topics with them. They made my life as a graduate student full of happiness and joy. Mainly, I'd like to thank senior lab members – Yoonseung Kim, Youngju Song, Dongkwon Lee, and Sehun Kim. I believe they are the members who made the atmosphere of the two groups healthy and made the members get along well for a long time. I am a beneficiary of the healthy atmosphere.

Finally, I thank my parents for supporting my study. They also gave me thoughtful advice when help was necessary. Also, I thank my brother for being grown as a great mechanical engineer.

Contents

Chapter 1

Introduction

The formal semantics of a programming language is a mathematical specification providing the semantics of programs written in it. There are several approaches. Denotational semantics defines a function from programs to a partially ordered set that is called domain. Axiomatic semantics defines a set of axioms describing assertions that must hold after a program's execution. Operational semantics defines a set of rules describing state transitions on an abstract machine during the execution of a program. There are other variants as well, and the defined semantics works as a foundation for formal verification, static analysis and compiler correctness.

Ideally, people should define the formal semantics when they devise a new programming language. However, most of the real-world programming languages have their semantics written in prose first, and then their formal semantics follows. The reason is that the human-friendly form is more efficient in carrying the high-level intuition of the programming language's semantics. For example, to formally define the meaning of reading a memory location in operational semantics, one needs to define a value, pointer, and memory in mathematical terms first. However, programmers already have an intuition about what reading a memory means, and the formal definition of memory read should not be very different from this. Therefore, a simple specification stating the intuition in prose suffices for explaining the majority of programs.

Then, when is defining formal semantics necessary? To talk about this, imagine a case that *sometimes* happens – reading an out-of-bounds pointer in C. In x86 assembly language, reading a pointer raises a fault if the page is protected, or silently continues otherwise. We cannot use this semantics for C however, because it makes important compiler optimizations incorrect[1]. An alternative option is to simply state that a C program that reads an out-of-bounds pointer is a badly written program that a compiler does not need to care about. This is not what an assembler does, but it is not problematic because C's abstract machine does not need to be that of assembly. Instead, the design choice must be explicitly stated somewhere, so programmers are aware of this. These details are described in prose in the official documents of C, C++, Rust, and many other languages.

However, the full semantics written in prose is not handy in a programmer's perspective. Since modern programming languages have rich features, they are more than small textbooks, making programmers sometimes hard to know the exact result of a program. This implies that language semantics should be described in a *machine-friendly* way so that programmers can simply run their programs and see the results. Sometimes, text specifications are ambiguous or omit important details as well. In the case of C, unclear definition of a pointer caused lengthy discussions and publications [1–5].

[1]For example, register promotion can decrease the memory usage of a program, potentially changing the result of reading an out-of-bounds pointer from success to an abnormal exit.

A formally defined semantics is unambiguous by construction because it is defined using mathematical terms. It is machine-friendly as well – finding a correct implementation of an executable semantics is reduced to properly choosing data structures that precisely and efficiently represent its mathematical objects. It fits well with formal verification techniques such as machine-checked proof writing or automated theorem proving. One can prove important properties of the semantics such as type preservation using a proof assistant. Due to these reasons, formal semantics of real-world languages have become robust foundations for formal verification and compiler correctness.

In this thesis, we introduce the formal semantics of a compiler's intermediate representation (IR). IR is a language as well as an internal data structure of a compiler for representing compiling programs. Each compiler has its own IR(s) that is appropriate to represent programs written in the desired source languages. Compiler translates source programs into IR, performs optimizations on it, and emits assembly[2]. Showing the correctness of these three steps relies on the precise semantics of IR.

One of hardships in devising semantics for IR is that it is non-trivial to find the mathematical definition of the high-level information that is derived from the source language's specification. Translation to IR should preserve guarantees from the source language's specification because they enable various optimizations. This naturally makes IR a language that is rich with high-level information. Finding its mathematical definition is not trivial because its definition written in prose cannot be directly translated. Furthermore, we found that the semantics of important high-level features in IR were sometimes not rigorously defined in the official documents. This caused existing compiler optimizations to rely on different semantics, even introducing end-to-end miscompilation bugs.

[2]In fact, compilers have more than three steps because they have several IRs.

Another challenge is dealing with compiler optimizations that are many and changing fast. Compiler optimizations are constantly changing since a modern compiler evolves quickly. Also, it is unrealistic for compiler developers to manually write a proof whenever they update an optimization. To explain optimizations, we should devise a way of quickly *validing* formal semantics with respect to the current compiler optimizations. This is connected to finding a good compiler verification technique that fits our goal.

Among real-world compilers, we target the IR of the LLVM compiler infrastructure because it is extensively used by frontend languages as well as formal verification frameworks. Also, its prose semantics is described in the official specification in detail [6]. Note that our work is not the first formalization of LLVM IR. Our work has two important contributions that are absent from previous work, which are summarized below.

1. Implementing our new IR semantics in LLVM. We found critical problems in the semantics of LLVM IR, proposed solutions for these, and implemented our new formal semantics in the official releases of LLVM. The previous semantics of LLVM IR had problems in the notion of undefined behavior and its memory model. To resolve these problems, we proposed a new formal semantics that fixes these issues. We implemented our semantics in LLVM via a number of patches (Sections 3.7 and 4.8).

2. A validated IR semantics. We rigorously validated our semantics using *translation validation*. Since LLVM is written in C++ and constantly evolving via daily contributions, formally verifying the codebase is not realistic with the current verification technology. Instead, we implemented Alive2 that is an automatic compiler optimization validator. It uses an SMT solver to automatically check the agreement between our formal semantics and the transformations that happened during compilation of a *specific* program. It supports most of integer and float operations, memory operations, function calls, and branches. Many incorrect transformations in LLVM are fixed to comply with the validated semantics and the official specification was updated to address the issues we have found (Sections 5.9.2 and 5.9.3). Currently, Alive2 is being used daily by LLVM developers to show the validity of optimizations during the code review process.

1.1 Intermediate Representation

Compilation consists of a sequence of individual transformations. Intermediate representation (IR) is a language that is used in these internal steps. IR's syntax and semantics is tailored for easy implementation of a high-performance compiler.

LLVM has several intermediate representations, each of which is tailored to a different kind of transformations during compilation. Among them, LLVM IR is the top-most target-independent language where most of the inter/intraprocedural optimizations such as loop unroll/unswitch, inlining, auto-vectorization, global value numbering, and scalar replacement of aggregates happen. It also works as a language that various frontends – including Clang, Swift, and Rust – emit.

Fig. 1.1 illustrates a subset of features of LLVM IR. It has a function `@fn` taking two 32-bit integer arguments and returning a 32-bit integer. It contains three basic blocks (`ENTRY`, `THEN`, `ELSE`) each of which ends with either a branch instruction (`br`) or return instruction (`ret`). LLVM IR's instructions typically have two input operands on the right hand side and one output register on the left-hand side.

```
define i32 @fn(i32 %a, i32 %b) {
ENTRY:
  %t = add i32 %a, %a      ; %t := %a + %a
  %c = icmp eq i32 %t, 0   ; %c := Is %t == 0?
  ; If %c is true, goto THEN, ELSE otherwise
  br i1 %c, label %THEN, label %ELSE

THEN:
  %q = shl i32 %a, 2       ; %q := %a << 2
  ret i32 %q

ELSE:
  %r = and i32 %b, 1       ; %r := %b & 1
  ret i32 %r
}
```

Figure 1.1: Example LLVM IR function

```
int y;
if (cond)
  y = a + b;
else
  y = a + c;
```

```
ENTRY:
  br i1 %cond, label %BB1, label %BB2
BB1:
  %tmp1 = add i32 %a, %b
  br label %BB
BB2:
  %tmp2 = add i32 %a, %c
  br label %BB
BB:
  %y = phi i32 [%tmp1, %BB1], [%tmp2, %BB2]
```

Figure 1.2: A simple C program and an equivalent LLVM IR that uses a ϕ node.

Static Single Assignment. To facilitate the development of compiler optimizations, LLVM IR has a static single assignment (SSA) form [7]. In the SSA form, each variable is assigned exactly once at its definition. When the variable is to be used, the assignment operation must have been executed before the use site is reached. In other words, the definition must *dominate* the use site. Using the SSA form is beneficial because it is not necessary to implement an expensive flow-sensitive analysis to track the value of a variable. Also, the dominance constraint prevents hoisting uses above its definition, blocking incorrect code motions.

```
                              define i32 @_Z1fRi(i32* nonnull align 4
int f(int &x) {                                 dereferenceable(4) %x) {
  int y = 0;                    %y = alloca i32, align 4
  x = 1;                        store i32 0, i32* %y, align 4
  // This returns 0.            store i32 1, i32* %x, align 4
  return y;                     %temp = load i32, i32* %y, align 4
}                               ret i32 %temp
                              }
```

Figure 1.3: A simple C++ function and its translation to LLVM IR with -O0. For readability, a temporary storage for `x` is omitted and variables are renamed.

To represent a variable whose value changes in a flow-sensitive manner, a ϕ node is necessary. A ϕ node is a pseudo-instruction whose result depends on the control flow. The `phi` instruction takes a list of values as well as the predecessor blocks. Fig. 1.2 shows a simple C program where the variable y is assigned differently depending on the control-flow and an equivalent LLVM IR that uses the `phi` instruction.

Guarantees From Source Language's Specification. Compiler can rely on the specification of a frontend language for better code generation. For example, the C standard enforces that pointers must be aligned according to the pointee types. Given this information, memory accesses can be translated into efficient assembly commands that use a specific alignment.

These informations are recorded as assumptions in IR programs. In general, the `llvm.assume` instruction can be used to state that its condition operand must hold at a program point. There are two additional ways to specify assumptions in the LLVM IR. First, a function attribute or instruction flag specific to the information can be attached to the corresponding place in IR. Second, if such guarantee is common across various source languages, it is directly supported by the semantics of basic instructions.

For example, consider a simple C++ function `f` (Fig. 1.3). There are three interesting guarantees that C++ standard gives for `f`.

1. `x` is a 4-bytes dereferenceable non-null pointer because it has a reference type. As shown on the right, this is encoded in LLVM IR using the `nonnull` and `dereferenceable(4)` function attributes.

2. `x` and `y` are 4-bytes aligned. This is represented by attaching '`align 4`' to the function argument as well as `alloca`, `load`, and `store` instructions.

3. Writing a value to `x` cannot affect the value stored at `y` and vice versa, because y is a freshly allocated storage object. Unlike the previous two guarantees, this does not appear in the IR program syntactically. Rather, it is the IR's underlying memory model that must support.

Formally defining the semantics of an IR includes mathematically describing the meaning of these language constructs.

1.2 Formally Defining the Semantics of an IR

To describe the behavior of a program, we are going to define an abstract machine for the language. In this thesis, the formal semantics of LLVM's IR is described in an operational semantics style. A program state and transition rules for the abstract machine are defined.

Memory Model. A memory model defines a pointer, memory, and the behavior of memory operations. For x86-64, a pointer is defined as an 64-bit non-negative integer. A memory is a function from pointers to 8-bit non-negative integers. The semantics of a load instruction that did not trap is written as a transition rule stating that it reads the bytes at the location from the memory with a given size, interprets it in little endian, and stores the value in a register.

After the transition, a register file is updated to contain the loaded value. Store is similarly defined and it represents how a memory is updated. These instructions can fail (raise a fault) if the location is protected. This behavior must be described in separate rules.

Then, can we reuse assembly's memory model for LLVM IR? It turns out that we cannot, because pointers in LLVM IR must carry additional information: aliasing locations. As mentioned before, a pointer that is passed as a function argument cannot alias a local allocation. In assembly's memory model, there is no such guarantee. They can alias because the caller can guess the address of a stack variable and pass it. Therefore, the assembly's memory model cannot support optimizations in LLVM.

Whatever memory model we choose for an IR, we must check whether the memory model is consistent with respect to the implementation. If the new memory model makes some compiler optimization unsound, there is a mismatch between the model and compiler developers' reasoning. In this case, either (1) the optimization must be fixed, or (2) the memory model must be amended to support it. It depends on how crucial the optimization is for programs' performance.

Undefined Behavior. Defining the formal semantics of IR is linked to rigorously defining the meaning of assumption-carrying language constructs. This is important for compiler correctness because many optimizations rely on them. For example, we have the `nonnull` attribute to state that a pointer argument cannot be `NULL`. Optimizers can rely on this attribute to simplify pointer comparisons (e.g., '`p == null`').

However, it is possible at run time a null pointer was passed to a `nonnull` argument. It is because `nonnull` is not a syntactic constraint: we do not know whether an unknown pointer variable will have `NULL` or not in general. What is the state of an abstract machine after a null pointer is passed to `nonnull` argument? A straightforward solution would be that the machine is stuck. Another common way to express this is that such program has *undefined behavior*. Since the abstract machine is stuck in that case, the following instructions can safely assume that the pointer argument is never a null pointer in their execution. This definition seems reasonable, and this is what LLVM's official document stated in the past as well.

However, there are other transformations using `nonnull` in LLVM as well. If a value analysis concludes that an argument is given non-null pointers only, it attaches `nonnull` to the function argument. A question is whether the transformation based on non-null pointer analysis is consistent with the intepretation. It turns out that it is not, and the mismatch was detected during validation of its semantics (Section 5.9.3). The official definition of `nonnull` is fixed to use an alternative semantics after our findings.

1.3 Validating IR Semantics

The semantics of an IR is validated if the compiler implementation is correct with respect to the semantics. Naturally, validation of IR semantics is deeply related to techniques for ensuring correctness of compilers.

Existing approaches for ensuring correctness of compilers can be categorized into three groups: compiler fuzzing, compiler verification, and translation validation. First, compiler fuzzing is a technique to randomly generate a program as well as its input and check whether the compiled binary has an expected behavior. These tools have been very successful in finding bugs in optimizers [8–11], but they cannot ensure the *absence* of bugs. Second, compiler verification is an approach to verifying the compiler implementation using a formal verification technique. However, formally verifying the LLVM implementation is very challenging because (1) it is a large software written in C++, (2) the implementation is constantly changing, and (3) it relies on fairly complex data structures to boost compilation time whose specification must be defined as well.

The third technique – the one we would like to use – is translation validation. Translation validation sits in-between compiler fuzzing and compiler verification. Given a source program and target (generated) program, a validator proves that *the* compilation is correct. A challenge is whether the validatior can prove the correctness without any help from human. We found that, thanks to the advances in automatic theorem proving, this is becoming a realistic goal for middle-sized programs.

A validator is an independent software and it is normally unimpacted by the changes in the LLVM implementation. Validating the semantics using translation validation is lightweight because it does not require verification of compiler's code. Also, if the existing semantics is found invalid, we can quickly switch to an alternative semantics by fixing the validator with a low cost. Furthermore, making validators easy to use allows compiler developers to participate in checking the validity of IR semantics without knowledge in theorem proving.

1.4 Contributions

This thesis presents a validated formal semantics for LLVM IR. We introduce the two contributions of our work.

The first contribution is devising a formal semantics of LLVM IR's undefined behavior (Chapter 3) and memory model (Chapter 4). The old semantics of undefined behavior and memory model in the IR were unclear, causing inconsistencies in important compiler optimizations. This caused miscompilations of real-world applications as well. We present new formal semantics that effectively addresses the problems. The new formal semantics of undefined behavior has been adopted by LLVM (Section 3.7). Also, the problem in LLVM IR's memory model we have found is shared with compiler developers and motivated several patches (Section 4.8).

The second one is Alive2, an SMT-based *bounded* translation validation framework. (Chapter 5). Given source and optimized IR functions, the correctness of the transformation is encoded into an SMT formula and checked by an SMT solver. It supports most of integer and float operations, memory operations, function calls, and branches. A challenge is how to encode the SMT formula properly so that an SMT solver can effectively solve it.

We closely worked with LLVM developers to make LLVM consistent with our validated semantics. Many LLVM optimizations that are incorrect with respect to the new semantics were finally removed or properly fixed (Sections 5.9.2 and 5.9.3). By June 2021, Alive2 had found more than 50 miscompilation bugs in LLVM, and the online version of Alive2[3] has been used by more than 100 LLVM patches during the code review phase.

Published papers This thesis is based on the following publications.

1. Taming Undefined Behavior in LLVM, PLDI'17 [12].

2. Reconciling High-level Optimizations and Low-level Code in LLVM, OOPSLA'18 [13].

3. Alive2: Bounded Translation Validation for LLVM, PLDI'21 [14]

4. An SMT Encoding of LLVM's Memory Model for Bounded Translation Validation, CAV'21 [15].

[3] `https://alive2.llvm.org`

The text of this thesis contains copies of sentences, paragraphs, figures, and experimental results from them.

Chapter 2

Background

In this chapter, we overview compiler intermediate representation, formal semantics of programming languages, compiler correctness, and verification of compilers.

2.1 Intermediate Representation

Intermediate representation (IR) is a language that is used internally by compilers. IR's syntax and semantics is tailored for easy and correct implementation of a high-performance compiler.

Static Single Assignment. A significant improvement of IR syntax was made by the invention of static single assignment (SSA) form [7]. If an IR program is in the SSA form, each variable can be assigned only once, and its uses must be dominated by the assignment. This form allows compiler optimizations to find the value of a variable without performing flow-sensitive analyses. Several extensions have been made to make it suitable with loops [18], information [19], arrays [20], and memory accesses [21].

Converting non-SSA programs into SSA form requires inserting ϕ nodes whose values depend on the control flow. Minimally inserting ϕ nodes is important for faster compilation and smaller memory footprint. There have been several works to minimize the number of inserted ϕ nodes with small time complexity [16, 17].

Memory SSA. Values stored in memory and their uses can be represented in SSA form [21]. This is called Memory SSA. There are three instructions in Memory SSA: `DEF`, `USE`, or `PHI`. `DEF` creates a new memory state. IR instructions updating memory is represented as `DEF` in Memory SSA. `USE(M)` states that the corresponding IR instruction reads a value from memory `M`. `PHI` corresponds to the ϕ node in the SSA.

Using Memory SSA, we can sparsely represent the alias information between memory accesses. Consider this program:

```
; Assume that p and q do not overlap
store i32 10, p ; M1 = DEF(M0)
store i32 20, q ; M2 = DEF(M1)
r = load p      ; USE(M2)      ; optimize this to USE(M1)
```

The first store is represented as `DEF(M0)` because it updates the initial memory `M0`. `M1` is a memory after the store to `M0`, and `M2` is a memory after `M1`'s update. The load is represented as `USE(M2)` because it reads from the latest memory `M2`. Note that it is valid to optimize `USE(M2)` to `USE(M1)`. Since `q` does not alias `p`, the load does not need the second store's update.

Another valid optimization is transforming `DEF(M1)` to `DEF(M0)`. This is analogous to allowing reordering between the two stores. However, if the program is too large, this optimization can lead to insertion of many `PHI` instructions. Due to this reason, LLVM does not do this optimization. It tracks only one live

DEF node per basic block [22].

IRs in LLVM. Besides LLVM IR, LLVM has three more intermediate representations: SelectionDAG, MachineIR, and MCInst.

In LLVM, instruction selection translates each basic block in LLVM IR into a graph representation called SelectionDAG. In SelectionDAG, instructions are represented as graph nodes and their relation as edges. In the beginning of instruction selection, SelectionDAG mostly contains target-independent nodes. At the end of instruction selection, majority of them are lowered into target-dependent ones. As in LLVM IR, SelectionDAG layer has many optimizations. However, their scope is limited to a single basic block because a graph contains instructions in a block only.

At the end of instruction selection, SelectionDAG is translated into MachineIR. MachineIR is a low-level representation of a program whose structure is similar to that of LLVM IR. It tracks low-level information such as constant pool and jump tables. Register allocation maps virtual registers in MachineIR into physical registers. MachineIR is translated into MCInst, a data structure for emitting assembly.

IRs in GCC. GCC has three main IRs: GENERIC, GIMPLE, and RTL. GENERIC is an IR for representing programs generated from compiler frontends. GENERIC contains high-level language constructs such as loops and OpenMP directives. GIMPLE is a three-address representation lowered from GENERIC via gimplification. Its syntactic property changes over transformations. Most of target-independent compiler optimizations happen after GIMPLE is converted into SSA form. RTL is a low-level intemediate representation that may contain target-dependent instructions.

MLIR. Multi-Level Intermediate Representation (MLIR) [23] is a compiler infrastructure for defining custom intermediate representation and reusing it across different compilers. In MLIR, one can define a dialect that is a small set of instructions and types for a specific purpose. For example, `tensor` dialect contains a tensor type as well as basic operations on it. `memref` dialect contains a pointer (`memref`) type as well as related operations such as allocation, deallocation, load, and store. Then, an IR program is represented using a composition of multiple dialects.

The implementation of a dialect typically contains compiler optimizations as well. For example, `memref` dialect implementation contains simple optimizations on memory access operations, and compiler developers can reuse them. Also, there are transformations across dialects that can be reused as well. Lowering `tensor` operations to `memref` operations is called bufferization because it realizes conceptual tensor operations into memory accesses.

Formalization of IRs. Vellvm [24] is a formalization of parts of the LLVM IR in Coq, and K-LLVM [25] is a formalization of LLVM IR in K framework [26]. Both formalizations do not attack the inconsistencies described in this thesis.

CompCert [27] includes full formalization of multiple three-address code-based IRs that are used throughout its pipeline. There is also an extension to CompCert that includes the formalization of an SSA-based IR [28].

2.2 Formal Semantics of Programming Languages

The formal semantics of a language describes the meaning of a program in a mathematical notation. The formal semantics of C [5, 27] with concurrency [29–31], Java [32], Rust [33, 34] as well as low-level languages such as x86-64 [35], ARM, RISC-V [36] have been devised recently. They mathematically define the behavior of a program which becomes a rigorous foundation for compiler correctness.

In this section, we will introduce three important concepts: undefined behavior, nondeterminism, and a memory model.

2.2.1 Undefined Behavior

Some programming languages define a set of erroneous operations that may cause the abstract machine to misbehave. These operations are said to have undefined behaviors (UB), and it is the result of design choices that can simplify the implementation of a platform. The burden of avoiding these behaviors is then placed upon the platform's users.

The best-known examples of undefined behaviors in programming languages come from C and C++, ranging from simple local operations (overflowing signed integer arithmetic) to global program behaviors (race conditions and violations of type-based aliasing rules). Undefined behaviors facilitate optimizations by permitting a compiler to assume that programs will only execute defined operations.

There are two different ways in operational semantics to state that the behavior of executing an instruction is undefined. The first one is to use a notion of 'stuck'. A program is stuck if there is no applicable small-step rule at a non-terminal state[1]. In this scheme, any execution that is unspecified in the language semantics has undefined behavior. The second one is to explicitly define an undefined state and state in the small-step rules that certain operations reach to the state. Each method is just a different representation of the other one. In this thesis, the first one ('stuck') is used unless it is stated otherwise.

[1]A terminal state is the state of a program that has safely exited.

2.2.2 Nondeterminism

A program has nondeterministic execution if it has more than one reachable state from the same input. For example, imagine a program that runs two threads in parallel.

```
// thread 1          |   // thread 2
print("1\n")         |   print("2\n")
```

If this programming language has interleaving semantics without any scheduling policy, one of these threads will be nondeterministically chosen and executed. Assuming that `print` is an atomic operation[2], the trace of this program is either '`print("1\n")`,`print("2\n")`' or '`print("2\n")`,`print("1\n")`'.

In the real world, the operating system's scheduler (as well as CPU's one if it has) will algorithmically choose which thread to run. Therefore, multithreaded programs in the real world are not purely nondeterministic. If one wants to take the role of a scheduler into consideration, the scheduler is typically described as an oracle that returns which thread to execute next. An oracle describes the interaction of a program with outer space (e.g., keyboard inputs). In this thesis, executions depending on an oracle are not considered nondeterministic.

In terms of compiler correctness, it is allowed for a compiler to remove one or more traces from them. For the above example, it is valid to translate it into assembly that always prints `2` after `1`. This supports linking to a thread library that can schedule threads deterministically, various optimizations on atomic operations, compilation to ISA having strong synchronizations only, and the "roach motel" ordering in Java Memory Model [29]. The precise definition of compiler correctness with respect to nondeterminism will be described in Section 2.3.

[2] Otherwise, the program is racy. In C/C++, the behavior of a racy program is undefined.

2.2.3 Memory Model

The memory model for a programming language determines how programs are permitted to observe and modify storage. From the perspective of operational semantics, the memory model of a language is described as a mathematical definition of memory and inference rules describing the output state after executing memory-related instructions.

Different languages have different memory models. For example, references in Java are pointer-like in that they uniquely identify objects in memory, but programs are not allowed to construct references into the middle of objects or to fabricate references from scratch. In contrast, assembly language allows arbitrary memory locations to be inspected and modified with no restrictions whatsoever on how addresses are computed.

C and C++ occupy an interesting niche. They are intended to be low-level languages; systems software—operating system kernels, virtual machine managers, embedded firmware, programming language runtimes, etc.—tends to be built in one or the other. To support these applications, pointers into objects can be constructed using pointer arithmetic, and pointers may be converted to integers and integers to pointers. However, despite their low-level character, the C and C++ memory models also incorporate higher-level features. For example, the compiler is permitted to assume (with some restrictions) that a pointer to one allocated object is not used as the basis for creating a pointer to another object. This causes finding a valid formal memory model for C and C++ a hard problem [5,37]

It is challenging to develop memory models for concurrent programming languages. Since hardwares as well as compiler optimizations can reorder memory operations, a multithreaded program may exhibit behavior that cannot be explained using interleaving semantics. A promising semantics [29] explains such behavior by introducing a notion of *promise*. A thread can nondeterministically promise to write a value to a certain location if it can fulfill its promise in the future. The promised store is indistinguishable from other stores in other threads' perspectives. The semantics is extended to explain concurrent programs in ARM/RISC-V [38], support global optimizations [30], a stronger notion of data-race freedom [31], and non-volatile memory accesses [39].

Chakraborty and Vafeiadis [40] formalize the semantics of parts of the concurrency-related instructions of LLVM IR.

2.3 Compiler Correctness

To state the correctness of compilation, we need to define a relation that must hold between the behavior of a source (input) and target (output) program. The relation must be strict enough to filter out incorrect transformations but weak enough to allow valid compiler optimizations. The relation is called refinement because the source program acts as a specification of the target program. A commonly used relation is behavioral refinement that relates the observable behavior of the source and target program [27].

2.3.1 Behavioral Refinement

Compilation from a source program P_s to a target program P_t is correct if refinement holds between the observable behaviors of P_s and P_t. An observable behavior of a program is either (1) a possibly infinite trace of observable events (e.g., system calls, volatile memory accesses) during its execution, or

(2) undefined[3][4]. In small-step operational semantics, a program's behavior is undefined if the program reaches to a state has no applicable rule (a.k.a. *stuck*).

For closed programs P_s and P_t, behavioral refinement holds if for any input I, either (1) P_s's behavior is undefined, or (2) P_t's behavior is defined and the observable behavior of P_s and P_t is equivalent. We will use notation $\mathbf{B}(P_s) \sqsupseteq \mathbf{B}(P_t)$ to represent the behavioral refinement between P_s and P_t. Behavioral refinement is transitive: for any programs P_1, P_2, P_3, if $\mathbf{B}(P_1) \sqsupseteq \mathbf{B}(P_1)$ and $\mathbf{B}(P_2) \sqsupseteq \mathbf{B}(P_3)$ holds, then $\mathbf{B}(P_1) \sqsupseteq \mathbf{B}(P_3)$ holds. This naturally implies that if all transformations in a compiler are individually correct, so does the end-to-end compilation.

Nondeterministic Behavior. If either the source or target language can exhibit nondeterminism, the definition of behavioral refinement is expanded. Behavioral refinement is defined in terms of set inclusion. $\mathbf{B}(P_s) \sqsupseteq \mathbf{B}(P_t)$ holds if for any input I (1) P_s's observable behavior set $\mathbf{B}(P_s)$ contains undefined behavior, or (2) P_t's observable behavior set $\mathbf{B}(P_t)$ does not have undefined behavior and $\mathbf{B}(P_s) \supseteq \mathbf{B}(P_t)$ holds.

2.3.2 Simulation Relation

A technique that is frequently used to prove the behavioral refinement of two programs is to (1) define a simulation relation and (2) use adequacy property [42]. Informally speaking, we relate equivalent program states in P_s and P_t's executions including their terminated states, and show that the initial states of P_s and P_t are indeed related. We will use notation $s \overset{e}{\hookrightarrow} s'$ to represent a small-step

[3]One missing case is an infinite loop without having any observable event. It is undefined behavior in C/C++, but allowed behavior in type-safe languages such as Rust. This case is omitted for simplicity.

[4]It is not trivial to find a definition of a program behavior that is (1) general enough to encompass nondeterministic behavior and infinite events (2) friendly to writing machine-checked proofs. [41] has a nice introduction about this issue and provides its own solution.

transition from a program state $s \in \text{State}$ to another state $s' \in \text{State}$ raising an event e. If $e = \tau$, the step has no event. The program text and program counter are embedded in State. State_s is a set of program states of P_s and State_t is a set of program states of P_t.

A relation $R \in \text{State}_s \times \text{State}_t$ is a simulation if for any $(s_1, s_2) \in R$, the following predicate holds[5]:

$$\forall s_2' \in \text{State}_t \,.\, s_2 \overset{e}{\hookrightarrow} s_2' \implies \exists s_1', (s_1', s_2') \in R \wedge s_1 \overset{\tau}{\hookrightarrow}^* \overset{e}{\hookrightarrow} \overset{\tau}{\hookrightarrow}^* s_1'$$

$\overset{\tau}{\hookrightarrow}^*$ is a multi-step transition consisting of zero or more silent small steps (a.k.a. *stuttering*). We will use $s_1 \sim_R s_2$ to state that s_2 simulates s_1 ($(s_1, s_2) \in R$).

Construction of R. As mentioned before, every terminated program state is related to each other. A simulation relation can be built in an incremental manner from this base case. However, this cannot relate non-terminating executions because they will never terminate. In order to deal with this issue, R and an event trace must be coinductively defined. The detail of this case is out of the scope of this thesis.

Another approach to constructing R is to define which states in the source and target program are *similar*. In CompCert, each transformation defines its own `match_states`(s_1, s_2) that is a predicate stating that s_1 and s_2 are similar. Unlike the generic definition of a simulation, its definition only depends on the current states. Its definition is loose enough to allow slightly different states, but the predicate itself must be a simulation relation.

`match_states` is tailored to the optimization. For example, if an optimization affects registers' values only, source and target memories are simply stated as equal.

[5]For brevity, we omit the undefined behavior and terminated cases.

Proving Behavioral Refinement. Once R is built, we can check the behavioral refinement of P_s and P_t by checking whether their initial states are related by R. This is called adequacy property. If P_s and P_t are closed programs, the initial states are simply the program states at the beginning of `main` function. If they are open programs, they are initial states in the function entry with properly related input variables and memories.

2.3.3 Contextual Refinement

It is common that a program consists of multiple source files. In order to verify the compilation of an open program, contextual refinement is often used. An open program t contextually refines s if $C[t]$ refines $C[s]$ for all closing program contexts C [43]. Note that this is stronger than the behavioral refinement: proving contextual refinement for all modules means that behavioral refinement holds for the full programs, but not vice versa.

A simulation relation can be used to prove contextual refinement as well. We can define match_states for relating program states at function call boundaries. In Section 5.6, we will define *state refinement* which is a kind of match_states that is general enough to cover intraprocedural optimizations.

2.4 Verifying Compilers

Ensuring the correctness of compilers is crucial for the correctness of software relying on it. Compiler bugs have changed the behavior of `SQLite`'s memory allocator [44], `git`'s `diff` [45], introduced exploitable security holes in web browsers' JavaScript Just-In-Time compilers [46–48], and used to introduce a backdoor in `sudo` [49]. An even more complicated kind of bug is that a compiler miscompiled itself and the resulting compiler miscompiled another program. [50].

In order to ensure the correctness of compilers, various efforts have been made in the research community. Existing approaches can be categorized into three groups: compiler fuzzing, compiler verification, and translation validation.

2.4.1 Compiler Fuzzing

Fuzzing tools have been very successful in finding bugs in optimizers. Randomly generated programs are compiled, carefully chosen inputs are given to the binaries, and their outputs are compared with previously known answers or outputs from other compilers' binaries. Tools like Csmith [8], EMI [9], YARPgen [10] and SPE [11] have found hundreds of bugs in commercial compilers, including GCC, LLVM, and MSVC. Marcozzi et al. [51] studied the impact of bugs found by fuzzers and verification tools.

However, they cannot ensure the absence of bugs. Since there is an infinite number of valid programs as well as possible inputs, testing alone cannot entirely give a correctness guarantee.

2.4.2 Compiler Verification

An alternative approach is compiler verification, where the compiler/optimizer is verified once and for all. CompCert [27] is a compiler for C that is formalized and verified in Coq [52]. Its core property – behavioral refinement – is proven by the developers and mechanically checked. Further work has extended CompCert with verified peephole optimizations [53], verified polyhedral model-based optimizations [54], and a verified SSA-based middle-end optimizer [55]. However, it still lacks important optimizations such as vectorization because a human must write its correctness proof manually. Automatically writing the proof is still a far-reaching goal due to its sheer complexity.

There are several frameworks tailored for verifying a limited set of compiler optimizations. Cobalt [56] and its successor Rhodium [57] are frameworks to specify and automatically verify peephole optimizations and dataflow analyses. PEC [58] extends this work with support for loop-manipulating optimizations by reusing some of the TVOC's techniques [59].

Alive [60] is an automatic verification tool for LLVM's peephole optimizations. AliveInLean [61] is a reimplementation of Alive that was specified and verified in Lean. Newcomb et al. [62] present an automatic verification tool for soundness and termination of Halide's rewriting system. CORK [63] is an automatic equivalence checker that supports loop optimizations over rational numbers.

2.4.3 Translation Validation

Translation validation is a technique that sits in-between compiler fuzzing and compiler verification. Given a source program and its compiled assembly, a validator mechanically checks whether the semantic preservation holds for *the* pair.

Early translation validation (TV) tools supported only transformations that did not change the control-flow or the loop structure (e.g., loop unroll, software pipeline) of the program [64]. TV tools were then extended to accept hints from the compiler (witnesses) to simplify their job [65–68]. Crellvm [69] proposes a witnessed TV framework for LLVM whose validator is formally verified in Coq. Witnesses are especially useful for validating optimizations that change the loop structure. Witnesses do not have to be correct, since they are validated by the TV tool. When the compiler provides sufficient information, validation can be done mostly syntactically [69]; there is something of a tradeoff between the amount of changes required in the compiler and the computational and implementation complexity of the TV tool.

Some tools, such as CoVaC [70], Counter [71], DDEC [72], JTFG [73], and trace alignment [74], search for cut points between source and target programs such that some relation between the two programs can be automatically synthesized. When such relations are found, verification can be split into smaller tasks. Moreover, this technique supports some control-flow-changing transformations. TVOC [59,75] also has heuristics to support transformations that change loop structure.

LLVM-MD [76] and Peggy [77] are TV tools for LLVM that work by rewriting the source program until it is syntactically equal to the source. This process—equality saturation—is similar to how many first-order theorem provers work (e-matching). egg [78] is a library that implements equality saturation. These tools are limited to proving equivalence.

Coeus [79] implements verification of relational program properties with reinforcement learning. Klebanov et al. [80] proposed a CEGAR-based approach for the verification of program equivalence. Inter-procedural equivalence checking with mutual summaries was proposed in [81, 82]. Compositional Lifter [83] validates binary-to-LLVM IR lifting tools.

A different class of TV tools are ones that are specific to a particular transformation. For example, there are TV tools specific for lazy code motion [84], software pipelining [85,86], and optimizations for scientific programs [87]. The advantage of being specific is that these tools are simpler than generic ones. Moreover, some of these TV tools are formally verified, which is harder to do for generic tools. Sewell et al. [88] implemented TV for a specific program (seL4 kernel) for its compilation from C to ARM assembly.

While the majority of work so far on TV has focused on equivalence checking, there is one that introduced support for some forms of UB [89]. This work shows that adding support for UB (even if partially) reduces the number of false alarms substantially.

Chapter 3

Undefined Behavior in the IR

LLVM heavily relies on undefined behavior to utilize the assumptions made by the frontend languages. For example, C/C++ standard assumes that a reasonable program will never divide a number by zero, stating that such program has undefined behavior. A compiler can benefit from this and can analyze that a value is never zero if it is used by division. It makes removal of a dead division instruction valid as well because it is allowed for compiler to remove undefined behavior from the source program. This possibly makes a programmer surprised because it changes the equivalent assembly program from raising a trap to silently finishing the execution.

Undefined behavior in LLVM IR falls into two categories. First, “immediate UB” for serious errors, such as dividing by zero or dereferencing an invalid pointer, that have consequences such as a processor trap. Second, “deferred UB” for operations that produce unpredictable values but are otherwise safe to execute. For example, shifting by a number that is larger than the bitwidth yields a value that depends on the target architecture, but it does not immmediately crash the system. Deferred UB is necessary to support speculative execution, such as hoisting potentially undefined operations out of loops. Deferred UB in LLVM comes in two forms: an *undef* value that models a register with indeterminate value, and *poison*, a slightly more powerful form of UB that taints the dataflow graph and triggers immediate UB if it reaches a side-effecting operation such as division.

The presence of two kinds of deferred UB, and in particular the interaction between them, has often been considered to be unsatisfying, and has been a persistent source of discussions and bugs. LLVM has long contained optimizations that are inconsistent with the documented semantics and that are inconsistent with each other. To prevent miscompilation and permit rigorous reasoning about LLVM IR, we redefined the UB-related parts of LLVM's semantics in such a way that:

- Compiler developers can understand and work with the semantics.
- Long-standing optimization bugs can be fixed.
- Few optimizations currently in LLVM need to be removed.
- Compilation time and execution time of generated code are largely unaffected.

The new formal semantics of undefined behavior has been adopted by the LLVM (Section 3.7). This chapter describes and evaluates our efforts.

3.1 Undefined Behavior in the IR

Undefined-behavior-related compiler optimizations are often thought of as black magic, even by compiler developers. In this section we introduce IR-level undefined behavior and show examples where it enables useful optimizations.

3.1.1 Undefined Behavior $\neq$ Unsafe Programming

Despite the very poor example set by C and C++, there is no inherent connection between undefined behavior (UB) and unsafe programming. Rather, UB simply reflects a refusal to systematically trap program errors at one particular level of the system: the responsibility for avoiding these errors is delegated to a higher level of abstraction. For example, of course, many safe programming languages have been compiled to machine code, the unsafety of which in no way compromises the high-level guarantees made by the language implementation. Swift and Rust are compiled to LLVM IR; some of their safety guarantees are enforced by dynamic checks in the emitted code, other guarantees are made through type checking and have no representation at the LLVM level. Even C can be used safely if some tool in the development environment ensures—either statically or dynamically—that it will not execute UB.

The essence of undefined behavior is the freedom to avoid a forced coupling between error checks and unsafe operations. The checks, once decoupled, can be optimized, for example by being hoisted out of loops or eliminated outright. The remaining unsafe operations can be—in a well-designed IR—mapped onto basic processor operations with little or no overhead. As a concrete example, consider this Swift code:

```
func add(a : Int, b : Int)->Int {
  return (a & 0xffff) + (b & 0xffff)
}
```

Although a Swift implementation must trap on integer overflow, the compiler observes that overflow is impossible and emits this LLVM IR:

```
define i64 @add(i64 %a, i64 %b) {
  %0 = and i64 %a, 65535
  %1 = and i64 %b, 65535
  %2 = add nuw nsw i64 %0, %1
  ret i64 %2
}
```

Not only has the checked addition operation been lowered to an unchecked one, but in addition the `add` instruction has been marked with LLVM's `nsw` and `nuw` attributes, indicating that both signed and unsigned overflow are undefined. In isolation these attributes provide no benefit, but they may enable additional optimizations after this function is inlined. When the Swift benchmark suite[1] is compiled to LLVM, about one in eight addition instructions has an attribute indicating that integer overflow is undefined.

In this particular example the `nsw` and `nuw` attributes are redundant since an optimization pass could re-derive the fact that the `add` cannot overflow. However, in general these attributes and others like them add real value by avoiding the need for potentially expensive static analyses to rediscover known program facts. Also, some facts cannot be rediscovered later, even in principle, since information is lost at some compilation steps.

3.1.2 Enabling Speculative Execution

The C code in Fig. 3.1 executes undefined behavior if `x` is `INT_MAX` and `n` > 0, because in this case the signed addition `x + 1` overflows. A straightforward translation of the C code into LLVM IR, also shown in Fig. 3.1, has the same domain of definedness as the original code: the `nsw` modifier to the `add` instruction indicates that it is defined only when signed overflow does not occur.

We would like to optimize the loop by hoisting the invariant expression `x + 1`. If integer overflow triggered immediate undefined behavior, this transformation would be illegal because it makes the domain of definedness smaller: the code would execute UB when `x` was `INT_MAX`, even if `n` was zero. LLVM works around

[1] https://swift.org/blog/swift-benchmark-suite/

```c
for (int i = 0; i < n; ++i) {
  a[i] = x + 1;
}
```

```llvm
init:

  br %head

head:
  %i = phi [ 0, %init ], [ %i1, %body ]
  %c = icmp slt %i, %n
  br %c, %body, %exit

body:
  %x1 = add nsw %x, 1
  %ptr = getelementptr %a, %i
  store %x1, %ptr
  %i1 = add nsw %i, 1
  br %head
```

Figure 3.1: C code and its corresponding LLVM IR. We want to hoist the invariant addition out of the loop. The `nsw` attribute means the `add` is undefined for signed overflow.

this problem by adding the concept of deferred undefined behavior: the undefined addition is allowed, but the resulting value cannot be relied upon. It is easy to see that after hoisting the `add`, the code remains safe in the $\texttt{n} = 0$ case, because `x1` is not used. While deferred UB is useful, it is not appropriate in all situations. For example, division by zero can trigger a processor trap and an out-of-bounds store can corrupt RAM. These operations, and a few others in LLVM IR, are immediate undefined behaviors and programs must not execute them.

3.1.3 Undefined Value

We need a semantics for deferred undefined behavior. A reasonable choice is to specify that an undefined value represents any value of the given type. A

```
int x;
if (cond)
  x = f();

if (cond2)
  g(x);
```

(a)

```
entry:
  br %cond, %ctrue, %cont

ctrue:
  %xf = call @f()
  br %cont

cont:
  %x = phi [ %xf, %ctrue ],
           [ undef, %entry ]
  br %cond2, %c2true, %exit

c2true:
  call @g(%x)
```

(b)

```
  ; test cond
  testb   %dil, %dil
  je      ctrue
  ; return value goes in %eax
  callq   f

ctrue:
  ; test cond2
  testb   %bl, %bl
  je      exit
  ; whatever is in %eax
  ; gets passed to g()
  movl    %eax, %edi
  callq   g
```

(c)

Figure 3.2: If `cond2` implies `cond`, the C code in (a) does not perform UB by accessing `x` before it is assigned a value. (b) is Clang's translation into LLVM IR and (c) is the eventual x86-64.

number of compiler IRs support this abstraction; in LLVM it is called undef.[2][3].

Undef is useful because it lets the compiler avoid materializing an arbitrary constant in situations—such as the one shown in Fig. 3.2—where the exact value does not matter. In this example, assume that `cond2` implies `cond` in some non-trivial way such that the compiler cannot see it. Thus, there is no need to initialize variable `x` at its point of declaration since it is only passed to `g` after being assigned a value from `f`'s return value. If the IR lacked an undef value, the compiler would have to use an arbitrary constant, perhaps zero, to initialize `x` on the branch that skips the first `if` statement. This, however, increases the code size by one instruction and two bytes on x86 and x86-64. Little optimizations like this can add up across a large program. Undef is used to represent the values of padding in structures, arrays, and bit fields as well.

[2] `http://nondot.org/sabre/LLVMNotes/UndefinedValue.txt`

[3] Actually, undef in LLVM is defined as a set of values of the type. Section 5.2 describes its formal definition in detail.

3.1.4 Beyond Undef

In C and C++, we can assume that the expressions `a + b > a` and `b > 0` always yield the same value because signed overflow is undefined (assuming `a` and `b` are of a signed type like `int`). If the original expression is translated to this LLVM IR:

```
%add = add %a, %b
%cmp = icmp sgt %add, %a
```

the optimization to:

```
%cmp = icmp sgt %b, 0
```

becomes illegal since the `add` instruction wraps around on overflow. Moreover, this problem cannot be fixed by defining a version of `add` that returns `undef` when there is a signed integer overflow.

To see the inadequacy of undef, let `a` = `INT_MAX` and `b` = 1. The addition overflows and the expression simplifies to `undef` > `INT_MAX`, which is always false since there is no value of integer type that is larger than `INT_MAX`. However, the desired optimized expression, `b` > 0, simplifies to 1 > 0, which is true. Thus, the optimization is illegal: it would change the semantics of the program.

To justify this transformation, LLVM has a second kind of deferred undefined behavior, the *poison value*. The original expression is compiled to this code instead:

```
%add = add nsw %a, %b
%cmp = icmp sgt %add, %a
```

The `nsw` (no signed wrap) attribute on the `add` instruction indicates that it returns a poison value on signed overflow. Poison values, unlike undef, are not restricted to being a value of a given type. Most instructions including `icmp` return poison if any of their inputs is poison. Thus, poison is a stronger form of

```
for (int i = 0; i <= n; ++i) {
  a[i] = 42;
}
```

```
entry:
  br %head

head:
  %i = phi [ 0, %entry ], [ %i1, %body ]
  %c = icmp sle %i, %n
  br %c, %body, %exit

body:
  %iext = sext %i to i64
  %ptr = getelementptr %a, %iext
  store 42, %ptr
  %i1 = add nsw %i, 1
  br %head
```

Figure 3.3: C code and corresponding LLVM IR on x86-64. We want to eliminate the `sext` instruction in the loop body.

UB than undef. In the previous example with `nsw`, the result of the comparison becomes poison whenever the addition overflows and thus the optimization is justified.

Fig. 3.3 shows another example motivating the poison value. The `getelementptr` instruction (GEP for short) performs pointer arithmetic. The GEP there is computing $\texttt{a} + \texttt{i} * 4$, assuming that `a` is an array of 4-byte integers.

The sign-extend operation `sext` in the loop body handles the mismatch in bitwidth between the 32-bit induction variable and the 64-bit pointer size. It is the low-level equivalent of casting an `int` to `long` in C. Therefore, the GEP in the program is actually computing $\texttt{a} + \texttt{sext}(\texttt{i}) * 4$. We would like to optimize away the `sext` instruction since sign extension, unlike zero extension, is usually not free at runtime.

If we convert the loop induction variable `i` into `long` we can remove the sign extension within the loop body (at the expense of adding a sign extend of `n` to the `entry` basic block). This transformation improves performance by up to 39%, depending on the microarchitecture, since we save one instruction per iteration (`cltq` — sign extend `eax` into `rax`).

The transformation is only valid if pointer arithmetic overflow is undefined. If it is defined to wrap around, the transformation is not semantics-preserving, since a sequence of values of a signed 32-bit counter is different from a signed 64-bit counter's. Therefore, we would be changing the set of stored locations in case of overflow.

For a compiler to perform the aforementioned transformation, it needs to prove that either the induction variable does not overflow, or if it does it is a signed operation and therefore it does not matter. As we have seen before, signed integer overflow cannot be immediate UB since that would prevent hoisting math out of loops. If signed integer overflow returns undef, the resulting semantics are too weak to justify the desired optimization: on overflow we would obtain `sext(undef)` for `%iext`, which has all the most-significant bits equal to either zero or one. Therefore, the maximum value `%i1` could take would be `INT_MAX` and thus the comparison at `%c` would always be true if `%n` = `INT_MAX`. On the other hand, the comparison with 64-bit integers would return false instead.

If overflow is defined to return poison, an induction variable overflow would result in `%iext` = `sext(poison)`, which is equal to `poison`, which would make the comparison at `%c` equal to `poison` as well. Therefore, this semantics justifies induction variable widening.

3.2 Inconsistencies in LLVM

In this section we present several examples of problems with the current LLVM IR semantics.

3.2.1 Duplicate SSA Uses

In some CPU micro-architectures, addition is cheaper than multiplication. It may therefore be beneficial to rewrite $2 \times x$ as $x + x$. In LLVM IR we want to rewrite:

```
%y = mul %x, 2
```

as:

```
%y = add %x, %x
```

Algebraically, these two expressions are equivalent. However, consider the case where `%x` is undef. In the original code, the result can be any even number, while in the transformed code the result can be any number. Therefore, the transformation is wrong because we have increased the set of possible outcomes.

This problem happens because each use of undef in LLVM can yield a different result. Therefore, it is not correct in general to increase the number of uses of a given SSA register in an expression tree, unless it can be proved to not hold the undef value. Even so, LLVM incorrectly performs similar transformations.

There are, however, multiple advantages to defining `undef` as yielding a possibly different value on each use. For example, it helps reduce register pressure since we do not need to hold the value of an `undef` in a register to give the same value to all uses. Secondly, peephole optimizations can easily assume that an `undef` takes whatever value is convenient to do a particular transformation, which they could not easily do if `undef` had to remain consistent over multiple uses. Another advantage is to allow duplication of memory loads given that loads from uninitialized memory yield `undef`. If `undef` was defined to return a consistent value for all uses, a duplicated load could potentially return a different value if loading from uninitialized memory, which would be incorrect.

3.2.2 Global Value Numbering vs. Loop Unswitching

When `c2` is loop-invariant, LLVM's loop unswitching optimization transforms code of this form:

```
while (c) {
  if (c2) { foo }
  else    { bar }
}
```

to:

```
if (c2) {
  while (c) { foo }
} else {
  while (c) { bar }
}
```

This transformation assumes that branching on poison is not UB, but is rather a non-deterministic choice. Otherwise, if `c2` was poison, then loop unswitching would be introducing UB if `c` was always false (i.e., if the loop never executed).

The goal of global value numbering (GVN) is to find equivalent expressions and then pick a representative one and remove the remaining (redundant) computations. For example, in the following code, variables `t`, `w`, and `y` all hold the same value within the "then" block:

```
t = x + 1;
if (t == y) {
  w = x + 1;
  foo(w);
}
```

Therefore, GVN can pick `y` as the representative value and transform the code into:

```
t = x + 1;
if (t == y) {
  foo(y);
}
```

However, if `y` is a poison value and `w` is not, we have changed the code from using a regular value as function argument to passing a poison value to `foo`. If GVN followed loop unswitching's interpretation of branch-on-poison (non-deterministic branch), the transformation would be unsound. However, if we decide instead that branch-on-poison is UB, then GVN is fine, since the comparison "`t == y`" would be poison and therefore the original program would be already executing UB. This, however, contradicts the assumption made by loop unswitching. In other words, loop unswitching and GVN require different semantics for branch on poison in LLVM IR in order to be correct. By assuming different semantics, they perform conflicting optimizations, enabling end-to-end miscompilations.[4]

3.2.3 Select and Poison

LLVM's ternary `select` instruction, like the `?:` operator in C/C++, uses a Boolean to choose between its arguments. Either choice for how `select` deals with poison—producing poison if its not-selected argument is poison, or not—could be used as the basis for a correct optimizer. However, LLVM's optimization passes have not consistently implemented either choice. The LLVM Language Reference Manual[5] implies that if either argument to a `select` is poison, the output is poison.

The SimplifyCFG pass tries to convert control flow into select instructions:

```
  br %cond, %true, %false
true:
```

[4] `http://llvm.org/PR27506` and `http://llvm.org/PR31652`
[5] `http://llvm.org/docs/LangRef.html`

```
  br %merge
false:
  br %merge
merge:
  %x = phi [ %a, %true ], [ %b, %false ]
```

Gets transformed into:

```
  br %merge
merge:
  %x = select %cond, %a, %b
```

For this transformation to be correct, select on poison cannot be UB if branching on poison is not. Moreover, it can only be poison when the chosen value at runtime is poison (in order to match the behavior of phi).

LLVM also performs the reverse transformation, usually late in the pipeline and for target ISAs where it is preferable to branch rather than do a conditional move. For this transformation to be correct, branch on poison can only be UB if select on a poison condition is also UB. Since we want both transformations to be feasible, we can conclude that the behavior of branching on poison and select with a poison condition has to be equivalent.

If select on a poison condition is UB, it makes it very hard for the compiler to introduce select instructions in replacement of arithmetic. E.g., the following transformation that replaces an unsigned division with a comparison would be invalid (which ought to be valid for any constant `%C` < 0):

```
%r = udiv %a, %C
```

to:

```
%c = icmp ult %a, %C
%r = select %c, 0, 1
```

This transformation is desirable since it removes a potentially expensive operation like division. However, if select on poison is UB, the transformed program would execute UB if `%a` was poison, while the original program would not. As we have seen previously, if select (and therefore branch) on poison is not UB, GVN is unsound, but that is incompatible with the transformation above.

Finally, it is often desirable to view select as arithmetic, allowing transformations like: `%x = select %c, true, %b` to `%x = or %c, %b`. This property of equivalence with arithmetic, however, requires making the return value poison if any of the arguments is poison, which breaks soundness for the phi/branch to select transformation (SimplifyCFG in LLVM) above.

There is a tension between the different semantics that select can take and which optimizations can be made sound. Currently, different parts of LLVM implement different semantics for select, which originates end-to-end miscompilations. [6]

Finally, it is very easy to make mistakes when both undef and poison are involved. LLVM currently performs the following substitution:

```
%v = select %c, %x, undef
```

to:

```
%v = %x
```

This is wrong because `%x` could be poison, and poison is stronger than undef.[7]

3.2.4 Summary

In this section we showed that undefined behavior, which was added to LLVM's IR to justify certain desirable transformations, is exceptionally tricky and has lead to conflicting assumptions among compiler developers. These conflicts are reflected in the code base.[8] Although the LLVM developers almost always fix

[6] http://llvm.org/PR31632
[7] http://llvm.org/PR31633
[8] e.g., http://llvm.org/PR31181 and http://llvm.org/PR32176

overt problems that can be demonstrated to lead to end-to-end miscompilations, the latent problems we have shown here are long-standing and have so far resisted attempts to fix them (any fix that makes too many existing optimizations illegal is unacceptable). In the next section we introduce a modified semantics for UB in LLVM that we believe fixes all known problems and is otherwise acceptable.

3.3 Proposed Semantics

In Section 3.1 we showed that undef and poison enable useful optimizations that programmers might expect. In Section 3.2, however, we showed that undef and poison, as currently defined, are inconsistent with other desirable transformations (or combinations of transformations) and that they interact poorly with each other. Our proposal—arrived at after many iterations and much discussion, and currently under discussion with the broader LLVM community—is to tame undefined behavior in LLVM as follows:

- Remove undef and use poison instead.

- Introduce a new instruction:

  ```
  %y = freeze %x
  ```

  `freeze` is a nop unless its input is poison, in which case it non-deterministically chooses an arbitrary value of the type. All uses of a given freeze return the same value, but different freezes of a value may return different constants.

- All operations over poison unconditionally return poison except `phi`, `select`, and `freeze`.

- Branching on poison is immediate UB.

Our experience is that the presence of two kinds of deferred undefined behavior is simply too difficult for developers to reason about: one of them had to go. We define `phi` and `select` to conditionally return poison, and branching on poison to be UB, because these decisions reduce the number of freeze instructions that would otherwise be needed.

Defining branching on poison to be UB further enables analyses to assume that predicates used on branches hold within the target basic block, which would not be possible if we had defined branching on poison to be a non-deterministic choice. For example, for code like `if (x > 0) \{ /* foo */ \}`, we want to allow analyses to assume that `x` is positive within the "then" block (and not positive in the "else" block).

A risk of using freeze is that it disables subsequent optimizations that take advantage of poison. Our observation is that many of these optimizations were illegal anyway, and that it is better to disable them explicitly rather than implicitly. Also, as we show later, we usually do not need to introduce many freeze instructions. We experimentally show that freeze does not unduly impact performance.

3.3.1 Syntax

Fig. 3.4 gives the partial syntax of LLVM IR statements. LLVM IR is typed, but we omit operand types for brevity (in this section and throughout the paper) when these are implicit or non-essential. The IR includes standard unary/binary arithmetic instructions, load/store operations, a phi node, a comparison operator, multiple type casting instructions, conditional branching, instructions to access and modify vectors, etc. We also include the new **freeze** instruction and the new **poison** value, while removing the old **undef** value.

$$
\begin{array}{rcl}
\mathit{stmt} & ::= & \mathit{reg} = \mathit{inst} \;\mid\; \textbf{br}\; \mathit{op}, \mathit{label}, \mathit{label} \;\mid\; \textbf{store}\; \mathit{op}, \mathit{op} \\
\mathit{inst} & ::= & \mathit{binop}\; \overline{\mathit{attr}}\; \mathit{op}, \mathit{op} \;\mid\; \mathit{conv}\; \mathit{op} \;\mid\; \textbf{bitcast}\; \mathit{op} \;\mid \\
 & & \textbf{select}\; \mathit{op}, \mathit{op}, \mathit{op} \;\mid\; \textbf{icmp}\; \mathit{cond}, \mathit{op}, \mathit{op} \;\mid \\
 & & \textbf{phi}\; \mathit{ty}, [\mathit{op}, \mathit{label}] \ldots, [\mathit{op}, \mathit{label}] \;\mid\; \textbf{freeze}\; \mathit{op} \;\mid \\
 & & \textbf{getelementptr}\; \mathit{op}, \ldots, \mathit{op} \;\mid\; \textbf{load}\; \mathit{op} \;\mid \\
 & & \textbf{extractelement}\; \mathit{op}, \mathit{constant} \;\mid \\
 & & \textbf{insertelement}\; \mathit{op}, \mathit{op}, \mathit{constant} \\
\mathit{cond} & ::= & \textbf{eq} \;\mid\; \textbf{ne} \;\mid\; \textbf{ugt} \;\mid\; \textbf{uge} \;\mid\; \textbf{slt} \;\mid\; \textbf{sle} \\
\mathit{ty} & ::= & \textbf{i}\mathit{sz} \;\mid\; \mathit{ty}{*} \;\mid <\mathit{sz} \times \textbf{i}\mathit{sz}> \mid <\mathit{sz} \times \mathit{ty}{*}> \\
\mathit{binop} & ::= & \textbf{add} \;\mid\; \textbf{udiv} \;\mid\; \textbf{sdiv} \;\mid\; \textbf{shl} \;\mid\; \textbf{and} \;\mid\; \textbf{or} \\
\mathit{attr} & ::= & \textbf{nsw} \;\mid\; \textbf{nuw} \;\mid\; \textbf{exact} \\
\mathit{op} & ::= & \mathit{reg} \;\mid\; \mathit{constant} \;\mid\; \textbf{poison} \\
\mathit{conv} & ::= & \textbf{zext} \;\mid\; \textbf{sext} \;\mid\; \textbf{trunc}
\end{array}
$$

Figure 3.4: Partial syntax of LLVM IR statements. Types include arbitrary bitwidth integers, pointers $\mathit{ty}{*}$, and vectors $<\mathit{elems} \times \mathit{ty}>$ that have a statically-known number of elements elems.

3.3.2 Semantics

We first define the semantic domains as follows.

$$
\begin{array}{lcl}
\mathrm{Num}(\mathit{sz}) & ::= & \{\, i \mid 0 \le i < 2^{\mathit{sz}} \,\} \\
[\![\textbf{i}\mathit{sz}]\!] & ::= & \mathrm{Num}(\mathit{sz}) \uplus \{\, \textbf{poison} \,\} \\
[\![\mathit{ty}{*}]\!] & ::= & \mathrm{Num}(32) \uplus \{\, \textbf{poison} \,\} \\
[\![\langle \mathit{sz} \times \mathit{ty} \rangle]\!] & ::= & \{0, \ldots, \mathit{sz} - 1\} \to [\![\mathit{ty}]\!] \\
\mathrm{Mem} & ::= & \mathrm{Num}(32) \nrightarrow [\![\langle 8 \times \textbf{i}1 \rangle]\!] \\
\mathrm{Name} & ::= & \{\, \texttt{\%x}, \texttt{\%y}, \ldots \,\} \\
\mathrm{Reg} & ::= & \mathrm{Name} \to \{\, (\mathit{ty}, v) \mid v \in [\![\mathit{ty}]\!] \,\}
\end{array}
$$

Here $[\![\mathit{ty}]\!]$ denotes the set of values of type ty, which are either `poison` or fully defined for base types, and are element-wise defined for vector types. The memory Mem is bitwise defined since it has no associated type. Specifically, Mem partially maps a 32-bit address to a bitwise defined byte (we assume, with no loss of generality, that pointers are 32 bits). The register file Reg maps a name to a type and a value of that type.

We define two meta operations: conversion between values of types and low-level bit representation. These operations are used later for defining semantics of instructions.

$$\begin{array}{rcl} ty{\downarrow} & \in & [\![ty]\!] \to [\![\langle \text{bitwidth}(ty) \times \mathbf{i1} \rangle]\!] \\ ty{\uparrow} & \in & [\![\langle \text{bitwidth}(ty) \times \mathbf{i1} \rangle]\!] \to [\![ty]\!] \end{array}$$

$$\begin{array}{rcl} \mathbf{i}sz{\downarrow}(v) \text{ or } ty{*}{\downarrow}(v) & = & \begin{cases} \lambda\_.\,\mathbf{poison} & \text{if } v = \mathbf{poison} \\ (std) & \text{otherwise} \end{cases} \\ \langle sz \times ty \rangle{\downarrow}(v) & = & ty{\downarrow}(v[0]) +\!\!+ \ldots +\!\!+\, ty{\downarrow}(v[sz-1]) \end{array}$$

$$\begin{array}{rcl} \mathbf{i}sz{\uparrow}(b) \text{ or } ty{*}{\uparrow}(b) & = & \begin{cases} \mathbf{poison} & \text{if } \exists i.\, b[i] = \mathbf{poison} \\ (std) & \text{otherwise} \end{cases} \\ \langle sz \times ty \rangle{\uparrow}(b) & = & \langle ty{\uparrow}(b_0), \ldots, ty{\uparrow}(b_{sz-1}) \rangle \\ & & \text{where } b = b_0 +\!\!+ \ldots +\!\!+\, b_{sz-1} \end{array}$$

For base types, $ty{\downarrow}$ transforms poison into the bitvector of all poison bits, and defined values into their standard low-level representation. For vector types, $ty{\downarrow}$ transforms values element-wise, where $+\!\!+$ denotes the bitvector concatenation. Conversely, for base types, $ty{\uparrow}$ transforms bitwise representations with at least one poison bit into poison, and transforms fully defined ones in the standard way. For vector types, $ty{\uparrow}$ transforms bitwise representations element-wise.

Now we give semantics to selected instructions in Fig. 3.5. It shows how each instruction updates the register file $R \in \text{Reg}$ and the memory $M \in \text{Mem}$, denoted $R, M \hookrightarrow R', M'$. The value $[\![op]\!]_R$ of operand op over R is given by:

$$\begin{array}{rcll} [\![r]\!]_R & = & R(r) & //\ \text{register} \\ [\![C]\!]_R & = & C & //\ \text{constant} \\ [\![\mathbf{poison}]\!]_R & = & \mathbf{poison} & //\ \text{poison} \end{array}$$

The load operation $\text{Load}(M, p, sz)$ successfully returns the loaded bit representation only if p is a non-poison address pointing to a valid block of bitwidth at least sz in the memory M. The store operation $\text{Store}(M, p, b)$ successfully stores the bit representation b in the memory M and returns the updated memory only if p is a non-poison address pointing to a valid block of bitwidth at least $\text{bitwidth}(b)$.

$$\boxed{\begin{array}{c} (r = \textbf{freeze}\ \textbf{i}sz\ op) \\ \dfrac{[\![op]\!]_R = \textbf{poison} \quad v \in \mathrm{Num}(sz)}{R, M \hookrightarrow R[r \mapsto v], M} \\ \dfrac{[\![op]\!]_R = v \neq \textbf{poison}}{R, M \hookrightarrow R[r \mapsto v], M} \end{array}}$$

$$\boxed{\begin{array}{c} (r = \textbf{freeze}\ ty\ op)\ \text{for}\ ty = \langle n \times \textbf{i}sz \rangle \\ \dfrac{[\![op]\!]_R = \langle v_0, \ldots, v_{n-1} \rangle \quad \left[\begin{array}{c} \forall i.\ (v_i = \textbf{poison} \land v'_i \in \mathrm{Num}(sz)) \\ \lor\ (v_i = v'_i \neq \textbf{poison}) \end{array} \right]}{R, M \hookrightarrow R[r \mapsto \langle v'_0, \ldots, v'_{n-1} \rangle], M} \end{array}}$$

$$\boxed{\begin{array}{c} (r = \textbf{phi}\ ty\ [op_1, L_1],\ \ldots,\ [op_n, L_n]) \\ \dfrac{[\![op_i]\!]_R = v_i}{R, M \hookrightarrow R[r \mapsto v_i], M}\ (\text{coming from } L_i) \end{array}}$$

$$\boxed{\begin{array}{c} (r = \textbf{select}\ op,\ ty\ op_1,\ op_2) \\ \dfrac{[\![op]\!]_R = \textbf{poison}}{R, M \hookrightarrow R[r \mapsto \textbf{poison}], M} \quad \dfrac{[\![op]\!]_R = 1 \quad [\![op_1]\!]_R = v_1}{R, M \hookrightarrow R[r \mapsto v_1], M} \quad \dfrac{[\![op]\!]_R = 0 \quad [\![op_2]\!]_R = v_2}{R, M \hookrightarrow R[r \mapsto v_2], M} \end{array}}$$

$$\boxed{\begin{array}{c} (r = \textbf{and}\ \textbf{i}sz\ op_1,\ op_2) \\ \dfrac{[\![op_1]\!]_R = \textbf{poison}}{R, M \hookrightarrow R[r \mapsto \textbf{poison}], M} \quad \dfrac{[\![op_2]\!]_R = \textbf{poison}}{R, M \hookrightarrow R[r \mapsto \textbf{poison}], M} \\ \dfrac{[\![op_1]\!]_R = v_1 \neq \textbf{poison} \quad [\![op_2]\!]_R = v_2 \neq \textbf{poison}}{R, M \hookrightarrow R[r \mapsto v_1\ \&\ v_2], M} \end{array}}$$

$$\boxed{\begin{array}{c} (r = \textbf{add nsw}\ \textbf{i}sz\ op_1,\ op_2) \\ \dfrac{[\![op_1]\!]_R = \textbf{poison}}{R, M \hookrightarrow R[r \mapsto \textbf{poison}], M} \quad \dfrac{[\![op_2]\!]_R = \textbf{poison}}{R, M \hookrightarrow R[r \mapsto \textbf{poison}], M} \\ \dfrac{[\![op_1]\!]_R = v_1 \quad [\![op_2]\!]_R = v_2 \quad v_1 + v_2\ \text{overflows (signed)}}{R, M \hookrightarrow R[r \mapsto \textbf{poison}], M} \\ \dfrac{[\![op_1]\!]_R = v_1 \quad [\![op_2]\!]_R = v_2 \quad v_1 + v_2\ \text{no signed overflow}}{R, M \hookrightarrow R[r \mapsto v_1 + v_2], M} \end{array}}$$

$$\boxed{\begin{array}{c} (r = \textbf{bitcast}\ ty_1\ op\ \textbf{to}\ ty_2) \\ \dfrac{[\![op]\!]_R = v}{R, M \hookrightarrow R[r \mapsto ty_2{\uparrow}(ty_1{\downarrow}(v))], M} \end{array}}$$

$$\boxed{\begin{array}{c} (r = \textbf{load}\ ty, ty{*}\ op) \\ \dfrac{\mathrm{Load}(M, [\![op]\!]_R, \mathrm{bitwidth}(ty))\ \text{fails}}{R, M \hookrightarrow \mathrm{UB}} \\ \dfrac{\mathrm{Load}(M, [\![op]\!]_R, \mathrm{bitwidth}(ty)) = v}{R, M \hookrightarrow R[r \mapsto ty{\uparrow}(v)], M} \end{array}} \quad \boxed{\begin{array}{c} (\textbf{store}\ ty\ op_1, ty{*}\ op) \\ \dfrac{\mathrm{Store}(M, [\![op]\!]_R, ty{\downarrow}([\![op_1]\!]_R))\ \text{fails}}{R, M \hookrightarrow \mathrm{UB}} \\ \dfrac{\mathrm{Store}(M, [\![op]\!]_R, ty{\downarrow}([\![op_1]\!]_R)) = M'}{R, M \hookrightarrow R, M'} \end{array}}$$

Figure 3.5: Semantics of selected instructions

The rules shown in Fig. 3.5 follow the standard operational semantics notation. For example, the first rule says that the instruction $r =$ **freeze i**sz op, if the operand value $[\![op]\!]_R$ is poison, updates the destination register r with an arbitrary value v (i.e., updates the register file R to $R[r \mapsto v]$) leaving the memory M unchanged; and if $[\![op]\!]_R$ is a non-poison value v, it updates the register r with the operand value v.

3.4 Illustrating the New Semantics

In this section we show how the proposed semantics enable optimizations that cannot be performed soundly today in LLVM. We also show how to encode certain C/C++ idioms in LLVM IR for which changes are required in the frontend (Clang), as well as optimizations that need tweaks to remain sound.

3.4.1 Loop Unswitching

We showed previously that GVN and loop unswitching could not be used together. With the new semantics, GVN becomes sound, since we chose to trigger UB in case of branch on poison value. Loop unswitching, however, requires a simple change to become correct. When a branch is hoisted out of a loop, the condition needs to be frozen. E.g.,

```
while (c) {
  if (c2) { foo }
  else    { bar }
}
```

is transformed into:

```
if (freeze(c2)) {
  while (c) { foo }
} else {
  while (c) { bar }
}
```

By using the `freeze` instruction, we avoid introducing UB in case `c2` is poison and force a non-deterministic choice between the two loops instead. This is a refinement of the original code, which would trigger UB if `c2` was poison and the loop executed at least once.

Freeze can be avoided if the branch on `c2` is placed in the loop pre-header (since then the loop is guaranteed to execute at least once). The compiler further needs to prove that the branch on `c2` is always reachable (i.e., that all function calls before the "`if (c2)`" statement always return).

3.4.2 Reverse Predication

In some CPU architectures it is beneficial to compile a `select` instruction into a set of branches rather than a conditional move. We support this transformation using freeze:

```
%x = select %c, %a, %b
```

can be transformed to:

```
  %c2 = freeze %c
  br %c2, %true, %false
true:
  br %merge
false:
  br %merge
merge:
  %x = phi [ %a, %true ], [ %b, %false ]
```

Freeze ensures that no UB is triggered if `%c` is poison. We believe, however, that this kind of transformation may be delayed to lower-level IRs where poison usually does not exist.

3.4.3 Bit Fields

C and C++ have bit fields in structures. These fields are often packed together to form a single word-sized field (depending on the ABI). Since in our semantics loads of uninitialized data yield poison, and bit-field store operations also require a load (even the first store), extra care is needed to ensure that a store to a bit field does not always yield poison.

Therefore we propose to lower the following C code:

```
mystruct.myfield = foo;
```

into:

```
%val  = load %mystruct
%val2 = freeze %val
%val3 = ; ...combine %val2 and %foo...
store %val3, %mystruct
```

We need to freeze the loaded value, since it might be the first store to the bit field and therefore it might be uninitialized. If the stored value `foo` is poison, this bit field store operation contaminates the adjacent fields when it is combined through bit masking operations. This is fine, however, since if `foo` is poison then UB must have already occurred in the source program and so we can taint the remaining fields.

An alternative way of lowering bit fields is to use vectors or use the structure type. These are superior alternatives, since they allow perfect store-forwarding (no freezes), but currently they are both not well supported by LLVM's backend. E.g., with vectors:

```
%val  = load <32 x i1> %mystruct
%val2 = insertelement %foo, %val, ...
store %val2, %mystruct
```

Here we assume the word size is 32 bits, and therefore we ask LLVM to load a vector of 32 bits instead of loading a whole word. Since our semantics for vectors define that poison is determined per element, a poison bit field cannot contaminate adjacent fields.

3.4.4 Load Combining and Widening

Sometimes it is profitable to combine or widen loads to align with the word size of a given CPU. However, if the compiler chooses to widen, say, a 16-bit load into a 32-bit load, then care must be taken because the remaining 16 bits may be poison or uninitialized and they should not poison the value the program was originally loading. To solve the problem, we also resort to vector loads, e.g.,

```
%a = load i16, %ptr
```

can be transformed to:

```
%tmp = load <2 x i16>, %ptr
%a = extractelement %tmp, 0
```

As for bit fields, vector loads make it explicit to the compiler that we are loading unrelated values, even though at assembly level it is the still the same load of 32 bits.

3.4.5 Pitfall 1: Freeze Duplication

Duplicating freeze instructions is not allowed, since each freeze instruction may return a different value if the input is poison. For example, this blocks loop sinking optimization (dual of loop invariant code motion). Loop sinking is beneficial if, e.g., a loop is rarely executed. For example, it is not sound to perform the following transformation:

```
x = a / b;
y = freeze(x);
while (...) {
  use(y)
}
```

to:

```
while (...) {
  x = a / b;
  y = freeze(x);
  use(y)
}
```

3.4.6 Pitfall 2: Semantics of Static Analyses

Static analyses in LLVM usually return a value that holds only if all of the analyzed values are not poison. For example, if we run the `isKnownToBeAPowerOfTwo` analysis on value "`%x = shl 1, %y`", we get a statement that `%x` will be always a power of two. However, if `%y` is poison, then `%x` will also be poison, and therefore it could take any value, including a non-power-of-two value.

Many LLVM analyses are not sound over-approximations with respect to poison. The main reason is that if poison was taken into account then most analyses would return the worst result (top) most of the time, rendering them useless.

This semantics is generally fine when the result of the analyses are used for expression rewriting, since the original and transformed expressions will yield poison when any of the inputs is poison. However, this is not true when dealing with code movement past control-flow. For example, we would like to hoist the division out of the following loop (assuming `a` is loop invariant):

```
while (c) {
  b = 1 / a;
}
```

If the `isKnownToBeAPowerOfTwo` analysis states that `a` is always a power of two, we are tempted to conclude that hoisting the division is safe since `a` cannot possibly be zero. However, `a` may be poison, and therefore hoisting the division would introduce UB if the loop did not execute.

In summary, there is a trade-off for the semantics of static analysis regarding how they treat poison. LLVM is considering extending APIs of relevant analyses to return up-to results with respect to poison, i.e., the result of an analysis is sound if a set of values is non-poison. Then it is up to the client of the analysis to ensure this is the case if it wants to use the result of the analysis in a way that requires the value to be non-poison (e.g., to hoist instructions that may trigger UB past control-flow).

3.5 Prototype Implementation

We prototyped our new semantics in LLVM 4.0 RC4.[9] We made the following modifications to LLVM, changing a total of 578 lines of code:

- Added a new freeze instruction to the IR and to SelectionDAG (SDAG), and added appropriate translation from IR's freeze into SDAG's freeze and then to MachineInstruction (MI).
- Fixed loop unswitching to freeze the hoisted condition (as described in Section 3.4.1).
- Fixed several unsound InstCombine (peephole) transformations handling select instructions (e.g., the problems outlined in Section 3.2.3).
- Added simple transformations to InstCombine to optimize spurious uses of freeze, such as transforming `freeze(freeze(x))` to `freeze(x)` and `freeze(const)` to `const`.

We made a single change to Clang, modifying just one line of code: we changed the lowering of bit field stores to freeze the loaded value (as described in Section 3.4.3).

Lowering Freeze LLVM IR goes through two other intermediate languages before assembly is finally generated. Firstly, LLVM IR is lowered into Selection

[9]Code available from `https://github.com/snu-sf/{llvm-freeze,clang-freeze}/tree/pldi`

DAG (SDAG) form, which still represents code in a graph like LLVM IR but where operations may already be target dependent. Secondly, SDAG is lowered into MachineInstruction (MI) through standard instruction selection algorithms, followed by register allocation.

We introduced a freeze operation in SDAG, so a freeze in LLVM IR maps directly into a freeze in SDAG. Additionally, we had to teach type legalization (SDAG level) to handle freeze instructions with operands of illegal type (for the given target ISA). For instruction selection (i.e., when going from SDAG to MI), we convert poison values into pinned undef registers, and freeze operations into register copies. At MI level there is no poison, but instead there are undef registers, which may yield a different value for each use like LLVM IR's undef value. Since taking a copy from an undef register effectively freezes undefinedness (i.e., all uses of the copy observe the same value), we can lower freeze into a register copy.

Optimizations We had to implement a few optimizations to recover some performance regressions we observed in early prototypes. These regressions were due to LLVM optimizers not recognizing the new freeze instruction and conservatively giving up. For example, on x86 it is usually preferable to lower a branch on an and/or operation into a pair of jumps rather than do the and/or operation and then do a single jump. This transformation got blocked if the branch was done on a frozen and/or operation. We modified CodeGenPrepare (a phase right before lowering IR to SDAG) to support freeze.

For x86, a comparison used only by a conditional branch is usually moved so that it is placed right before the branch, since it is often preferable to repeat the comparison (if needed) than save the result to reuse later. Since freeze instructions cannot be sunk into loops, this transformation is blocked if the branch is over a frozen comparison. We changed CodeGenPrepare to transform "`freeze(icmp %x, const)`" to "`icmp(freeze %x), const`" when deemed profitable. Note that we cannot do this transformation early in the pipeline since it would break some static analyses (like scalar evolution)—the transformed expression is a refinement of the original one.

We changed the inliner to recognize freeze instructions as zero cost, even if they may not always be free. With this change, we avoid changing the behavior of the inliner as much as possible.

Testing the Prototype To test the correctness of the prototype, we used the LLVM and Clang test suites. We also used opt-fuzz[10] to exhaustively generate all LLVM functions with three instructions (over 2-bit integer arithmetic) and then we used Alive [60] to validate both individual passes (InstCombine, GVN, Reassociation, and SCCP) and the collection of passes implied by the `-O2` compiler flag. This way we increase confidence that Alive and LLVM agree on the semantics of the IR. This technique was also very useful during the development of the semantics since it enabled us to quickly try out different solutions and check which optimizations would be invalid.

Limitations of the Prototype Our prototype has a few limitations that make it unsound in theory, even though we did not detect any end-to-end miscompilations. These limitations do not reflect fundamental problems with our proposed semantics, but they require more extensive changes to LLVM than we have performed so far. Also, bear in mind that LLVM was already unsound before our changes, but in ways that are harder to fix.

InstCombine performs a few transformations taking a select instruction

[10] `https://github.com/regehr/opt-fuzz`

and producing arithmetic operations. For example, "`select %c, true, %x`" is transformed into "`or %c, %x`". This transformation is incorrect if `%c` may be poison. A safe version requires freezing `%c` for the `or` operation. Alternatively, we could just remove these transformations, but that would likely require improvements to other parts of the compiler to make them recognize the idiom to produce efficient code (since at the moment the backend and other optimizations may not be expecting this non-canonical code).

Another limitation is related to vectors. We have shown that widening can be done safely by using vector operations. However, LLVM does not yet handle vectors as first-class values, which frequently results in generation of sub-optimal code when vectors are used. Therefore, we did not fix any widening done by LLVM (e.g., in GVN, in Clang's lowering of bit-fields, or in Clang's lowering of certain parameters that require widening by some ABIs).

3.6 Performance Evaluation

This section evaluates the performance of our prototype in terms of compile time and size and speed of generated code.

3.6.1 Experimental Setup

Environment We used two machines with different micro-architectures for evaluation. Machine 1 had an Intel Core i7 870 CPU at 2.93 GHz, and Machine 2 had an Intel Core i5 6600 CPU at 3.30 GHz. Both machines had 8 GB of RAM and were running Ubuntu 16.04. To get consistent results, we disabled Hyper-Threading, SpeedStep, Turbo Boost, and address space layout randomization (ASLR). We used the `cpuset` tool[11] to grant exclusive hardware resources to the benchmark process. Machines were disconnected from the network while

[11] `https://github.com/lpechacek/cpuset.git`

running the benchmarks.

Benchmarks We used three benchmarks: SPEC CPU 2006, LLVM Nightly Test (LNT), and five large single-file programs ranging from 7k to 754k lines of code each.[12] SPEC CPU consists of 12 integer-only (CINT) and seven floating-point (CFP) benchmarks (we only consider C/C++ benchmarks). LNT consists of 281 benchmarks with about 1.5 million lines of code in total.

Measurements We measured running time and peak memory consumption of the compiler, running time of compiled programs, and generated object file size.

To estimate compilation and running time, we ran each benchmark three times (except LNT, which we ran five times to cope with shorter running times) and took the median value. To estimate peak memory consumption, we used the `ps` tool and recorded the `rss` and `vsz` columns every 0.02 seconds. To measure object file size, we recorded the size of `.o` files and the number of IR instructions in LLVM bitcode files. All programs were compiled with `-O3` and the comparison was done between our prototype and the version of LLVM/Clang from which we forked.

3.6.2 Results

Compile time On both machines, compile time was largely unaffected by our changes. Most benchmarks were in the range of ±1%. There were a few exceptions with small files, such as the "Shootout nestedloop" benchmark, where compilation time increased by 19% to 29 ms. The reason was that an optimization (jump threading) did not kick in because of not knowing about freeze, which

[12] `http://people.csail.mit.edu/smcc/projects/single-file-programs/` and `https://sqlite.org/2016/sqlite-amalgamation-3140100.zip`

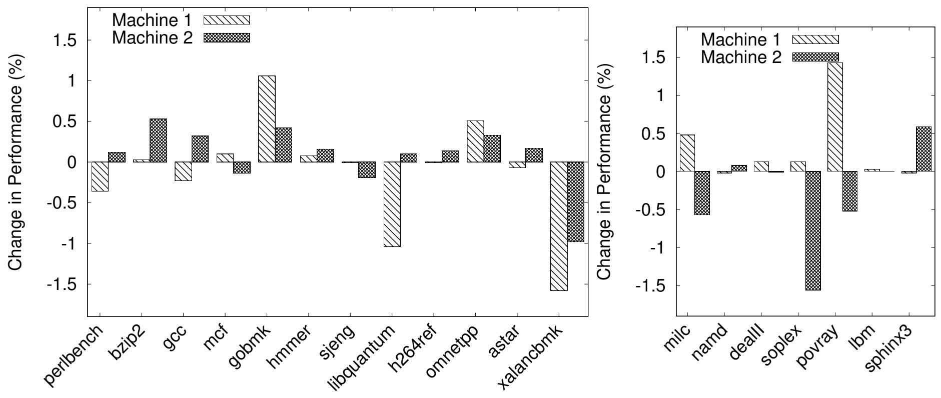


Figure 3.6: Change in performance in % for SPEC CPU 2006: CINT on the left, CFP on the right. Positive values indicate that performance improved, and negative values indicate that performance degraded.

then caused a different set of optimizations to fire in the rest of the pipeline.

Memory consumption For most benchmarks, peak memory consumption was unchanged, and we observed a maximum increase of 2% for bzip2, gzip, and oggenc.

Object code size We observed changes in the range of ±0.5%. Freeze instructions represented about 0.04%–0.06% of the total number of IR instructions. The gcc benchmark, however, had 3,993 freeze instructions (0.29% of total), since it contains a large number of bit-field operations.

Run time Change in performance for SPEC CPU 2006 is shown in Fig. 3.6. The results are in the range of ±1.6%, with slightly different results on the two machines.

For LNT benchmarks, only 26% had different IR after optimization, and only 82% of those produced different assembly (21% overall resulted in a different binary). Excluding noisy tests, we observed a range of -3% to 2% performance change on machine 1 and -3% to 1.5% on machine 2, except for one case: "Stanford Queens." This test showed a significant speedup (6% on machine 1 and 8% on machine 2) because the introduction of a single freeze instruction caused a change in allocated registers (r13 vs r14). According to the Intel Optimization Reference Manual, the latency and throughput of the LEA instruction is worse with certain registers.[13]

It is normal that run time results fluctuate a bit when a new instruction is added to an IR, since some optimizations and heuristics need to learn how to handle the new instruction. We did only a fraction of the required work, but the results are already reasonable, which shows that the semantics can be deployed incrementally.

3.7 Implementing Our Semantics in LLVM

The new formal semantics has been adopted by LLVM, and many optimizations are fixed because they were incorrect with respect to the semantics. In this section, we describe the efforts we made to carry our formal semantics to LLVM. We actively sent patches to the LLVM project in order to fix incorrect transformations and address performance regressions after their land. Several incorrect transformations are fixed by LLVM developers as well.

3.7.1 Adding Freeze Instruction and Poison Constant

The proposed `freeze` instruction is officially added into LLVM 10.0 in November, 2019. We wrote patches for adding freeze to LLVM IR and SelectionDAG[14], both of which were successfully landed. Instruction selection was updated to handle `freeze` with an operand whose type is illegal in the target architecture. A `freeze`

[13] `https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf`, §3.5.1.3 Using LEA

[14] `https://reviews.llvm.org/rG58acbce3def63a207b8f5a69318a99666a4aac53`, `https://reviews.llvm.org/rG7802be4a3d86743242273593d43a78df84ece8c1`

node in SelectionDAG is lowered into a register copy in MachineIR. However, this translation turned out to be unsound because MachineIR's `IMPLICIT_DEF` has undef-like behavior. To fully fix this, further updates in MachineIR's transformations are necessary.

We also implemented `isGuaranteedNotToBeUndefOrPoison` analysis in ValueTracking to make removal of redundant `freeze` instructions easy. It is a flow-sensitive analysis returning true if the value must be well defined. A similar function is added to SelectionDAG framework as well.

Also, `poison` constant is officially added to LLVM in December, 2020[15]. In the past, there was no way to present `poison` constant in LLVM IR. After this patch, constant folding and instruction combining are updated to deal with `poison`.

Several transformations and APIs are updated to use `poison` instead of `undef` to represent a don't-care value. For example, `IRBuilder::CreateShuffleVector` is updated to use `poison` as the second vector operand if the shuffle operation does not choose elements from the second vector. Fully replacing `undef` with `poison` is ongoing work.

3.7.2 Disabling Select to And/Or Folding

As described in Section 3.5, transforming "`select %c, true, %x`" into "`or %c, %x`" is incorrect because "`or %c, %x`" is more poisonous than "`select %c, true, %x`" if `%c` is true and `%x` is `poison`. Similarly, transforming "`select %c, %x, true`" into "`and %c, %x`" is also unsound. There are two solutions for this. The first one is to remove this transformation, and the second one is to freeze `%c`.

The first solution, simply removing the `and`/`or` transformation, required

[15] `https://reviews.llvm.org/D71126` - this patch is written by a coauthor of our memory model and Alive2 paper.

significant updates in the codebase to avoid performance regression. Transformations and analyses were recognizing "`or %c, %x`" but not "`select %c, true, %x`" as a disjunction (similarly, "`and %c, %x`" but not "`select %c, %x, true`" as well). Also, the backend's code generation algorithm could emit optimized assembly code for "`and`/`or`" but not for `select`. However, this solution had one important strength: it does not remove undefinedness from source programs.

Compared to the first solution, the second solution (freezing `%c`) did not require updates in recognizing conjunction/disjunction patterns. However, insertion of `freeze` still caused suboptimal assembly because optimizations often had to look through the frozen condition. Besides that, inserted freeze instructions blocked further optimization because they permanently removed undefinedness from the program.

After discussions with LLVM developers, we chose to pursue the first solution. We updated transformations/analysis to recognize the select form of `and`/`or`. The backend was updated to generate optimized code as well.

Some optimizations were valid only when the input expression was in `and`/`or` form, but not in the select form. To fix them, we made two kinds of efforts. First, we again folded `select` to `and`/`or` only when it is correct to do so. The transformation is correct if `%x` being poison implies `%c` is also poison. We implemented a new `impliesPoison(x, y)` function that returns true if `x` being poison means `y` is also poison. Then, "`select %c, true, %x`" is optimized to "`or %c, %x`" if `impliesPoison(%x, %c)` holds (similarly for `and`). Second, in rare cases, `select` was transformed into to `and`/`or` with freeze.

We had to fix incorrect transformations that directly insert `and`/`or` as well. For example, SimplifyCFG pass was inserting `and`/`or` when short-circuiting two branch conditions ("`if (e1) { if (e2) { .. }}`" in C), which is poison-unsafe as described before. We fixed them to insert `select` instead.

In summary, we wrote about 20 patches and pushed a few without-review commits for minor improvements. The transformation was fully removed in May 2021.

3.7.3 Branch on Undef or Poison Is UB

We clarified in LLVM Language Reference Manual [6] that branching on `undef` or `poison` has undefined behavior[16]. For `switch` instruction, it has undefined behavior if the condition is either fully or partially undefined.

After the clarification, we fixed incorrect transformations that introduce undefined behavior according to our semantics. The transformation in CodeGenPrepare that converts `select` to a conditional branch is fixed to insert `freeze` instead. Also, JumpThreading is fixed to introduce `freeze` only when linking-time optimization is enabled. Since JumpThreading runs multiple times during the optimization pipeline, unconditionally inserting `freeze` could cause an observable performance regression.

Another incorrect transformation SimplifyCFG is fixed by a student who participated as a mentee in our Google Summer of Code project[17].

```
  %A = icmp ne i32 %mode, 0
  %B = icmp ne i32 %mode, 51
  %C = select i1 %A, i1 %B, i1 false
  %D = select i1 %C, i1 %Cond, i1 false
  br i1 %D, label %T, label %F
=>
  br i1 %Cond, label %switch.early.test, label %F
switch.early.test:
  switch i32 %mode, label %T [
          i32 51, label %F
          i32 0, label %F
  ]
```

[16] https://reviews.llvm.org/D76973

[17] In the summer of 2021, I and Nuno worked as mentors of 3 Google Summer of Code projects for fixing miscompilations in LLVM.

This transformation is incorrect if `%Cond` is `poison` and `%C` is false. We fixed this by freezing `%Cond`[18]. Previously, this transformation was inactivated if memory sanitizer option is turned on because it caused false positives. We fully activated the transformation in our patch.

To address performance regression after insertion of `freeze`, we added new optimizations to SelectionDAG. First of all, the semantics of branching on `undef` or `poison` in SelectionDAG is defined as a nondeterministic jump unlike IR[19]. This is okay because optimizations in SelectionDAG do not exploit branch conditions. After the clarification, we sent patches that remove `freeze` away if the instruction has only one use which is a conditional branch. This helped the backend do better code generation.

Fixing loop unswitch using `freeze` is ongoing work. The first trial to fix it was made in early 2020, but it was reverted due to performance regression. There are several patches under review to address the performance regression (Aug. 2021).

3.7.4 Dealing With Freeze in Loops

LLVM's loop optimizations heavily rely on loop analyses. Among them, scalar evolution is an analysis that finds a recurrence relation for an induction variable as well as its bounds. Scalar evolution helps simplify expressions containing induction variables as well as modeling the cost of the loop.

For precise modeling of an induction variable, scalar evolution assumes that a loop does not continue if it has a branch on `poison`.

```
loop:
  %i = phi i32 [0, %entry], [%i.next, %loop]
  %i.next = add nsw i32 %i, 1
```

[18] https://reviews.llvm.org/D104569
[19] https://reviews.llvm.org/D92015

```
  %cond = %i.next <= %n
  br i1 %cond, %loop, %exit
```

Given the above program, scalar evolution concludes that `%i` increases from zero to `%n` $- 1$ by one. A tricky case is when `%n` is 2^{32}. In this case, the last iteration executes a conditional branch on `poison`. To justify this case, we can use our branch-on-poison semantics. Since the last iteration executes a branch on `poison`, it has undefined behavior. As other value analyses do[20], scalar evolution's result holds for well-defined executions only. Since the last iteration is undefined, we only need to consider the remaining iterations.

This implies that the validity of scalar evolution's result relies on the undefinedness of the latch condition (`%cond`). If it is frozen (e.g., "`freeze %cond`"), or the added result is frozen (e.g., "`freeze %i.next`"), there is no undefined behavior anymore. This means the analysis result must be $\top$. This resulted in disabling many loop optimizations.

To fix this regression, we implemented a pass that pushes `freeze` out of a loop[21]. If the induction variable `%i.next` was frozen and used by `%cond` previously, the new pass removes `freeze` by stripping off poison-generating flags and freezing the initial value.

```
  ; Initial value 0 is already a well-defined constant, so
  ; it isn't necessary to be frozen
loop:
  %i = phi i32 [0, %entry], [%i.next, %loop]
  %i.next = add i32 %i, 1
  %cond = %i.next <= %n
  br i1 %cond, %loop, %exit
```

This pass is activated by default if -O1 or higher optimization flag is set.

[20]LLVM's value analysis on assumptions (`llvm.assume`) relies on undefined behavior as well, for example.

[21]https://reviews.llvm.org/D77523

3.7.5 Annotating NoUndef Attribute

LLVM has `noundef` function attribute stating that the annotated function argument or returned value must be well-defined. For example, passing "`undef | 1`" to a `noundef` argument has undefined behavior. To help never-undef-or-poison analysis, we wrote patches that annotate certain library functions's arguments and return value as `noundef`[22].

We can go one step further, and all user-defined functions in C/C++ can be marked as `noundef`. LLVM uses `undef` and `poison` to represent the result of an operation that (1) has undefined behavior, (2) yields a trap representation, or (3) yields an unspecified value in C/C++. The third case is unsound because the result is too undefined[23]. Assuming that translation of an unspecified value is fixed to use `freeze`(`poison`) instead, functions' arguments and return values can be annotated with `noundef`. Memory sanitizer experts wrote an initial version of a patch for this. In July 2021, a mentee of our Google Summer of Code project took the patch and he is completing the patch.

3.7.6 Optimizing Expressions Including Freeze

A majority of optimizations on arithmetic expressions pass are still sound if one or more of its input is `freeze`. For example, "`sub (freeze x), x`"[24] can be folded to 0.

There are two ways to show the correctness of such optimizations. The

[22] https://reviews.llvm.org/D97045
https://reviews.llvm.org/D87984
https://reviews.llvm.org/D85894
https://reviews.llvm.org/D85345

[23] C17 §3.19.3 says that an unspecified value is a valid value of the relevant type and using an unspecified value has unspecified behavior. According to this definition, using `undef` or `poison` is unsound because they raise undefined behavior if used by a few operations including `br`. To fix this, "`freeze poison`" must be used instead.

[24] For brevity, `%` is omitted from the variable names in this subsection.

first way is by applying sound unit transformations[25]. First, we can replace an operand of instruction with a frozen one because it makes the program more defined. We will name this unit transformation `freeze-replace`. Second, it is sound to merge two identical `freeze` instructions into one. We will name it `freeze-merge`.

```
      x' = freeze x
      r  = sub x', x

==>   x' = freeze x
      x2 = freeze x
      r  = sub x', x2 ; by freeze-replace

==>   x' = freeze x
      r  = sub x', x' ; by freeze-merge

==>   x' = freeze x
      r  = 0          ; by 'sub x, x => 0'
```

The second way is by proving the refinement relation between the resulting values. We are going to show that 0 refines the result of "`sub (freeze x), x`" for any x.

1. If x is a well-defined value, `freeze`(x) is equivalent to x. Since both of the source and target value are 0, refinement trivially holds.

2. If x is `poison`, "`sub (freeze x), x`" is `poison`, therefore refinement holds.

3. If x is a (partially) undefined value S, `freeze`(x) yields an element $s \in S$. "`sub (freeze x), x`" returns a set of value $\{s - s' \mid s' \in S\}$. The set includes 0, therefore refinement holds.

Although this optimization is valid, this is not added into the LLVM mainstream because it did not help any benchmark.

[25]Note that this is slightly different from equational reasoning because compiler transformations are not reversible.

Pushing Freeze Forwardly and Backwardly. Consider a transformation that pushes a freeze instruction backwardly:

```
                                        x.fr = freeze x
r   = op x, y                    ⇒      y.fr = freeze y
out = freeze r                          out  = op x.fr, y.fr
```

This transformation is correct if `op` never returns undef or poison when well-defined inputs are given. We can prove its soundness using `freeze-replace`.

```
      r = op x, y
      out = freeze r

⟹    x.fr = freeze x
      y.fr = freeze y
      r = op x.fr, y.fr ; by freeze-replace
      out = freeze r

⟹    x.fr = freeze x
      y.fr = freeze y
      r = op x.fr, y.fr ; by the definition of op
```

This transformation is added into InstCombine[26]. ValueTracking's `canCreateUndefOrPoison` is used to check the validity of the second step.

On the other hand, pushing freeze forwardly is hard to justify. Consider this transformation:

```
x.fr = freeze x
y.fr = freeze y                           ⇒    r   = icmp slt x, y
; slt is 'signed less-than'                    out = freeze r
out  = icmp slt x.fr, y.fr
```

If `y` is `INT_MAX`, `out` in the source program is true regardless of `x`. However, in the target program, `out` can be false if `x` was `poison`. Therefore, this transformation can be done only when `y` is known to be smaller than `INT_MAX`[27].

[26] https://reviews.llvm.org/D105392
[27] https://reviews.llvm.org/D105344

3.7.7 Folding Select Undef

As described in Section 3.2.3, folding "`select %c, undef, %x`" to `%x` is incorrect because the latter is more poisonous. This transformation is fixed so that it is folded only if `%x` is guaranteed not to be `poison`. Also, the backend is updated to support better assembly generation for such select pattern.

3.8 Undefined Behavior in Other Compilers

Most compilers have a concept like LLVM's undef, since it is simple, innocent-looking, and has tangible benefits. There are two common semantics for undef: one where each use of undef may get a different value, as in LLVM and Microsoft Phoenix; and another where all uses of undef get the same value, as in Firm [90], the Microsoft Visual C++ compiler (MSVC), and the Intel C/C++ Compiler (ICC).

GCC attempts to initialize uninitialized variables to zero, or give them a consistent value otherwise. However, this does not appear to be part of GCC's semantics because optimizations like SCCP can assume multiple values for the same uninitialized variable.[28]

Firm additionally has the concept of a "Bad" value,[29] the use of which triggers UB. This semantics is stronger than LLVM's poison (where the use of poison is not necessarily UB; arithmetic operations taking poison as input often just yield poison).

Signed overflow UB is exploited by ICC,[30] MSVC,[31] and GCC.[32] As far as we know, these compilers do not have their semantics formalized, but they

[28] `https://godbolt.org/g/r4PX4A`
[29] `http://pp.ipd.kit.edu/firm/Unknown_and_Undefined`
[30] `https://godbolt.org/g/egCqqm`
[31] `https://godbolt.org/g/ojfRVd`
[32] `https://godbolt.org/g/gtEbXx`

appear to use concepts similar to LLVM's poison. At least MSVC seems to suffer from similar problems as the ones we have outlined in this work for LLVM. It is likely that MSVC could fix their IR in a way similar to our solution. Similarly, Firm's developers acknowledge several bugs with their handling of "Bad" values; it is not clear whether it is a fundamental problem with the semantics of their IR or if these are implementation bugs.

CompCert [27] IR also has a deferred UB value called *undef*, which is essentially the same as poison in LLVM. Since branching on undef triggers UB in CompCert, certain optimizations like loop unswitching are unsound and thus not performed by CompCert. Mullen et al. [91] describe how the undef value gets in the way of peephole optimizations in CompCert.

In summary, most modern compiler IRs support reasoning based on undefined behavior, but this reasoning has received little up-front design work or formal attention.

3.9 Conclusion

Undefined behavior in a compiler IR, which is not necessarily related to undefined behavior in any given source language, gives optimizers the freedom to perform desirable transformations. We have presented the first detailed look at IR-level undefined behavior that we are aware of, and we have described difficult, long-standing problems with the semantics of undefined behavior in LLVM IR. These problems are present to some extent in other modern optimizing compilers. We developed and prototyped a modified semantics for undefined behavior that meets our goals of justifying most of the optimizations that LLVM currently performs, putting the semantics of LLVM IR on firm ground, and not significantly impacting either compile time or quality of generated code.

Chapter 4

A Memory Model for the IR

Precisely defining the memory model of IR is crucial because many important optimizations rely on it. Both GCC and LLVM have memory models that are informally specified, and they share an inconsistency in their semantics, leading to end-to-end miscompilation (Appendix A.1) This not only affects C and C++ but also type-safe languages like Rust (Appendix A.2)

The culprit is a new bug we found in LLVM's global value numbering (GVN) optimization. GVN propagates equalities of pointers (as well as of integers) from branch conditions, replacing pointers with value-equal ones. This, however, can change the behavior of a program, since pointers that compare equal are not necessarily equivalent. In the Rust example, GVN incorrectly propagates the equality in the condition of the last `if` statement (i.e., it replaces `q` with `p`), which then results in the program producing an incorrect result. The miscompilation of the C example can be traced back to a second bug we found where LLVM incorrectly assumes that `(int*)(intptr_t)p` is equal to `p`.

Fixing these miscompilations within the current IR semantics would be possi-

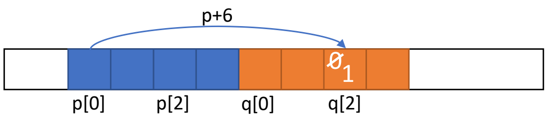


Figure 4.1: In a flat memory model, storing a `1` into `p[6]` can overwrite the `0` in `q[2]`

ble, but would necessitate disabling useful optimizations. The main contribution, which builds on insights developed by [5], is a new, formalized IR memory model for LLVM that departs from the current design in two ways. First, it uses *deferred bounds checking* to relax restrictions on the creation of out-of-bounds pointers in such a way that useful code motion optimizations can be performed soundly. Second, it uses *twin allocation*, which formalizes the idea that the value of a pointer has to be observed directly, it cannot be guessed. Twin allocation supports aggressive optimization of LLVM-based languages in the presence of low-level code such as integer-to-pointer casts. We have adapted LLVM to the new semantics in order to show that it does not require major changes to the compiler and it also does not degrade the performance of generated code.

4.1 Background

In this section we describe the design space for low-level sequential memory models and explain that existing designs are inadequate for managing the tension between low-level memory access and high-level optimizations. Then, in Section 4.2, we describe the basis for the new memory model for LLVM IR that we will formalize in Section 4.3. Our examples are written in a C-like syntax to make them easier to read, even though the scope of the paper is to specify a memory model for compiler IRs rather than for C.

4.1.1 Flat Memory Models

The two main questions a memory model needs to answer are (1) what is the return value of a load instruction, and (2) under what conditions is a memory-accessing instruction well-defined. A consequence is that the memory model should define which memory locations a store instruction writes to.

For example, what does the code below print? Or, alternatively, can the assignment `p[6] = 0` change any byte of the object pointed to by `q`?

```
char *p = malloc(4);
char *q = malloc(4);
q[2] = 0;
p[6] = 1;
print(q[2]); // prints 0 or 1?
```

In a *flat* memory model, the program would print `1` if `q == p + 4`, and `0` otherwise. A flat memory model treats pointers like integers: a memory-accessing instruction can access any (unprotected) location in memory, and therefore the program is allowed to guess the location of objects (as shown in Fig. 4.1). Some assembly languages have a flat memory model; others, such as those for machines with segmented memory, do not.

While a flat memory model is conceptually simple and is a good match for low-level programming, it hinders high-level optimizations that are routinely performed by and considered essential in modern compilers.

4.1.2 Data-Flow Provenance Tracking

In the previous example, we showed that the program can print either `0` or `1` depending on where the memory allocator places the allocated blocks. This dependence on the run-time behavior of the allocator overconstrains the compiler, blocking it from performing important optimizations such as store forwarding. For example, we want the compiler to be able to propagate the store `q[2] = 0` to the print instruction. Hence, the memory model needs a way to prevent the store to p[6] from accessing q[2] regardless of where p and q end up pointing at run time. For example, rules to this effect have been a part of C since C89.

Data-flow provenance tracking provides a way to prevent objects from being accessed via pointers derived from unrelated objects. The idea is that each pointer is a pair of two values: the object to which it can point to, and the memory address (or an offset within that object). It is undefined behavior (UB) to try to access memory with a pointer that is out-of-bounds of its object. This semantics is sufficient to allow the compiler to conclude that p[6] cannot access q[2], regardless of the fact that at runtime they may end up referring to the same location.

Data-flow provenance tracking could be defined like this:

```
char *p = malloc(4);   // (val=0x10, obj=p)
char *q = malloc(4);   // (val=0x14, obj=q)
char *q2 = q + 2;      // (val=0x16, obj=q)
char *p6 = p + 6;      // (val=0x16, obj=p)

*q2 = 0;  // OK
*p6 = 1;  // UB, since out-of-bounds of obj p
print(*q2);  // can be replaced with print(0);
```

The first store through q2 succeeds, since it is within the bounds of object q. The second store, however, triggers UB because the pointer is out-of-bounds of its base object (p), even though the program correctly guessed the address of a valid object (as shown in Fig. 4.2). Finally, the compiler can safely propagate the store *q2 = 0 to the print instruction since there are no well-defined store instructions in between.

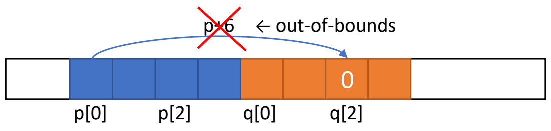


Figure 4.2: In a memory model with data-flow provenance tracking, `p[6]` is not allowed to alias `q[2]`

4.1.3 Extending Provenance to Integers

The model we showed in the previous section does not support low-level language features like integer to pointer casts. To support this functionality, we can extend integers with provenance information: [1]

```
char *p = malloc(4);  // (val=0x10, obj=p)
char *q = (int*)0x10; // (val=0x10, obj=nil)

*q = 0; // UB, since obj=nil
if (p == q)
    *q = 1; // still UB; obj=nil

int v = (int)p;        // (val=0x10, obj=p)
int w = v + 2;         // (val=0x12, obj=p)

*(char*)w = 3;  // OK

char *r = malloc(4);   // (val=0x14, obj=r)
int x = v + (int)r;    // (val=0x24, obj=??)
int y = x - (int)r;    // (val=0x10, obj=??)
```

In this model, each integer and pointer variable tracks a numeric value plus the object it refers to, or `nil` if none. As in the previous model, addresses of objects, even if stored as integer variables, need to be derived from an object. Hence, the stores through `q` are UB. The accesses through `w` are well-defined since the value of this integer variable derives (data-flow wise) from a valid

[1]We will use `int` to represent an integer type that is sufficiently large to hold a pointer for brevity.

object. The last lines of the example show that provenance tracking breaks down when doing integer arithmetic operations. It is hard to assign meaningful semantics to cases like these.

A drawback of this model—fatal in practice—is that it blocks many integer optimizations, such as propagation of equalities as done by, e.g., global value numbering (GVN) or range analysis. For example, transforming "`(a == b) ? a : b`" into "`b`" is incorrect in this model: even if two integer variables compare equal, they may still have different provenances. We give a more complete example to demonstrate the problem:

```
char *p = malloc(4);  // (val=0x10, obj=p)
char *q = malloc(4);  // (val=0x14, obj=q)
int v = (int)p + 4;   // (val=0x14, obj=p)
int w = (int)q;       // (val=0x14, obj=q)

if (v == w)
    *(int*)w = 2;
```

In this program, `v` and `w` happen to have the same value, but they differ in their provenance. Hence it is not safe to replace "`*(int*)w = 2`" with "`*(int*)v = 2`". Doing so would introduce UB since "`v`" is only allowed to access object `p` and its offset is out-of-bounds. However, this sort of equality propagation is routinely done by GVN. In fact, a transformation similar to this one performed by GVN was responsible for miscompiling the Rust code shown in Appendix A.2.

4.1.4 Wildcard Provenance

The previous memory model has the advantage of supporting low-level operations and enabling high-level memory optimizations. However, since integer variables now carry provenance, it makes some integer optimizations unsound. One way to solve this problem is to remove provenance from integer variables, as follows:

```
char *p = malloc(4);    // (val=0x10, obj=p)
char *q = malloc(4);    // (val=0x14, obj=q)
int v = (int)p + 4;     // (val=0x14)
int w = (int)q;         // (val=0x14)

if (v == w) {
    char *r = (int*)w;  // (val=0x14, obj=*)
    *r = 2;
}
```

The differences to the example in the previous section are that (1) integers only carry a numeric value, and (2) a pointer obtained by casting from an integer can access any object (represented with a `*`). Therefore, `v` and `w` can be used interchangeably, and GVN for integers becomes sound again. This model has a major disadvantage: precise alias analysis becomes very difficult as soon as a single integer-to-pointer cast has been performed by the program being compiled. In the next sections we explore how to recover precision.

4.1.5 Inbounds Pointers

In the previous section, we presented a model with wildcard provenance; this works, but it impedes precise alias analysis. In this section we explain the model currently used by LLVM where pointer arithmetic is optionally "inbounds," allowing some precision to be recovered by making out-of-bounds pointer arithmetic undefined:

```
char *p = malloc(4);  // (val=0x10, obj=p)
char *q = foo(p);     // (val=0x13, obj=p)
char *r = q +inb 2;   // poison: 0x15 is out of bounds of p

p[1] = 0;
*r = 1;               // UB
print(p[1]);          // prints 0 or 1?
```

When doing inbounds pointer arithmetic (`+inb`), the base pointer and the result must be within bounds of the same object (or one past its end). This is not the case in `r`, hence the result of the operation is **poison**, which then makes the dereference of this pointer UB [6].

Even if the compiler does not know the value of `q`, because of the inbounds pointer arithmetic it now knows that the minimum offset of `r` has to be two, since both `q` and `r` have to be in bounds of the same object (i.e., $0 \leq o_q \leq n$ and $0 \leq o_r \leq n$, with o_q and o_q being the offsets of `q` and `r` respectively within the object, and n the object size). Since the access to `p[1]` only accesses offset one of an object, and `*r` can only access offset two or beyond, the compiler can conclude these accesses do not alias (and that the program prints `0` always).

4.2 A Memory Model for LLVM

This section informally describes a modified IR-level memory model that: enables high-level optimizations while still supporting low-level code; does not restrict movement of pointer arithmetic instructions (which remain pure functions); and, does not inhibit any standard integer optimizations. Section 4.3 formalizes the new model.

4.2.1 Deferred Bounds Checking

A drawback of LLVM's current inbounds pointer checking is that it prevents reordering of pointer arithmetic instructions and allocation functions:

```
char *p = malloc(4);   // (val=0x10, obj=p)
char *q = malloc(4);   // (val=0x14, obj=q)

char *r = (char*)((int)p + 5);  // (val=0x15, obj=*)
char *s = r +inb 1;             // (val=0x16, obj=q)
*s = 0; // OK
```

In this example, `s` is a valid pointer (i.e., it is in bounds of an object).

However, if we move the definitions of `r` and `s` across that of `q`, `s` becomes out-of-bounds, and thus gets assigned **poison**.[2]

Constraining the movement of instructions is not desirable, since it inhibits optimizations like code hoisting. LLVM, as it turns out, freely moves pointer arithmetic instructions around. This is unsound. Our new model fixes the problem by instead using *deferred* bounds checking, in contrast with LLVM's current *immediate* bounds checking.

In deferred bounds checking, we allow out-of-bounds pointers to be created and manipulated; undefined behavior is only triggered when such a pointer is dereferenced. It is now OK to reorder pointer arithmetic across allocation functions in the previous example:

```
char *p = malloc(4);             // (val=0x10, obj=p)

char *r = (char*)((int)p + 5);  // (val=0x15, obj=*)
char *s = r +inb 1;              // (val=0x16, obj=*, inb={0x15,0x16})

char *q = malloc(4);             // (val=0x14, obj=q)

*s = 0; // OK since 0x15 and 0x16 are inbounds of same object
```

For pointers with `obj=*`, we now track a set of addresses that have to be within bounds of the same object when the pointer is dereferenced. On every inbounds pointer arithmetic operation, we record in the `inb` field the base pointer as well as the resulting pointer. A memory access operation is UB if not all the addresses in `inb` are within bounds of the same object. Therefore, the inbounds check is *delayed* until the pointer is dereferenced. While deferred bounds checking achieves the same effect as immediate bounds checks, it allows

[2]Note that while this example is not correct in memory models with data-flow provenance tracking, it is ok in our model. Even though we build a pointer into `q` based on `p`, this might be the result of the compiler propagating an equality that established that `p == q + 4`.

free movement of pointer arithmetic instructions since they now do not depend on the memory state.

Precision could be further increased by replacing `*` provenance with the object(s) the cast pointer refers to. In the first example above we could define r to have value `(val=0x15, obj=q)` instead. However, this is also a form of immediate bounds checking with the same movement restriction. Moreover, some addresses are in bounds of two objects, like `0x14` in the example (corresponds to `p + 4` and `q`), adding further complexity to the model. Therefore, we do not use these semantics.

4.2.2 Preventing Address Guessing

This section introduces *twin allocation*, a technique that allows our model to prevent a program from guessing addresses of objects without data-flow provenance tracking.

A problem with wildcard provenance is that a pointer formed out of an integer can access any object. Consequently, a program may be able to guess the address of any object and access it. This makes precise alias analysis very difficult.

A simple idea to prevent guessing is to exploit the fact that allocation functions return a non-deterministic value:

```
char *p = malloc(4);  // (val=*, obj=p)
char *q = 0x10;
*q = 0; // UB if val(p) != 0x10
```

This program can guess the address of `p` in a possible execution where `malloc` returns `0x10`. However, there is at least one execution where the program fails to guess the address of `p` (e.g., `malloc` returns `0x20`), and so the program would trigger UB. Since there is at least one execution where the program would trigger UB, the compiler can assume `q` cannot alias with `p` (or with any object at all).

Even with non-deterministic allocations, a program can still (desirably) observe the address of an object such that a pointer created by casting from an integer can alias with that object, e.g.:

```
char *p = malloc(4);  // (val=*, obj=p)
*p = 0;
int v = 0x10;
if ((int)p == v)
    *(int*)v = 1;
print(*p); // can print 0 or 1
```

This program is well-defined and may print `0` or `1`, depending on the return value of `malloc`. The comparison is sufficient for the program to observe the address of object `p`, and so `*(int*)v = 1` will not trigger UB in any execution.

However, this semantics still has a caveat: it allows programs to guess addresses through a “side-channel leak.” The leak happens when the memory only has a single address left where a new object can be allocated. For example, assume that a system has an 8-bit heap segment as well as 8-bit pointers, that heap address `0x00` is legal, and that the allocations below succeed:

```
char *p = malloc(0x80);
char *q = malloc(0x80);

*q = 0;
int v = ((int)p == 0x00) ? 0x80 : 0x00;
*(char*)v = 1;

print(*q); // prints 1
```

Since each heap cell is half the size of the address space, there are only two possible heap configurations: `p`-first or `q`-first. Therefore, a single test allows the program to guess the address of `q` without having to explicitly observe it. In other words, when memory is finite, returning a non-deterministic value from allocation functions is not sufficient to prevent programs from guessing addresses.

Our solution is to change allocation functions to reserve (at least) *two* blocks instead of a single one, as it happens in run-time implementations. We call this technique *twin allocation*, and we use it to formalize the notion that a program cannot guess the address of an object.

We will informally explain the concept of twin allocation with the following example:

```
char *p = malloc(1);
char *q = malloc(1);

*q = 0;
int v = (int)p + 1;  // equal to q?
*(char*)v = 1;

print(*q); // prints 0 or 1?
```

Since the address of `q` was not observed, we would like the compiler to be able to conclude that the program can only print `0`. However, as we have seen previously, if the memory is full, observing the address of one object may implicitly disclose the address of another object. In Fig. 4.3(a) we show a possible memory configuration before our allocations, and (b) shows the configuration after allocating `p` and `q`.

With twin allocation, each allocation function reserves at least two blocks. Non-deterministically, one of the blocks is used and its address is returned, and the remaining blocks are marked as unreachable (i.e., it is UB to access those memory regions). By reserving two blocks, we guarantee there is enough non-determinism left such that the program cannot guess the address of a block even if the memory is full.

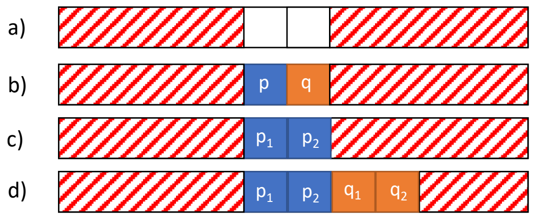


Figure 4.3: Memory configuration: (a) almost full with only two bytes left, (b) after allocating `p` and `q` in (a), (c) after allocating `p` with twin allocation semantics in (a), (d) alternative configuration where twin allocation had enough space for both objects.

In Fig. 4.3(c), we show that with twin allocation our previous example would simply run out of memory, and thus the program cannot continue and try to guess the address of `q`. In (d) we show another memory configuration where the space left was just enough to allocate two blocks for each allocation. Since we have two blocks per object, `malloc` can still return one of the two addresses non-deterministically, effectively inhibiting the program from guessing the address of an object. Even if a program is able to guess the address of, say, $\mathtt{p}_1$ (and even $\mathtt{p}_2$ as well), it is not able to guess which of the remaining addresses points to `q`: It cannot know which of $\mathtt{q}_1$ and $\mathtt{q}_2$ was used.

4.2.3 Summary

We have informally presented our memory model for LLVM IR. To support both high-level optimizations and low-level code, we split pointers into two categories. First, logical pointers, derived from allocation sites, for which we do data-flow dependence tracking. That is, a pointer `q` obtained by pointer arithmetic operations from `p` (e.g., `q = p + x`) can only access the same object as `p`. Second, physical pointers, derived from integer-to-pointer casts, for which we do not do data-flow dependence tracking, since that would block standard integer optimizations such as equality propagation. We employ two new techniques to recover precision instead: delayed bounds checking (to restrict the set of objects a pointer can point to, while keeping pointer arithmetic operations pure), and twin memory allocation (to prevent address guessing).

4.3 Semantics and Transformations

In this section, we present a formal view of the modified memory model for LLVM that we informally presented in Section 4.2. Our top-level design goal was to support low-level operations required by C and C++, such as casting between integers and pointers, while also enabling high-level memory optimizations. Additional goals were: not interfering with integer optimizations, not constraining opportunities for code motion, not requiring major changes to make LLVM conform to the new model, and finally, avoiding significant regressions in compile time and in quality of generated code.

4.3.1 Logical and Physical Pointers

As we saw in Section 4.1.4, we have two types of pointers. *Logical* pointers are obtained by calling an allocation function or by doing pointer arithmetic on a logical pointer. In Section 4.1.4 these are the pointers with provenance of one object, e.g., `(val=0x10, obj=p)`. The second type of pointer is the *physical* pointer, which is the result of an integer-to-pointer cast. In Section 4.1.4, these are the pointers with wildcard provenance, e.g., `(val=0x10, obj=*)`; they can access any object.

In Fig. 4.4 we show the definitions for our model. Logical and physical pointers are represented, respectively, by $\mathsf{Log}(l, o, s)$ and $\mathsf{Phy}(o, s, I, cid)$.

Logical Pointers A logical pointer $\mathsf{Log}(l, o, s)$ consists of a logical block id l, an offset o within the block, and the address space s it corresponds to (explained

$$
\begin{array}{lcl}
\text{Num}(sz) & ::= & \{\, i \mid 0 \leq i < 2^{sz} \,\} \\
\text{Time} & ::= & \mathbb{N} \\
\text{BlockID} & ::= & \mathbb{N} \\
\text{CallID} & ::= & \mathbb{N} \\
\text{Mem} & ::= & \text{Time} \times (\text{BlockID} \nrightarrow \text{Block}) \times (\text{CallID} \nrightarrow \text{Time} \uplus \{\textbf{None}\}) \\
\text{AddrSpace} & ::= & \mathbb{N} \\
\text{Block} & ::= & \{\, (t, r, n, a, c, P) \mid t \in \{\, \texttt{stack}, \texttt{heap}, \texttt{global}, \texttt{function} \,\} \wedge \\
 & & r \in (\text{Time} \times (\text{Time} \uplus \{\infty\})) \wedge n \in \mathbb{N} \wedge a \in \mathbb{N} \wedge c \in \text{Byte}^n \wedge \\
 & & P \in (\text{AddrSpace} \nrightarrow \text{Num}(64)^{N+1}) \,\} \\
\text{LogAddr}(s) & ::= & \{\, \mathsf{Log}(l, o, s) \mid l \in \text{BlockID} \wedge o \in \text{Num}(\mathsf{ptrsz}(s)) \,\} \\
\text{PhyAddr}(s) & ::= & \{\, \mathsf{Phy}(o, s, I, cid) \mid o \in \text{Num}(\mathsf{ptrsz}(s)) \wedge I \subset \text{Num}(\mathsf{ptrsz}(s)) \wedge \\
 & & cid \in \text{CallID} \uplus \{\textbf{None}\} \,\} \\
\text{Addr}(s) & ::= & \text{LogAddr}(s) \uplus \text{PhyAddr}(s) \\
[\![\textbf{i}sz]\!] & ::= & \text{Num}(sz) \uplus \{\, \textbf{poison} \,\} \\
[\![\langle sz \times ty \rangle]\!] & ::= & \{0, \ldots, sz - 1\} \rightarrow [\![ty]\!] \\
[\![ty*]\!] & ::= & \text{Addr}(0) \uplus \{\, \textbf{poison} \,\} \\
\text{Name} & ::= & \{\, \texttt{\%x}, \texttt{\%y}, \ldots \,\} \\
\text{Reg} & ::= & \text{Name} \rightarrow \{\, (ty, v) \mid v \in [\![ty]\!] \,\} \\
\text{Byte} & ::= & \text{Bit}^8 \\
\text{Bit} & ::= & [\![\textbf{i1}]\!] \uplus \text{AddrBit} \\
\text{AddrBit} & ::= & \{\, (p, i) \mid \exists s.\ p \in \text{Addr}(s) \wedge (0 \leq i < \mathsf{ptrsz}(s)) \,\}
\end{array}
$$

Figure 4.4: Definitions. $\mathsf{ptrsz}(s)$ is the pointer size (in bits) for a given address space s (e.g., 64). The set of all possible values of a type ty is given by $[\![ty]\!]$.

later in Section 4.3.2). A logical pointer corresponds to the address $P + o$ on the physical machine, where P is the base address of block l.

Logical pointers realize the rule that pointers derived from one object can never be used to modify other objects. This is achieved by making it impossible to change the value of l: Pointer arithmetic only affects the offset o.

Physical Pointers As we have seen in Section 4.1.4, tracking objects in a pointer obtained by an integer-to-pointer cast is not viable because (1) some addresses may be within bounds of multiple objects, and (2) it prevents reordering of instructions. Hence we introduce physical pointers, roughly corresponding to pointers in a flat memory model.

A physical pointer $\mathsf{Phy}(o, s, I, cid)$ consists of an offset o within the address space s (i.e., the physical address), as well as two additional fields I and cid to restrict the set of objects the pointer can access. This allows us to recover alias analysis precision, and it enables several of the optimizations allowed by the C and C++ standards. Field I is a set of physical addresses that corresponds to the `inb` field we used to specify deferred bounds checking in Section 4.2.1. When the pointer is dereferenced, each address in I must be inbounds of the same object as o. Field cid is a call id, which corresponds to the time stamp when the pointer was passed as argument to a function, or **None** if the pointer did not originate from an argument. The intent is to show that pointers received as arguments do not alias locally allocated objects, e.g.:

```
int f(int *p) {
    int a = 0;
    if (&a == p)
        *p = 1;
    return a;  // returns 0 or 1?
}
```

Since a physical pointer can access any object, if there was no call id restriction, this function could return either `0` or `1`. However, since `p` has a call id of a function call still on the call stack, it cannot access any object created after the call.

The motivation to have an indirection in the time stamp of the call time in the pointer (cid indexes in memory M to retrieve the time stamp) is to support escaping pointers. Escaped physical pointers should behave as their cids being **None** after termination of the function call. This supports moving function calls across other function calls. If a function stores a pointer received by argument in a global variable, we did not want to have to record that fact and change all such pointers when the function returns. This way we only need to change the mapping between *cid* and time stamp on function return (set it to **None**).

4.3.2 Address Spaces

LLVM uses address spaces to represent distinct memories, and our memory model also supports this feature. For example, a machine might use one address space for the CPU and another for the GPU, or one address space for code and another for data. Since both memories may have overlapping address ranges (e.g., they may both use addresses in the range $[0, 2^{64})$), the address space field in pointers is used to disambiguate between the two.

The main memory of the CPU is assigned the address space zero. It is possible that a physical memory region is mapped into multiple address spaces. In this case the application can use an address cast instruction to convert pointers between address spaces.

A consequence of possible overlapping of address spaces is that pointers belonging to different address spaces may alias. To improve precision of alias analysis, we parameterize our model by the overlap.

4.3.3 Memory Blocks

We define memory $M = \text{Time} \times (\text{BlockID} \nrightarrow \text{Block}) \times (\text{CallID} \nrightarrow \text{Time} \uplus \{\textbf{None}\})$ as a triple of a time stamp, a map from logical block ids to memory blocks, and a map recording the time stamp of each function call (indexed on *cid* of physical pointers). When a new memory block is created (e.g., with **malloc** or **alloca**) or deallocated, the time stamp is incremented by one.

A memory block is a tuple (t, r, n, a, c, P), where t is the block type (e.g., stack or heap allocated), r is the life range of the block, n is the block size in bytes, a is the alignment, c the contents of the block (the actual data), and P has the addresses of the block.

$(\iota = \text{“}r = \textbf{call i}8{*}\ \textbf{malloc}(\textbf{i}64\ \mathit{len})\text{”})$

MALLOC

$$\frac{\begin{array}{c} n = [\![\mathit{len}]\!]_R \quad c = \textbf{i}(8 \times n)\!\downarrow\!(\textbf{poison}) \\ P \text{ unallocated physical addresses}, \quad l \text{ fresh} \\ m' = m[l \mapsto (\texttt{heap}, (\tau_{cur}, \infty), n, a, c, P)] \end{array}}{R, (\tau_{cur}, m, C) \overset{\iota}{\hookrightarrow} R[r \mapsto \mathsf{Log}(l, 0, 0)], (\tau_{cur} + 1, m', C)}$$

$(\iota = \text{“}\textbf{call void free}(\textbf{i}8{*}\ \mathit{ptr})\text{”})$

FREE-LOGICAL

$$\frac{\begin{array}{c} \mathsf{Log}(l, 0, 0) = [\![\mathit{ptr}]\!]_R \\ m(l) = (\texttt{heap}, (b, \infty), n, a, c, P) \\ m' = m[l \mapsto (\texttt{heap}, (b, \tau_{cur}), n, a, c, P)] \end{array}}{R, (\tau_{cur}, m, C) \overset{\iota}{\hookrightarrow} R, (\tau_{cur} + 1, m', C)}$$

$(\iota = \text{“}r = \textbf{ptrtoint}\ \mathit{ty}{*}\ \mathit{op}\ \textbf{to i}\mathit{sz}\text{”})$

PTRTOINT-LOGICAL

$$\frac{\begin{array}{c} \mathsf{Log}(l, o, s) = [\![\mathit{op}]\!]_R \\ \mathsf{cast2int}_M(l, o, s) = j \end{array}}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto j\%2^{sz}], M}$$

$(\iota = \text{“}r = \textbf{inttoptr i}\mathit{64}\ \mathit{op}\ \textbf{to}\ \mathit{ty}{*}\text{”})$

INTTOPTR

$$\frac{\begin{array}{c} i = [\![\mathit{op}]\!]_R \\ p = \mathsf{Phy}(i, 0, \emptyset, \textbf{None}) \end{array}}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto p], M}$$

$(\iota = \text{“}r = \textbf{icmp eq}\ \mathit{ty}{*}\ \mathit{op_1}\ \mathit{op_2}\text{”})$

ICMP-PTR-LOGICAL

$$\frac{\begin{array}{c} \mathsf{Log}(l, o_1, s) = [\![\mathit{op}_1]\!]_R \\ \mathsf{Log}(l, o_2, s) = [\![\mathit{op}_2]\!]_R \end{array}}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto (o_1 = o_2)], M}$$

$(\iota = \text{“}r = \textbf{icmp eq}\ \mathit{ty}{*}\ \mathit{op_1}\ \mathit{op_2}\text{”})$

ICMP-PTR-LOGICAL’

$$\frac{\begin{array}{c} \mathsf{Log}(l_1, o_1, s) = [\![\mathit{op}_1]\!]_R \\ \mathsf{Log}(l_2, o_2, s) = [\![\mathit{op}_2]\!]_R \\ l_1 \neq l_2 \end{array}}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto \textbf{false}], M}$$

$(\iota = \text{“}r = \textbf{icmp eq}\ \mathit{ty}{*}\ \mathit{op_1}\ \mathit{op_2}\text{”})$

ICMP-PTR-PHYSICAL

$$\frac{\begin{array}{c} \mathsf{Phy}(o_1, s, I_1, \mathit{cid}_1) = [\![\mathit{op}_1]\!]_R \\ \mathsf{Phy}(o_2, s, I_2, \mathit{cid}_2) = [\![\mathit{op}_2]\!]_R \end{array}}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto (o_1 = o_2)], M}$$

$(\iota = \text{“}r = \textbf{icmp ule}\ \mathit{ty}{*}\ \mathit{op_1}\ \mathit{op_2}\text{”})$

ICMP-ULE-PTR-PHYSICAL

$$\frac{\begin{array}{c} \mathsf{Phy}(o_1, s, I_1, \mathit{cid}_1) = [\![\mathit{op}_1]\!]_R \\ \mathsf{Phy}(o_2, s, I_2, \mathit{cid}_2) = [\![\mathit{op}_2]\!]_R \end{array}}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto (o_1 \leq_u o_2)], M}$$

$(\iota = \text{“}r = \textbf{icmp ule}\ \mathit{ty}{*}\ \mathit{op_1}\ \mathit{op_2}\text{”})$

ICMP-PTR-ULE-LOGICAL

$$\frac{\begin{array}{c} \mathsf{Log}(l, o_1, s) = [\![\mathit{op}_1]\!]_R \\ \mathsf{Log}(l, o_2, s) = [\![\mathit{op}_2]\!]_R \end{array}}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto (o_1 \leq_u o_2)], M}$$

$(\iota = \text{“}r = \textbf{icmp eq}\ \mathit{ty}{*}\ \mathit{op_1}\ \mathit{op_2}\text{”})$

ICMP-PTR-LOGICAL-NONDET-TRUE

$$\frac{\begin{array}{cc} \mathsf{Log}(l_1, o_1, s) = [\![\mathit{op}_1]\!]_R & l_1 \neq l_2 \\ \mathsf{Log}(l_2, o_2, s) = [\![\mathit{op}_2]\!]_R & \big((o_1 = n_1 \wedge o_2 = 0)\ \vee o_1 > n_1 \vee \\ m(l_1) = (t_1, r_1, n_1, a_1, c_1, P_1) & (o_1 = 0 \wedge o_2 = n_2)\ \vee\ o_2 > n_2 \vee \\ m(l_2) = (t_2, r_2, n_2, a_2, c_2, P_2) & r_1, r_2 \text{ disjoint}\big) \end{array}}{R, (\tau_{cur}, m, C) \overset{\iota}{\hookrightarrow} R[r \mapsto \textbf{true}], (\tau_{cur}, m, C)}$$

Figure 4.5: Selected rules of our operational semantics

($\iota =$ "$r = \textbf{gep}\ ty{*}\ op_1\ \textbf{i}sz\ op_2$")
GEP-LOGICAL

$$\frac{\mathsf{Log}(l,o,s) = [\![op_1]\!]_R \quad i = [\![op_2]\!]_R \qquad o' = (o + \mathsf{bytewidth}(ty) * i)\,\%\,2^{\mathsf{ptrsz}(s)}}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto \mathsf{Log}(l,o',s)], M}$$

($\iota =$ "$r = \textbf{gep}\ ty{*}\ op_1\ \textbf{i}sz\ op_2$")
GEP-PHYSICAL

$$\frac{\mathsf{Phy}(o,s,I,cid) = [\![op_1]\!]_R \quad i = [\![op_2]\!]_R \qquad o' = (o + \mathsf{bytewidth}(ty) * i)\,\%\,2^{\mathsf{ptrsz}(s)}}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto \mathsf{Phy}(o',s,I,cid)], M}$$

($\iota =$ "$r = \textbf{gep inbounds}\ ty{*}\ op_1\ \textbf{i}sz\ op_2$")
GEP-INBOUNDS-LOGICAL

$$\frac{\begin{matrix}\mathsf{Log}(l,o,s) = [\![op_1]\!]_R \\ o' = o + \mathsf{bytewidth}(ty) * i\end{matrix} \qquad \begin{matrix} i = [\![op_2]\!]_R \\ \mathsf{inbounds}_M(l,o) \\ \mathsf{inbounds}_M(l,o')\end{matrix}}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto \mathsf{Log}(l,o',s)], M}$$

($\iota =$ "$r = \textbf{gep inbounds}\ ty{*}\ op_1\ \textbf{i}sz\ op_2$")
GEP-INBOUNDS-PHYSICAL

$$\frac{\mathsf{Phy}(o,s,I,cid) = [\![op_1]\!]_R \qquad i = [\![op_2]\!]_R \qquad o' = o + \mathsf{bytewidth}(ty) * i \qquad 0 \le o' < 2^{\mathsf{ptrsz}(s)}}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto \mathsf{Phy}(o',s,I \cup \{\, o, o' \,\}, cid)], M}$$

($\iota =$ "$r = \textbf{psub}\ ty{*}\ op1,\ op2$")
PSUB-LOGICAL

$$\frac{\mathsf{Log}(l,o_1,s) = [\![op_1]\!]_R \qquad \mathsf{Log}(l,o_2,s) = [\![op_2]\!]_R \qquad i = (o_1 - o_2)\,\%\,2^{\mathsf{ptrsz}(s)}}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto i], M}$$

($\iota =$ "$r = \textbf{psub}\ ty{*}\ op1,\ op2$")
PSUB-LOGICAL-POISON

$$\frac{\mathsf{Log}(l_1,o_1,s) = [\![op_1]\!]_R \qquad \mathsf{Log}(l_2,o_2,s) = [\![op_2]\!]_R \qquad l_1 \neq l_2}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto \textbf{poison}], M}$$

($\iota =$ "$r = \textbf{psub}\ ty{*}\ op1,\ op2$")
PSUB-PHYSICAL

$$\frac{\mathsf{Phy}(o_1,s,I_1,cid_1) = [\![op_1]\!]_R \qquad \mathsf{Phy}(o_2,s,I_2,cid_2) = [\![op_2]\!]_R \qquad i = (o_1 - o_2)\,\%\,2^{\mathsf{ptrsz}(s)}}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto i], M}$$

Figure 4.6: Selected rules of our operational semantics (cont.)

When a block is deallocated (e.g., with **free** or on function exit), it is not deleted from memory. [3] Instead, we set the end of the lifetime range to the current memory time stamp, and increment it as well. FREE-LOGICAL in Fig. 4.5 shows the semantics of **free**. If a physical pointer $\mathsf{Phy}(o, s, I, cid)$ is given to **free**, it is equivalent to freeing a dereferenceable block whose base address is o. Double-freeing a block or **free** with a logical pointer with non-zero offset is UB. **free** with NULL is no-op.

Memory allocation functions reserve at least two blocks: The block actually observed by the program, and N additional *twin blocks*. The number N of twin blocks is a parameter of the semantics. (We will discuss in Section 4.3.12 why $N = 1$ might not be enough.) The block's base addresses in address space s are stored in $P(s)$, a sequence of physical addresses: $P(s)_0$ is the actual base address used and observed by the program, while the remaining addresses in the sequence are the base addresses of the twin blocks.

Crucially, we maintain the invariant that for P, P' of any pair of different live (non-deallocated) blocks, the address ranges $[P(s)_i, P(s)_i + n)$ and $[P'(s)_j, P'(s)_j + n)$ are disjoint. Thus, **malloc** reserves space for both the block and its twins (Fig. 4.5). The rest of the semantics only depends on $P(s)_0$ and ignores the remaining base addresses.

In Fig. 4.5, notation $[\![op]\!]_R$ represents the evaluation of an instruction's operand:

$$[\![v]\!]_R = \begin{cases} R(v) & \text{if } v \text{ is a register} \\ v & \text{if } v \text{ is a constant or } v = \textbf{poison} \end{cases}$$

[3]Note that memory is deallocated at run time as expected. Also, in our semantics addresses can be reused because allocation functions allocate blocks with addresses that are disjoint from all other *live* blocks only.

4.3.4 Pointer Arithmetic

LLVM IR has a single instruction for pointer arithmetic, **getelementptr**, or **gep** for short. In Fig. 4.5 we show the semantics of several cases. For a logical pointer, the result is also a logical pointer where only the offset is updated (GEP-LOGICAL). Likewise for physical pointers (GEP-PHYSICAL).

Function $\mathsf{inbounds}_M(l, o)$ checks if a given offset o is within bounds of object l in memory $M = (\tau, m, C)$. If $m(l) = (t, r, n, a, c, P)$, $\mathsf{inbounds}_M(l, o)$ is true iff $0 \leq o \leq n$.

When **gep** has the **inbounds** tag (e.g., when compiling C/C++ code or most cases in safe languages), the compiler can assume that the input pointer is valid (within bounds of some object or else one element past the end) and also that the resulting pointer is valid. For logical pointers, the bounds checking is immediate (GEP-INBOUNDS-LOGICAL). If either of the inbounds conditions fails, the result is **poison** (not shown).

If the pointer is physical, we use deferred bounds checking: input and output offsets are added to I. They are checked to be inbounds only when the pointer is dereferenced. As seen in Section 4.2.1, this allows free movement of **gep** instructions since they do not depend on the memory state.

4.3.5 Casting

LLVM has two pointer/integer casting instructions: **ptrtoint** and **inttoptr**. The semantics of casting a logical pointer to an integer is given in PTRTOINT-LOGICAL. Function $\mathsf{cast2int}_M(l, o, s)$ converts $\mathsf{Log}(l, o, s)$ to the integer $P(s)_0 + o$ based on block l. If the operation $P(s)_0 + o$ overflows, it wraps around. This can happen if the pointer is out of bounds. Instruction '**ptrtoint** $\mathsf{Phy}(o, s, I, \mathit{cid})$' yields o. If the size of the destination type **i**sz is larger than the pointer width, the result is zero-extended. If it is smaller, the most significant bits are truncated.

Casting from an integer to a pointer returns a physical pointer with no provenance information (INTTOPTR). No check is done on whether the pointer refers to a valid location or not. This avoids a dependency on the memory state, and thus allows code motion.

The NULL pointer used in, e.g., C is defined as '**inttoptr** 0' because C programs use '`(void*)0`' as the null pointer value.

Casting a pointer to a different address space through **addrspacecast** translates the pointer's offset(s) using a target-specific mapping function. If the pointer is logical, it preserves the block id and offset if in bounds, otherwise it yields **poison**. If the pointer is physical, '**ptrtoint** $\mathsf{Phy}(o, s, I, cid)$' updates o as well as the offsets in I.

In our semantics, all casting instructions can be freely moved, removed, or introduced.

4.3.6 Pointer Comparison

Consider the following program compares two logical pointers that point to different objects. Pointer optimizations would like to fold this comparison to false:

```
char *p = malloc(4);
char *q = malloc(4);

char *pp = some expr over p;
char *qq = some expr over q;
if (pp == qq) { /* always  false? */ }
```

If `pp` and `qq` are dereferenceable, i.e., their offsets are in the range $[0, 4)$, the comparison should yield `false` since the resulting machine addresses cannot be the same (since objects `p` and `q` cannot overlap). However, what if `pp == p + 4`, `qq == q`, and the objects `p` and `q` were allocated consecutively (i.e., `p + 4 == q`)? Even though `pp` and `qq` are conceptually very different pointers, their

	Integer comparison	Non-deterministic
Fold $\mathsf{p} = \mathsf{q}$ to false if p.bid $\neq$ q.bid	No	Yes
Fold $\mathsf{p} + i = \mathsf{q} + i$ to $\mathsf{p} = \mathsf{q}$	Yes	No
Fold $(int)\mathsf{p} = (int)\mathsf{q}$ to $\mathsf{p} = \mathsf{q}$	Yes	No
Fold $\mathsf{p} < \mathsf{q} \wedge \mathsf{p} \neq \mathsf{q}$ to $\mathsf{p} < \mathsf{q}$	Yes	No
Fold $\mathsf{p} < \mathsf{q} \wedge \mathsf{q} \neq null$ to $\mathsf{p} < \mathsf{q}$	Yes	Potentially
Run-time aliasing checks	Yes	Correct, but not useful
Analysis of pointers cast from integers	Harder	Easy

Table 4.1: Comparison of two semantics for pointer comparison.

underlying machine addresses are the same. Therefore, if the compiler lowers pointer comparison into a comparison of the respective machine addresses in assembly, the comparison would yield `true`.

There are two solutions for this. The first solution is relying on nondeterminism of their allocated locations. In twin allocation, there exists a twin execution such that `p + 4 != q`. Therefore, we can fold the comparison into `false` by the definition of refinement[4]

The second solution is to define the comparison `p + n == q` to yield a nondeterministic value, justifying both the lowering to machine address comparison, and the desired optimization. This way the compiler can always fold comparisons between different objects to `false` without having to prove that they cannot have the same machine address.

We found that there are pros and cons of both semantics for the comparison of pointers of different blocks, and that neither of them covers all optimizations that LLVM performs. Table 4.1 summarizes the effects on each of the optimizations. We will explain each solution in detail.

[4]This might require more than one twin. Section 4.3.12 has more details about this issue.

Provenance-Agnostic Comparison

The first semantics, defining pointer comparison as an operation that simply compare the operands' integer addresses, is simple and a friendly definition to compiler developers. This explains LLVM's arithmetic optimizations on pointer comparisons as depicted in Table 4.1. Most importantly, the definition is consistent with the current wording of pointer comparison in LLVM Language Reference[5].

The cons of this definition is that it cannot support 'high-level' optimizations. Given two pointers p and q, if a program learns that q is placed right after p in memory, the program can potentially change the contents of q without the compiler realizing it. Therefore, alias analysis must answer very conservatively if a pointer is compared. Also, alias analysis must give up when they encounter an integer-to-pointer cast, which is what LLVM's alias analyses already do.

LLVM has a transformation that is incorrect under this definition. Unless `p` is compared only once, `p == q` cannot be folded to `false` (the first optimization in Table 4.1). Therefore, we must remove the optimization which is what LLVM already does today from LLVM to make it sound. Furthermore, C++ standard allows an unspecified value when comparing two pointers from different objects. LLVM optimizations cannot exploit this power under the integer comparison semantics.

Provenance-Aware Comparison

The second solution, defining the comparison `p + n == q` to yield a non-deterministic value, addresses these concerns. Formally speaking, rule ICMP-PTR-LOGICAL' defines that comparing logical pointers to different blocks can always evaluate to `false`. Furthermore, ICMP-PTR-LOGICAL-NONDET-TRUE states that

[5]For this reason, we chose this semantics for Alive2 (Section 5.3).

if the offset of either pointer is not dereferenceable, the comparison can *also* evaluate to `true`.

A caveat of this choice of semantics is that even if a pointer comparison returns `true`, we cannot assume that two pointers have the same value. However, this was already the case because pointer comparison ignores, e.g., the extra precision fields in physical pointers like *cid*. This makes propagation of pointer equalities (e.g., GVN) unsound. We show later in this section how to make GVN for pointers correct.

Also, making pointer comparison non-deterministic violates the C standard. However, both GCC and LLVM (in C mode) will fold a comparison to false even when the pointers compare equal, effectively choosing code quality over standards conformance. The C semantics makes programs harder to optimize since it causes pointer comparisons to leak information about the memory layout, therefore requiring compilers to conservatively assume that most compared pointers escape (which inhibits many optimizations, such as store forwarding and dead store elimination). Therefore, to give more optimization opportunities to LLVM, we adopt this provenance-aware model for pointer comparison semantics.

For pointer inequality comparison (e.g., `p <= q`), for two logical pointers of the same block, if their offsets are inbounds the result is simply the comparison of their offsets (ICMP-PTR-ULE-LOGICAL). If the offsets are not inbounds, pointer values may overflow in hardware and hence produce a different result than if comparing the offsets. Hence, if one of the offsets is not inbounds, the result of the comparison is nondeterministic, allowing both the compiler to optimize comparisons based on the pointer offsets, as well as efficient compilation of pointer comparisons to assembly.

Comparison of logical pointers into different blocks yields a non-deterministic value. We cannot compare their integer values since that would leak information about the memory layout. We do not make the comparison yield **poison** in this case because optimizations like vectorization introduce comparisons between pointers of potentially different blocks to check at run time if vectorized accesses overlap or not (to check if it is safe to run the vectorized code).

Comparing two physical pointers is equivalent to comparing their integer representations (ICMP-PTR-PHYSICAL, ICMP-ULE-PTR-PHYSICAL). If one pointer is logical and the other physical, the logical pointer is converted to a physical pointer and then they are compared. This naturally supports the definition of comparison with pointer-integer roundtrip in the C/C++ standard: Given a valid pointer `p`, `(void*)(int)p == p` should be true. The pointer-integer round trip yields a physical pointer in our semantics, hence the comparison is always true.

4.3.7 Pointer Subtraction

Pointers can be subtracted to compute the difference between their offsets. This can be soundly implemented by first casting the pointers to integers and then performing an integer subtraction. In fact, this is what LLVM/Clang does today.

However, there is potential for improvement: The C/C++ standard permits an indeterminate result for the subtraction of pointers into different blocks. However, if the operation was realized as an integer subtraction, such an operation would be well defined and therefore leak information about the memory layout. Therefore, lowering pointer subtraction to subtraction of pointers cast to integers fundamentally loses precision.

In our semantics, if the program subtracts logical pointers from different blocks, the result is a **poison** value. To this end, we introduce a new instruction **psub** that takes two pointers and returns their difference. When given two logical pointers, **psub** gives the difference of their offsets if they refer to the same block (PSUB-LOGICAL). Otherwise, if the pointers refer to different logical blocks, **psub** returns **poison** (PSUB-LOGICAL-POISON). If at least one of the pointers is physical, both pointers are cast to integers and their difference is computed.

Besides the theoretical precision improvement by having a dedicated pointer subtraction instruction, there is also a practical advantage. Compiler analyses are naturally imprecise. In particular, when they see a pointer-to-integer or an integer-to-pointer cast, they tend to bail out. By introducing a new instruction for pointer subtraction, we are able to reduce the number of these casts significantly (as shown later in the evaluation).

4.3.8 Memory Block Lifetime

Besides the `p + n == q` case, the machine addresses of different logical pointers may also be equal when addresses get reused by the allocator. For example:

```c
char *p = malloc(4);
free(p);
char *q = malloc(4);
if (p == q) { /* ... */ }
```

We would again like to optimize the comparison to `false` because we are comparing the results of two separate calls to `malloc`. And again, in an actual execution, the addresses may be equal because `malloc` can reuse memory after `free`.

A common solution to this problem is to let the behavior of pointer equality depend on whether the block that `p` points to is still allocated. However, the problem with that approach is that it makes comparison not freely reorderable with deallocation. Another solution is to define comparisons with freed blocks as UB (as C and C++ standards do), but that limits movement of pointer comparisons as well. Instead, we add the concept of a *lifetime* to our memory blocks. The memory time stamp gets incremented on every memory allocation and deallocation. The end of the lifetime of a block is initially ∞ and gets set when the block is deallocated.

In the above situation, because the lifetimes of the two blocks do not overlap, we again make pointer comparison non-deterministic, justifying the optimization. This is reflected in ICMP-PTR-LOGICAL-NONDET-FALSE, which also applies if the lifetimes of the blocks are disjoint.

This lifetime-based handling of pointer comparison has the caveat that a `free` can no longer be moved up above a `malloc` of a different block, since that would make the two blocks' lifetimes no longer overlap, possibly affecting program behavior. Although this an interesting optimization to support (to reduce peak memory consumption and potentially reuse cached memory), LLVM does not perform it.

4.3.9 Load and Store

To perform a memory access of size $sz > 0$ through a pointer p on memory M, the pointer must be *dereferenceable*, written $\mathsf{deref}_M(p, sz)$. If p is a logical pointer $\mathsf{Log}(l, o, s)$ where block l is not freed and $\mathsf{inbounds}_M(l, o) \wedge \mathsf{inbounds}_M(l, o + sz)$, then we have $\mathsf{deref}_M(p, sz)$.

If p is a physical pointer $\mathsf{Phy}(o, s, I, cid)$, there must be a still-alive block $(t, (b, \infty), n, a, c, P)$ with id l and an offset o_l such that $P(s)_0 + o_l = o$ and $\mathsf{inbounds}_M(l, o_l) \wedge \mathsf{inbounds}_M(l, o_l + sz)$. (Note that l and o_l are uniquely determined since memory blocks are disjoint and $sz > 0$.) Moreover, all addresses $o' \in I$ must be inbounds of the same block, i.e., $\forall o' \in I$, $\mathsf{inbounds}_M(l, o' - P(s)_0)$. If the pointer was derived from a parameter (i.e., $cid \neq \textbf{None}$) and the function corresponding to that parameter did not return yet (i.e., $M(cid) \neq \textbf{None}$), then $b < M(cid)$, i.e., the block must have been allocated before the function call identified by cid started. If all these requirements are satisfied, we have $\mathsf{deref}_M(p, sz)$.

To support conversion between values and low-level bitwise representation, we define two meta operations, $ty{\downarrow} \in (\llbracket ty \rrbracket \rightarrow \text{Bit}^{\text{bitwidth}(ty)})$ and $ty{\uparrow} \in (\text{Bit}^{\text{bitwidth}(ty)} \rightarrow \llbracket ty \rrbracket)$. For base types, $ty{\downarrow}$ transforms **poison** into the bitvector of all **poison** bits, and defined values into their standard low-level representation (Fig. 4.7a). $\text{getbit}\, v\, i$ is a partial function that returns the ith bit of a value v. If v is a pointer, $\text{getbit}\, v\, i$ returns either **poison** if v is **poison** or a pair (p, i) which is an element of AddrBit denoting the ith bit of a non-poison pointer p. For vector types, $ty{\downarrow}$ transforms values element-wise, where $+\!\!+$ denotes the bitvector concatenation.

$\mathbf{i}sz{\uparrow}(b)$ transforms a bitwise value b to an integer of type $\mathbf{i}sz$ (Fig. 4.7b). Notation $n.i$ is used to represent the ith bit of a non-**poison** integer n. Type punning from pointer to integer yields **poison**. This is needed to justify redundant load-store pair elimination.[6] If any bit of b is **poison**, the result of $\mathbf{i}sz{\uparrow}(b)$ is **poison**. For vector types, $ty{\uparrow}$ transforms bitwise representations element-wise.

$ty{*}{\uparrow}(b)$ transforms a bitvector b into a pointer of type $ty*$. If b is exactly all the bits of a pointer p in the right order, it returns p. Otherwise, it returns **poison**.

Now we define semantics of **load**/**store** operations. Function $\text{Load}(M, p, sz, a)$ returns the bits that correspond to pointer p if $\mathsf{deref}_M(p, sz)$ and if p is a-aligned (i.e., $p \,\%\, a = 0$). **load** yields v if $\text{Load}(M, p, sz, a)$ returns a value v, or UB otherwise. The store operation $\text{Store}(M, p, b, a)$ stores the bit representation b into the memory M and returns the updated memory M' if p is dereferenceable and a-aligned. **store** is UB if $\text{Store}(M, p, b, a)$ fails, and updates the memory to

[6]Redundant load-store pair elimination means removing '`v = load i64 ptr; store v, ptr`'. If reading a logical pointer as integer implicitly casts the pointer, removing this load-store pair would eliminate a cast and hence be illegal. This is discussed further in Section 4.7.

$$\begin{array}{rcl} \mathbf{i}sz{\downarrow}(v) \text{ or } ty{*}{\downarrow}(v) & = & \lambda i.\, \mathrm{getbit}\, v\, i \\ \langle sz \times ty \rangle{\downarrow}(v) & = & ty{\downarrow}(v[0]) \mathbin{+\!\!+} \ldots \\ & & \mathbin{+\!\!+}\, ty{\downarrow}(v[sz-1]) \end{array}$$

(a) Converting a value to a bit vector

$$\mathbf{i}sz{\uparrow}(b) = \begin{cases} n & \text{if } \forall_{0 \le i < sz}\, b[i] \neq \mathbf{poison} \\ & \text{such that } \forall_{0 \le i < sz}\, b[i] = n.i \\ \mathbf{poison} & \text{otherwise} \end{cases}$$

(b) Converting a bit vector to an integer

$$\frac{\begin{array}{c}(\iota = \text{“}r = \mathbf{load}\ ty, ty{*}\ op,\ \mathbf{align}\ a\text{”}) \\ \mathrm{Load}(M, [\![op]\!]_R, \mathrm{bitwidth}(ty), a) \text{ fails}\end{array}}{R, M \overset{\iota}{\hookrightarrow} \mathrm{UB}}$$

$$\frac{\mathrm{Load}(M, [\![op]\!]_R, \mathrm{bitwidth}(ty)) = v}{R, M \overset{\iota}{\hookrightarrow} R[r \mapsto ty{\uparrow}(v)], M}$$

$$\frac{\begin{array}{c}(\iota = \text{“}\mathbf{store}\ ty\ op_1, ty{*}\ op,\ \mathbf{align}\ a\text{”}) \\ \mathrm{Store}(M, [\![op]\!]_R, ty{\downarrow}([\![op_1]\!]_R), a) \text{ fails}\end{array}}{R, M \overset{\iota}{\hookrightarrow} \mathrm{UB}}$$

$$\frac{\mathrm{Store}(M, [\![op]\!]_R, ty{\downarrow}([\![op_1]\!]_R)) = M'}{R, M \overset{\iota}{\hookrightarrow} R, M'}$$

Figure 4.7: Semantics of **load** and **store**

M' otherwise.

4.3.10 Justifying Transformations in Practice

We now illustrate how our semantics can be used in practice in a compiler, and how one can informally reason about the correctness of optimizations. For example, we would like to justify that the following transformation performed by GVN is correct:

```
char *p = malloc(4);                char *p = malloc(4);
int v = (int)p;                     int v = (int)p;
if (v == 10)              ⇒         if (v == 10)
    *(int*)v = 0;                       *(int*)10 = 0;
```

Under our semantics, this transformation is legal because integers do not track provenance. Moreover, the address of object p is observed by the comparison, and therefore the store through (int*)10 is guaranteed to write to object p.

A useful way to think about this is that our semantics effectively take *control-flow* dependencies into account when determining which objects have had their addresses observed. In our example, the address of p is observed within the then block since it was used in a comparison along the control-flow that leads to the block. Hence, we allow (int*)10 to refer to object p. This sort of reasoning can be directly implemented in a compiler.

4.3.11 Preventing Accesses via Guessed Addresses

We now show how twin allocation semantics prevents unobserved blocks from being accessed via guessed addresses. This aspect is essential to enable compiler optimizations, otherwise any pointer could potentially access any object.

A block is *unobserved* if any value derived from (i.e., any value that has data or control dependence on) the block's logical address created at allocation is never used in any of the *address-observing* operations: being (*i*) cast to an integer, (*ii*) compared to a physical address, and (*iii*) subtracted to/from a physical address. For example, in the following program, the first block of size 10 is unobserved because its address a is never used in an address-observing operation, while the second block of size 5 is observed because its address b is compared to the physical address 0x200.

```
char *a = malloc(10);
char *b = malloc(5);
if (b == (char*)0x200)
    *(char*)0x100 = 1;
```

A *guessed* address is necessarily a physical address. We will now show that in the twin allocation model, compilers can safely assume that unobserved blocks cannot be accessed via guessed addresses. Since compilers are allowed to assume anything when the source program has an execution raising UB, it suffices to show that whenever an unobserved block b is accessed via a guessed address p in some execution, there is an alternative execution raising UB. The alternative execution is to take the address of the twin block b' instead of that of b at allocation. After allocating blocks b and b', the twin executions (i.e., the original and the alternative) have exactly the same program state except that:

(i) the logical address created at allocation corresponds to a different underlying machine address (i.e., that of b and b' respectively), and

(ii) validity of b and b' are swapped.

Condition (i) does not make any difference in the twin executions because the machine address is only ever used in address-observing operations. More specifically, our semantics is carefully designed in such a way that all operations other than the address-observing ones are independent of the underlying machine address of any logical address. Thus only condition (ii) can make a difference during execution. This can happen only when one of the blocks b and b' is accessed via a physical address. Since only one of b and b' is accessible in each of the twin executions, at least one of the executions will raise UB when block b is accessed via a guessed address.

For instance, in the above example, suppose that at line 1, two blocks of size `10` are allocated at `0x100` and `0x150` with the former activated. Then at line 4 the block is successfully accessed via the physical address `0x100`, while in the twin execution where the block at `0x150` is activated, the access at `0x100` at line 4 raises UB. Therefore, the compiler can conclude that the store at line 4 cannot access object `a`.

Finally, we discuss why allocating two blocks is necessary even though one of them is always made invalid. The reason is because in order to have equivalent twin executions modulo accesses via guessed addresses as described above, the twin executions should have identical memory layouts. For example, instead of allocating two blocks, consider allocating a single block in such a way that there is enough space left for allocating another one of the same size; and raising out-of-memory if such allocation is not possible. In this semantics, one may think that we can still simulate a twin execution as described above using the free space. However, it does not work due to different memory layouts. Specifically, in the above example program, suppose that before executing line 1, the memory only has two segments of free space: `0x100` $\sim$ `0x109` and `0x200` $\sim$ `0x209`. Then at line 1, the block of size `10` can be allocated at `0x100` or `0x200` because there is enough space left for allocating another one. Then at line 2, the block of size `5` can be allocated at `0x200` or `0x205` in the former case and at `0x100` or `0x105` in the latter case because there is enough space left. Among those twin executions, one of them accesses the unobserved block of size `10` via the guessed address `0x100` at line 4 while the others do not reach line 4 thereby raising no UB. Therefore, with this simpler semantics the compiler would (annoyingly) have to conclude that line 4 can access object `a`.

4.3.12 Sometimes Two Blocks Are Not Enough

Justifying the correctness of some optimizations requires more than two blocks. We give an example of an optimization that requires triple allocation ($N = 2$ in our semantics). This optimization removes single observations of addresses of local variables through comparison with a function argument. For example, in the following function on the left, if the address of `c` is unobserved except at line 3, the comparison can be folded to `false` to get the optimized code on the right. This enables further optimizations since the address of `c` becomes completely unobserved in the target code and thus the compiler can assume that the block `c` cannot be accessed via a guessed address.

```
1: void foo(int i) {                    void foo(int i) {
2:   char c[4];                           char c[4];
3:   if (c == (char*)i) {                 if (false) {
4:     ...                                  ...
5:   } else {                  ⇒          } else {
6:     *(char*)0x200 = 0;                   *(char*)0x200 = 0;
7:   }                                    }
8:   ...                                  ...
9: }                                    }
```

The reason why two blocks are not enough for this optimization is that `c` can be accessed in line 6 of the original code when memory has only space for two blocks: The observation of the address of `c` in line 3 can forbid one of the twin executions from reaching line 6, which does not happen in the optimized code. To see this clearly, suppose that `i = 0x100` and the twin blocks for `c` can be only allocated at `0x100` and `0x200` because the memory had space for just those blocks. Then, in the original code, none of the twin executions triggers UB for the guessed access at line 6, while in the optimized code, the execution with `c = 0x100` triggers UB at line 6 now that guessing is prevented, as usual, by the twin allocations.

However, if we have triple twin allocations, `c` cannot be accessed via a guessed address even if one of the triple twin executions is ruled out by a single observation: We still have two twin executions left. For example, in the above setting, if `c` is allocated at `0x100`, `0x200` and `0x300`, in the original code the execution with `c = 0x300` triggers UB for the guessed access at line 6.

We can argue as follows that triple twin allocations enable the optimization above for a fresh block `c` when the address of `c` is observed only once in the comparison. If one of the triple executions in the optimized code accesses `c` via a guessed address (causing UB in the optimized code), we can always trigger UB for the guessed access in one of the three possible executions of the original code as discussed above. If not, the triple executions behave exactly the same in the optimized code because there is no observation of the address of `c`, and the behavior is simulated by one of the triple executions of the original code that makes the comparison `c == p` false.

Finally, we note that over-approximating the number of twin blocks is sound. However, as we have seen in the previous example, under-approximating it is not. In practice, due to the limited reasoning power of compilers in general, and LLVM in particular, we believe 3 blocks are sufficient. [7]

4.4 Prototype Implementation

We implemented two prototype versions of LLVM to study the impact of adapting the compiler to match the new semantics. First, we removed optimizations that are invalid in our semantics in order to create a version of LLVM that soundly implements its memory model. We confirmed this version fixes all the related bugs we have found. Second, we modified the "sound" compiler to regain some performance by adding optimizations that were not previously supported. We implemented our prototypes using LLVM 6.0.[8]

Making LLVM sound. LLVM propagated pointer equalities in several places; these had to be disabled. For example, InstSimplify (a peephole optimizer that does not create new instructions) transforms the IR equivalent of `(x == y) ? x : y` into `y`. In our semantics this transformation is correct for integers, but

[7]We are only aware of one optimization in LLVM and GCC that may require more than three blocks, and further investigation is ongoing to determine whether it is valid (see `http://llvm.org/PR35102`).

[8]`https://github.com/snu-sf/llvm-twin` and `https://github.com/snu-sf/clang-twin`

not for pointers, because it breaks data-flow provenance tracking. Similarly, we turned off GVN for pointer-typed variables.

A second kind of transformation that we had to disable were some integer-pointer conversions. LLVM (and GCC to some extent) treat a round-trip of pointer-to-integer and then integer-to-pointer casts as a no-op. This is incorrect under our semantics, and we removed this transformation—the IR equivalent of turning `(int*)(int)p` into `p`—in InstSimplify. Also related is a transformation from InstCombine (a peephole optimizer that potentially creates new instructions), which rewrites `(int)p == (int)q` into `p == q` for pointers `p` and `q`. This transformation was also removed, since comparing physical pointers is not equivalent to comparing logical pointers in our semantics.

A third kind of change is related to compiler-introduced type punning. There are optimizations in LLVM that transform, for example, load/store instructions of pointers into that of pointer-sized integers. This includes GVN when equivalent variables have different types, transforming a small `memcpy` into a pair of load/store instructions (e.g., where the intermediate value is an integer, and the target type is a pointer), etc. As we will discuss in Section 4.7, we had to disable all of these.

Finally, we disabled a few additional transformations that became invalid in our semantics including folding a pointer comparison against a stack-allocated object within a loop and a handful of peephole optimizations that changed pointer provenance. In total, we changed 419 lines of code.

Regaining performance. Making LLVM sound with respect to our memory model caused some small performance regressions in SPEC CPU 2017. We created a second prototype that attempts to regain that performance by adding optimizations that conform to our new semantics.

We changed how LLVM/Clang handles C/C++ pointer subtractions. The unmodified Clang lowers pointer subtraction into the IR equivalent of `(int)p - (int)q`. This is sound under our model, but since LLVM is very conservative when it encounters these kind of casts (it assumes the pointers escape), we unnecessarily lose precision in, e.g., alias analysis. The problem becomes even worse in our "sound" prototype. We added the **psub** instruction for pointer subtraction (described in Section 4.3.7) and modified Clang to use it. Since in our semantics **psub** does not escape if both operands are logical pointers, more aggressive optimization are now allowed.

We augmented alias analysis so that it becomes more precise when it encounters integer-to-pointer casts. Alias analysis can now conclude that a pointer cast from integer never aliases an unescaped object. Moreover, we also improved the analysis to assume that the pointer comparison instruction does not make its operands escape if they are both logical pointers.

Finally, we re-enabled GVN for pointer types for the following specific cases where it is guaranteed to be sound (for pointers `p` and `q` in the same equivalence class, where `p` is replaced with `q`):

- `q` is `NULL` or the result of an integer-to-pointer cast.
- `p` and `q` are logical pointers, and both are either dereferenceable or point to the same block.
- `p` and `q` are both computed by the **gep inbounds** with same base pointer.
- Either `p` or `q` is computed by a series of **gep inbounds** with positive offsets, based on the same base pointer.

The number of lines changed in this prototype against the previous one totals about 1274 lines. Overall, our "sound and fast" prototype requires changing less than 1700 lines of code in LLVM to make it sound against our memory model with minor impact on run time performance.

4.5 Performance Evaluation

We show that our memory model does not interfere with the quality of generated code.

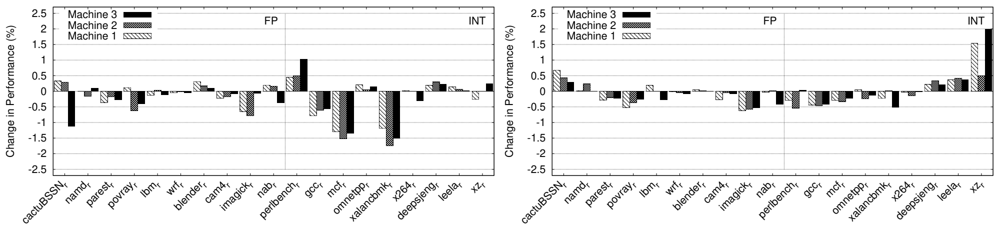
4.5.1 Setup

We used three machines, with the following CPUs, to measure performance: (1) Intel i3-6100 (Skylake), (2) Intel i5-6600 (Skylake), and (3) Intel i7-7700 (Kaby Lake). All machines have 8 GB RAM and Ubuntu 16.04 installed. To get more consistent results, Intel Hyperthreading, TurboBoost, SpeedStep, and Speed Shift were disabled, and the CPU scaling governor was set to "performance." We also disabled non-essential system services and address space layout randomization.

As benchmarks, we used SPEC CPU 2017 and the LLVM Nightly Test suite (293 C/C++ benchmarks). For SPEC CPU, we used only the SPECrate benchmarks because of RAM limitations (SPECspeed requires more memory than our machines had). The Fortran benchmarks were compiled first with `gfortran`.

Each benchmark was compiled with `-O3`, executed three times, and the median running time is used to compare performance. To run LLVM Nightly Tests, we used the `cpuset` utility to pin the benchmark to a single core. Unlike SPEC, these short-running benchmarks were seeing significant variance due to migrations between cores. Also, we used a RAM disk to minimize fluctuations due to disk access time. Some benchmarks in the LLVM suite have short running time, so using this setting is helpful to stabilize the results.

When comparing performance for LLVM Nightly Tests, we exclude bench-



(a) After making LLVM sound (b) Sound prototype + optimizations

Figure 4.8: Change in run time of SPEC CPU 2017 for our two prototypes. Higher bars indicate speedup relative to baseline LLVM, lower bars indicate slowdown.

marks where our changes to Clang and LLVM did not result in any changes to the resulting object code. Some benchmarks had high performance variation ($> 5\%$) across runs, most of these were benchmarks that run too quickly for accurate timing. We excluded these from our results as well.

4.5.2 Performance of Generated Code

Performance after making LLVM sound. Fig. 4.8a shows the effect that our first (sound but unoptimized) prototype had on the performance of SPEC CPU 2017. On average, across all benchmarks and machines, we observed a 0.2% slowdown. The worst slowdown was a 1.75% regression in xalancbmk on machine 2. There was also a $0.5 \sim 1.5\%$ consistent slowdown in gcc, mcf, and xalancbmk.

For the LLVM Nightly Tests, the worst slowdown was 4.3%, and the best speedup was 4.0%. On average, we observed a 0.3% slowdown. One benchmark, Oscar, had a huge speedup (25%). This was because our fixes prevented loop vectorization from working in this case, which happened to make the benchmark run faster. This benchmark is omitted in the numbers mentioned before.

In the LLVM Nightly Tests, only 25% (464/1853) of object files were different than when compiled with the baseline compiler, and only 27% (80/293)of benchmarks had any object file changed.

Across all of our benchmarks, the baseline version of LLVM's GVN pass performs about 24,700 replacements. About 28% of these correspond to propagation of pointer equalities, with the remaining being propagation of integer equalities. Of the pointer replacements, 44% correspond to replacing a pointer with `NULL`, which is a sound transformation under our semantics (even though it was disabled in this prototype).

Performance after adding optimizations. Fig. 4.8b shows the change in performance of SPEC CPU 2017 for our second prototype ("sound and fast") relative to baseline LLVM. The three changes that contributed the most to recovering lost performance were as follows. Adding **psub** changed the average slowdown on SPEC from -0.2% to -0.1%. The main winners due to this optimization were mcf and xalancbmk, having a 1.7% and 1.1% speedup, respectively.

We found "jpeg-6.a" in LLVM Nightly Tests showed 2.2% slowdown. It was because removing transformations from load/store of pointers into load/store of pointer-sized integers blocked SLP vectorization of heterogeneous types. We can support this by allowing type punning between a physical pointer and integer. Formally, we can define load of integer i as pointer to yield $\mathsf{Phy}(i, 0, \emptyset, \textbf{None})$ rather than **poison**, and load of $\mathsf{Phy}(i, 0, \emptyset, \textbf{None})$ as integer to return i. With this update, it is valid to canonicalize load/store of pointer-sized integers into load/-store of pointers. After the canonicalization, SLP vectorization can fire because load/stores now have homogeneous types. We implemented this canonicalization pass before SLP vectorization and checked that the slowdown disappeared. It did not, however, improve the performance of the SPEC benchmarks nor the average slowdown of the LLVM Nightly Tests. This canonicalization can be a

Table 4.2: Number of instructions with different prototypes

		Baseline	**i2p**(**p2i** *p*) fold disabled	Sound	add **psub**
`-O0`	Total	21,219,982	21,219,982	21,219,982	21,194,610
	inttoptr	1,554	1,554	1,554	1,554
	ptrtoint	39,726	39,726	39,726	14,338
	psub	0	0	0	12,806
`-O3`	Total	15,015,064	15,030,020	14,950,405	14,750,101
	inttoptr	29,976	44,255	2,580	2,573
	ptrtoint	38,249	45,425	64,585	9,431
	psub	0	0	0	29,117

good workaround for reenabling vectorization, but in the future we would like to pursue more generalized solution for this, as discussed in Section 4.7.

Finally, reactivating GVN for safe cases removed the slowdown in perlbench and cactuBSSN. With the Nightly Tests, the performance regressed slightly, increasing the average slowdown to 0.07%.

This experiment shows that it is possible to make LLVM adhere to our memory model without significant performance regression. In fact, some benchmarks get a small performance improvement.

4.5.3 Number of Instructions

We investigated how the number of pointer cast instructions (**inttoptr** / **ptrtoint**) and **psub** instructions evolved with our modifications. Table 4.2 shows the total number of instructions over all the benchmarks, when compiled without optimizations (`-O0`) and with optimizations (`-O3`).

In the first column, we have the numbers for the baseline (vanilla LLVM). In the second column, we can observe the number of casts does not increase significantly after disabling the (incorrect) optimization that removes the round-trip cast of `(int*)(int)p` to `p`, for a pointer `p`.

The third column shows the result for our first prototype, where all incorrect optimizations were removed. The number of **inttoptr** instructions is reduced because we disabled the introduction of type punning by the optimizer. The increase in the number of **ptrtoint** casts is the result of removing unsound optimizations like the one that rewrites `(int)p == (int)q` into `p == q`.

In the last column, we show the data for a prototype that uses the new **psub** instruction. Not only are the number of casts reduced in the non-optimized build (since we changed Clang to lower pointer subtraction into **psub**), but the optimized build also shows a reduction since we also added support for the optimizers to create **psub** when needed.

Finally, we note the number of cast instructions is still larger in the optimized build since optimizations like loop unrolling and inlining can soundly duplicate instructions.

4.5.4 Compile Time

We measured the time it takes to compile each file of all the benchmarks in an optimized build. We exclude files with a very small compile time ($< 0.15\,\mathrm{s}$) in these results.

In the first prototype, the average compile time had a 0.1% slowdown. The worst slowdown was 10.5% and the best speedup 13.9%. For the second prototype, we observed an average slowdown of compilation time of 1.1%. The worst slowdown increased to 9.3%. A reason for the second prototype regressing further on the compilation time is due to extra analysis required to propagate pointer equalities. Also, our prototype implementations were not optimized to reduce compilation time.

	high-level mem optimizations	int-to-ptr casts	low-level ptr manipulation	all integer optimizations	ptr arith is pure
Flat		✓	✓	✓	✓
Memarian et al.	✓	✓			✓
CompCert	✓			✓	✓
Besson et al.	✓	✓			✓
Kang et al.	✓	†	✓	✓	
Ours	✓	✓	✓	✓	✓

Table 4.3: Comparison between different memory models: whether they enable high-level memory optimizations (such as store forwarding), whether they support integer-to-pointer casts, whether they support languages with low-level pointer manipulation (such as doing XORs of pointers), whether they support all standard integer optimizations (such as equality propagation and reassociation), and whether pointer arithmetic and cast operations are pure (since otherwise they constrain movement of such instructions). † does not support cast of $p + n$.

4.6 Related Work

Much work has been done on defining and clarifying the semantics of C, including the semantics of pointer provenance [2–4, 92–95]. However, focusing on source language semantics involves different considerations than that of a compiler IR. For example, many of the optimizations presented in this chapter are not sound under the proposed C semantics. That is not a problem, as long as C can efficiently be compiled to an IR like LLVM's that allows the desired optimizations.

CompCert [27, 96] is a compiler for a subset of C that is formalized in Coq. It defines a pointer as a pair of block id and offset, which is equivalent to our logical pointer. CompCert does not, however, support pointer-to-integer casts, thereby avoiding the problems we address in this chapter. [97] extend CompCert with a weak memory model.

Vellvm [24] formalizes the LLVM IR in Coq. It adopts CompCert's memory model, inheriting its weaknesses.

Besson et al. [98–101] propose an extension of the CompCert memory model supporting casts between integers and pointers. When a pointer is cast to an integer, a fresh symbol is created to represent the underlying physical address and henceforth keep the result of arithmetic or bitwise operations over it as symbolic expressions. When a concrete value is needed for a symbolic expression, if it uniquely determines an integer value, then that value is returned; otherwise, poison is returned. They verify CompCert optimizations and programs performing bitwise operations on pointers with their memory model. However, their model is limited in practice because observing the underlying address of a pointer is not allowed. For example, using a pointer value as a hash key is undefined in their model.

[5] proposed a low-level memory model that supports pointer-to-integer casts. However, the model does not support casting a pointer one-past-the-end of an object to an integer and back: this round-trip resulted in **poison** in their semantics, while it is well-defined in C. Moreover, in their semantics, `ptrtoint` is an effectful operation that cannot be removed even if the result is not used, and `inttoptr` is not freely movable across `malloc` and `free`. Our work fixes both of these problems by combining logical and physical pointers in one memory model.

4.7 Discussion and Future Work

Type Punning In our semantics, loading, e.g., from a memory location that contains a pointer at an integer type results in **poison** due to type punning. One may wonder why we did not specify the load to, instead, perform an implicit pointer-to-integer cast. Disallowing implicit pointer casts is necessary to justify load-store elimination:

```
int q, **x = malloc(sizeof(int*));
*x = &q;
```

```
int i = *(int*)x // implicit pointer-to-integer cast
*(int*)x = i;    // remove; it is redundant
```

So, while the original code performed a pointer-to-integer cast and then wrote back an integer (or, equivalently, a physical pointer with no provenance information), the changed program leaves the original provenance information intact. In other words, load-store elimination can result in increasing provenance restrictions on pointers, which can in turn result in introducing UB. To fix this, we let load return **poison** instead of performing an implicit cast.

The downside of disallowing implicit casts is that there is no longer a type that can actually hold any data that can be present in memory. LLVM typically uses integer types for that purpose but, as discussed above, that does not work in our semantics. One option to fix this is adding a new "byte" type that only permits a few bitwise operations, and that can hence hold both pointer and integer bits. We intend to pursue this avenue further in the future.

Non-deterministic Values We have defined some operations to yield a non-deterministic result, e.g., **icmp ule** of pointers of different blocks. This blocks duplication of instructions since they could then return different results. Solutions to this problem includes introducing a notion of "entangled" instructions that need to be folded together. We leave this exploration for future work.

Out of memory A technical problem with our memory model is that any transformation that may eliminate an out-of-memory condition is unsound. However, the necessary conditions for achieving miscompilation via this unsoundness are difficult to meet. First, the compiler must perform a transformation based on knowledge that a program path triggers out-of-memory (LLVM does not do this at present, and seems unlikely to do so in the future). Second, it would need to perform an optimization eliminating the out-of-memory condition (these transformations, such as eliminating unused allocations, are performed routinely). Designing a memory model that is sound in this respect seems challenging and it is not clear the result would be useful in practice.

As an example, consider this function:

```
unsigned int oom(unsigned int n) {
  int a[128];
  return n == 0 ? 0 : oom(n-1) + (unsigned int)&a;
}
```

Since the LLVM compiler analyzes the function as `readnone` (i.e., not accessing memory), it optimizes away `oom(n)` if its return value is unused. For instance, `(oom(UINT_MAX); print(0))` is optimized to `print(0)`, which is unsound in our model because out-of-memory behavior of the source is eliminated in the target. In theory, allowing such optimizations can be problematic because `print(1)` can be translated to `(oom(UINT_MAX); print(1))` by some code motion, which is equivalent to `(oom(UINT_MAX); print(0))` because both behaves exactly the same (i.e., always producing out-of-memory), which can be translated to `print(0)` by eliminating `oom(UINT_MAX)`. Thus, `print(1)` can be translated to `print(0)`, which is clearly a problem. However, in practice, compilers never recognize equivalence between `(oom(UINT_MAX); print(1))` and `(oom(UINT_MAX); print(0))` because they never perform out-of-memory analysis.

Finding a semantic model that allows translating `(oom(UINT_MAX); print(0))` to `print(0)`, but not `(oom(UINT_MAX); print(1))` to `(oom(UINT_MAX); print(0))` is a challenging problem.

4.8 Implementing Our Memory Model in LLVM

In this section, we describe the efforts we made to carry our memory model to LLVM. We delivered our work to the LLVM community by (1) giving a talk at EuroLLVM'18, (2) leading and actively participating in online discussions, and (3) writing patches and sending them to the LLVM project.

4.8.1 Removing Provenance-Changing Transformations

Our proposal motivated developers to remove a few transformations that are unsound because they change pointer provenance. LLVM InstCombine and InstSimplify were incorrectly folding '`gep p, (q-p)`' to '`q`'. The transformations were finally removed by Rust developers[9].

To prevent unsound replacement of a pointer using equality comparison, `canReplacePointersIfEqual` is added to LLVM's value analysis (`ValueTracking.h`). Given two pointers, this function returns true if it is sound to replace a pointer with the other one. However, the function is not used by optimizers due to a possible performance regression. To allow pointer replacements in general, a new provenance-laundering operation is necessary. Interestingly, a proposal for C's `restrict` keyword support in clang contains an operation for this. The operation can be used after the patch is landed in LLVM.

4.8.2 Safely Optimizing Casts between Pointers and Integers

We delivered our idea about folding `inttoptr(ptrtoint p)` to LLVM developers via our EuroLLVM'18 talk and online discussions. It motivated compiler developers to write a few patches in LLVM.

First of all, `clang-tidy` is updated so that it detects integer-to-pointer

[9] `https://reviews.llvm.org/D98588` and `https://reviews.llvm.org/D98611`. `https://reviews.llvm.org/D93820` has a link to our paper.

casts and warns that LLVM may not optimize this code well[10]. It also suggests its alternative form. For example, instead of writing a pointer masking operation as '`(char*)(ofs | (intptr_t)p)`', `clang-tidy` suggests using '`p + (((intptr_t)p | ofs) - (intptr_t)p)`' because the alternative form preserves pointer provenance. Also, to succinctly represent a pointer masking expression, a new `llvm.ptrmask` instrinsic function is added[11]. Its semantics is equivalent to the above alternative form.

In this summer, we are working on writing patches for the removal of the pointer cast roundtrip folding. A student who participated as a mentee in Google Summer of Code[12] added a new '`-disable-i2p-p2i-opt`' flag to the official LLVM implementation[13]. If the commandline flag is given, the pointer cast roundtrip folding is disabled. She also added optimizations to InstCombine in order to help reduce performance regressions after the flag is enabled.

4.8.3 Type Punning

A few transformations incorrectly performing type punning of loads and stores are finally removed by compiler developers. InstCombine's type punning transformation, which was the source of the majority of `inttoptr` casts, was finally removed[14]. Similar transformations in other optimizations were removed as well.

In this summer, we are working with a student who is interested in implementing the 'byte' type (Section 4.7). He participated as a mentee in Google Summer of Code and is working on implementing its prototype.

[10] `https://reviews.llvm.org/D91055`
[11] `https://reviews.llvm.org/D59065`
[12] As described in Section 3.7, I and Nuno worked as mentors of 3 Google Summer of Code projects for fixing miscompilations in LLVM.
[13] `https://reviews.llvm.org/D105771`
[14] `https://reviews.llvm.org/D88789`

4.9 Conclusion

Languages like C, C++, and Rust give the programmer low-level control over how memory is arranged and accessed while also giving the compiler freedom to perform high-level memory optimizations. This is challenging and current toolchains based on LLVM and GCC are not always able to strike the right balance between performance of generated code and strength of guarantees made to programs. Part of the problem is that the memory models for the internal representations in these compilers—where high-level optimizations are performed—are informally specified and thus easily misunderstood.

We created a new memory model for LLVM IR that we believe provides sufficiently strong guarantees that it can be efficiently targeted by front-ends while also permitting many desirable optimizations. We have implemented a prototype of this model and have shown that the implementation is not invasive, that code quality remains good, and that several known miscompilations due to the current, informal memory model are fixed.

Chapter 5

Validating the IR Semantics

Ensuring that LLVM is correct is crucial for the safety and reliability of the software ecosystem. There has been significant work towards this goal including, e.g., formally specifying the semantics of the LLVM IR (intermediate representation). This entails describing precisely what each instruction does and how it handles special cases such as integer overflows, division by zero, or dereferencing out-of-bounds pointers [12, 13, 24, 25, 102]. There has also been work on automatic verification of classes of optimizations, such as peephole optimizations [60, 61], semi-automated proofs [103], translation validation [69, 76, 77, 104], and fuzzing [8, 9]. All this work uncovered several hundred bugs in LLVM.

Alive2 is the first fully automatic bounded translation validation tool for LLVM. It relies on our formal semantics of undefined behavior and memory model. Alive2 supports all of its forms of undefined behavior (UB). Handling UB is important because LLVM's optimizers frequently take advantage of it. UB is heavily used by frontends to communicate invariants about the code to optimizers. Therefore, in practice, any optimization verification tool that targets

LLVM (or any other modern compiler) needs to support UB; otherwise the number of false alarms would make the tool impractical.

Our goal is a zero false-alarm rate, so we err on the side of soundness. We make use of *bounded* translation validation to bound the resources used by each verification task. For example, we unroll loops up to a given bound, and we limit input variables to be either fully or never undef, but not partially.

We also present the first SMT encoding of our LLVM's memory model[1]. It is precise enough to validate LLVM's intraprocedural memory optimizations. The design of the encoding was guided by practical insights of the common aliasing cases in BTV to achieve better performance. For example, we observed that in most cases we can cheaply infer whether a pointer aliases with a locally-allocated or a global object (but not both). Therefore, our encoding case-splits itself on this property rather than leaving that to the SMT solver, as we can cheaply resolve the case split for over 95% of the cases.

Another contribution of our memory model encoding is a new semantics for heap allocation for the verification of optimizations for real-world C/C++ programs. Although our memory model has a reasonable semantics for heap allocations, we realized it was not suitable for verifying optimizations. In some programming styles, the result of functions such as `malloc` is not checked against `NULL` and the resulting pointer is dereferenced right away. Since `malloc` can return `NULL` in some executions, we could end up proving that some undesirable optimizations were correct since the program triggers undefined behavior in at least one execution. We propose a new semantics for heap allocations in this chapter that is better suited for the verification of optimizations.

Alive2 has emphasized transitioning formal methods tools and their results

[1]Currently, our encoding deals with logical pointers only. Finding an efficient encoding for physical pointers is ongoing work.

into the LLVM community, including:

- Creating several Alive2-based tools such as plugins for `opt` (LLVM's standalone IR optimizer) and Clang (LLVM's C/C++ frontend) that can be used to check refinement after every optimization pass, and a refinement checker for pairs of files containing LLVM IR.
- Since summer of 2019, continuously monitoring the LLVM unit test suite using Alive2, resulting in 54 bug reports, of which 28 have been fixed.
- Contributing 7 bug fixes to LLVM.
- Contributing 11 patches to LLVM IR's specification document and leading the discussion for fixing several other inconsistencies in the semantics.
- Running larger-scale experiments doing translation validation while compiling small applications.
- Engaging with the LLVM community on mailing lists and at their annual developers' meeting, where we have presented three talks about our work.
- Releasing Alive2 as open-source software (`https://github.com/AliveToolkit/alive2`) and also making it available through a web site (`https://alive2.llvm.org/`), obviating the need for most developers to compile our tools.

5.1 Overview

Consider the functions below in `C`:[2] a source (original) function on the left and a target (optimized) function on the right. According to the semantics of high-level languages, and also of LLVM IR, a pointer received as argument or a callee

[2] We use the syntax of `C` for many of the examples in this chapter to make them easier to read, even though we consider the semantics of LLVM IR.

cannot guess the address of a memory region allocated within a function. That is, pointer `q` is not aliased with `p`, `r`, nor touched by `g(p+1)`. Although the caller of `f` may guess the address of `q` in practice, that behavior is excluded by the language semantics because `p`'s object (*provenance*) cannot be a fresh one like `q`. If `p` happens to alias `q`, accessing such pointer triggers UB.

```
1: int f(int *p) {               | 1': int f(int *p) {
2:   int *q = malloc(4);         | 2':   // q removed
3:   *q = 42;                    | 3':
4:   int *r = g(p+1);            | 4':   int *r = g(p+1);
5:   *r = 37;                    | 5':   *r = 37;
6:   return *q;                  | 6':   return 42;
7: }                             | 7': }
```

The provenance rules allow LLVM to forward the stored value in line 3 to line 6, and therefore line 6′ simply returns 42. As the value stored to `*q` is not used anymore and pointer `q` does not escape, LLVM also removes the heap allocation.

Next we show how to verify this example. Note that we do not require the two programs to be aligned; the example is aligned to make it easier to understand.

5.1.1 Verifying The Example Transformation

We start by defining two auxiliary functions that encode the effect of memory operations on the program state. Let state $S = (m, ub)$ be a pair, where m is a memory and ub a boolean that tracks whether the program has already executed UB or not. Let p be the accessed pointer, and v the stored value. The definition of functions $\overline{\textbf{load}}$ and $\overline{\textbf{store}}$ is as follows:

$$\begin{aligned} \overline{\textbf{load}}\ p\ S\ &::= (\ \textbf{load}(p, S.\mathsf{m})\ ,\ (S.\mathsf{m},\ S.\mathsf{ub} \lor \neg\ \textbf{deref}(p, \texttt{sizeof}(*p), S.\mathsf{m})\ ))\\ \overline{\textbf{store}}\ p\ v\ S\ &::= (\ \textbf{store}(p, v, S.\mathsf{m})\ ,\ S.\mathsf{ub} \lor \neg\ \textbf{deref}(p, \texttt{sizeof}(*p), S.\mathsf{m})\ ) \end{aligned}$$

$\overline{\textbf{load}}$ returns a pair with the loaded value and the updated state, where ub

#	**Inputs:** p, m_0, ub_0	#	**Inputs:** p', m'_0, ub'_0
2	$S_1 := (m_0, ub_0)$ $A_1 :=$ q is fresh	$2'$	-
3	$S_2 := \overline{\textbf{store}}\ q\ 42\ S_1$	$3'$	-
4	$S_3 := (m_\text{g}, S_2.\texttt{ub} \lor ub_\text{g})$ $A_2 :=$ r is not aliased with q $\land$ m_g agrees with $S_2.\texttt{m}$ on q	$4'$	$S'_1 := (m'_\text{g}, ub'_0 \lor ub'_\text{g})$
5	$S_4 := \overline{\textbf{store}}\ r\ 37\ S_3$	$5'$	$S'_2 := \overline{\textbf{store}}\ r'\ 37\ S'_1$
6	$\text{O} := \overline{\textbf{load}}\ q\ S_4$	$6'$	$\text{O}' := (42, S'_2)$

Table 5.1: States and axioms after executing each of the lines of `f`.

is further constrained to ensure that pointer p is dereferenceable for at least the size of the loaded type. Similarly, $\overline{\textbf{store}}$ returns the updated state. The gray boxes ($\cdots$) encode SMT expressions; we describe these in the next section.

1. Encoding the output states. Table 5.1 shows the state after executing each of the programs' lines. p, m_0, and ub_0 are SMT variables for the input pointer, and function `f` caller's memory and UB flag, respectively. The target's corresponding variables are primed. Meta variables are upper-cased and SMT variables are lower-cased.

On line 2, q is assigned a pointer to a new object (encoded in axiom A_1). On line 3, '`*q = 42`' updates the state using $\overline{\textbf{store}}$.

On line 4, the return value, output memory, and UB of `g(p+1)` are represented with fresh variables r, m_g, and ub_g, respectively. Axiom A_2 encodes the provenance rules: the return value cannot alias with locally non-escaped pointers (`q`) and only the remaining objects are modified. Line $4'$ does not need these axioms because there are no locally-allocated objects in the target function.

Finally, the outputs O and O' are a pair of return value and state.

2. Relating the source and target's states. To prove correctness of a transformation, we must first establish refinement between the input states of the source/target functions. Refinement ($\sqsupseteq$) is used rather than equality because it is allowed for the source's caller to give less defined inputs than the target's.

$$A_{\text{in}} := \boxed{p \sqsupseteq p'} \wedge \boxed{m_0 \sqsupseteq m'_0} \wedge (ub'_0 \implies ub_0)$$

The inputs and outputs of function calls are also related using refinement. For any pair of calls in the source and target functions, if the target's inputs refine those of the source, the target's output also refines the source's output. The example only has one function call pair:

$$A_{\text{call}} := \Big(\boxed{S_2.\mathsf{m} \sqsupseteq m'_0} \wedge \boxed{p+1 \sqsupseteq p'+1} \implies \boxed{m_{\text{g}} \sqsupseteq m'_{\text{g}}} \wedge \boxed{r \sqsupseteq r'} \wedge (ub'_{\text{g}} \implies ub_{\text{g}}) \Big)$$

We can now state the correctness theorem for the example transformation. For any input, if the axioms hold, the output of the target must refine that of the source for some internal nondeterminism in the source (e.g., the address of pointer q). Output is refined iff (*i*) the source triggers UB, or (*ii*) the target triggers no UB, and the target's return value and memory refine those of the source.

$$\forall p, p', m_0, m'_0, ub_0, ub'_0, m_{\text{g}}, m'_{\text{g}}, ub_{\text{g}}, ub'_{\text{g}} \,. \quad \exists q \,. \, (A_1 \wedge A_2 \wedge A_{\text{in}} \wedge A_{\text{call}}) \implies O \sqsupseteq O'$$

5.1.2 Efficiently Encoding LLVM's Memory Model and Refinement

We now present our key ideas for efficiently encoding LLVM's memory model and refinement (the gray boxes) in SMT, which is one of our main contributions.

1. Pointers. We represent a pointer as a pair (bid, o) of a block id (i.e., its provenance) and an offset within, so that we can easily detect out-of-bound accesses: accessing (bid, o) in memory m triggers UB unless $0 \le o < m[bid]$.size, from which $\boxed{\mathbf{deref}((bid, o), sz, m)}$ naturally follows.

2. Bounding the number of blocks. Our first observation is that we can safely bound the number of memory blocks for *bounded* translation validation since loops are unrolled for a fixed number of iterations. As a result, we can use a (fixed-length) bit-vector to encode block ids.

For the example source function, four blocks are sufficient: three for pointers `p`, `q`, `r` as they may all point to different blocks, and an extra to represent all the other blocks that are not syntactically present but are accessible by function `g`.

For the sake of simplifying the example, we ignore that `p`, `q`, `r` may be **null**. Our model does not make such assumption; we explain later how null is handled.

3. Aliasing rules. Several of the aliasing rules are encoded for free as we can distinguish most blocks by construction. First, we use the most significant bit of the block ids to distinguish local (1) from non-local (0) blocks. Second, we assign constant ids whenever possible (e.g., global variables and stack allocations).

For the example source function, (without loss of generality) we set the block ids of `q`, `p` and the extra block to $100_{(2)}$, $000_{(2)}$, and $011_{(2)}$ (in binary format), respectively. However, we cannot fix the block id of `r` and instead give the constraint that it should be either $000_{(2)}$ or $001_{(2)}$ since `r` may alias with `p` but not with `q`. This establishes the alias constraints in A_1 and A_2 for free.

4. Memory accesses. In order to leverage the fact that each pointer may range over a small number of blocks as seen above, we use one SMT array per block (from an offset to a byte) instead of using a single global array (from a pointer to a byte). For the latter, it becomes harder to exploit non-aliasing guarantees since all stores to different blocks are grouped together.

For the example source function, m_0 consists of four arrays $m_0^{(100)}$, $m_0^{(000)}$, $m_0^{(001)}$, $m_0^{(011)}$ for the four blocks. Then since q's block id is $100_{(2)}$, $\overline{\textbf{store}}\ q\ 42\ S_1$ at line 3 only updates the array $m_0^{(100)}$, leaving the others unchanged. Similarly, $\overline{\textbf{store}}\ r\ 2\ S_3$ at line 5 only updates $m_0^{(000)}$ and $m_0^{(001)}$ using the SMT if-then-else expression on r's block id. Finally, $\overline{\textbf{load}}\ q\ S_4$ at line 6 reads from the updated array at $100_{(2)}$, thereby easily realizing that the read value is 42.

5. Refinement. The value/memory refinement $\sqsupseteq$ is defined based on a map-

$$\begin{array}{lll}
\text{Num}(sz) & ::= & \{\, i \mid 0 \leq i < 2^{sz} \,\} \\
\text{BlockID} & ::= & \mathbb{N} \\
\text{Offset} & ::= & \text{Num}(64) \\
\text{PtrAttr} & ::= & \{\texttt{nocapture}, \texttt{readonly}, \texttt{readnone}\} \\
\text{Pointer} & ::= & \text{BlockID} \times \text{Offset} \times 2^{\text{PtrAttr}} \\
\text{DefinedValue} & ::= & \text{Int} \uplus \text{Pointer} \uplus \text{Float} \\
\text{Value} & ::= & \mathcal{P}(\text{DefinedValue}) \uplus \{\, \textbf{poison} \,\} \uplus \text{Aggregate} \\
\text{Aggregate} & ::= & \text{list Value} \\
\text{Memory} & ::= & \text{BlockID} \rightarrow \text{MemBlock} \\
\text{RegFile} & ::= & \text{string} \rightarrow \text{Value} \\
\text{State} & ::= & \text{RegFile} \times \text{Memory} \times \text{bool} \\
\text{ValueNoRet} & ::= & \text{Value} \uplus \{\, \text{noreturn} \,\} \\
\text{FinalState} & ::= & \text{ValueNoRet} \times \text{Memory} \times \text{bool}
\end{array}$$

Figure 5.1: Definitions of important sets. $\mathcal{P}$ is the power set operation. The full definitions are at Fig. 5.5.

ping between source and target blocks, which we efficiently encode leveraging the alignment information between source and target as much as possible (Section 5.6).

5.2 Encoding LLVM IR Semantics in SMT

In this section we explain how Alive2 encodes the state of an LLVM IR function and how the semantics of the IR are specified. Fig. 5.1 defines values, the register file, and the program state. A program state consists of a register file, a memory, and a flag stating whether the program has executed UB. The program state is updated after the execution of each instruction.

A register file assigns a valuation to each register. A value is either **poison** or a set of integer/floating-point numbers or pointers. The set of values is not a singleton if the value is **undef**. When a value is used, one of the elements of the set is picked non-deterministically. Memory is a map from block id to its properties, including the block's data, size, alignment, whether it is alive, etc. Each allocation gets a fresh block.

($\iota =$ "$r = \textbf{add } \mathbf{i}sz\ op_1,\ op_2$")

$$\textsc{add-poison}\quad \frac{[\![op_1]\!]_R = \textbf{poison} \lor [\![op_2]\!]_R = \textbf{poison}}{\langle R, M, b\rangle \overset{\iota}{\hookrightarrow} \langle R[r \mapsto \textbf{poison}], M, b\rangle}$$

$$\textsc{add}\quad \frac{\begin{array}{c}[\![op_1]\!]_R = v_1 \quad [\![op_2]\!]_R = v_2 \quad v_1, v_2 \in \mathcal{P}(\text{Int}) \\ v' = \{\, (i_1 + i_2) \bmod 2^{sz} \mid i_1 \in v_1 \land i_2 \in v_2 \,\}\end{array}}{\langle R, M, b\rangle \overset{\iota}{\hookrightarrow} \langle R[r \mapsto v'], M, b\rangle}$$

($\iota =$ "$r = \textbf{add nuw } \mathbf{i}sz\ op_1,\ op_2$")

$$\textsc{add-nuw-overflow}\quad \frac{\begin{array}{c}[\![op_1]\!]_R = v_1 \quad [\![op_2]\!]_R = v_2 \quad v_1, v_2 \in \mathcal{P}(\text{Int}) \\ \exists i_1 \in v_1, i_2 \in v_2 \,.\, i_1 + i_2 \geq 2^{sz}\end{array}}{\langle R, M, b\rangle \overset{\iota}{\hookrightarrow} \langle R[r \mapsto \textbf{poison}], M, b\rangle}$$

($\iota =$ "$r = \textbf{udiv } \mathbf{i}sz\ op_1,\ op_2$")

$$\textsc{udiv-ub}\quad \frac{\begin{array}{c}[\![op_2]\!]_R = v_2 \\ v_2 = \textbf{poison} \lor 0 \in v_2\end{array}}{\langle R, M, b\rangle \overset{\iota}{\hookrightarrow} \langle R, M, \textbf{true}\rangle} \qquad \textsc{udiv-poison}\quad \frac{\begin{array}{c}[\![op_1]\!]_R = \textbf{poison} \quad [\![op_2]\!]_R = v_2 \\ v_2 \in \mathcal{P}(\text{Int}) \land 0 \notin v_2\end{array}}{\langle R, M, b\rangle \overset{\iota}{\hookrightarrow} \langle R[r \mapsto \textbf{poison}], M, b\rangle}$$

($\iota =$ "$r = \textbf{freeze } \mathbf{i}sz\ op$")

$$\textsc{freeze-poison}\quad \frac{\begin{array}{c}[\![op]\!]_R = \textbf{poison} \\ v \in \text{Num}(sz)\end{array}}{\langle R, M, b\rangle \overset{\iota}{\hookrightarrow} \langle R[r \mapsto \{v\}], M, b\rangle} \qquad \textsc{freeze-pick}\quad \frac{\begin{array}{c}[\![op]\!]_R = v \quad v' \in v \\ v \in \mathcal{P}(\text{DefinedValue})\end{array}}{\langle R, M, b\rangle \overset{\iota}{\hookrightarrow} \langle R[r \mapsto \{v'\}], M, b\rangle}$$

Figure 5.2: Semantics of selected instructions

We give the semantics for a few example instructions in Fig. 5.2. Let tuple $S = \langle R, M, b\rangle$ be a program state (resp. register file, memory, UB flag). The notation $S \overset{\iota}{\hookrightarrow} S'$ defines the resulting state S' of executing instruction ι on state S.

The final state is similar to a register valuation, but includes the symbol noreturn. In LLVM, functions can end with a call instruction to a function that does not return (e.g., `exit`). This is a code-size optimization as it enables the compiler to skip inserting code to cleanup the stack and return to the callee, for example.

5.2.1 Register File

For each program register in the register file, we maintain a pair of SMT expressions: $(\mathsf{value}, \mathsf{ispoison})$. The second element is a Boolean indicating whether the value is poison or not. The first element's value is only meaningful when the $\mathsf{ispoison}$ flag is false. The value is an SMT expression of appropriate type, depending on the program register's type. Integers are encoded with bit-vectors, while floats use SMT's FPA theory. Aggregates (arrays, vectors, and structs) are encoded by converting each element to a bit-vector and then concatenating them. Similarly, pointers are encoded with a bit-vector concatenation of the individual components (block id and offset).

5.2.2 Function Arguments

We assume function arguments can be arbitrary, and therefore these can be either **undef**, **poison**, or well-defined. We use four SMT variables to encode each function argument: two Booleans to indicate if the argument is **undef** or **poison**, a variable to hold the well-defined value, and a fresh quantified variable to represent all the values of a type for the **undef** case.

Putting it together, the encoding of a function argument `%a` is the pair: $(\mathsf{ite}(\mathsf{isundef}_{\%\mathsf{a}}, \mathsf{undef}_1, \%\mathsf{a}), \mathsf{ispoison}_{\%\mathsf{a}})$. For aggregates, we compute this expression element-wise, allowing for example each element to be **poison** or not independently.

Our encoding for **undef** values is an under-approximation, since we only allow an argument to be either fully **undef** or not **undef** at all. This disallows behaviors where, e.g., only one of the input bits is **undef** (like the result of "`and i32 undef, 1`"). Partial **undef** values spawn a doubly-exponential state space ($2^{2^n} - 1$ for each n-bit integer). By supporting only fully **undef** values, we reduce the complexity to a "mere" exponential state space. We believe the potential for missed bugs is small, and that this is a good tradeoff.

5.2.3 Undef Values

We have seen that **undef** can yield a different value each time it is observed. Therefore, we need to create a fresh variable for each **undef** each time a value is observed. We keep track of the undef SMT variables used for each expression in the register file. When we lookup a value in the register file, we rewrite all undef variables with fresh variables.

For example, assume that we have the following value in the register file for %a:

$$R[\%\mathsf{a}] = (\mathsf{ite}(\mathsf{isundef}_{\%\mathsf{a}}, \mathsf{undef}_1, \%\mathsf{a}), \mathsf{ispoison}_{\%\mathsf{a}})$$

Evaluating the instruction `%b = add %a, %a` yields the following expression for the value (ignoring the poison bit):

$$\mathsf{ite}(\mathsf{isundef}_{\%\mathsf{a}}, \mathsf{undef}_2, \%\mathsf{a}) + \mathsf{ite}(\mathsf{isundef}_{\%\mathsf{a}}, \mathsf{undef}_3, \%\mathsf{a})$$

Variables $\mathsf{undef}_\mathsf{i}$ are appropriately quantified so they can take any value of the type. We describe this process later in Section 5.6.

The **freeze** instruction stops propagation of both **undef** and **poison**. The only difference between **freeze undef** and **undef** is that the former evaluates to the same (arbitrary) value on every use. Therefore, we just need to clear the set of undef SMT variables in the register file such that the undef variables are not replaced with fresh ones on each lookup.

Detecting **undef** boils down to detecting if an expression can evaluate to more than one value. The straightforward way for checking this for an expression e with the set of undef variables v is to check if $\exists v, w \,.\, e \neq e[w/v]$ is satisfiable (with w being a set of fresh variables). To reduce the number of variables, we use an alternative encoding: we replace variables v with a constant, $e_k = e[k/v]$.

If the comparison $e = e_k$ is valid, then e is not **undef** since there is no model for the undef variables that makes e different than a base value (e_k). We tried the values 0 and 1 for this constant, but neither worked well because they are identity/absorbent for some arithmetic operations and are folded away by the solver. This tends to destroy syntactic similarity between e and e_k, confusing our SMT solver's quantifier instantiation algorithm. The constant 2 also does not work well: it is a power of 2 and thus simplifies, e.g., multiplications. Therefore, we replace **undef** values with 3 when creating e_k.

5.2.4 Control Flow

The flow of execution in LLVM is controlled using (conditional) branch, `switch`, function calls (Section 5.7), and exceptions (`invoke` instruction).

As LLVM's IR is in SSA form, merge of values from different paths through the control-flow graph (CFG) is already explicit through the `phi` instruction. We merge the multiple SMT expressions from the incoming paths of a basic block trivially using the `phi` instructions, ending up with a single SMT expression per register per function. We do not fork expressions across paths in the CFG.

5.2.5 Floating-Point Numbers

LLVM's floating-point (IEEE-754) instructions trivially map to SMT's FPA theory. A notable exception is LLVM's remainder operation (equivalent to C's `fmod`) which has different rounding behavior than SMT's (equivalent to C's `remainder`). Additionally, we do not support non-IEEE-754 types such as x86's 80-bit floats, as SMT's FPA theory does not support those either.

In IEEE-754, the bit representation of NaN is not unique, and different CPUs use different bit patterns in practice. This leaves us with the question of what should be the semantics of the `bitcast` instruction from float to integer (not to be confused with an arithmetic cast, where the float's value is truncated to an integer).

There are essentially two choices:

- `bitcast` from integer to float and then back to integer preserves the bit pattern (such a round-trip is a NOP).

- `bitcast` does not preserve NaN's bit pattern. When a NaN is bit cast to integer, it gets assigned a non-deterministic bit pattern.

Unfortunately, either semantics makes some of LLVM's optimizations incorrect. At the time of writing there was no consensus in the community on which of the semantics to adopt. We chose the second one in Alive2, as it supports processors that canonicalize NaN bit patterns when a NaN is loaded into a floating-point register.

5.2.6 Return Value

As previously mentioned, a function in LLVM may either return a value or reach a "no-return" instruction. We therefore compute two expressions: the returned value, and a Boolean indicating in which cases the function reaches a "no-return" instruction. To compute the final state, we merge the states of each of the return instructions through a linear chain of ite expressions.

The SMT encoding for the register file of the function shown in Fig. 1.1 is:

$$\begin{array}{lcl}
R[\%\mathsf{a}] & = & (\mathsf{ite}(\mathsf{isundef}_{\%\mathsf{a}}, \mathsf{undef}_1, \%\mathsf{a}), \mathsf{ispoison}_{\%\mathsf{a}}) \\
R[\%\mathsf{b}] & = & (\mathsf{ite}(\mathsf{isundef}_{\%\mathsf{b}}, \mathsf{undef}_2, \%\mathsf{b}), \mathsf{ispoison}_{\%\mathsf{b}}) \\
R[\%\mathsf{t}] & = & \big(\mathsf{ite}(\mathsf{isundef}_{\%\mathsf{a}}, \mathsf{undef}_3, \%\mathsf{a}) + \mathsf{ite}(\mathsf{isundef}_{\%\mathsf{a}}, \mathsf{undef}_4, \%\mathsf{a}), \\
 & & \;\; \mathsf{ispoison}_{\%\mathsf{a}}\big) \\
R[\%\mathsf{c}] & = & \big(\mathsf{ite}(\mathsf{ite}(\mathsf{isundef}_{\%\mathsf{a}}, \mathsf{undef}_5, \%\mathsf{a}) + \\
 & & \qquad\quad \mathsf{ite}(\mathsf{isundef}_{\%\mathsf{a}}, \mathsf{undef}_6, \%\mathsf{a}) = 0, 1, 0), \mathsf{ispoison}_{\%\mathsf{a}}\big) \\
R[\%\mathsf{q}] & = & (\mathsf{shl}(\%\mathsf{a}, 2), \mathbf{false}) \\
R[\%\mathsf{r}] & = & (\mathsf{ite}(\mathsf{isundef}_{\%\mathsf{b}}, \mathsf{undef}_7, \%\mathsf{b}) \;\&\; 1, \mathsf{ispoison}_{\%\mathsf{b}})
\end{array}$$

Each use of an undef value creates a new fresh variable to account for the case that each observation may yield a different value. Perhaps the most surprising is the value of $R[\%\mathsf{q}]$ as it ignores the cases when $\%\mathsf{a}$ is **undef** or **poison**. This is an optimization: since branching on a non-well-defined value is UB, we can assume that $\%\mathsf{t}$ is well-defined and transitively $\%\mathsf{a}$ as well.

Another expression that might be surprising is that of $R[\%\mathsf{c}]$. LLVM has no Boolean type and uses 1-bit integers instead, which explains the extra ite to convert the Boolean expression to a bit-vector.

The final state for the same function is as follows:

$$\begin{array}{lcl}
\mathit{retval} & = & \big(\mathsf{ite}(\%\mathsf{a} + \%\mathsf{a} = 0, \mathsf{shl}(\%\mathsf{a}, 2), \\
 & & \qquad \mathsf{ite}(\mathsf{isundef}_{\%\mathsf{b}}, \mathsf{undef}_7, \%\mathsf{b})\ \&\ 1), \\
 & & \quad \mathsf{ite}(\%\mathsf{a} + \%\mathsf{a} = 0, \mathbf{false}, \mathsf{ispoison}_{\%\mathsf{b}})\big) \\
\mathit{ub} & = & \mathsf{isundef}_{\%\mathsf{a}} \vee \mathsf{ispoison}_{\%\mathsf{a}} \\
\mathit{noret} & = & \mathbf{false}
\end{array}$$

We perform the same optimization as described for the register file. By taking advantage of the cases that are guaranteed to be UB, we are able to simplify the formulas for the return value.

5.2.7 Additional Optimizations

We do several optimizations to shrink the size of SMT formulas by taking advantage of invariants that are deduced during verification condition generation. For example, we track whether a register is **undef** or **poison** in a flow-sensitive way. This information is deduced from the cases where it would be UB if a register was **undef** or **poison** (such as when branching on that register). It does not matter whether a value is well-defined if the program already triggered UB.

Another set of facts we propagate in a flow-sensitive way is the set of unused $\mathsf{undef}_\mathsf{i}$ variables to avoid rewriting expressions on each register file lookup. When we lookup the value of, say, $\%\mathsf{r}$ in the register file, the $\mathsf{undef}_\mathsf{i}$ variables are not rewritten if this register was not used before in the current basic block or in any predecessor. This reduces the number of formula rewrites and the number of quantified variables.

As shown in the previous example, we attempt to compute closed-form expressions to determine when registers are **undef** for common patterns. For example, the expression $\mathsf{ite}(\mathsf{isundef}_{\%\mathsf{a}}, \mathsf{undef}_1, \%\mathsf{a}) = 0$ is **undef** iff $\mathsf{isundef}_{\%\mathsf{a}}$ is true. This expression is simpler than the general formula and does not use quantified variables.

We instantiate the $\mathsf{isundef}_{\%\mathsf{r}}$ variables in the final SMT formula (i.e., replace $\exists x\ .\ f(x)$ with $\exists x\ .\ (\neg x \wedge f(\mathbf{false})) \vee (x \wedge f(\mathbf{true})))$, up to a bound to limit the exponential growth. This helps refinement proofs substantially as the non-**undef** case often becomes trivial and the fully **undef** case also becomes simpler (as the SMT solver can replace, e.g., $\mathsf{undef}_1 + \mathsf{undef}_2$ with undef' as these $\mathsf{undef}_\mathsf{i}$ variables often only appear once in the formula).

5.3 Encoding Memory Blocks and Pointers in SMT

We describe our new encoding of LLVM's memory model in SMT over the next few sections. We use the theories of UFs (uninterpreted functions), BVs (bit-vectors), and arrays with lambdas [105], with first order quantification. Moreover, we consider that the scope of verification is a single function without loops (or where loops have been previously unrolled). We only consider logical pointers and a single address space.

5.3.1 Memory Blocks

Each memory block is assigned a distinct identifier (a bit-vector number). We further split memory blocks into local and non-local. Local blocks are all those that are allocated within the function under consideration, either on the stack

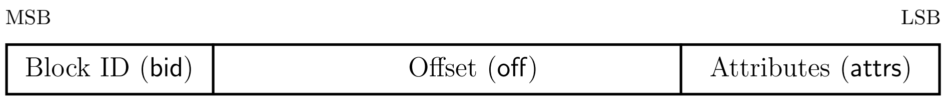


Figure 5.3: The bit-vector representation of a pointer.

or the heap. Non-local blocks are the remaining ones, including global variables, heap/stack allocations in callers and heap allocations in callees (stack allocations in callees are not observable, since they are deallocated when the called function returns, hence there is no need to consider them).

We use the most significant bit (MSB) to encode whether a block is local (1) or non-local (0). This representation allows the null block to have $\mathsf{bid} = 0$ and be non-local. We refer to the short block id, or $\widetilde{\mathsf{bid}}$, to refer to bid without the MSB. This is used in cases where it has already been established whether the block is local or not. Example with 4-bit block ids:

```
int g;                  // bid(g) = 0001
void f(int *p) {        // bid(p) = 0xyz (with xyz = arbitrary)
  int a[2];             // bid(a) = 1000
  int *q = malloc(4);   // bid(q) = 1001
}
```

The separation of local and non-local block ids is an efficient way to encode the constraint that pointers of these groups cannot alias with each other. In the example above, argument `p` cannot alias with either `a` or `q`.

As we only consider functions without loops, block ids can be statically assigned for each allocation site.

5.3.2 Pointers

A pointer $\mathsf{ptr} = (\mathsf{bid}, \mathsf{off}, \mathsf{attrs})$ is encoded as a single bit-vector consisting in the concatenation of the three elements. The offset is interpreted as a *signed* number (which is why blocks cannot be larger than half of the address space). Each attribute (such as **readonly**) is encoded with a bit. Example with 2-bit block ids and offsets, and a single attribute (we use `.` to visually separate the elements):

```
void f(char readonly *p, char *q) { // p = 0x.ab.1, q = 0y.cd.0
  char *r = p + 2;                  // r = 0x.(ab+2).1
  char *s = q + 3;                  // s = 0y.(cd+3).0
  char *t = malloc(4);              // t = 10.00.0
}
```

Let $\widetilde{\mathsf{off}}$ be a truncated offset where the least significant bits corresponding to the greatest common divisor of the alignment and sizes of all memory operations are removed. For example, if all operations are 4-byte aligned and they access either 4- or 8-byte values, then $\widetilde{\mathsf{off}}$ has less 2 bits than off (as these are guaranteed to be always zero when accessing the memory).

5.3.3 Block Properties

Each block has seven associated properties: size, alignment, read-only, liveness, allocation type (heap, stack, global), physical address, and value. Block properties are looked up and updated by memory operations. For example, when doing a store, we need to check if the access is within the bounds of the block.

Except for liveness and value, properties are fixed at allocation time. Liveness is encoded with a bit-vector (one bit per block), and value with arrays (indexed on $\widetilde{\mathsf{off}}$). We use a multi-memory encoding, where we have one array per bid.

The encoding of fixed properties differs for local and non-local blocks. For non-local blocks, we use a UF symbol per property, taking $\widetilde{\mathsf{bid}}$ as argument. For local blocks, we cannot use UFs because for the refinement check some of these would have to be quantified (c.f. Section 5.6) and most, if not all, SMT solvers do not support quantification of UF symbols. Therefore, we encode each of the remaining properties of local blocks as an if-then-else (ITE) expression, which is tailored for each use (e.g., each time an operation needs to lookup a local block's size, we build an ITE expression for the given $\widetilde{\mathsf{bid}}$).

Using ITE expressions to encode properties is less concise than using UFs. However, it is not a disaster for two reasons. Firstly, we only need to consider the local blocks that have been allocated beforehand, since the program cannot access blocks allocated afterward. Secondly, pointers are usually not fully arbitrary. Oftentimes we know statically which type of block they refer to, and even what is the block id, given that pointer arithmetic operations do not change the block id. Therefore, the ITE expressions are usually small in practice. Example with 4-bit block ids and offsets of a source program:

```
int g;                          // g = 0001.0000, size_src(001) = 4
void f() {
  char p[2];                    // p = 1000.0000
  char q[3];                    // q = 1001.0000
  char *r = ... p or q or g ...
  r[2] = 0;
  char t[1];                     // t = 1010.0000
}
```

The store in this program is only well defined if the size of block pointed by `r` is greater than 2. This is encoded in SMT as follows:

$$\mathsf{ite}(\mathbf{islocal}(r), \mathsf{ite}(\widetilde{\mathbf{bid}}(r) = 0, 2, 3), \mathsf{size}_{src}(\widetilde{\mathbf{bid}}(r)) > 2)$$

Function $\mathbf{islocal}(p)$ is encoded with the SMT $\mathsf{extract}$ expression to fetch the MSB of the pointer. Similarly, $\widetilde{\mathbf{bid}}(p)$ extracts the relevant bits from a pointer. The expression for local blocks only needs to consider local blocks 0 and 1, since block 2 (`t`) is only allocated afterward. This allows a simple single pass through the code to generate optimized ITE expressions.

Value

Value is defined as an array from short offset to byte (described later in Section 5.5.1). For non-local blocks, only those that are constant are initialized with the respective value. The remaining blocks are allowed to take almost any value. The exception is for pointers: non-local blocks cannot initially have local pointers stored, since the calling environment cannot fabricate local pointers.

Local blocks are initialized with **poison** values using a constant array (i.e., an array that yields the same value for all indexes).

5.3.4 Physical Addresses

If a program observes addresses (through, e.g., pointer-to-integer casting or pointer comparison[3]), we need additional constraints to ensure that addresses of blocks that overlap in time are disjoint. Since we are doing translation validation, we have two programs with potentially different sets of locally allocated blocks. Therefore, we need to ensure that non-local blocks' addresses are disjoint from those of local blocks of both programs. This makes the disjointness constraints quite complex.

As an optimization, we split the address space in two: local blocks have MSB=1 and non-locals have MSB=0. Since the encoding of address disjointness is quadratic in the worst case (cross-product of blocks), halving the number of blocks is significant. This optimization, however, is an under-approximation of the program's behavior (Section 5.8). After investigating LLVM's optimizations, we believe it is highly unlikely this approximation will cause false negatives.

If a program does not observe any pointer's physical address, neither the block's physical address property nor the disjointness axioms are instantiated.

However, when dereferencing a pointer, we need to check if the physical address is sufficiently aligned. When physical addresses are not created, we resort to checking alignment of both of the pointer's block and offset. Since in this case physical addresses are not observed (and therefore not constrained by the program using, e.g., pointer comparisons), a block's physical address

[3]We decided to implement the integer comparison semantics. Section 4.3.6 explains the details about pros and cons of two pointer comparison definitions.

can take any value, and therefore blocks and offsets must be both sufficiently aligned to ensure that physical pointers are aligned in all program executions. This argument justifies why we can soundly discard physical addresses.

5.3.5 Bounding the Maximum Number of Blocks

Since we assume that programs do not have loops, we can statically bound the maximum number of both local and non-local blocks a program may observe.

The maximum number of local blocks in the source and target programs, respectively, N^{src}_{local} and N^{tgt}_{local}, is computed by counting the number of heap and stack allocation instructions. Note that this is an upper-bound because not all allocation sites may be reachable in practice.

For non-local blocks, we cannot see their definitions as with local blocks, except for global variables. Nevertheless, we can still bound the maximum number of observed blocks. It is sufficient to count the number of instructions that may return non-local pointers, such as function calls and pointer loads. In addition, we consider a null block when needed (if the null pointer may be observed).

To encode the behavior of source and target programs, we need $N^{src}_{nonlocal} + N^{tgt}_{nonlocal}$ non-local blocks in the worst case, as all referenced pointers may be distinct. However, correct transformations will not have the target program observe more blocks than the source. If the target observes a pointer to a non-local block that was not observed in the source, we can set that pointer to **poison** because its value is not restricted by the source. Therefore, $N^{src}_{nonlocal}$ non-local blocks are sufficient to allow the target to exhibit *an* incorrect behavior.

The bit-width of $\widetilde{\mathsf{bid}}$ is: $w_{\widetilde{bid}} = \lceil \log_2(max(N^{src}_{nonlocal}, max(N^{src}_{local}, N^{tgt}_{local})))\rceil$. When only local or non-local pointers are used, $w_{bid} = w_{\widetilde{bid}}$, as we know statically if the pointer is local or not. Otherwise, $w_{bid} = w_{\widetilde{bid}} + 1$.

5.3.6 Function Attributes

LLVM's function attributes constrain the pre/postcondition of a function. For example, if a pointer argument is marked as **readonly**, the function should not write any byte via that pointer (triggers UB otherwise). We implemented the most common memory-related attributes: **nonnull**, **dereferenceable**(n), **byval**, **readonly**, **readnone**, **nocapture**, and **nofree**.

5.3.7 C Library Functions

We encoded a few commonly used library functions that LLVM recognizes and optimizes, including **memcpy**, **memset**, **memcmp**, and **strlen**. The semantics of these is not specified in the LLVM manual, so we created a reasonable semantics from the C99 standard and tested the semantics against LLVM's implementation.

LLVM unrolls small **memcmp** (and **bcmp**) into a sequence of integer loads and comparisons. To explain this transformation, we set the result to **poison** if any of the loaded bytes is **poison** (as opposed to, e.g., UB). On the other hand, we defined **strlen** to raise UB on poison bytes and it did not cause any failure (note that raising UB rather than returning poison leads to a more efficient encoding in SMT).

Encoding the exact semantics of these library functions in SMT requires the use of quantifiers. For example, $n = \textbf{strlen}(p)$ should constrain n so that all bytes between p and $p + n$ are non-zero. To avoid an introduction of quantifiers, we approximated these functions by unrolling them a constant number of times, except for **memcpy** and **memset** where we use lambdas.

5.4 Memory Allocation

In LLVM, memory blocks can be allocated on the stack (**alloca**), in the heap (e.g., **malloc**, **calloc**, etc), or as global variables. It is surprisingly non-trivial to find a semantics for memory allocations that allows all of LLVM's optimizations, and rejects undesired transformations. For example, we have to support allocation removal and splitting, introduce new stack allocations and new constant global variables, etc. We explore multiple semantics and show their merits and shortcomings in the context of proving correctness of program transformations.

5.4.1 Heap Allocation

Heap allocation is done through functions such as **malloc**, **calloc**, C++'s `new` operator, etc. We describe semantics for **malloc**; remaining functions can be described in terms of it.

First of all, it is important to note that there are two common idioms used in practice by C programmers when doing memory allocation:

```
int *p = malloc(4);      |    int *p = malloc(4);
*p = 0;                  |    if (p) { *p = 0; }
```

In some programs, like the example on the left, **malloc** is assumed to never return **null**, say non-null assumption. This is mainly because the program does not consume too much memory and it is expected that the computer has enough memory/swap space. In other programs like the one on the right, **malloc** is expected to sometimes return **null**, say may-null assumption. Therefore, the program performs null-ness checks.

Since both programming styles are prevalent, we would like optimizations to be correct for both. This is non-trivial, as the two assumptions are conflicting: with the non-null assumption, it is sound to eliminate **null** checks, but not with the may-null assumption. We now explore several possible semantics to find one that works for both programming styles.

A. Malloc always succeeds. Based on the non-null assumption, in this semantics we only consider executions where there is enough space for all allocations to succeed. Regardless of whether the target uses more or less memory than the source, all calls to **malloc** yield non-null pointers. Therefore, for example, deleting unused **malloc** calls is allowed.

However, removing **null** checks of **malloc** is also allowed in this semantics. For example, optimizing the right example above into the left one is sound. This transformation, however, is obviously undesirable.

B. Malloc only succeeds if there is enough free space. To solve the problem just described, based on the may-null assumption, we can simulate the behavior of dynamic memory allocation and define **malloc** to return a pointer to a newly created block if there is an empty space in memory, and **null** otherwise. This semantics prevents the removal of **null** checks of **malloc** as it may return **null**.

However, this semantics does not explain removal of unused allocations. It aligns both source and target programs' allocations such that any change in the allocation sequence disrupts the program alignment and thus makes verification fail. For example, the following transformation removing unused **malloc** instructions and replacing comparisons of their output with **null** is not supported:

```
int *x = malloc(4);              ⇒   // remove x (unused)
if (x != nullptr) { ... }            if (true) { ... }
```

In case there were 0 bytes left in memory, `x` would be **null**, but since LLVM assumes that the program cannot observe the state of the allocator it folds the comparison `x != nullptr` to `true` after eliminating the allocation. This optimization would be flagged as incorrect in this semantics.

LLVM assumes very little about the run-time behavior of memory allocators. This is to support, for instance, garbage collectors, where an allocation may fail but if repeated it may succeed because memory was reclaimed in between. This explains why LLVM folds comparisons with **null** of unused memory blocks, and also contradicts the linear view of allocations of this semantics.

C. Malloc non-deterministically returns null. This semantics abstracts the behavior of the memory allocator by (1) allowing **malloc** to non-deterministically return **null** even if there is available space, and (2) only considering executions where there is enough space for all allocations to succeed. This semantics prevents the removal of null checks of **malloc**, which fixes the shortcomings of semantics A, and also allows the removal of unused allocations, which fixes those of semantics B. However, this semantics is too weak and therefore allows other undesirable transformations, like the following:

```
p = malloc(4);                  ⇒   exit();
*p = 0;
```

For the sake of proving refinement (Section 5.6), we need just one trace triggering UB (i.e., one particular realization of the non-deterministic choices) for a given input to be able to transform the source program into anything for that input. Informally speaking, refinement always picks the worst-case execution for each input. Since the source program executes UB when `p` is **null**, it is correct to transform the source into any program although that is obviously undesirable.

This semantics is too weak in practice since many programs are written without **null** checks, either assuming the program will not run out of memory, or assuming the program will terminate if it runs out memory. It is not reasonable in practice to allow compilers to break all such programs.

Our solution. As we have seen, there is no single semantics that both allows all desired transformations and rejects undesired ones. While semantics B prevents desired optimizations like allocation removal, semantics A and C allow undesired optimizations, but in a complementary way. For example, removing null checks of **malloc** is allowed in A but not in C. On the other hand, transforming an access of a **malloc**-allocated block without a **null** check beforehand into arbitrary code is allowed in C but not in A.

Therefore, we obtain a good semantics by requiring both A and C: an optimization is correct if it passes the refinement criteria with each of the two semantics. Intuitively, this definition requires the compiler to support the two considered coding styles: semantics A supports the non-null assumption, while semantics C the may-null assumption.

5.4.2 Stack Allocation

The semantics of **alloca**, the stack-allocation instruction, is slightly different from that of **malloc**. LLVM assumes that stack allocations always succeed, since the program will likely crash if there is a stack overflow. That is, **alloca** never returns a **null** pointer.

LLVM performs more optimizations on stack allocations than on heap ones. For example, LLVM can split an allocation into multiple smaller ones or increase the alignment. These transformations can increase memory consumption.

5.5 Encoding Loads and Stores in SMT

We encode the value of memory blocks with several arrays (one per bid): from short offset to byte. We next give the definition of byte and the encoding of memory accessing instructions in SMT.

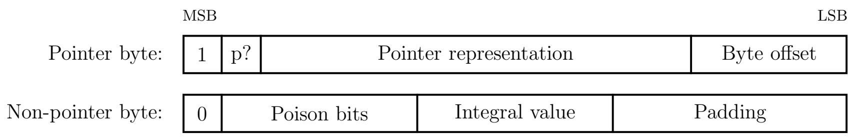


Figure 5.4: Bit-wise representation of a byte. A pointer byte is poison if ‘p?’ is zero. A non-pointer byte tracks poison bit-wise.

5.5.1 Byte

There are two types of bytes: *pointer* bytes and *non-pointer* bytes, cf. Fig. 5.4.

A **pointer byte** has the most significant bit (MSB) set to one. The following bit states whether the byte is poison or not. Next is the pointer representation as described in Section 5.3.2 $(\mathsf{bid}, \mathsf{off}, \mathsf{attrs})$.

Pointers are often longer than one byte, so when storing a pointer to memory we write multiple consecutive bytes. Each of these bytes records the same pointer, but with a different byte offset (the last bits of the byte) to distinguish between the partial bytes of the pointer.

For **non-pointer bytes**, we track whether each of the bits is poison or not. This is not required for pointers, since LLVM does not allow pointer values to be manipulated bit-wise. Non-pointer values can be manipulated bit-wise (e.g., using vectors with element types smaller than 8 bits). Each bit of the integral value is only significant if the corresponding poison bit is zero.

5.5.2 Load and Store Instructions

Load and store instructions are trivially encoded using SMT arrays. These arrays store bytes as described in the previous section. We next describe how LLVM values are encoded to and decoded from our byte representation.

We define two functions, $ty{\downarrow}(v)$ and $ty{\uparrow}(b)$, which convert a value v into a byte array and a byte array b back to value, respectively. We show below $ty{\downarrow}(v)$ when $v \neq \mathbf{poison}$. $\mathbf{i}sz$ stands for the integer type with bit-width sz. If sz is not a multiple of 8 bits, v is zero-extended first. When v is poison, all poison bits are set to one. BitVec(n, b) stands for number n with bit-width b. Pointer's byte offset is 3 bits because we assume 64-bit pointers.

$$\begin{array}{rcl} \mathbf{i}sz{\downarrow}(v) \text{ or } \mathbf{float}{\downarrow}(v) & = & \lambda i.\, 0 \mathbin{+\!\!+} 0^8 \mathbin{+\!\!+} \text{bitrepr}(v)[8\times i \ldots 8\times(i+1)-1] \mathbin{+\!\!+} \text{padding} \\ ty{*}{\downarrow}(v) & = & \lambda i.\, 1^2 \mathbin{+\!\!+} \text{bitrepr}(v) \mathbin{+\!\!+} \text{BitVec}(i, 3) \end{array}$$

$\mathbf{i}sz{\uparrow}(b)$ and $\mathbf{float}{\uparrow}(b)$ return **poison** if any bit is **poison**, or if any of the bytes is a pointer. Otherwise, these functions return the concatenation of the integral values of the bytes.

$ty{*}{\uparrow}(b)$ returns **poison** if any of the bytes is **poison** or not a pointer, there is more than one distinct pointer value in b, or one of the bytes has an incorrect byte offset (they have to be consecutive, from zero to byte size minus one). An exception is reading a non-pointer zero byte, which is interpreted as a null pointer byte. This allows initialization of, e.g., arrays with null pointers with **memset** (which is an idiom commonly used in LLVM IR).

Instructions **memset** and **memcpy** are encoded using lambdas.

5.5.3 Multi-Array Memory

As already described, we use a multi-array encoding for memory, with one array per block id, each indexed on $\widetilde{\mathsf{off}}$. A simpler encoding would have used a single array indexed on ptr. The multi-array encoding is beneficial when we can cheaply compute small aliasing sets for each memory access. In that case, we reduce the case-splitting work on bid that the SMT solver needs to do, and it enables further formula simplifications like store forwarding.

The multi-array encoding may, however, end up in a larger encoding overall if several of the accesses may alias with too many blocks. For load operations that alias multiple blocks the resulting expression is a linear combination of the loads of each block, e.g., $\mathsf{ite}(\mathsf{bid} = 0, \mathbf{load}(m_0, \widetilde{\mathsf{off}}), \mathsf{ite}(\mathsf{bid} = 1, \mathbf{load}(m_1, \widetilde{\mathsf{off}}), \ldots))$. In this case, it would be more compact to use the single-array encoding. Note that even if we do not know the specific block id, we often know whether a pointer refers to a local or non-local block (e.g., pointers received as argument have unknown block id, but are known to be non-local), and hence splitting the memory in two is usually a good idea (c.f. Section 5.9).

We perform several optimizations that are enabled with this multi-array encoding. We do partial-order reduction (POR) to shrink the potential aliasing of pointers with unknown block id. For example, consider a function with two pointer arguments (`x` and `y`) and one global variable. We assign $\mathsf{bid} = 1$ to the global variable. Then, we estipulate that `x` can only alias blocks with $\mathsf{bid} \leq 2$, which is sufficient to access the global variable or another unknown block. Argument `y` is also constrained to only alias blocks with $\mathsf{bid} \leq 3$, allowing it to alias with the global variable, the same block as `x`, or a different block. The same is done for function calls that return pointers. This POR technique greatly reduces the potential aliasing of unknown pointers without losing precision.

5.6 Verifying Correctness of Optimizations

To verify correctness of LLVM optimizations, we establish a refinement relation between source (original) and target (optimized) code. We cannot simply check for equivalence because UB-related transformations are ubiquitous.

Given functions f_{src} and f_{tgt}, a set of input and output variables I_{src}/I_{tgt} and O (which include, e.g., memory, side effects, and the return value), a set of non-determinism variables N_{src}/N_{tgt}, f_{src} is refined by f_{tgt} iff:

$$\begin{aligned}\forall I_{src}, I_{tgt}, O\,.\ \ & (I_{src} \sqsupseteq I_{tgt}\ \wedge\ \exists N_{tgt}\,.\, f_{tgt}(I_{tgt}, N_{tgt}, O)) \implies \\ & (\exists N_{src}\,.\, f_{src}(I_{src}, N_{src}, O))\end{aligned}$$

In other words, for any fixed input I_{src}, if the target function f_{tgt} produces a given output O with some internal non-determinism N_{tgt} and refined input I_{tgt} (equal to or more defined than I_{src}), the source function must produce the same output for some internal non-determinism N_{src}. Therefore, the target function is allowed to remove non-determinism so it generates fewer outputs for a given input, but not the other way around.

We only support intraprocedural optimizations, and therefore checking refinement of each function individually is sufficient to establish refinement of an entire program. The definition above ensures compositionality: if a function's inputs are refined, so are the outputs. This definition is assumed at call sites, and established for each function, justifying why checking each function individually is sufficient.

In LLVM, loops tagged with the **mustprogress** attribute must either terminate or perform externally observable actions infinitely often. The function triggers UB otherwise. Therefore, every function in LLVM with only such loops terminates, perhaps triggering UB. In practice, the compiler can only prove non-termination of simple cases and therefore we only need to support those. Moreover, given that we do *bounded* translation validation, we do not support non-terminating loops without the **mustprogress** attribute.

5.6.1 Refinement of Program State

We start by defining refinement between values. A value v is refined by another value v' if v' is equivalent to or more defined than v. Fig. 5.6 gives rules for the definition.

For integer and floating-point numbers, refinement holds between two equal numbers (ELEMENT-NONPTR). For pointers, we cannot simply use equality because local pointers (such as pointers to stack-allocated memory blocks) in

$\text{Num}(sz) ::= \{i \mid 0 \le i < 2^{sz}\}$	$\text{BlockID} ::= \mathbb{N}$	$\text{Addr} ::= \text{Num}(64)$	$\text{Offset} ::= \text{Num}(64)$
$\text{PtrAttr} ::= \{\texttt{nocapture}, \texttt{readonly}, \texttt{readnone}\}$			
$\text{Pointer} ::= \text{BlockID} \times \text{Offset} \times 2^{\text{PtrAttr}}$	$\text{DefinedValue} ::= \text{Int} \uplus \text{Pointer} \uplus \text{Float}$		
$\text{Value} ::= \text{Aggregate} \uplus \text{Int} \uplus \text{Pointer} \uplus \text{Float} \uplus \{\textbf{poison}\}$	$\text{Aggregate} ::= \text{list Value}$		
$\text{PtrByte} ::= (\text{Pointer} \times \{i \mid 0 \le i < 8\}) \uplus \{\textbf{poison}\}$	$\text{NonPtrByte} ::= \text{Num}(8) \times \text{Num}(8)$		
$\text{Byte} ::= \text{PtrByte} \uplus \text{NonPtrByte}$	$\text{Bytes} ::= \text{Offset} \rightarrow \text{Byte}$	$\text{Size} ::= \text{Num}(64)$	
$\text{Align} ::= \{i \mid 0 \le i < 64\}$	$\text{Kind} ::= \{\texttt{stack}, \texttt{malloc}, \texttt{new}, \texttt{global}\}$	$\text{Live} ::= \text{bool}$	
$\text{Writable} ::= \text{bool}$	$\text{MemBlock} ::= \text{Addr} \times \text{Align} \times \text{Kind} \times \text{Live} \times \text{Writable} \times \text{Size} \times \text{Bytes}$		
$\text{Memory} ::= \text{BlockID} \rightarrow \text{MemBlock}$	$\text{UB} ::= \text{bool}$		
$\text{ValueNoRet} ::= \text{Value} \uplus \{\text{ noreturn }\}$	$\text{FinalState} ::= \text{ValueNoRet} \times \text{Memory} \times \text{UB}$		

$p \in \text{Pointer}$	$ag \in \text{Aggregate}$	$\nu \in \text{DefinedValue}$	$v \in \text{Value}$	$r \in \text{ValueNoRet}$
$pb \in \text{PtrByte}$	$nb \in \text{NonPtrByte}$	$b \in \text{Byte}$	$mb \in \text{MemBlock}$	
$M \in \text{Memory}$	$ub \in \text{UB}$	$\mu \in \text{BlockID} \nrightarrow \text{BlockID}$		

Figure 5.5: Type Definitions and Variable Naming Conventions. $\mathcal{P}$ is the power set operation.

$$\textsc{element-nonptr}\quad \frac{\nu \in \text{Int} \uplus \text{Float}}{\textbf{poison} \sqsupseteq^{\mu}_{e} \nu} \qquad \textsc{element-ptr}\quad \frac{\nu, \nu' \in \text{Pointer} \quad \nu \sqsupseteq^{\mu}_{\text{ptr}} \nu'}{\nu \sqsupseteq^{\mu}_{e} \nu'}$$

$$\textsc{value-poison}\quad \frac{v \in \text{Value}}{\textbf{poison} \sqsupseteq^{\mu} v} \qquad \textsc{value-undef}\quad \frac{v, v' \in \mathcal{P}(\text{DefinedValue}) \qquad \forall \nu' \in v' \,.\, \exists \nu \in v \,.\, \nu \sqsupseteq^{\mu}_{e} \nu'}{v \sqsupseteq^{\mu} v'} \qquad \textsc{value-aggregate}\quad \frac{v, v' \in \text{Aggregate} \qquad |v| = |v'| \quad \forall i \,.\, v[i] \sqsupseteq^{\mu} v'[i]}{v \sqsupseteq^{\mu} v'}$$

$$\textsc{final-state-ub}\quad \frac{r, r' \in \text{ValueNoRet} \qquad M, M' \in \text{Memory} \qquad ub' \in \text{UB}}{\langle r, M, \textbf{true} \rangle \sqsupseteq_{\text{st}} \langle r', M', ub' \rangle} \qquad \textsc{final-state-noret}\quad \frac{r = \text{noreturn} \qquad M, M' \in \text{Memory} \qquad ub = \textbf{false}}{\langle r, M, ub \rangle \sqsupseteq_{\text{st}} \langle r, M', ub \rangle}$$

$$\textsc{final-state}\quad \frac{r, r' \in \text{Value} \qquad M, M' \in \text{Memory} \qquad \exists \mu,\ r \sqsupseteq^{\mu} r' \wedge M \sqsupseteq^{\mu}_{\text{mem}} M'}{\langle r, M, \textbf{false} \rangle \sqsupseteq_{\text{st}} \langle r', M', \textbf{false} \rangle}$$

Figure 5.6: Refinement of value and final state.

source and target are internal to each of the functions. Even if they have the same block identifier, they may refer to different allocation sites in the functions (ELEMENT-PTR). Similarly, the refinement of the final state should consider this difference between local pointers. To address this, we track a mapping μ between escaped local blocks of the two functions, which will be described in

Section 5.6.3.

poison is refined by any value (VALUE-POISON). A (partially) undef value is refined by another undef value that is equally or more defined (VALUE-UNDEF). Aggregate values are compared element-wise (VALUE-AGGREGATE).

Next, we define refinement between final states. Fig. 5.5 shows the definition of final program state which is a tuple of return value, return memory, and UB. A memory is a function from block id to a memory block. A memory block has seven attributes that are described in Section 5.3.3. A final state is defined as a tuple $\langle r, M, ub \rangle$, where r is the return value, M the memory at the return site, and ub a Boolean flag indicating whether the function triggered UB before returning. A memory M is refined by M', $M \sqsupseteq^{\mu}_{\text{mem}} M'$, if refinement holds between blocks in M and M' (value and remaining attributes) with respect to mapping μ. A final state s is refined by s', or $s \sqsupseteq_{\text{st}} s'$, if (1) s is undefined (FINAL-STATE-UB), (2) both s and s' never return, or (3) refinement between their respective return values and memories holds with respect to some mapping μ (FINAL-STATE).

Using $\sqsupseteq_{st}$, we define correctness of an optimization as follows. If functions f_{src} and f_{tgt} are deterministic, f_{src} is refined by f_{tgt} if:

$$\forall I_{src}, I_{tgt} \,.\; I_{src} \sqsupseteq I_{tgt} \;\wedge\; \text{valid}(I_{src}, I_{tgt}) \implies \\ [\![f_{src}]\!](I_{src}) \sqsupseteq_{st} [\![f_{tgt}]\!](I_{tgt})$$

The valid predicate encodes the global precondition. For example, it states that global variables should be assigned non-null and disjoint addresses. $[\![f]\!](I)$ is the final state after executing function f with input I.

5.6.2 Nondeterministic Execution

The previous definition of correctness does not take non-determinism, such as **undef** values and **freeze** instructions, into account. Let N_{src} and N_{tgt} be the set of variables used to encode non-determinism in functions f_{src} and f_{tgt}, respectively. We extend the previous definition of refinement to support non-determinism as follows:

$$\begin{aligned}\forall I_{src},\ & I_{tgt}, O_{tgt} \,.\, I_{src} \sqsupseteq I_{tgt} \ \land\ \text{valid}(I_{src}, I_{tgt}) \ \land \\ & (\exists N_{tgt} \,.\, \text{pre}_{\text{tgt}}(I_{tgt}, N_{tgt}) \ \land\ [\![f_{tgt}]\!](I_{tgt}, N_{tgt}) = O_{tgt}) \\ \implies\ & (\exists N_{src} \,.\, \text{pre}_{\text{src}}(I_{src}, N_{src}) \ \land\ [\![f_{src}]\!](I_{src}, N_{src}) \sqsupseteq_{st} O_{tgt})\end{aligned}$$

Predicate pre represents the precondition of a function, which is used to constrain the non-determinism and the inputs a function can take. For example, in LLVM a function's argument can be marked as non-null. Constraints like this are added to pre.

Sometimes the precondition for the source function does not hold for a particular input I_{src} for any non-determinism but it may hold for the target function. For example, LLVM is allowed to remove the non-null attribute of a function's argument. The formula above fails in that case, since then pre_{src} makes the right-hand side of the implication become false. To support such cases, we extend the previous refinement condition to arrive at the final version that Alive2 actually uses[4]:

$$\begin{aligned}\forall I_{src},\ & I_{tgt}, O_{tgt} \,.\, \text{valid}(I_{src}, I_{tgt}) \ \land \\ & \big(\exists N_{src},\ N_{tgt} \,.\ \ \text{pre}_{\text{src}}(I_{src}, N_{src}) \ \land\ \text{pre}_{\text{tgt}}(I_{tgt}, N_{tgt}) \ \land \\ & \qquad\qquad [\![f_{tgt}]\!](I_{tgt}, N_{tgt}) = O_{tgt}\big) \\ \implies\ & (\exists N_{src} \,.\, \text{pre}_{\text{src}}(I_{src}, N_{src}) \ \land\ [\![f_{src}]\!](I_{src}, N_{src}) \sqsupseteq_{st} O_{tgt})\end{aligned}$$

5.6.3 Refinement of Memory

Checking refinement of non-local memory blocks is simple as blocks are the same in the source and target functions (e.g., global variables have the same

[4]It is an open question whether this definition satisfies transitivity. We leave this question to people who are interested in implementing formally verified validators.

(POINTER)
$$\frac{\begin{array}{c} p.\text{block.live} \Rightarrow p'.\text{block.live} \\ p.\text{offset} = p'.\text{offset} \\ \left[\begin{array}{l} (\text{isNonLocal}(\{p,p'\}) \land p.\text{block.id} = p'.\text{block.id}) \\ \lor\ (\text{isLocal}(\{p,p'\}) \land p.\text{block.id} = \mu[p'.\text{block.id}]) \end{array}\right] \end{array}}{p \sqsupseteq^{\mu}_{\text{ptr}} p'}$$

(MEMORY-MAP)
$$\frac{\begin{array}{c} \left[\begin{array}{l} \forall bid,\ \text{isNonLocal}(bid) \\ \quad \Longrightarrow M[bid] \sqsupseteq^{\mu}_{\text{blk}} M'[bid] \end{array}\right] \\ \left[\begin{array}{l} \forall bid,\ \text{isLocal}(bid) \land \mu[bid] \text{ defined} \\ \quad \Longrightarrow M[\mu[bid]] \sqsupseteq^{\mu}_{\text{blk}} M'[bid] \end{array}\right] \end{array}}{M \sqsupseteq^{\mu}_{\text{mem}} M'}$$

(BYTE-PTR)
$$\frac{\begin{array}{c} pb.\text{byteoff} = pb'.\text{byteoff} \\ pb.\text{ptr} \sqsupseteq^{\mu}_{\text{ptr}} pb'.\text{ptr} \end{array}}{pb \sqsupseteq^{\mu}_{\text{byte}} pb'}$$

(BYTE-NONPTR)
$$\frac{\begin{array}{c} nb'.\text{p} \mid nb.\text{p} = nb.\text{p} \\ nb.\text{v} \mid nb.\text{p} = nb'.\text{v} \mid nb.\text{p} \end{array}}{nb \sqsupseteq^{\mu}_{\text{byte}} nb'}$$

(BYTE-ZERO)
$$\frac{\begin{array}{c} \text{isZeroByte}(b) \\ \text{isZeroByte}(b') \end{array}}{b \sqsupseteq^{\mu}_{\text{byte}} b'}$$

(BYTE-POISON)
$$\frac{\text{isPoisonByte}(b)}{b \sqsupseteq^{\mu}_{\text{byte}} b'}$$

(BYTES)
$$\frac{\left[\begin{array}{l} \forall\, 0 \le i < mb.\text{size}, \\ mb.\text{bytes}[i] \sqsupseteq^{\mu}_{\text{byte}} mb'.\text{bytes}[i] \end{array}\right]}{mb \sqsupseteq^{\mu}_{\text{bytes}} mb'}$$

(BLOCK)
$$\frac{\begin{array}{cc} mb.\text{live} \Rightarrow mb'.\text{live} & mb.\text{size} = mb'.\text{size} \\ mb.\text{kind} = mb'.\text{kind} & mb.\text{writable} = mb'.\text{writable} \\ mb.\text{align} \le mb'.\text{align} & mb.\text{live} \Rightarrow mb \sqsupseteq^{\mu}_{\text{bytes}} mb' \end{array}}{mb \sqsupseteq^{\mu}_{\text{blk}} mb'}$$

Figure 5.7: Refinement of memory and pointers.

ids in the two functions). Therefore, one just needs to compare blocks of source and target functions with the same id pairwise.

Checking refinement of local blocks is harder but needed when, e.g., the function returns a locally-allocated heap block. This is legal, but block ids in the two functions may not be equal as allocations may have happened in a different order. Therefore, we cannot simply compare local blocks with the same ids.

To check refinement of local blocks, we need to align the two functions' allocations, i.e., we need to find a correspondence between local blocks of the two functions. We introduce a mapping $\mu \in \text{BlockID} \nrightarrow \text{BlockID}$ between target and source local block ids.

Local blocks become related on function calls and return statements, which is when local pointers may be observed. For example, if a function is called with a pointer to a local block as the first argument, μ should relate that pointer with the first argument of an equivalent function call in the target function.

Fig. 5.7 gives the definition of memory refinement, $M \sqsupseteq^{\mu}_{\text{mem}} M'$, as well as other related relations between memory blocks and pointers. The first rule POINTER describes refinement between source pointer p and target pointer p' with respect to μ. The following four rules define refinement between bytes b and b'. In rule BYTE-NONPTR, '$a \,|\, b$' is the bitwise OR operation, and it is used to check the equality of only those bits that are not **poison**. Predicate isZeroByte(b) holds if b is a **null** pointer or if it is a zero-valued non-pointer byte. This is needed because stores of **null** pointers can be optimized to **memset** instructions.

Rules BYTES and BLOCK define refinement between memory blocks' values and memory blocks, respectively. Rule MEMORY-MAP describes memory refinement with respect to local block mapping μ. $M[bid]$ stands for the memory block with block id *bid*.

The well-formedness of μ is established in the refinement rules for function calls and return statements. We show these for function calls in Section 5.7. We note that there might be multiple well-formed μ due to non-determinism. As an optimization, we track the memory locations that may contain escaped pointers throughout the function such that we can focus only on those locations when searching for escaped blocks to be related.

5.6.4 SMT Encoding

To check refinement using an SMT solver, the last formula from the previous subsection is negated and the SMT solver is asked to prove unsatisfiability. Rather than running a monolithic query, we check refinement as a sequence of simpler queries; this helps provide detailed error messages to users and also reduces the burden on the solver. We check if:

CALL

$$\frac{\begin{array}{cc} args, args' \in \text{list Value} & (M_o, v_o, ub_o) = \texttt{call}(fn,\ args,\ M) \\ M, M' \in \text{Memory} & (M'_o, v'_o, ub_{o'}) = \texttt{call}(fn,\ args',\ M') \\ M \sqsupseteq^{\mu}_{\text{mem}} M' & \forall i\ .\ args[i] \sqsupseteq^{\mu}_{\text{arg}} args'[i] \\ \mu_{out} = \text{extend}(\mu, M, M', args, args') & \end{array}}{M_o \sqsupseteq^{\mu_{out}}_{\text{mem}} M'_o\ \wedge\ v_o \sqsupseteq^{\mu_{out}} v'_o\ \wedge\ (ub'_o \implies ub_o)}$$

Figure 5.8: Refinement between the inputs and outputs of two function calls. Fig. 5.9 shows the definition of $args[i] \sqsupseteq^{\mu}_{\text{arg}} args'[i]$ which is refinement between function call arguments in source and target programs. Section 5.7.3 shows the definition of extend().

1. Any of the preconditions is always false; this can happen because of bugs or limitations in the encoding.
2. The target triggers UB only when the source does.
3. The return domain of the target is equal to that of the source, except for when the source triggers UB.
4. The return value of the target is **poison** only when the source's return value is **poison**.
5. The return value of the target is **undef** only when the source's return value is **undef** or **poison**.
6. The return value of the source and target are equal when the source value is neither **undef** nor **poison**.
7. Finally, if memory is refined.

5.7 Function Calls

A call to an unknown function may change the memory in an arbitrary way. We model the semantics of call instructions as a pure function that takes its arguments as well as the current memory as inputs, and returns a value and a fresh memory. Let $(M_o, v_o, ub_o) = \texttt{call}(f,\ args,\ M)$ denote a call.

5.7.1 Relating Two Function Calls

When considering multiple calls to the same function, we may need to relate the impact each has on the program state. For the purpose of refinement checking, there are three cases to consider: (1) two calls in source, (2) two calls in target, (3) a call in source and another in target.

For the first case, we consider all pairs of calls in the source to a same function and constrain their behavior such that their outputs are refined if the inputs are refined (c.f. Fig. 5.8). We need to consider this case because LLVM has an optimization that removes duplicated calls. For example, if a function is known to not read or write to memory and it is called twice with the same arguments and one call dominates the other, LLVM removes the dominated call. In practice, LLVM's optimizer only de-duplicates calls with equal inputs (rather than refined), so similarly we make the condition for input refinement stronger (for performance reasons) without introducing false alarms.

Relating two calls in the target is not necessary: these calls must already exist in source (with potentially less defined inputs) and therefore they are already appropriately related, or else refinement does not hold as it is illegal to introduce new function calls. Relating calls in source and target is similar to the first case, but we need to use the full refinement rule as shown in Fig. 5.8.

5.7.2 SMT Encoding

Each call in the source function gets a fresh variable for each of its outputs (memory, return value, and UB flag). Calls in the target are encoded differently, as these must refine at least one of the source's calls since no new calls can be introduced.

Focusing just on poison, a value v is refined by v' iff v is **poison** or $v = v'$. Same reasoning applies for **undef**. Therefore, we consider the two parts of the encoding differently: (1) the poison flag, and (2) the value (only meaningful when the poison flag is false). For the poison flag, we introduce a fresh variable for each call in the target. For the value part, we have to pick one of the source

$$\textsc{(nonptr-arg)}\ \frac{\begin{bmatrix} v, v' \notin \\ \text{Pointer} \end{bmatrix} \quad v \sqsupseteq^{\mu} v'}{v \sqsupseteq^{\mu,sz}_{\text{arg}} v'} \qquad \textsc{(ptr-arg-mapped)}\ \frac{p \sqsupseteq^{\mu}_{\text{ptr}} p'}{p \sqsupseteq^{\mu,sz}_{\text{arg}} p'}$$

$$\textsc{(ptr-arg-unmapped)}\ \frac{\text{isLocal}(\{p, p'\}) \quad p.\text{offset} = p'.\text{offset} \quad M[p.\text{bid}] \sqsupseteq^{\mu}_{\text{blk}} M'[p'.\text{bid}]}{p \sqsupseteq^{\mu,sz}_{\text{arg}} p'}$$

$$\textsc{(ptr-arg-byval)}\ \frac{\begin{matrix} sz > 0 \quad o = p.\text{offset} \quad o' = p'.\text{offset} \\ mb = M[p.\text{bid}] \quad mb' = M'[p'.\text{bid}] \\ \begin{bmatrix} \forall 0 \leq i < sz, \\ mb.\text{bytes}[o{+}i] \sqsupseteq^{\mu}_{\text{byte}} mb'.\text{bytes}[o'{+}i] \end{bmatrix} \end{matrix}}{p \sqsupseteq^{\mu,sz}_{\text{arg}} p'}$$

Figure 5.9: Refinement between function arguments.

call's values or use a fresh one in case the source's value is poison.

Given that the non-deterministic variables of source and target are bound to different quantifiers in the refinement query, it is not trivial to encode the value part. Let C^f_{src} be the set of calls to function f in the source. We introduce a fresh variable i such that $0 \leq i \leq |C^f_{src}|$. Then, in the precondition, we encode that $i = j$ holds iff the jth call in $|C^f_{src}|$ is refined by the target's call, except for $i = |C^f_{src}|$ that holds iff no source's call is refined. The target's value is encoded with an ite expression that ranges over i and yields the corresponding source's value. The target triggers UB if $i = |C^f_{src}|$, i.e., when the call does not correspond to any of the source's.

A limitation of our current implementation is that local variables are never modified by function calls even if their address has escaped.

5.7.3 Encoding Memory Updates

A call to an unknown function may change the memory arbitrarily (except for, e.g., constant variables and non-escaped local blocks). The outputs in the source and target are, however, related: if the target's inputs refine those of the source, refinement holds between their outputs as well.

Let (M_{in}, v_{in}) and (M_{out}, v_{out}) be the input and output of a function call in the source, and their primed versions, (M'_{in}, v'_{in}) and (M'_{out}, v'_{out}), those of a function call in the target. Let μ_{in} be a local block mapping before executing the calls. To state that the outputs are refined if the inputs are refined, we add the following formula to the target's precondition:

$$\Big(M_{in} \sqsupseteq^{\mu_{in}}_{\text{mem}} M'_{in} \wedge \forall i\,.\; v_{in}[i] \sqsupseteq^{\mu_{in},sz[i]}_{\text{arg}} v'_{in}[i]\Big) \implies \big(M_{out} \sqsupseteq^{\mu_{out}}_{\text{mem}} M'_{out} \wedge v_{out} \sqsupseteq^{\mu_{out}} v'_{out}\big)$$

A call to a function with a pointer to a local block as argument escapes this block, as the callee may, e.g., store that pointer to a global variable. Moreover, any pointer stored in this block also escapes as the callee may traverse the block and grab any pointer stored there, and do so transitively. The updated mapping $\mu_{out} = \text{extend}(\mu_{in}, M_{in}, M'_{in}, v_{in}, v'_{in})$ returns μ_{in} updated with the relationship between the newly escaped blocks in source and target functions. These can come either from the arguments or from the memory as the caller may have previously stored addresses of local blocks in non-local blocks or in escaped local blocks. As an optimization, we keep track of blocks that may be escaped on each store instruction to shrink the initial set of locations we need to search for escaped blocks.

Fig. 5.9 shows the definition of refinement between function call arguments in source and target programs. The first rule relates non-pointer arguments. The second one handles pointers that have escaped before these calls. The third rule handles local pointers of blocks that did not escape before these calls, and therefore we need to check if the contents of these block are refined.

The fourth refinement rule handles **byval** pointer arguments. These arguments get a freshly allocated block and the contents of the pointer are copied from the pointer's offset onwards.

5.7.4 Optimizations

The number of function call pairs that may need to be related grows quadratically with the number of calls to the same function. We use a dataflow analysis to cheaply prune call pairs that definitely do not have their inputs refined. For each call, we compute (in a path-sensitive way) the min/max number of calls of each function that were made in all preceding paths. Then we only consider call pairs with overlapping ranges. If ranges do not overlap, one of the calls must have had at least one more call beforehand which could have changed the memory that is read by the function under consideration. Therefore, this is a sound over-approximation of possibly-related call pairs.

5.8 Approximating Program Behavior

In order to speedup verification, we approximate programs' behaviors, which can result in false negatives. We believe none of these approximations has a significant impact because we do not consider the compiler to be malicious (which may not be true in certain contexts).

1. Physical addresses of local memory blocks have the MSB set to 1, and non-locals set to 0. This is reasonable if we assume the compiler is not malicious and therefore will not exploit our approximation.

2. We do not consider the case where a (portion of a) global variable is initially **undef**, only **poison** or a regular value.

3. Library functions **strlen**, **memcmp**, and **bcmp** are unrolled for a constant number of times. A precondition is added to constrain the input to be smaller than the unroll factor. In the case of **strlen**, the input pointer is often a constant array. We compute the result straight away in this case.

5.9 Implementation and Evaluation

This section describes the implementation of Alive2, and evaluates its utility and its impact on the LLVM compiler and community.

5.9.1 Implementation

Alive2 consists in about 23,000 lines of C++ and uses Z3 [106] for SMT solving. Besides its own source code, the trusted computing base for Alive2 includes Z3 and functionality from LLVM for parsing binary and textual LLVM IR into an in-memory representation. Because a premise of our project is that we do not trust LLVM to be correct, Alive2 does not rely on code from LLVM to perform tasks such as computing points-to sets or dominator trees.

Tools Alive2 includes multiple tools for different use cases:

- A plugin for `clang` (LLVM's C/C++ frontend) that validates all optimizations performed by LLVM.
- A plugin for `opt` (LLVM's standalone optimization tool). We add a `-tv` argument so the user can choose after which optimizations Alive2 should run (to support batching), e.g., `opt -tv -sroa -instcombine -tv -gvn -tv file.ll`.
- `alive-tv`, a standalone tool that takes two LLVM IR files and checks refinement between each function present in the two files.
- A pair of compiler drivers, `alivecc` and `alive++`, that respectively invoke `clang` and `clang++` with options that cause our translation validation plugin to be loaded, and then to validate every intra-procedural IR-level transformation that the compiler performs.

Our LLVM plugins implement a trivial (but effective) additional optimization beyond those described earlier in this chapter, which is to avoid running Alive2 at all when an LLVM pass does not make any changes.

5.9.2 Translation Validation of LLVM's Unit Tests

As of version 11, the LLVM test suite contains around 168,000 functions in LLVM IR that are variously optimized, analyzed, and compiled to machine code during testing. About 36,000 of these functions test IR-level transformations and this subset has been the focus of our translation validation efforts.

LLVM's Unit Test Framework This is a typical test case:

```
; RUN: opt < %s -instsimplify -S | FileCheck %s

define i1 @max1(i32 %x, i32 %y) {
; CHECK-LABEL: @max1(
; CHECK:         ret i1 false
;
  %c = icmp sgt i32 %x, %y
  %m = select i1 %c, i32 %x, i32 %y
  %r = icmp slt i32 %m, %x
  ret i1 %r
}
```

It ensures that the instruction simplifier—a collection of peephole optimizations—can recognize that the maximum of two integer values cannot be smaller than the first of those values. The first line specifies that the `opt` tool should run the instruction simplifier and pipe its output through the `FileCheck` tool, which fails the test unless the function is optimized to return false.

To run one or more of these test cases through Alive2, we ask `lit`, the LLVM unit test runner, to call a program that we wrote, instead of calling `opt`. This program runs `opt` with our plugin and skips unsupported transformations (e.g., inter-procedural optimizations). Our plugin then works in three stages. First, it translates the original LLVM IR into Alive2 IR and stores it in memory. Second, it runs the specified (unmodified) LLVM transformations (managed by LLVM's pass manager itself). Finally, it translates the optimized LLVM code into Alive2

IR and checks if it refines the original IR that was saved earlier. Thus, the LLVM unit tests can be run seamlessly through Alive2.

Results We detected 121 violations of refinement in the unit tests:

- 43 optimizations that are incorrect when **undef** is given as input or constant
- 18 optimizations that introduce a branch on **undef** or **poison**, which is UB
- 9 bugs due to the mishandling of vector operations
- 5 UB-related bugs while optimizing `select` instructions
- 4 incorrect arithmetic operations
- 3 occurrences of incorrect handling of floating point "fast-math" flags
- 3 bugs due to the ambiguous semantics of bitcast between integer and floating points
- 4 loop optimizations incorrectly handling memory accesses
- 17 memory-related miscompilations
- 15 failures due to bugs in Alive2, or tests that are designed to fail when an external module like Alive2 is invoked

We reported 54 miscompilation bugs to the LLVM community after identifying the root causes of unit test failures. We did not report every bug detected by Alive2 as some were already known. In April 2021, 28 of the bugs we reported have been fixed, including 7 patches that we wrote ourselves. The remaining fixes were done by LLVM developers, and we actively led discussions around finding good solutions for the bugs. In several cases, compiler developers used Alive2 to help validate that their fixes were correct. Moreover, several members of the LLVM community have become Alive2 users, and have gone on to fix LLVM bugs detected by Alive2, even though we never reported them.

Selected Bug #1: Vectorization Here `%x` is a pointer to an array of 8-bit integers:

```
%a = load i8* %x
%b = load i8* (%x+1)                 %v = load <4 x i8>* %x
%c = load i8* (%x+2)           ⇏     %w = %v[0:1] +nsw %v[2:3]
%d = load i8* (%x+3)                 %r = %w[0]   +nsw %w[1]
%r = %a +nsw %b +nsw
      %c +nsw %d
```

This transformation, which exploits the associativity of addition to reduce the number of instructions using vector addition, is not a refinement. The problem is that LLVM's addition operator, when qualified by the `nsw` flag (which turns signed overflows into poison values), is not associative. The fix was to drop the `nsw` flag from the code on the target side of this transformation.

Selected Bug #2: Floating point This transformation is not a refinement:

```
%c = fmul nsz %a, %b                 %c = fmul nsz %a, %b
%r = fadd %c, +0.0             ⇏
ret %r                               ret %c
```

The `nsz` (non-signed-zero) flag is an assertion that `%c` is nondeterministically $+0.0$ or -0.0 if `%a` $\times$ `%b` $= 0$. However, `%r` is $+0.0$ even if `%c` $= -0.0$ due to the definition of floating point addition. Thus, the target code displays a behavior not observed in the source, violating refinement.

Selected Bug #3: Eliminating a load This transformation is incorrectly removing a load:

```
// i32 *x, *y, *z;                   // i32 *x, *y, *z;
i32 *p = (*x < *y ? x : y);    ⇏     i32 r = (*x < *y ? *x : *y);
*(i64*)z = *(i64*)p;                 *z = r;
```

The bit-width of the store is accidentally shrunk from 64 to 32 bits, which is incorrect since the last 32 bits were not copied. This happened because of the

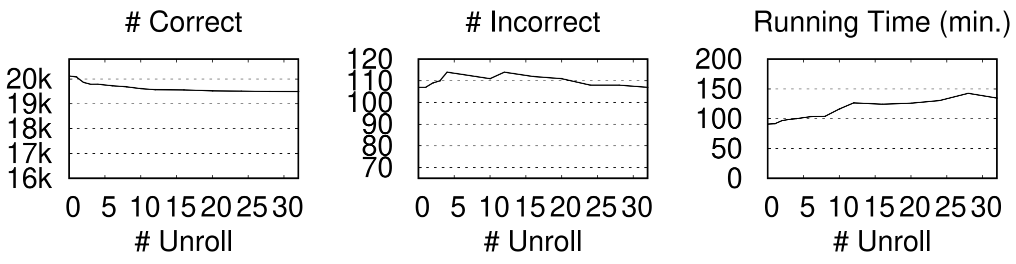


Figure 5.10: Effect of changing the unroll factor when validating LLVM's unit tests.

mismatch in the load and store's bit-widths. The checks in place were not tight enough to catch this bug.

Selected Bug #4: Phi optimization This transformation is not a refinement:

```
BB: %z = phi i1 [1, B1], [%c, B2]
    %r = select %z, i32 %a, i32 %b
```

$$\not\sqsupseteq$$

```
B2: br %c, BB, B3
B3: br BB
BB: %r = phi i32 [%a, B1], [%a, B2], [%b, B3]
```

Here, a `select` instruction is replaced by a conditional branch. This introduces UB when `%c` is **undef** or **poison**. The transformation can be fixed by adding a "$\%c' =$ **freeze**$(\%c)$" instruction and branching on $\%c'$ instead. This bug is partially fixed, due to concerns about regressing the quality of the generated code, with further work ongoing for a full fix.

Performance Running the LLVM unit test suite under Alive2 takes about 2.5 hours on an 8-core Intel workstation. Fig. 5.10 shows the trend of the number of passed tests, refinement failures, and running time when increasing the unroll factor. The number of passed tests decreases with the unroll factor due to timeout or out-of-memory. The wall-clock time increases in a linear manner.

Discussion One might wonder why the LLVM unit tests—which seldom fail on the main LLVM development branch—would be a fruitful place to look for compiler bugs. The answer is that Alive2 is a far more discerning test oracle than are the syntactic oracles, such as the `CHECK:` line in the example at the start of this subsection, that are built into the unit tests. Moreover, Alive2 has the virtue of being consistent: it expects all test cases to follow the same rules.

5.9.3 Updates to the LLVM IR Semantics

When we found ambiguities in the LLVM Language Reference Manual, we initiated discussions in order to clarify the document. Overall, we wrote 8 patches and contributed advice or ideas to 3 patches written by LLVM developers.

Nonnull Attribute While testing our tool, we found a mismatch in the semantics of the **nonnull** attribute between LLVM's documentation and LLVM's code. The documentation specified that passing a null pointer to a **nonnull** argument triggered UB. However, as illustrated below, LLVM adds **nonnull** to a pointer that may be **poison**. This is incorrect because **poison** can be optimized into any value including null.

```
p = gep inbounds q, 1          ⇏     p = gep inbounds q, 1
f(p)                                 f(nonnull p) ; UB if p poison
```

We proposed a new semantics to the LLVM developers, where non-conforming pointers would be considered **poison** rather than UB. This was accepted, and we have contributed patches to fix the docs and the incorrect optimizations.

Nocapture Attribute While implementing the **nocapture** attribute, we found that the definition of pointer capture was not explicit in the LLVM Language Reference Manual. We wrote and shared a draft of patch to the document for the precise definition of pointer capture. After discussions, we could list four cases that capture pointers and updated the patch to explicitly mention them. The patch was accepted, and we have contributed it to the LLVM Reference Manual.

Lifetime Intrinsics We found that LLVM IR's document does not clearly define the semantics of instructions manipulating lifetimes of stack-allocated memory objects (**lifetime.start**, **lifetime.end**). We defined formal semantics for them based on what LLVM's stack coloring algorithm assumes. The validity of the new semantics is checked using LLVM unit tests as well as through manual inspection of LLVM's optimizations. We proposed our new semantics to the LLVM community, and after its acceptance we wrote a patch to the document.

GEP When we started our work, it was not clear whether LLVM's **gep inbounds** operator for pointer arithmetic interpreted its index argument and the base pointer's offset value as signed or unsigned integers, for purposes of computing "inboundedness" of the resulting address. Also, the assumptions that an object cannot be larger than half of the size of its address space, and that no "inbounds" address computation can overflow an unsigned value, needed to be clarified.

Vectors and UB LLVM's **shufflevector** instruction supports permuting two input vectors, returning an output vector that has the same number of elements as its mask argument:

```
; Shuffles two vectors with mask <3, 2, 1, 2>
%v = shufflevector <10, 20>, <30, 40>, <3, 2, 1, 2>
; result: %v = <40, 30, 20, 30>
```

Initially, we believed that when the mask operand contained one or more **undef** values, **poison** elements in the input vectors would be propagated to the output. We reported optimizations that were incorrect under these semantics, and this led to discussions with LLVM developers followed by a decision that **undef** in the mask operand does not result in the propagation of **poison** values.

Other Changes We helped make several clarifications regarding the interaction between **undef** and padding in aggregates. For example, **freeze** has no effect on padding values. Additionally, we clarified that a pointer given to a load or store instruction is not allowed to be a non-deterministic value.

5.9.4 Translation Validation for Applications

Although our focus has been on validating transformations for core elements of LLVM IR, we also wanted to see how Alive2 would work while compiling applications. We chose five single-file benchmarks: bzip2, gzip, oggenc,[5] ph7 2.1.4,[6] and SQLite 3.30.1 amalgamation,[7] and compiled them at the `-O3` optimization level, and using the `-fno-strict-aliasing` flag to disable type-based alias analysis. We extracted pairs of IR files corresponding to the code before and after every optimization pass ran on every function in the code being compiled. We timed out individual invocations of Z3 after one minute and limited its RAM usage to 1 GB.

We batched optimization passes for oggenc, ph7, and SQLite, in order to reduce the total verification time. Instead of calling Alive2 after each optimization,

[5] `http://people.csail.mit.edu/smcc/projects/single-file-programs`
[6] `http://www.symisc.net/downloads/ph7-amalgamation-2001004.zip`
[7] `https://www.sqlite.org/2019/sqlite-amalgamation-3300100.zip`

Prog.	LoC	Pairs	Diff	Time	✓	✗	TO	OOM	Unsup.
bzip2	5.1K	282K	2.2K	1.26	333	10	540	195	1,125
gzip	5.3K	371K	2.6K	1.74	884	4	905	60	754
oggenc	48K	215K	1.8K	1.63	440	4	588	72	663
ph7	43K	1.7M	5.6K	3.15	1,393	28	1,337	35	2,755
SQLite3	141K	3.9M	12.2K	6.37	2,314	38	2,102	100	7,543

Figure 5.11: Results for single-file benchmarks. From left to right, the columns indicate the program name, the number of lines of code, the total number of source/target function pairs (intraprocedural optimizations only), the number of non-identical pairs considered for translation validation, total wall-clock time taken (in hours), pairs successfully validated, violations of refinement, timeouts, out-of-memory conditions, and pairs containing at least one feature unsupported by Alive2.

we batched optimizations between pairs of unsupported optimizations, such that only supported transformations occurred between those two optimizations. Batching, however, incurs a slight risk of hiding bugs, as an optimization may accidentally fix the miscompilation of a previous optimization.

The results of this experiment, run on an 8-core machine, are shown in Fig. 5.11. Quite a few functions in these programs make use of features not yet supported by Alive2; fixing these is a matter of ongoing work. The most common unsupported features are function pointers and missing semantics for some string and I/O library functions. Furthermore, LLVM IR has a long tail of features that have been added over the years, and supporting them requires significant engineering effort.

The last five columns almost add up to the “Diff” column, which is the number of function pairs for which we run Alive2. The remaining few pairs not shown in the table could not be proved correct or incorrect due to Z3 giving up because of incomplete handling of quantifiers.

We manually inspected every failure of refinement observed during this experiment. The bulk of them are due to an incorrect transformation done

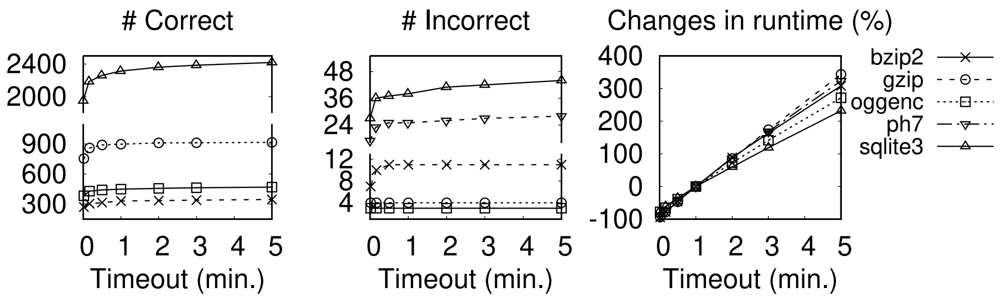


Figure 5.12: Effect of changing the SMT solver timeout for the single-file benchmarks. In the third graph, at the right, running times are normalized to the time taken by that benchmark when the timeout is set to one minute.

by LLVM, where in some cases **select** instructions with Boolean operands are replaced with **and**/**or** instructions. As discussed in Section 3.7, they are fully removed and no longer happens in the later version of LLVM.

5.9.5 Alias Sets

To show that splitting the memory into multiple arrays is beneficial, we gathered statistics of the alias sets in our benchmarks. More than 96% of the dereferenced pointers turned out to be only local or non-local, but not both. This shows that splitting the memory into local and non-local simplifies the memory encoding.

We also counted the number of memory blocks pointers may alias with. Half of the pointers were aliased with just one block. About 80% of the pointers aliased with at most 3 blocks. This is much less than the median number of blocks functions have. The median of the number of memory blocks was $7 \sim 13$ (varying over programs), and only 10% of the functions had fewer than 3 blocks.

Measuring the effect of SMT solver timeout To better understand the impact of the SMT solver's timeout, we ran the single-file benchmarks with timeouts varying from one second to five minutes; the results are shown in Fig. 5.12. While the running time of Alive2 increases approximately linearly with the solver timeout, the number of times Alive2 reached a definitive result plateaus once the timeout reaches one minute. Increasing the solver timeout from one to five minutes increased the number of pairs proved correct and incorrect by less than 5% and 17%, respectively.

5.9.6 Z3 Bugs Found While Developing Alive2

Although finding defects in Z3 was not one of our goals, we did encounter solver bugs while performing this work. We found six soundness bugs, six crashes, and one timeout violation. All but one of these bugs have been fixed; two by us, and the rest by the Z3 developers.

We also hit some performance issues in Z3. We found that one of Z3's internal timer mechanisms was incurring significant overhead because it created a new helper thread on every use of the timer. By patching Z3 to use a thread pool for its `scoped_timer` abstraction, we realized a 20–30% speedup for a collection of Alive2 processes running on a large multicore. We also found an issue in the structural hashing mechanism that was having too many collisions in the hash table. Fixing this issue led to a 3x speedup in the sqlite3 benchmark. Finally, we introduced a new API in Z3 to reset the solver's memory, to reduce memory fragmentation; this fixed an issue where the performance of long Alive2 runs degraded over time. All these patches have been upstreamed.

5.10 Conclusion

Software development is a fundamentally human process, and there are many opportunities for a large, decentralized group of compiler developers, who primarily coordinate using a mailing list and an English-language specification, to introduce subtle defects into their implementation. To assist the LLVM community in creating a coherent semantics for their IR, and making their toolchain respect it, we have created and deployed Alive2, a tool for bounded translation validation for LLVM IR. Running Alive2 over LLVM's unit test suite has revealed 54 bugs of which 28 have been fixed so far. Moreover, there have been a number of cases where the LLVM IR specification was either vague or defective, and we have worked with the community to fix these. The goals are to create a formalization of the intended semantics of LLVM IR, to bring the compiler implementation into conformance with these semantics, and to give the LLVM community tools that it can use to prevent deviations from its specification in the future.

Chapter 6

Conclusion

LLVM IR is a language that is used internally by LLVM to represent programs. Precisely defining the semantics of LLVM IR is crucial for rigorously checking the correctness of compilation.

This thesis proposes (1) a new formal semantics of LLVM IR's undefined behavior and memory model, and (2) an SMT-based translation validation tool for LLVM IR based on the semantics. We show that the old undefined behavior and memory model are inconsistent with the LLVM implementation. To fix the inconsistency of the old undefined behavior model, we suggest removing undef value and introducing a new 'freeze' instruction. To fix the inconsistency of the memory model, we suggest removing transformations that incorrectly assume pointers and integers are equivalent. The new formal semantics of undefined behavior has been adopted by LLVM. Also, the problem of the old memory model was shared with compiler developers, and patches have landed in LLVM to fix it.

To help compiler developers explore our semantics, we propose Alive2, an automatic translation validation tool for LLVM. Since Alive2 does not require any hints from compiler, it does not require any change in LLVM. To make the tool practical, various optimizations in its SMT encoding were necessary. The tool found dozens of miscompilation bugs in LLVM, many of them have already been fixed by compiler developers.

Appendix A

Appendix

A.1 End-to-end miscompilation by both LLVM and GCC

This C program is miscompiled by both GCC and LLVM:[1]

```
// b.c
void f(int *x, int *y) {}

// a.c
#include <stdint.h>
#include <stdio.h>

void f(int *, int *);

int main(void) {
  int a = 0, y[1], x = 0;
  uintptr_t pi = (uintptr_t)&x;
  uintptr_t yi = (uintptr_t)(y + 1);
  int n = pi != yi;

  if (n) {
```

[1]Please note that the program may produce different results on different platforms, since the miscompilation is only observable with some stack layouts. Changing the order of declaration of variables x and y is often sufficient to observe the miscompilation.

```
    a = 100;
    pi = yi;
  }

  if (n) {
    a = 100;
    pi = (uintptr_t)y;
  }

  *(int *)pi = 15;

  printf("a=%d x=%d\n", a, x);

  f(&x, y);

  return 0;
}
```

```
$ clang-5.0 -Wall -O2 a.c b.c ; ./a.out
a=0 x=0
$ gcc-7 -Wall -O2 a.c b.c ; ./a.out
a=0 x=0
```

The result produced by both compilers is incorrect. There are only two possible outcomes, depending on whether `x` and `y` are allocated consecutively (`n` is false) or not (`n` is true):

- Case 1: `n` is true and the program must print

  ```
  a=100 x=0
  ```

- Case 2: `n` is false and the program must print

  ```
  a=0 x=15
  ```

No other output is permitted. In both compilers, the root cause is insufficiently conservative treatment of a pointer derived from an integer. The problem is difficult to solve without throwing away desirable pointer optimizations. This program has been reported in the bug tracking systems for both compilers. The semantics given in this chapter solve this problem, and we have confirmed that the prototype fixes this bug.

A.2 Safe Rust Program Miscompiled by LLVM

This function uses low-level language features, but it is still in the safe subset of Rust. It is miscompiled because of a bug in LLVM's GVN optimization:

```
pub fn test(gp1: &mut usize, gp2: &mut usize, b1: bool, b2: bool)
-> (i32, i32) {
    let mut g = 0;
    let mut c = 0;
    let y = 0;
    let mut x = 7777;
    let mut p = &mut g as *const _;

    {
        let mut q = &mut g;
        let mut r = &mut 8888;

        if b1 {
            p = (&y as *const _).wrapping_offset(1);
        }
        if b2 {
            q = &mut x;
        }

        *gp1 = p as usize + 1234;
        if q as *const _ == p {
            c = 1;
            *gp2 = (q as *const _) as usize + 1234;
            r = q;
        }
        *r = 42;
    }
    return (c, x);
}
```

This function first assigns a reference of `g` to `q`. It then creates a temporary object holding the number `8888` and `r` is assigned a reference to it. Function `wrapping_offset` performs pointer arithmetic that can safely go out of bounds of the base object (equivalent to LLVM's **gep** without **inbounds**). If `b2` is true, `q` is assigned a reference to `x`. Therefore, if the program enters in the following branch as well, `r` is assigned a reference to `x`, and thus the following store through `r` (`*r = 42;`) overwrites the value of `x`.

When called with `b1` and `b2` set to true, the optimized version of this code returns `c = 1, x = 7777`. This outcome is impossible; legal results are `c = 0, x = 7777` and `c = 1, x = 42`.

The miscompilation happens when LLVM's GVN pass exploits the condition of the last `if` statement and incorrectly replaces all uses of `q` within that branch with `p`. After the replacement, `*r` is assumed not to touch `x` because `r` now either contains a reference to the initial temporary object or to one of the references assigned to `p` (based on `g` and `y` only). Therefore, `x = 7777` is constant-propagated to the return statement, which causes the wrong output.

A.3 Coq Formalization and Proof

I formalized the memory model in Coq and proved several key claims of this chapter. The code is available from `https://github.com/snu-sf/llvmtwin-coq`. Note that I omit function calls and returns to simplify the formalization.

A.3.1 Definitions

The memory model is specified in file `Memory.v`. It differs in two ways from the presentation in this chapter: (1) it does not support address spaces for brevity, and (2) memory maintains the last used block id, which is used to create fresh ids on allocation. The number of twin blocks ($|P|$) is set to 3.

Well-formedness of a memory block is defined in `Ir.MemBlock.wf`, and states, e.g., that $|P|$ is always 3, and the size of blocks is larger than zero. Well-formedness of memory is defined in `Ir.Memory.wf` and states, e.g., that all existing memory blocks are well-formed, and alive blocks have no overlapping addresses. Well-formedness of a state is defined in `Ir.Config.wf`, and includes assertions like the program counter is valid, memory is well-formed, etc. We proved that the execution of any instruction preserves the well-formedness of the input state.

The small-step semantics is defined in file`SmallStep.v`. Given an input state, a step can result in one of the following results: (1) success, if the execution of the following instruction was successful and yielded a new state, (2) UB, if the program raised undefined behavior, (3) OOM, if the program raised out-of-memory, and (4) halt, if the program finished.

Two states s and s' are twin with respect to a memory block l iff they are equal except for the configuration of l where the addresses of l in s' (P') are a permutation of those of s (P) and the enabled address is different (i.e., $P_0 \neq P'_0$).

A block's address is observed if a pointer to that block is given as argument to one of the following instructions: **ptrtoint**, **psub**, or **icmp**. Moreover, for the latter two instructions, the other argument must be a physical pointer. A pointer is guessed if it points to an unobserved block.

A.3.2 Proofs

We proved the following theorems in Coq:

Theorem 1 (Twin allocation forbids pointer guessing) *Given two twin states s and s' w.r.t. l, where the next instruction dereferences a guessed pointer to l, the execution of either s or s' triggers UB.*

Two additional theorems are proved that establish the usefulness of twin states:

Theorem 2 (Malloc) malloc *either returns a NULL pointer, or a logical pointer* $\mathsf{Log}(l, o)$*. Moreover, the states yielded by the small-step execution are all twin w.r.t. l.*

Theorem 3 *Given two twin states s and s' w.r.t. l, the small-step execution of s and s' where the next instruction does not dereference a guessed pointer and does not observe block l either (1) halts, triggers UB or OOM, or (2) is successful and the successor states of s are twin with those of s'.*

We now show that certain instructions can be freely moved across (de)allocation functions:

Theorem 4 (Instruction reordering) *Instructions* **icmp eq**, **icmp ule**, **psub**, **inttoptr**, **ptrtoint**, *and* **gep** *can be moved across* **malloc** *and* **free** *in both directions (upward and downward). Moreover, a program P' obtained from P by doing such reordering is equivalent to P.*

Finally we prove sufficient conditions for soundness of GVN:

Theorem 5 (Soundness of GVN for pointers) *The four conditions given in Section 4.4 that state when it is sound to replace a pointer* `p` *with another pointer* `q` *are correct, i.e., the pointer replacement is a refinement.*

Bibliography

[1] WG14, “Indeterminate values and identical representations (defect report #260),” 2004. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_260.htm

[2] K. Memarian and P. Sewell, “Clarifying the C memory object model (revised version of WG14 N2012),” 2016. [Online]. Available: https://www.cl.cam.ac.uk/~pes20/cerberus/notes64-wg14.html

[3] D. Chisnall, J. Matthiesen, K. Memarian, P. Sewell, and R. N. M. Watson, “C memory object and value semantics: the space of de facto and ISO standards,” 2016. [Online]. Available: https://www.cl.cam.ac.uk/~pes20/cerberus/notes30.pdf

[4] K. Memarian and P. Sewell, “N2090: Clarifying pointer provenance (draft defect report or proposal for C2x),” 2016. [Online]. Available: https://www.cl.cam.ac.uk/~pes20/cerberus/n2090.html

[5] J. Kang, C.-K. Hur, W. Mansky, D. Garbuzov, S. Zdancewic, and V. Vafeiadis, “A formal C memory model supporting integer-pointer casts,” in *PLDI*, 2015.

[6] T. L. Project, "LLVM language reference manual," 2021. [Online]. Available: https://llvm.org/docs/LangRef.html

[7] 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.*, vol. 13, no. 4, p. 451–490, Oct. 1991.

[8] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *PLDI*, 2011.

[9] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *PLDI*, 2014.

[10] V. Livinskii, D. Babokin, and J. Regehr, "Random testing for C and C++ compilers with YARPGen," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020.

[11] Q. Zhang, C. Sun, and Z. Su, "Skeletal program enumeration for rigorous compiler testing," in *PLDI*, 2017.

[12] J. Lee, Y. Kim, Y. Song, C.-K. Hur, S. Das, D. Majnemer, J. Regehr, and N. P. Lopes, "Taming undefined behavior in LLVM," in *PLDI*, 2017.

[13] J. Lee, C.-K. Hur, R. Jung, Z. Liu, J. Regehr, and N. P. Lopes, "Reconciling high-level optimizations and low-level code in LLVM," *Proc. of the ACM on Programming Languages*, vol. 2, no. OOPSLA, Nov. 2018.

[14] N. P. Lopes, J. Lee, C.-K. Hur, Z. Liu, and J. Regehr, "Alive2: Bounded translation validation for LLVM," in *PLDI*, 2021.

[15] J. Lee, D. Kim, C.-K. Hur, and N. P. Lopes, "An SMT encoding of LLVM's memory model for bounded translation validation," in *CAV*, 2021.

[16] M. Braun, S. Buchwald, S. Hack, R. Leißa, C. Mallon, and A. Zwinkau, "Simple and efficient construction of static single assignment form," in *CC*, 2013.

[17] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. K. Zadeck, "An efficient method of computing static single assignment form," in *POPL '89*, 1989.

[18] K. J. Ottenstein, R. A. Ballance, and A. B. MacCabe, "The program dependence web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages," in *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, ser. PLDI '90. New York, NY, USA: Association for Computing Machinery, 1990, p. 257–271.

[19] C. S. Ananian, "The static single information form," 1999.

[20] K. Knobe and V. Sarkar, "Array SSA form and its use in parallelization," in *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '98. New York, NY, USA: Association for Computing Machinery, 1998, p. 107–120.

[21] D. Novillo, "Memory SSA – a unified approach for sparsely representing memory operations," in *Proc. of the GCC Developers' Summit*, 2007.

[22] "MemorySSA." [Online]. Available: https://llvm.org/docs/MemorySSA.html

[23] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: Scaling compiler infrastructure for domain specific computation," in *2021*

IEEE/ACM International Symposium on Code Generation and Optimization (CGO), 2021, pp. 2–14.

[24] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic, "Formalizing the LLVM intermediate representation for verified program transformations," in *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 427–440.

[25] L. Li and E. L. Gunter, "K-LLVM: A relatively complete semantics of LLVM IR," in *ECOOP*, 2020.

[26] G. Rosu, "K: A semantic framework for programming languages and formal analysis tools," in *Dependable Software Systems Engineering*, 2017.

[27] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, pp. 107–115, Jul. 2009.

[28] G. Barthe, D. Demange, and D. Pichardie, "Formal verification of an SSA-based middle-end for CompCert," *ACM Trans. Program. Lang. Syst.*, vol. 36, no. 1, Mar. 2014.

[29] J. Kang, C.-K. Hur, O. Lahav, V. Vafeiadis, and D. Dreyer, "A promising semantics for relaxed-memory concurrency," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 175–189.

[30] S.-H. Lee, M. Cho, A. Podkopaev, S. Chakraborty, C.-K. Hur, O. Lahav, and V. Vafeiadis, "Promising 2.0: Global optimizations in relaxed memory concurrency," in *Proceedings of the 41st ACM SIGPLAN Conference on*

Programming Language Design and Implementation, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 362–376.

[31] M. Cho, S.-H. Lee, C.-K. Hur, and O. Lahav, "Modular data-race-freedom guarantees in the promising semantics," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 867–882.

[32] D. Bogdanas and G. Roşu, "K-Java: A complete semantics of java," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 445–456.

[33] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, "RustBelt: Securing the foundations of the Rust programming language," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, Dec. 2017.

[34] R. Jung, H.-H. Dang, J. Kang, and D. Dreyer, "Stacked borrows: an aliasing model for Rust," in *POPL*, 2020.

[35] S. Dasgupta, D. Park, T. Kasampalis, V. S. Adve, and G. Roşu, "A complete formal semantics of x86-64 user-level instruction set architecture," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 1133–1148.

[36] A. Armstrong, T. Bauereiss, B. Campbell, A. Reid, K. E. Gray, R. M. Norton, P. Mundkur, M. Wassell, J. French, C. Pulte, S. Flur, I. Stark, N. Krishnaswami, and P. Sewell, "ISA semantics for ARMv8-a, RISC-v,

and CHERI-MIPS," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019.

[37] F. Besson, S. Blazy, and P. Wilke, "A concrete memory model for CompCert," in *ITP*, 2015.

[38] C. Pulte, J. Pichon-Pharabod, J. Kang, S. H. Lee, and C. Hur, "Promising-ARM/RISC-V: A simpler and faster operational concurrency model," in *PLDI 2019*. ACM, 2019, pp. 1–15.

[39] K. Cho, S.-H. Lee, A. Raad, and J. Kang, "Revamping hardware persistency models: view-based and axiomatic persistency models for Intel-x86 and Armv8," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021. Association for Computing Machinery, 2021.

[40] S. Chakraborty and V. Vafeiadis, "Formalizing the concurrency semantics of an LLVM fragment," in *CGO*, 2017.

[41] L.-y. Xia, Y. Zakowski, P. He, C.-K. Hur, G. Malecha, B. C. Pierce, and S. Zdancewic, "Interaction Trees: Representing recursive and impure programs in Coq," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, Dec. 2019.

[42] J. Kang, Y. Kim, C.-K. Hur, D. Dreyer, and V. Vafeiadis, "Lightweight verification of separate compilation," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 178–190.

[43] G. Neis, C.-K. Hur, J.-O. Kaiser, C. McLaughlin, D. Dreyer, and V. Vafeiadis, "Pilsner: A compositionally verified compiler for a higher-order imperative language," *SIGPLAN Not.*, vol. 50, no. 9, p. 166–178, Aug. 2015.

[44] "clang-11.0.0 miscompiles SQLite." [Online]. Available: https://sqlite.org/forum/forumpost/296b7c1e02

[45] "A bug in gcc miscompiling git's diff.c." [Online]. Available: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=93908

[46] "CVE-2020-16040." [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2020-16040

[47] "CVE-2019-11707." [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2019-11707

[48] F. Brown, J. Renner, A. Nötzli, S. Lerner, H. Shacham, and D. Stefan, "Towards a verified range analysis for JavaScript JITs," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 135–150.

[49] S. Bauer, P. Cuoq, and J. Regehr, "Deniable backdoors using compiler bugs," 2015.

[50] "[llvm-dev] a bug related with undef value when bootstrap MemorySSA.cpp." [Online]. Available: https://lists.llvm.org/pipermail/llvm-dev/2017-July/115497.html

[51] M. Marcozzi, Q. Tang, A. F. Donaldson, and C. Cadar, “Compiler fuzzing: How much does it matter?” *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019.

[52] “The Coq Proof Assistant.” [Online]. Available: https://coq.inria.fr/

[53] E. Mullen, D. Zuniga, Z. Tatlock, and D. Grossman, “Verified peephole optimizations for CompCert,” in *PLDI*, 2016.

[54] N. Courant and X. Leroy, “Verified code generation for the polyhedral model,” in *POPL*, 2021.

[55] G. Barthe, D. Demange, and D. Pichardie, “Formal verification of an SSA-based middle-end for CompCert,” *ACM Trans. Program. Lang. Syst.*, vol. 36, no. 1, Mar. 2014.

[56] S. Lerner, T. Millstein, and C. Chambers, “Automatically proving the correctness of compiler optimizations,” in *PLDI*, 2003.

[57] S. Lerner, T. Millstein, E. Rice, and C. Chambers, “Automated soundness proofs for dataflow analyses and transformations via local rules,” in *POPL*, 2005.

[58] S. Kundu, Z. Tatlock, and S. Lerner, “Proving optimizations correct using parameterized program equivalence,” in *PLDI*, 2009.

[59] L. Zuck, A. Pnueli, B. Goldberg, C. Barrett, Y. Fang, and Y. Hu, “Translation and run-time validation of loop transformations,” *Form. Methods Syst. Des.*, vol. 27, no. 3, pp. 335–360, Nov. 2005.

[60] N. P. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr, “Provably correct peephole optimizations with Alive,” in *Proceedings of the 36th ACM*

SIGPLAN Conference on Programming Language Design and Implementation, ser. PLDI '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 22–32.

[61] J. Lee, C.-K. Hur, and N. P. Lopes, "AliveInLean: A verified LLVM peephole optimization verifier," in *CAV*, 2019.

[62] J. L. Newcomb, A. Adams, S. Johnson, R. Bodik, and S. Kamil, "Verifying and improving halide's term rewriting system with program synthesis," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020.

[63] N. P. Lopes and J. Monteiro, "Automatic equivalence checking of programs with uninterpreted functions and integer arithmetic," *Int. J. Softw. Tools Technol. Transf.*, vol. 18, no. 4, pp. 359–374, Aug. 2016.

[64] A. Pnueli, M. Siegel, and E. Singerman, "Translation validation," in *TACAS*, 1998.

[65] M. C. Rinard and D. Marinov, "Credible compilation with pointers," in *RTRV*, 1999.

[66] G. C. Necula, "Translation validation for an optimizing compiler," in *PLDI*, 2000.

[67] A. Kanade, A. Sanyal, and U. P. Khedker, "Validation of GCC optimizers through trace generation," *SP&E*, vol. 39, no. 6, pp. 611–639, Apr. 2009.

[68] K. S. Namjoshi and L. D. Zuck, "Witnessing program transformations," in *SAS*, 2013.

[69] J. Kang, Y. Kim, Y. Song, J. Lee, S. Park, M. D. Shin, Y. Kim, S. Cho, J. Choi, C.-K. Hur, and K. Yi, "Crellvm: Verified credible compilation for LLVM," in *PLDI*, 2018.

[70] A. Zaks and A. Pnueli, "CoVaC: Compiler validation by program analysis of the cross-product," in *FM*, 2008.

[71] S. Gupta, A. Rose, and S. Bansal, "Counterexample-guided correlation algorithm for translation validation," in *OOPSLA*, 2020.

[72] R. Sharma, E. Schkufza, B. Churchill, and A. Aiken, "Data-driven equivalence checking," in *OOPSLA*, 2013.

[73] M. Dahiya and S. Bansal, "Black-box equivalence checking across compiler optimizations," in *APLAS*, 2017.

[74] B. Churchill, O. Padon, R. Sharma, and A. Aiken, "Semantic program alignment for equivalence checking," in *PLDI*, 2019.

[75] C. Barrett, Y. Fang, B. Goldberg, Y. Hu, A. Pnueli, and L. Zuck, "TVOC: A translation validator for optimizing compilers," in *CAV*, 2005.

[76] J.-B. Tristan, P. Govereau, and J. G. Morrisett, "Evaluating value-graph translation validation for LLVM," in *PLDI*, 2011.

[77] M. Stepp, R. Tate, and S. Lerner, "Equality-based translation validator for LLVM," in *CAV*, 2011.

[78] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, "Egg: Fast and extensible equality saturation," *Proc. ACM Program. Lang.*, vol. 5, no. POPL, Jan. 2021.

[79] J. Chen, J. Wei, Y. Feng, O. Bastani, and I. Dillig, "Relational verification using reinforcement learning," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019.

[80] V. Klebanov, P. Rümmer, and M. Ulbrich, "Automating regression verification of pointer programs by predicate abstraction," *Form. Methods Syst. Des.*, vol. 52, no. 3, pp. 229–259, Jun. 2018.

[81] C. Hawblitzel, M. Kawaguchi, S. K. Lahiri, and H. Rebêlo, "Towards modularly comparing programs using automated theorem provers," in *CADE*, 2013.

[82] T. Wood, S. Drossopolou, S. K. Lahiri, and S. Eisenbach, "Modular verification of procedure equivalence in the presence of memory allocation," in *ESOP*, 2017.

[83] S. Dasgupta, S. Dinesh, D. Venkatesh, V. S. Adve, and C. W. Fletcher, "Scalable validation of binary lifters," in *PLDI*, 2020.

[84] J.-B. Tristan and X. Leroy, "Verified validation of lazy code motion," in *PLDI*, 2009.

[85] R. Leviathan and A. Pnueli, "Validating software pipelining optimizations," in *CASES*, 2002.

[86] J.-B. Tristan and X. Leroy, "A simple, verified validator for software pipelining," in *POPL*, 2010.

[87] K. Shashidhar, M. Bruynooghe, F. Catthoor, and G. Janssens, "Geometric Model Checking: An automatic verification technique for loop and data reuse transformations," *ENTCS*, vol. 65, no. 2, 2002.

[88] T. Sewell, M. Myreen, and G. Klein, "Translation validation for a verified OS kernel," in *PLDI*, 2013.

[89] M. Dahiya and S. Bansal, "Modeling undefined behaviour semantics for checking equivalence across compiler optimizations," in *HVC*, 2017.

[90] M. Braun, S. Buchwald, and A. Zwinkau, "Firm—a graph-based intermediate representation," Karlsruhe Institute of Technology, Tech. Rep. 35, 2011. [Online]. Available: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000025470

[91] E. Mullen, D. Zuniga, Z. Tatlock, and D. Grossman, "Verified peephole optimizations for CompCert," in *PLDI*, 2016.

[92] K. Memarian, J. Matthiesen, J. Lingard, K. Nienhuis, D. Chisnall, R. N. Watson, and P. Sewell, "Into the depths of C: elaborating the de facto standards," in *PLDI*, Jun. 2016.

[93] C. Hathhorn, C. Ellison, and G. Roşu, "Defining the undefinedness of C," in *PLDI*, 2015.

[94] R. Krebbers and F. Wiedijk, "A typed C11 semantics for interactive theorem proving," in *CPP*, 2015, pp. 15–27.

[95] R. Krebbers, "Aliasing restrictions of C11 formalized in Coq," in *CPP*, 2013.

[96] X. Leroy and S. Blazy, "Formal verification of a C-like memory model and its uses for verifying program transformations," *Journal of Automated Reasoning*, vol. 41, no. 1, pp. 1–31, Jul 2008.

[97] J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell, "CompCertTSO: A verified compiler for relaxed-memory concurrency," *J. ACM*, vol. 60, no. 3, pp. 22:1–22:50, Jun. 2013.

[98] F. Besson, S. Blazy, and P. Wilke, "CompCertS: A memory-aware verified C compiler using pointer as integer semantics," in *ITP*, 2017.

[99] ——, "A verified CompCert front-end for a memory model supporting pointer arithmetic and uninitialised data," *Journal of Automated Reasoning*, Nov 2017.

[100] ——, "A concrete memory model for CompCert," in *ITP*, 2015.

[101] ——, "A precise and abstract memory model for C using symbolic values," in *APLAS*, 2014.

[102] S. Chakraborty and V. Vafeiadis, "Formalizing the concurrency semantics of an LLVM fragment," in *CGO*, 2017.

[103] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic, "Formal verification of SSA-based optimizations for LLVM," in *PLDI*, 2013.

[104] K. S. Namjoshi, G. Tagliabue, and L. D. Zuck, "A witnessing compiler: A proof of concept," in *RV*, 2013.

[105] R. E. Bryant, S. K. Lahiri, and S. A. Seshia, "Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions," in *CAV*, 2002.

[106] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS*, 2008.

초록

중간언어는 컴파일러가 변환 중인 프로그램을 내부적으로 나타내기 위해 사용하는 언어이다. 소스 프로그램을 중간언어로 번역할 때는 소스 언어의 명세로부터 얻을 수 있는 여러 고급 정보들을 잘 보존해야 하는데, 그 이유는 컴파일러 최적화가 이 정보들을 활용할 수 있도록 하기 위해서이다. 따라서 중간언어는 이러한 고급 정보를 표현할 수 있는 문법들을 가지고 있다.

기존의 LLVM은 중간언어로 번역된 프로그램에 표현된 고급 정보들의 의미가 무엇인지를 엄밀하게 정의하고 있지 않았었다. 이것은 고급 정보를 나타내는 구문의 의미를 컴파일러 최적화 마다 서로 다르게 이해하는 결과를 낳았고, 이들 간의 나쁜 상호작용은 여러 컴파일러 버그들의 원인이 되었다. 이 문제를 해결하기 위해서는 먼저 중간언어에 있는 고급 정보의 의미를 먼저 정확하고 엄밀하게 정의해야 한다. 그 다음에는, LLVM 내의 컴파일러 최적화들이 해당 정의를 기준으로 옳은지를 하나하나 엄밀하게 체크해야 한다. 하지만 LLVM은 빠르게 진화하는 방대한 소프트웨어이기 때문에 효과적으로 이 문제를 해결하기가 쉽지 않다.

이 학위 논문에서는 (1) LLVM 컴파일러의 중간언어 의미에 존재하던 치명적인 문제들을 해결하는 새로운 형식 의미를 제안하고 (2) 그에 기반한 컴파일러 최적화 번역 검증 (translation validation) 프레임워크를 소개한다. 우리는 LLVM 컴파일러 중간언어의 정의되지 않은 행동 (undefined behavior) 과 메모리 모델에 심각한 문제가 있음을 보이며, 이것을 해결한 새로운 형식 의미를 제안한다. 둘째로, 우리는 자동 번역 검증 프레임워크 Alive2를 개발하였다. 제안된 번역 검증 프레임워크는 자동 명제 증명기 (SMT solver)를 사용해서 최적화의 옳음성을 수학적으로 엄밀하게 확인하며 컴파일러로부터의 도움이 필요하지 않다.

우리가 제안한 정의되지 않은 행동의 형식 의미는 LLVM 컴파일러에 공식적으로 채택되었다. 본 논문에서 제안한 ‘freeze’ 명령어는 LLVM 10.0에 도입 되었으며,

공식 문서는 우리의 형식 의미를 사용하도록 업데이트 되었다. 또한, 기존의 LLVM 메모리 모델에서 찾은 치명적인 문제들은 컴파일러 개발자들에게 공유되었으며 여러 토론을 야기했고 이 문제를 해결하기 위한 패치가 LLVM 컴파일러에 적용되었다. 우리는 Alive2 를 이용해 수십개의 컴파일러 버그를 찾아낼 수 있었으며, 이 도구는 현재 LLVM 컴파일러 개발자들에 의해 코드 리뷰 과정에서 사용되고 있다.

주요어: 컴파일러, 프로그래밍 언어, 컴파일러 중간 언어, 형식 언어 의미, 컴파일러 검증, 자동 검증, 번역 검증, 메모리 모델

학번: 2016-21227