Provably Correct Peephole Optimizations with Alive

Nuno P. Lopes
Microsoft Research, UK
nlopes@microsoft.com

David Menendez Santosh Nagarakatte
Rutgers University, USA
{davemm,santosh.nagarakatte}@cs.rutgers.edu

John Regehr
University of Utah, USA
regehr@cs.utah.edu

Abstract

Compilers should not miscompile. Our work addresses problems in developing peephole optimizations that perform local rewriting to improve the efficiency of LLVM code. These optimizations are individually difficult to get right, particularly in the presence of undefined behavior; taken together they represent a persistent source of bugs. This paper presents Alive, a domain-specific language for writing optimizations and for automatically either proving them correct or else generating counterexamples. Furthermore, Alive can be automatically translated into C++ code that is suitable for inclusion in an LLVM optimization pass. Alive is based on an attempt to balance usability and formal methods; for example, it captures—but largely hides—the detailed semantics of three different kinds of undefined behavior in LLVM. We have translated more than 300 LLVM optimizations into Alive and, in the process, found that eight of them were wrong.

Categories and Subject Descriptors D.2.4 [*Programming Languages*]: Software/Program Verification; D.3.4 [*Programming Languages*]: Processors—Compilers; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords Compiler Verification, Peephole Optimization, Alive

1. Introduction

Compiler optimizations should be efficient, effective, and correct—but meeting all of these goals is difficult. In practice, whereas efficiency and effectiveness are relatively easy to quantify, correctness is not. Incorrect compiler optimizations can remain latent for long periods of time; the resulting problems are subtle and difficult to diagnose since the incorrectness is introduced at a level of abstraction lower than the one where software developers typically work. Although mainstream compilers use well-known algorithms, bugs arise due to misunderstandings of the semantics, incomplete reasoning about boundary conditions, and errors in the implementation of the algorithms.

Random testing [15, 23, 37] is one approach to improve the correctness of compilers; it has been shown to be effective, but of course testing misses bugs. A stronger form of insurance against compiler bugs can be provided by a proof that the compiler is correct (compiler verification) or a proof that a particular compilation was correct (translation validation). For example, CompCert [17] uses a hybrid of the two approaches. Unfortunately, creating CompCert required several person-years of proof engineering and the resulting tool does not provide a good value proposition for many real-world use cases: it implements a subset of C, optimizes only lightly, and does not yet support x86-64 or the increasingly important vector extensions to x86 and ARM. In contrast, production compilers are constantly improved to support new language standards and to obtain the best possible performance on emerging architectures.

This paper presents Alive: a new language and tool for developing correct LLVM optimizations. Alive aims for a design point that is both practical and formal; it allows compiler writers to specify peephole optimizations for LLVM's intermediate representation (IR), it automatically proves them correct with the help of satisfiability modulo theory (SMT) solvers (or provides a counterexample), and it automatically generates C++ code that is similar to hand-written peephole optimizations such as those found in LLVM's instruction combiner (InstCombine) pass. InstCombine transformations perform numerous algebraic simplifications that improve efficiency, enable other optimizations, and canonicalize LLVM code.

Alive is inspired by previous research on domain specific languages for easing compiler development such as Gospel [36], Rhodium [16], PEC [13], Broadway [9], and Lola [21]. Alive's main contribution to the state of the art is providing a usable formal methods tool based on the semantics of LLVM IR, with support for automated correctness proofs in the presence of LLVM's three kinds of undefined behavior, and with support for code generation.

While testing LLVM using Csmith, we found InstCombine to be the single buggiest file [37]; it was subsequently split into multiple files totaling about 15,600 SLOC. A similar pass, InstSimplify, contains about 2,500 more SLOC. An example InstCombine transformation takes $(x \oplus -1) + C$ and turns it into $(C - 1) - x$ where x is a variable, $\oplus$ is exclusive or, and C is an arbitrary constant as wide as x. If C is 3333, the LLVM input to this InstCombine transformation would look like this:

```
%1 = xor i32 %x, -1
%2 = add i32 %1, 3333
```

and the optimized code:

```
%2 = sub i32 3332, %x
```

The LLVM code specifying this transformation[1] is 160 bytes of C++, excluding comments. In Alive it is:

```
%1 = xor %x, -1
%2 = add %1, C
  =>
%2 = sub C-1, %x
```

[1] `http://llvm.org/viewvc/llvm-project/llvm/tags/RELEASE_350/final/lib/Transforms/InstCombine/InstCombineAddSub.cpp?view=markup#l1148`

$$
\begin{array}{rcl}
\mathit{prog} & ::= & \mathit{pre}\ \mathit{nl}\ \mathit{stmt} \implies \mathit{stmt} \\
\mathit{stmt} & ::= & \mathit{stmt}\ \mathit{nl}\ \mathit{stmt} \mid \mathit{reg} = \mathit{inst} \mid \mathit{reg} = \mathit{op} \mid \\
 & & \mathbf{store}\ \mathit{op}, \mathit{op} \mid \mathbf{unreachable} \\
\mathit{inst} & ::= & \mathit{binop}\ \overline{\mathit{attr}}\ \mathit{op}, \mathit{op} \mid \mathit{conv}\ \mathit{op} \mid \\
 & & \mathbf{select}\ \mathit{op}, \mathit{op}, \mathit{op} \mid \mathbf{icmp}\ \overline{\mathit{cond}}, \mathit{op}, \mathit{op} \mid \\
 & & \mathbf{alloca}\ \mathit{typ}, \mathit{constant} \mid \mathbf{bitcast}\ \mathit{op} \mid \\
 & & \mathbf{inttoptr}\ \mathit{op} \mid \mathbf{ptrtoint}\ \mathit{op} \mid \\
 & & \mathbf{getelementptr}\ \mathit{op}, \ldots, \mathit{op} \mid \mathbf{load}\ \mathit{op} \\
\mathit{cond} & ::= & \mathbf{eq} \mid \mathbf{ne} \mid \mathbf{ugt} \mid \mathbf{uge} \mid \mathbf{ult} \mid \\
 & & \mathbf{ule} \mid \mathbf{sgt} \mid \mathbf{sge} \mid \mathbf{slt} \mid \mathbf{sle} \\
\mathit{typ} & ::= & \mathbf{i}\mathit{sz} \mid \mathit{typ}* \mid [\mathit{sz} \times \mathit{typ}] \\
\mathit{binop} & ::= & \mathbf{add} \mid \mathbf{sub} \mid \mathbf{mul} \mid \mathbf{udiv} \mid \mathbf{sdiv} \mid \\
 & & \mathbf{urem} \mid \mathbf{srem} \mid \mathbf{shl} \mid \mathbf{lshr} \mid \mathbf{ashr} \mid \\
 & & \mathbf{and} \mid \mathbf{or} \mid \mathbf{xor} \\
\mathit{attr} & ::= & \mathbf{nsw} \mid \mathbf{nuw} \mid \mathbf{exact} \\
\mathit{op} & ::= & \mathit{reg} \mid \mathit{constant} \mid \mathbf{undef} \\
\mathit{conv} & ::= & \mathbf{zext} \mid \mathbf{sext} \mid \mathbf{trunc}
\end{array}
$$

Figure 1. Partial Alive syntax. Types include arbitrary bitwidth integers, pointers $typ*$, and arrays $[sz \times typ]$ that have a statically-known size sz.

The Alive specification is less than one third the size of the LLVM C++ code. Moreover, it was designed to resemble—both syntactically and semantically—the LLVM transformation that it describes. The Alive code is, we claim, much easier to understand, in addition to being verifiable by the Alive tool chain. This transformation illustrates two forms of abstraction supported by Alive: abstraction over choice of a compile-time constant and abstraction over bitwidth.

So far Alive has helped us discover eight previously unknown bugs in the LLVM InstCombine transformations. Furthermore, we have prevented dozens of bugs from getting into LLVM by monitoring the various InstCombine patches as they were committed to the LLVM subversion repository. Several LLVM developers are currently using the Alive prototype to check their InstCombine transformations. Alive is open source.[2]

2. The Alive Language

We designed Alive to resemble the LLVM intermediate representation (IR) because our user base—the LLVM developers—is already familiar with it. Alive's most important features include its abstraction over choice of constants, over the bitwidths of operands (Sections 2.2 and 3.2), and over LLVM's instruction attributes that control undefined behavior (Sections 2.4 and 3.4).

2.1 Syntax

Figure 1 gives the syntax of Alive. An Alive transformation has the form $A \implies B$, where A is the *source template* and B is the *target template*. Additionally, a transformation may include a precondition. Since Alive's representation, like LLVM's, is based on directed graphs of instructions in SSA format [5], the ordering of non-dependent instructions, and the presence of interleaved instructions not part of a template, are irrelevant.

Alive implements a subset of LLVM's integer and pointer instructions; since InstCombine does not modify control flow, Alive does not support branches.[3] In contrast to LLVM, Alive provides an explicit assignment instruction for copying temporaries (i.e., $\%tmp1 = \%tmp2$). Alive supports LLVM's `nsw`, `nuw`, and

[2] The latest version of Alive can be found at `https://github.com/nunoplopes/alive`.

[3] An experimental version of Alive supports branches, but does not yet support indirect branches and loops.

```
Pre: C1 & C2 == 0 && MaskedValueIsZero(%V, ~C1)
%t0 = or  %B,  %V
%t1 = and %t0, C1
%t2 = and %B,  C2
%R  = or  %t1, %t2
  =>
%R  = and %t0, (C1 | C2)
```

Figure 2. An example illustrating many of Alive's features. $((B \vee V) \wedge C1) \vee (B \wedge C2)$ can be transformed to $(B \vee V) \wedge (C1 \vee C2)$ when $C1 \wedge C2 = 0$ and when the predicate $MaskedValueIsZero(V, \neg C1)$ is true, indicating that an LLVM dataflow analysis has concluded that $V \wedge \neg C1 = 0$. $\%B$ and $\%V$ are input variables. $C1$ and $C2$ are constants. $\%t0$, $\%t1$, and $\%t2$ are temporaries. This transformation is rooted at $\%R$.

`exact` instruction attributes that weaken the behavior of integer instructions by adding undefined behaviors.

Scoping. The source and target templates must have a common *root variable* that is the root of the respective graphs. The remaining variables are either inputs to the transformation or else temporary variables produced by instructions in the source or target template. Inputs are visible throughout the source and target templates. Temporaries defined in the source template are in scope for the precondition, the target, and the remaining part of the source from the point of definition. Temporaries declared in the target are in scope for the remainder of the target. To help catch errors, every temporary in the source template must be used in a later source instruction or be overwritten in the target, and all target instructions must be used in a later target instruction or overwrite a source instruction.

Constant expressions. To allow algebraic simplifications and constant folding, Alive includes a language for constant expressions. A constant expression may be a literal, an abstract constant (e.g., `C` in the example on the previous page), or a unary or binary operator applied to one or two constant expressions. The operators include signed and unsigned arithmetic operators and bitwise logical operators. Alive also supports functions on constant expressions. Built-in functions include type conversions and mathematical and bitvector utilities (e.g., abs(), umax(), width()).

2.2 Type System

Alive supports a subset of LLVM's types. The types in Alive are a union of integer types ($\mathcal{I} = \{\texttt{i1}, \texttt{i2}, \texttt{i3}, \ldots\}$ for bitwidth 1,2,3,...), pointer types ($\mathcal{P} = \{t* \mid t \in \mathcal{T}\}$), array types ($\mathcal{A} = \{[n \text{ x } t] \mid n \in \mathbb{N} \wedge t \in \mathcal{T}\}$), and the void type. LLVM also defines the set of first-class types ($\mathcal{FC} = \mathcal{I} \cup \mathcal{P}$), which are the types that can be the result of an instruction. The set of all types is therefore $\mathcal{T} = \mathcal{FC} \cup \mathcal{A} \cup \{\texttt{void}\}$. Alive does not yet support floating point, aggregate types such as records and vectors, and labels.

Unlike LLVM, variables in Alive can be implicitly typed and do not need to have fixed bitwidth. An Alive transformation is polymorphic for all the types that satisfy the constraints imposed by the instructions. The Alive framework automatically checks the correctness of the transformation for all feasible types. A transformation can optionally provide types for the input operands of the instructions, which then constrain the set of feasible types. The typing rules for the Alive language are shown in Figure 3.

2.3 Built-In Predicates

Peephole optimizations frequently make use of the results of dataflow analyses. Alive makes these results available using a collection of built-in predicates such as isPowerOf2(), MaskedValueIsZero(), and WillNotOverflowSignedAdd(). The analyses producing these results are trusted by Alive: verifying their correctness

$$\frac{}{\Gamma, \%\mathtt{x} : t \vdash \%\mathtt{x} : t}\text{ var} \qquad \frac{\Gamma \vdash \%\mathtt{x} : t \quad \Gamma \vdash \%\mathtt{y} : t \quad t \in \mathcal{I}}{\Gamma \vdash \mathit{binop}\ \%\mathtt{x},\ \%\mathtt{y} : t}\text{ binop} \qquad \frac{\Gamma \vdash \%\mathtt{x} : t \quad t <: t'}{\Gamma \vdash \mathit{extend}\ \%\mathtt{x} : t'}\text{ extend} \qquad \frac{\Gamma \vdash \%\mathtt{x} : t \quad t' <: t}{\Gamma \vdash \mathtt{trunc}\ \%\mathtt{x} : t'}\text{ trunc}$$

$$\frac{\Gamma \vdash \%\mathtt{x} : t \quad t \in \mathcal{I} \quad t'* \in \mathcal{P}}{\Gamma \vdash \mathtt{inttoptr}\ \%\mathtt{x} : t'*}\text{ inttoptr} \qquad \frac{\Gamma \vdash \%\mathtt{x} : t* \quad t' \in \mathcal{I}}{\Gamma \vdash \mathtt{ptrtoint}\ \%\mathtt{x} : t'}\text{ ptrtoint} \qquad \frac{\Gamma \vdash \%\mathtt{x} : t \quad \Gamma \vdash \%\mathtt{y} : t \quad t \in \mathcal{I} \cup \mathcal{P}}{\Gamma \vdash \mathtt{icmp}\ \mathit{cond}\ \%\mathtt{x},\ \%\mathtt{y} : \mathtt{i1}}\text{ icmp}$$

$$\frac{\Gamma \vdash \%\mathtt{c} : \mathtt{i1} \quad \Gamma \vdash \%\mathtt{x} : t \quad \Gamma \vdash \%\mathtt{y} : t \quad t \in \mathcal{FC}}{\Gamma \vdash \mathtt{select}\ \%\mathtt{c},\ \%\mathtt{x},\ \%\mathtt{y} : t}\text{ select} \qquad \frac{\Gamma \vdash \%\mathtt{x} : t \quad t, t' \in \mathcal{FC} \quad \mathrm{width}(t) = \mathrm{width}(t')}{\Gamma \vdash \mathtt{bitcast}\ \%\mathtt{x} : t'}\text{ bitcast}$$

$$\frac{t \in \mathcal{FC}}{\Gamma \vdash \mathtt{alloca}\ t : t*}\text{ alloca} \qquad \frac{\Gamma \vdash \%\mathtt{x} : t \quad \overbrace{@\ldots@}^{n} t : t' \quad \Gamma \vdash \%\mathtt{y}_{1\ldots n} : \mathcal{I}}{\Gamma \vdash \mathtt{getelementptr}\ \%\mathtt{x}, \%\mathtt{y}_1, \ldots, \%\mathtt{y}_n : t'*}\text{ gep} \qquad \frac{\Gamma \vdash \%\mathtt{x} : t* \quad t \in \mathcal{FC}}{\Gamma \vdash \mathtt{load}\ \%\mathtt{x} : t}\text{ load}$$

$$\frac{\Gamma \vdash \%\mathtt{x} : t \quad t \in \mathcal{FC} \quad \Gamma \vdash \%\mathtt{y} : t*}{\Gamma \vdash \mathtt{store}\ \%\mathtt{x}, \%\mathtt{y} : \mathtt{void}}\text{ store} \qquad \frac{}{\Gamma \vdash \mathtt{unreachable} : \mathtt{void}}\text{ unreach} \qquad \frac{}{@t* : t}\text{ pointer} \qquad \frac{}{@[n \text{ x } t] : t}\text{ array}$$

Figure 3. Typing rules of the Alive language. The bitwidth of a type is given by the function $\mathrm{width}(.)$ (e.g., $\mathrm{width}(\mathtt{i3}) = 3$ and $\mathrm{width}(i5*)$ is the pointer size, say 32 bits). A type t is smaller than a type t', written $t <: t'$, if both are integer types and the bitwidth of t is smaller than that of t' (i.e., $t <: t'$ iff $t, t' \in \mathcal{I} \wedge \mathrm{width}(t) < \mathrm{width}(t')$). The extension operations are represented by $extend \in \{\mathtt{zext}, \mathtt{sext}\}$. The type dereferencing operator is denoted by @.

Instruction	Definedness Constraint
sdiv a, b	$b \neq 0 \wedge (a \neq \mathit{INT\_MIN} \vee b \neq -1)$
udiv a, b	$b \neq 0$
srem a, b	$b \neq 0 \wedge (a \neq \mathit{INT\_MIN} \vee b \neq -1)$
urem a, b	$b \neq 0$
shl a, b	$b <_u B$
lshr a, b	$b <_u B$
ashr a, b	$b <_u B$

Table 1. The constraints for Alive's arithmetic instructions to be defined. $<_u$ is unsigned less-than. B is the bitwidth of the operands in the Alive instruction. *INT_MIN* is the smallest signed integer value for a given bitwidth.

Instruction	Constraints for Poison-free execution
add nsw a, b	$\mathit{SExt}(a,1) + \mathit{SExt}(b,1) = \mathit{SExt}(a+b,1)$
add nuw a, b	$\mathit{ZExt}(a,1) + \mathit{ZExt}(b,1) = \mathit{ZExt}(a+b,1)$
sub nsw a, b	$\mathit{SExt}(a,1) - \mathit{SExt}(b,1) = \mathit{SExt}(a-b,1)$
sub nuw a, b	$\mathit{ZExt}(a,1) - \mathit{ZExt}(b,1) = \mathit{ZExt}(a-b,1)$
mul nsw a, b	$\mathit{SExt}(a,B) \times \mathit{SExt}(b,B) = \mathit{SExt}(a \times b,B)$
mul nuw a, b	$\mathit{ZExt}(a,B) \times \mathit{ZExt}(b,B) = \mathit{ZExt}(a \times b,B)$
sdiv exact a, b	$(a \div b) \times b = a$
udiv exact a, b	$(a \div_u b) \times b = a$
shl nsw a, b	$(a << b) >> b = a$
shl nuw a, b	$(a << b) >>_u b = a$
ashr exact a, b	$(a >> b) << b = a$
lshr exact a, b	$(a >>_u b) << b = a$

Table 2. The constraints for Alive's arithmetic instructions to be poison-free. $>>_u$ and $\div_u$ are the unsigned shift and division operations. B is the bitwidth of the operands in the Alive instruction. *SExt(a, n)* sign-extends a by n bits; *ZExt(a, n)* zero-extends a by n bits.

is not within Alive's scope. Predicates can be combined with the usual logical connectives. Figure 2 shows an example transformation that includes a built-in predicate in its precondition.

2.4 Well-Defined Programs and Undefined Behaviors

LLVM has three distinct kinds of undefined behavior. Together, they enable many desirable optimizations, and LLVM aggressively exploits these opportunities.

Undefined behavior in LLVM resembles undefined behavior in C/C++: anything may happen to a program that executes it. The compiler may simply assume that undefined behavior does not occur; this assumption places a corresponding obligation on the program developer (or on the compiler and language runtime, when a safe language is compiled to LLVM) to ensure that undefined operations are never executed. An instruction that executes undefined behavior can be replaced with an arbitrary sequence of instructions. When an instruction executes undefined behavior, all subsequent instructions can be considered undefined as well.

Table 1 shows when Alive's arithmetic instructions have defined behavior, following the LLVM IR specification. For example, the `shl` instruction is defined only when the shift amount is less than the bitwidth of the instruction. With the exception of memory access instructions (discussed in Section 3.3), the Alive instructions not listed in Table 1 are always defined.

The *undefined value* (`undef` in the IR) is a limited form of undefined behavior that mimics a free-floating hardware register than can return any value each time it is read. Semantically, `undef` stands for the set of all possible bit patterns for a particular type; the

(a)	(b)
%z = xor i8 undef, undef	%x = add i8 0, undef %y = add i8 0, undef %z = xor i8 %x , %y
%z = {0, 1, 2, ..., 255}	%z = {0, 1, 2, ..., 255}

(c)	(d)
%z = or i8 1, undef	br undef, l1, l2
%z = {1,3,...,253,255}	can branch to either l1 or l2

Figure 4. Illustration of the semantics of `undef`. The top half of each part of the figure presents an LLVM instruction; the bottom half indicates the possible results. In (a) and (b) the compiler can choose %z to have any value in $[0, 255]$. In (c), %z takes an odd 8-bit value. In (d) the compiler can choose either branch.

compiler is free to pick a convenient value for each use of `undef` to enable aggressive optimizations. For example, a one-bit undefined value, sign-extended to 32 bits, produces a variable containing either all zeros or all ones. Figure 4 illustrates the semantics of `undef`.

Poison values, which are distinct from undefined values, are used to indicate that a side-effect-free instruction has a condition that produces undefined behavior. When the poison value gets used by an instruction with side effects, the program exhibits true undefined behavior. Hence, poison values are deferred undefined behaviors: they are intended to support speculative execution of possibly-undefined operations. Poison values cannot be represented in the IR, but rather are ephemeral effects of certain incorrect operations. Poison values taint subsequent dependent instructions; unlike `undef`, poison values cannot be untainted by subsequent operations.

Instruction attributes modify the behavior of some LLVM instructions. The `nsw` attribute ("no signed wrap") makes signed overflow undefined. For example, this Alive transformation, which is equivalent to the optimization of `(x+1)>x` to `1` in C/C++ where `x` is a signed integer, is valid:

```
%1 = add nsw %x, 1
%2 = icmp sgt %1, %x
  =>
%2 = true
```

An analogous `nuw` attribute exists to rule out unsigned wrap. If an add, subtract, multiply, or shift left operation with an `nsw` or `nuw` attribute overflows, the result is a poison value. Additionally, LLVM's shift right and divide instructions have an `exact` attribute that requires an operation to not be lossy. Table 2 provides the constraints for the instructions to be poison-free. Developers writing Alive patterns can omit instruction attributes, in which case Alive infers where they can be safely placed.

2.5 Memory

LLVM's `alloca` instruction reserves memory in the current stack frame and returns a pointer to the allocated memory block. Heap memory, in contrast, is handled indirectly: there is no instruction in the IR for heap allocations; it is handled by library functions such as calls to `malloc`.

The `getelementptr` instruction supports structured address computations: it uses a sequence of additions and multiplications to compute the address of a specific array element or structure field. For example, an array dereference in C such as `val = a[b][c]` can be translated to the following LLVM code:

```
%ptr = getelementptr %a, %b, %c
%val = load %ptr
```

Unstructured memory accesses are supported by the `inttoptr` instruction. The `load` and `store` instructions support typed memory reads and writes. Out-of-bounds and unaligned loads and stores result in true undefined behavior, but a load from valid, uninitialized memory returns an `undef`.

3. Verifying Optimizations in Alive

The Alive checker verifies a transformation by automatically encoding the source and target, their definedness conditions, and the overall correctness criteria into SMT queries. An Alive transformation is parametric over the set of all *feasible types*: the concrete types satisfying the constraints of LLVM's type system. When computing feasible types, we place an upper limit on the width of each variable and constant (64 bits, by default).

3.1 Checking Correctness for a Feasible Type

In the absence of undefined behavior in the source or target of an Alive transformation, we can check correctness using a straightforward equivalence check: for each possible combination of values of input variables, the value of any variable that is present in both the source and target must be the same. However, checking equality of the values produced both in the source and target is not sufficient to prove correctness in the presence of any of the three kinds of undefined behavior described in Section 2.4. We use refinement to reason about optimizations in the presence of undefined behavior, following prior work [17]. The target of an Alive transformation *refines* the source template if all the behaviors of the target are included in the set of behaviors of the source.

When an instruction can produce or use an `undef` value, we need to ensure that the value produced in the target is one of the values that the source would have produced. In other words, an `undef` in the source represents a set of values and the target can refine it to any particular value. Poison values are handled by ensuring that an instruction in the target template will not yield a poison value when the source instruction did not, for any specific choice of input values. In summary, we check correctness by checking (1) the target is defined when the source is defined, (2) the target is poison-free when the source is poison-free, and (3) the source and the target produce the same result when the source is defined and poison-free.

3.1.1 Verification Condition Generation

To automatically check correctness of an Alive transformation, we need to encode the values produced, the precondition, and the correctness conditions into SMT. The semantics of the LLVM and Alive integer instructions closely match bitvector operations provided by SMT solvers. The encoding of these instructions is therefore straightforward and we omit the details. Predicates used in preconditions are also easily encoded as predicates over bitvector expressions in SMT.

For each instruction, Alive computes three SMT expressions: (1) an expression for the result of the operation (except for instructions returning void), (2) an expression representing the cases for which the instruction has defined behavior, and (3) an expression representing the cases for which the instruction does not return a poison value. The verification condition generator (VC Gen) generates a constraint for each instruction representing the cases for which it is defined (see Table 1). The VC Gen aggregates definedness conditions over the def-use chains, such that the definedness condition for each instruction is the conjunction of the definedness condition of the instruction and the definedness conditions of all of its operands. In summary, definedness constraints flow through def-use chains. The same applies to poison-free constraints.

Encoding `undef` values in SMT. An `undef` value represents a set of values of a particular type. The VC Gen encodes each `undef` value as a fresh SMT variable and adds it to a set $\mathcal{U}$. Variables in $\mathcal{U}$ are then appropriately quantified (see Section 3.1.2) over the correctness conditions to check refinement.

Encoding precondition predicates in SMT. The encoding of predicates depends on whether the underlying analysis is precise or is an over- or under-approximation. For example, the predicate `isPower2` is implemented in LLVM with a must-analysis, i.e., when `isPower2(%a)` is true, we know for sure that `%a` is a power of two; when it is false, no inference can be made. The VC Gen encodes the result of `isPower2(%a)` using a fresh Boolean variable p, and a side constraint $p \implies a \neq 0 \wedge a \mathbin{\&} (a-1) = 0$.

The encoding of may-analyses is similar. The VC Gen creates a fresh variable p to represent the result of the analysis and a side constraint of the form $s \implies p$ where s is an expression summarizing the may-analysis based on the inputs. For example, a simplified encoding of `mayAlias(%a,%b)` is $a = b \implies p$.

Most analyses in LLVM are precise when their inputs are compile-time constants. Therefore, we encode the result of these analyses precisely when we detect such cases (done statically by the VC Gen).

3.1.2 Correctness Criteria

Let ϕ be the precondition of a transformation, δ be the definedness constraint of the source instruction, ρ be the poison-free constraint, ι the result of executing a source instruction, and $\overline{\delta}$, $\overline{\rho}$, and $\overline{\iota}$ the respective constraints for the target.

Let $\mathcal{I}$ be the set of input variables from the source template including constants, P be the set of all fresh variables p used to encode approximating analyses in the precondition, and $\mathcal{U}$ and $\overline{\mathcal{U}}$ be the sets of variables created by the VC Gen to encode `undef` values of the source and target, respectively.

Let $\psi \equiv \phi \wedge \delta \wedge \rho$. A transformation specified in Alive is correct iff all of the following constraints hold for each instruction:

1. $\forall_{\mathcal{I},P,\overline{\mathcal{U}}}\, \exists_{\mathcal{U}} : \psi \implies \overline{\delta}$
2. $\forall_{\mathcal{I},P,\overline{\mathcal{U}}}\, \exists_{\mathcal{U}} : \psi \implies \overline{\rho}$
3. $\forall_{\mathcal{I},P,\overline{\mathcal{U}}}\, \exists_{\mathcal{U}} : \psi \implies \iota = \overline{\iota}$

The first constraint states that the target operation should be defined whenever the source operation is defined. The second constraint states that the target operation can only produce poison values for the inputs that the source operation does. Finally, the third constraint states that the source and target instructions should produce the same result whenever the precondition holds and the source operation is defined and poison-free.

The correctness conditions are universally quantified over the `undef` values in the target and existentially quantified over the `undef` values from the source. If the target has an `undef` value, we check that the correctness conditions hold for all possible values that a particular `undef` can take. In contrast, if the source has an `undef` value, `undef` can be instantiated with any value that enables the validity of the correctness conditions. Hence an existential quantifier is used for `undef` variables in the source, which occurs after the universal quantifier for the target `undef` variables. In summary, for each set of input variables, and `undef` values in the target, we can pick any value for the `undef` in the source that satisfies the correctness conditions.

We now state the correctness criteria for an Alive transformation:

THEOREM 1 (Soundness). *If conditions 1–3 hold for every instruction in an Alive transformation (without memory operations) and for any valid type assignment (as given by the type system of Figure 3), then the transformation is correct.*

3.1.3 Illustration of Correctness Checking

We illustrate the verification condition generation and correctness conditions with two examples.

```
Pre: C1 u>= C2
%0 = shl nsw i8 %a, C1
%1 = ashr %0, C2
  =>
%1 = shl nsw %a, C1-C2
```

The precondition for the above transformation is $\phi \equiv c_1 \geq_u c_2$, where c_1 and c_2 are bitvector variables corresponding to constants `C1` and `C2` in the transformation. The source instruction that produces `%0` is defined when `shl` is defined ($c_1 <_u 8$: the SMT variable c_1 is unsigned less than 8, the bitwidth of the operands). Likewise, instruction `%1` is defined when `ashr` is defined ($c_2 <_u 8$) and its operands are defined. Hence, the definedness conditions for the source are $\delta_{\%0} \equiv c_1 <_u 8$ and $\delta_{\%1} \equiv c_1 <_u 8 \wedge c_2 <_u 8$.

The source instruction `%0` is poison-free if no signed overflow occurs in the shift operation (because of the `nsw` attribute). The source instruction `%1` can only be poison if any of its operands is poison (since it has no `nsw` or `nuw` attributes). Hence, constraints for poison-free execution in the source are $\rho_{\%0} \equiv \rho_{\%1} \equiv (a << c_1) >> c_1 = a$, where a is the SMT variable for input `%a` in the source.

Similarly, the definedness and poison-free constraints for the target are $\overline{\delta}_{\%1} \equiv c_1 - c_2 <_u 8$ and $\overline{\rho}_{\%1} \equiv (a << (c_1 - c_2)) >> (c_1 - c_2) = a$ respectively. We trivially have $\overline{\delta}_{\%0} = \delta_{\%0}$ and $\overline{\rho}_{\%0} = \rho_{\%0}$. The constraints for the values produced by the source and target are $\iota_{\%0} \equiv \overline{\iota}_{\%0} \equiv a << c_1$, $\iota_{\%1} = (a << c_1) >> c_2$, and $\overline{\iota}_{\%1} = a << (c_1 - c_2)$. There are no `undef` values in this transformation.

Let $\psi \equiv \phi \wedge \delta_{\%1} \wedge \rho_{\%1}$. To ensure correctness, we need to check the validity of the following formulas:

1. $\forall_{a,c_1,c_2} : \psi \implies \overline{\delta}_{\%1}$
2. $\forall_{a,c_1,c_2} : \psi \implies \overline{\rho}_{\%1}$
3. $\forall_{a,c_1,c_2} : \psi \implies \iota_{\%1} = \overline{\iota}_{\%1}$

Since we have that $\delta_{\%0} = \overline{\delta}_{\%0}$, $\rho_{\%0} = \overline{\rho}_{\%0}$, and $\iota_{\%0} = \overline{\iota}_{\%0}$, the corresponding formulas for `%0` are trivially valid, and therefore we do not present them.

To check validity, we check if the negation of the above formulas are unsatisfiable; that is, we check if (1) $\exists a, \exists c_1, \exists c_2 : \psi \wedge \neg\overline{\delta}$, (2) $\exists a, \exists c_1, \exists c_2 : \psi \wedge \neg\overline{\rho}$, and (3) $\exists a, \exists c_1, \exists c_2 : \psi \wedge \iota \neq \overline{\iota}$ are unsat.

Example with `undef`. We illustrate verification condition generation and correctness checking for `undef` with the following example:

```
%r = select undef, i4 -1, 0
  =>
%r = ashr undef, 3
```

In this example, both the source and target are always defined and poison-free. However, there are `undef` values both in the source and the target. The VC Gen creates an SMT variable u_1 for the `undef` in the source and a variable u_2 for the `undef` in the target. Since both the source and the target are always defined and poison-free, checking refinement amounts only to checking that the values produced by the instructions are equal. The constraint for the value produced by the source is $\iota \equiv \mathsf{ite}(u_1 = 1, -1, 0)$, and for the target is $\overline{\iota} \equiv u_2 >> 3$ (where ite is the standard if-then-else SMT expression). We check the validity of the following formula: $\forall u_2 \exists u_1 : \iota = \overline{\iota}$. Since the formula is valid, the Alive transformation is correct. Note the use of universal quantification of the target `undef` (u_2) and existential quantification of the source `undef` (u_1).

3.1.4 Generating Counterexamples

When Alive fails to prove the correctness of a transformation, it prints a counterexample showing values for inputs and constants, as well as for each of the preceding intermediate operations. We bias the SMT solver to produce counterexamples with bitwidths such as four or eight bits. It is obvious that large-bitwidth examples are difficult to understand; we also noticed that, perhaps counter-intuitively, examples involving one- or two-bit variables are also not easy to understand, perhaps because almost every value is a corner case. Figure 5 shows an example.

3.2 Enumerating Feasible Types

Alive transformations are parametric over types. Hence, Alive must verify a transformation for all valid type assignments. We use

```
Pre: C2 % (1 << C1) == 0
%s = shl nsw %X, C1
%r = sdiv %s, C2
  =>
%r = sdiv %X, C2 / (1 << C1)

ERROR: Mismatch in values of i4 %r

Example:
%X i4 = 0xF (15, -1)
C1 i4 = 0x3 (3)
C2 i4 = 0x8 (8, -8)
%s i4 = 0x8 (8, -8)
Source value: 0x1 (1)
Target value: 0xF (15, -1)
```

Figure 5. Alive's counterexample for the incorrect transformation reported as LLVM PR21245

an SMT solver to perform type inference and to enumerate all possible type assignments (up to a bound of 64 bits for integers). We create SMT variables to represent types and create typing constraints between these variables according to the typing rules in Figure 3. Given this setup, enumerating all valid typings amounts to enumerating all (non-partial) models of an SMT formula. We solve the formula to obtain a model for the typing constraints and check the correctness of the Alive transformation for the typing. We explore all valid type assignments using the standard technique of iteratively strengthening the formula with the negation of each model until the formula becomes unsatisfiable.

3.3 Memory

We encode memory-handling instructions precisely in order to support arbitrary pointer arithmetic through structured (i.e., `getelementptr`) and unstructured (i.e., `inttoptr`) means. We use the SMT array theory to describe our encoding. Memory is represented by an array from the pointer length (e.g., 32 or 64 bits) to 8 bits, and is represented by m. Both the source and target start with an arbitrary, but equal, memory array m_0.

3.3.1 Verification Condition Generation

Alloca. Stack memory allocation is handled in two steps. First, a variable is created to represent the pointer resulting from an alloca instruction. This variable is constrained (1) to be different than zero, (2) to have a value that is properly aligned (e.g., a pointer to a 32-bit integer must be a multiple of 4), (3) to enforce that the allocated memory range does not overlap with other allocated regions (i.e., for two pointers p, p' and allocated sizes sz, sz', we have $p' \geq_u p + sz \vee p' + s' \leq_u p$), and (4) to ensure that the allocated memory range does not wrap around the memory space (i.e., for a pointer p and allocated size sz, we have $p \leq_u p + sz$). These constraints are added to α: a collection of constraints corresponding to stack memory allocation from alloca instructions. Second, the allocated region is marked as uninitialized because reading from an uninitialized memory location returns an `undef` value in LLVM. The VC Gen accomplishes this by creating a fresh bit vector with length equal to the allocation size and stores it to the memory region at the pointer value. This fresh variable is added to set $\mathcal{U}$. With this encoding, different loads of the same uninitialized memory location will return the same arbitrary value.

The size of the allocated memory block is equal to the number of allocated elements multiplied by the aligned allocation size of each element type. The allocation size of a type is computed by first rounding it to the nearest byte boundary (e.g., the allocation size of `i5` is 8 bits). Then, the size is rounded to the next valid ABI alignment. For example, in common x86 ABIs, the alignment for integers is usually 32 bits, and therefore the aligned allocation size of `i5` would be 32 bits. Verification is done parametrically on the ABI. If the allocation size exceeds the memory size, the operation has undefined behavior.

Load. Since memory is encoded as a byte array, load instructions are encoded as a concatenation of multiple array loads (select operation with the array theory). We use concat and $\mathsf{extract}$ operations from the bitvector theory to concatenate multiple bits and to extract a set of bits, respectively. For example, the instruction `%v = load i16* %p` is encoded as $v = \mathsf{concat}(\mathsf{select}(m, p+1), \mathsf{select}(m, p))$ (assuming a little-endian architecture, like x86). For loads of values whose size is not a multiple of 8 bits, an $\mathsf{extract}$ is added to the last loaded byte to trim its size. It is undefined behavior if a load operation is not within any known memory block, if the alignment of a load is larger than the alignment of the accessed memory block, or if the pointer is null.

Store. The encoding of store instructions is similar to that of the load instructions. Stored values are sliced into 8-bit fragments and then stored to the memory array. If undefined behavior has been observed previously, the memory is not modified. For example, the instruction `store %v, %p` yields the memory configuration $\mathsf{ite}(\delta, m'', m)$, with $m' = \mathsf{store}(m, p, \mathsf{extract}(v, 7, 0))$ and $m'' = \mathsf{store}(m', p+1, \mathsf{extract}(v, 15, 8))$.

Input memory blocks. An input variable in an Alive transformation can be a pointer. Such pointers can point to memory blocks that have been allocated without using the alloca instruction. Alive transformations can transform load and/or store instructions through these input pointer variables. Since we do not know anything about the memory regions referenced by these pointers (e.g., their size and alignment, whether they have been initialized before, whether any of the input pointers alias, etc.), none of the constraints described for the alloca instructions apply here. However, the VC Gen adds constraints that ensure that these pointers cannot alias pointers returned from alloca instructions (added to α).

Definedness constraint propagation with memory operations. Definedness constraints flow through the def-use chain. However, with the addition of memory operations, definedness constraints are also propagated by instructions with side-effects (e.g., stores and volatile loads). To propagate definedness constraints, the VC Gen has to maintain the order between instructions with side-effects. These instructions create sequence points. At each sequence point, the definedness constraints of the instructions and its operands are recorded and are also propagated to any subsequent instruction. Hence, the target can only perform limited reordering of instructions across sequence points.

3.3.2 Correctness Criteria

Let m and $\overline{m}$ be the final memory configurations of the source and the target. Let α be the alloca constraints given in the previous section for the source, and $\overline{\alpha}$ the respective constraints for the target. The three correctness constraints of Section 3.1.2 must be modified to include memory information (i.e., $\psi \equiv \phi \wedge \delta \wedge \rho \wedge \alpha \wedge \overline{\alpha}$). Moreover, an additional constraint must be discharged whenever the transformation uses memory-handling instructions:

4. $\forall_{\mathcal{I},\overline{\mathcal{U}},i} \, \exists_{\mathcal{U}} : \phi \wedge \alpha \wedge \overline{\alpha} \implies \mathsf{select}(m, i) = \mathsf{select}(\overline{m}, i)$

This constraint does not explicitly include constraints for limiting the equality to defined behavior like the correctness constraint 3 in Section 3.1.2. These are, however, already included in m. A store instruction only updates a memory block if no undefined behavior has been observed previously. Thus, no further constraints are required.

We now state the correctness criteria for Alive transformations with memory operations:

THEOREM 2 (Soundness). *If, for every valid type assignment in an Alive transformation, conditions 1–3 hold for every instruction and condition 4 holds for the final memory configurations, then the transformation is correct.*

3.3.3 Beyond the Array Theory

Many SMT solvers do not support the theory of arrays efficiently, especially in the presence of quantifiers. Hence, we explore encoding memory operations without arrays. This is usually accomplished with Ackermann's expansion [1]. In the worst case, this procedure can result in a quadratic increase in the size of the formula in terms of the number of load instructions, and requires an additional fresh SMT variable per load. SMT solvers usually perform Ackermannization lazily to avoid the quadratic blowup as much as possible.

In contrast to lazy Ackermannization as usually performed by SMT solvers, we use an eager Ackermannization encoding that is linear in the number of load and store instructions. Our encoding has the additional advantage of not requiring extra SMT variables, at the expense of potentially more complex SMT expressions.

With our eager encoding, store instructions are replaced with if-then-else expressions, such that the expression returns the stored value if pointer p is equal to the stored location, or returns the previous memory expression otherwise, i.e., $\mathsf{store}(m, q, v)$ becomes $\mathsf{ite}(p = q, v, m)$.

Load instructions take the memory expression built so far (a chain of ite expressions) and replace p with the loaded memory location, i.e., $\mathsf{select}(m, q)$ becomes $\mathsf{ite}(q = p_1, v_1, \mathsf{ite}(q = p_2, v_2, \ldots \mathsf{ite}(q = p_n, v_n, m_0)))$ for previously stored values $v_1, \ldots, v_n$ at locations $p_1, \ldots, p_n$ (potentially overlapping) and initial memory content m_0. Since the store operations are enqueued in order (most to least recent), the expression yields the most recent store in case there were multiple stores to the same location.

In our experiments, we observed that our eager encoding results in faster SMT solving when compared to the theory of arrays. However, our encoding does not ensure consistency across different loads to the same uninitialized memory location, but this is not required by any optimization we have analyzed.

3.4 Attribute Inference

By watching LLVM developers submit patches, we observed that placing the `nsw`, `nuw`, and `exact` attributes in LLVM optimizations is difficult enough that it often becomes a matter of trial and error, with developers frequently omitting attributes from the target side of transformations rather than determining whether they can be added safely. The result is that peephole optimizations in LLVM tend to strip away attributes, constraining the behavior of subsequent optimization passes that might have been able to exploit the attributes, had they been preserved.

Alive has support for adding the `nsw`, `nuw`, and `exact` attributes to the source and target of transformations when it is safe to do so. On the source side, our goal is to automatically synthesize the weakest precondition for a transformation in terms of instruction attributes. On the target side, we want to synthesize the optimal propagation of attributes: the strongest postcondition.

Our algorithm enumerates all models of a quantified SMT formula for correctness to infer attributes. As attributes only influence constraints generated for poison-free execution, we generate these poison-free constraints conditionally based on the presence or absence of the attributes. The algorithm introduces a fresh SMT Boolean variable f for each instruction and for each attribute. More concretely, we have $\rho \equiv f_1 \implies p_1 \wedge \ldots \wedge f_n \implies p_n$, where f_i are fresh Boolean variables, and p_i are the conditions that state when the instruction is poison-free when the respective attribute i is enabled.

```
$\Phi := \mathsf{true}$
for each type assignment do
    $f := \exists_{\mathcal{F},\overline{\mathcal{F}}} : \Phi \wedge c_1 \wedge c_2 \wedge c_3 \wedge c_4$
    $\mu := \mathsf{false}$
    while $f$ is satisfiable with model $m$ do
        $b := \{l \mid l \in \mathcal{F} \wedge m(l)\} \cup \{\neg l \mid l \in \overline{\mathcal{F}} \wedge \neg m(l)\}$
        $\mu := \mu \vee \bigwedge b$
        $f := f \wedge \neg \bigwedge b$
    $\Phi := \Phi \wedge \mu$
```

Figure 6. Optimal attribute inference algorithm. The set b accumulates SMT variables that indicate the presence of attributes in the source (for the precondition) and absence of attributes in the target in a given model (for the postcondition). The result (Φ) is a constraint that gives all possible attribute assignments.

Let $\mathcal{F}$ ($\overline{\mathcal{F}}$) be the set of all f variables from the source (resp. target). The disjunction of all models of the following formula is the optimal set of attributes:

$$\exists_{\mathcal{F},\overline{\mathcal{F}}} : c_1 \wedge c_2 \wedge c_3 \wedge c_4$$

Constraints c_1, c_2, c_3, c_4 are the conjunction of the constraints given as the correctness criteria in Section 3.1.2 for all instructions.

The algorithm for optimal attribute inference, shown in Figure 6, exploits the partial ordering between the attribute assignments. For example, if an optimization is correct without the `nsw` attribute in a source instruction, then the optimization is also correct with it. Similarly, if an attribute can be enabled in a target operation, then the optimization is also correct if it is turned off (although we are interested in enabling as many attributes in the target as possible).

The inner loop of the algorithm in Figure 6 generates all possible attribute assignments, which satisfy the correctness constraints and exploits the partial ordering of the attributes, for a given type assignment. The algorithm strengthens the set of possible attribute assignments with the disjunction of all possible attribute assignments for a given type assignment. The inner loop of the algorithm generates all possible attributes by iteratively enumerating all possible solutions by blocking each model at a time (i.e., $f := f \wedge \neg \bigwedge b$). The outer loop strengthens the attribute assignments explored for each type assignment to generate the constraint (Φ), whose solution gives the optimal attribute assignment. The algorithm can be further improved to reduce the number of queries by using techniques to bias the results of the SMT solver (e.g., [19]), which we leave for future work.

4. Generating C++ from Alive

After a transformation has been proved correct, Alive can turn it into C++ code that uses the same instruction pattern matching library that InstCombine uses; the generated code can be linked into LLVM and used as an LLVM optimization pass. The C++ code for an optimization has two parts. First, there is an `if`-statement checking if the optimization should fire, which is the case when a DAG of LLVM instructions matches the Alive source template and when the precondition, if any, is met. Second, the body of the transformation creates instructions from the target template and wires them up to the source and input variables. Figure 7 shows an example. The generated C++ code does not attempt to clean up any instructions

Alive transformation:

```
Pre: isSignBit(C1)
%b = xor %a, C1
%d = add %b, C2
   =>
%d = add %a, C1 ^ C2
```

Generated C++:

```
Value *a, *b;
ConstantInt *C1, *C2, *C3;
if (match(I,m_Add(m_Value(b),m_ConstantInt(C2))) &&
    match(b,m_Xor(m_Value(a),m_ConstantInt(C1))) &&
    C1->getValue().isSignBit()) {
  APInt C3_val = C1->getValue() ^ C2->getValue();
  C3 = ConstantInt::get(I->getType(), C3_val);
  BinaryOperator *d =
    BinaryOperator::CreateAdd(a, C3, "", I);
  I->replaceAllUsesWith(d);
}
```

Figure 7. An Alive transformation and its corresponding generated code. The C++ transformation is conditional on two match calls, one for each instruction in the source template, and also on the precondition. The target template has a single instruction and creates a new compile-time constant; both of these are directly reflected in the body of the C++ transformation.

that might have been rendered useless by the optimization; this task is left to a subsequent dead-code elimination pass.

Translating a source template. The code generator translates the source into conditions using LLVM's pattern-matching library, and binds variables for later use. For example, `match(I, m_Add(m_Value(b), m_ConstantInt(C2)))` in Figure 7 will return true if an LLVM instruction `I` is an addition of a value to a constant, and sets `b` and `C2` to point to the arguments. Matching begins with the root instruction in the source template and recursively matches operands until all non-inputs have been bound. Although LLVM allows arbitrarily nested pattern matching, Alive currently matches each instruction in a separate clause. Alive's predicates and logical connectives directly correspond to their C++ equivalents in LLVM.

Translating a target template. A new instruction is created for each instruction that is in the target template but not the source. The root instruction from the source is replaced by its counterpart in the target.

Constants. Constant expressions translate to `APInt` (LLVM's arbitrary-width literal values) or `Constant` values, depending on the context. Arithmetic expressions and functions translate to the corresponding operations for `APInt` values in LLVM.

Type unification. The LLVM constructors for constant literal values and conversion instructions require explicit types. In general, this information will depend on types in the source. As Alive transformations are parametric over types, and Alive provides support for explicit and named types, such information is not readily available. The Alive code generator uses a unification-based type inference algorithm to identify appropriate types for the operands and introduces additional clauses in the `if` condition to ensure the operands have the appropriate type before invoking the transformation. This type system ensures that the generated code does not produce ill-typed LLVM code.

The unification proceeds in three phases. First, the types of the operands in the source are unified according to the constraints in the source (e.g., the operands of a binary operator must have the same type) based on the assumption that source is a well-formed LLVM program. Second, the types of the operands in the target are similarly unified according to constraints of the target. Third, when the operands of a particular instruction in the target do not belong to the same class, then an explicit check requiring that the types are equal is inserted to the `if` condition in the C++ code generated. The explicit check is necessary as the target has type constraints that cannot be determined by the source alone.

5. Implementation

We implemented Alive in Python and used the Z3 SMT solver [7] to discharge both typing and refinement constraints. Alive is about 5,200 lines of open-source code.[4]

Typing constraints are over the QF_LIA (quantifier-free linear integer arithmetic) theory, and refinement constraints are either over the BV or QF_BV (quantified/quantifier-free bitvector) theories. The number of possible type assignments for a transformation is usually infinite. To ensure termination we constrain integer types to have bitwidths in the range $1..64$, and type nesting is limited to two levels.

Constraints c_1 to c_4 (Section 3.1.2) are negated before querying the SMT solver, effectively removing one quantifier alternation. Therefore, for transformations without undefined values in the source template, we obtain quantifier-free formulas, and formulas with a single quantifier otherwise. When performing attribute inference, formulas have either one or two sets of quantifiers.

Threats to validity. Alive tries to accurately reflect the semantics described in the LLVM Language Reference [18]; there could be differences between our formalization and the semantics intended by the LLVM developers. Alive's bounded verification could lead it to incorrectly verify an optimization, though in our experience it is uncommon to see operands wider than 64 bits in LLVM code.

6. Evaluation

We translated hundreds of peephole optimizations from LLVM into Alive. We verified them, we inferred optimal `nsw`/`nuw`/`exact` attributes for them, and we translated the Alive optimizations into C++ that we linked into LLVM and then used the resulting optimizer to build LLVM's test suite and the SPEC INT 2000 and 2006 benchmarks.

6.1 Translating and Verifying InstCombine

LLVM's InstCombine pass rewrites expression trees to reduce their cost, but does not change the control-flow graph. Table 3 summarizes the 334 InstCombine transformations that we translated to Alive. Eight (2.4%) of these could not be proved correct; we reported these erroneous transformations to the LLVM developers, who confirmed and fixed them. We re-translated the fixed optimizations to Alive and proved them correct. Out of the remaining 694 transformations that we did not translate, some (such as those using vectors, call/return instructions, and floating point) cannot yet be expressed in Alive and the rest we simply have not had time to translate.

The buggiest InstCombine file that we found was MulDivRem, which implements optimizations that have multiply, divide, and remainder instructions as the root of expression trees. Out of the 44 translated optimizations, we found that six of them (14%) were incorrect.

The most common kind of bug in InstCombine was the introduction of undefined behavior, where an optimization replaces an

[4] The version of Alive corresponding to this paper can be found at `https://github.com/nunoplopes/alive/tree/pldi15`.

```
Name: PR20186            Name: PR20189            Name: PR21242            Name: PR21243
%a = sdiv %X, C          %B = sub 0, %A           Pre: isPowerOf2(C1)      Pre: !WillNotOverflowSignedMul(C1, C2)
%r = sub 0, %a           %C = sub nsw %x, %B      %r = mul nsw %x, C1      %Op0 = sdiv %X, C1
  =>                       =>                       =>                     %r = sdiv %Op0, C2
%r = sdiv %X, -C         %C = add nsw %x, %A      %r = shl nsw %x, log2(C1)  =>
                                                                           %r = 0

Name: PR21245            Name: PR21255            Name: PR21256            Name: PR21274
Pre: C2 % (1<<C1) == 0   %Op0 = lshr %X, C1       %Op1 = sub 0, %X         Pre: isPowerOf2(%Power) && hasOneUse(%Y)
%s = shl nsw %X, C1      %r = udiv %Op0, C2       %r = srem %Op0, %Op1     %s = shl %Power, %A
%r = sdiv %s, C2           =>                       =>                     %Y = lshr %s, %B
  =>                     %r = udiv %X, C2 << C1   %r = srem %Op0, %X       %r = udiv %X, %Y
%r = sdiv %X, C2/(1<<C1)                                                     =>
                                                                           %sub = sub %A, %B
                                                                           %Y = shl %Power, %sub
                                                                           %r = udiv %X, %Y
```

Figure 8. We found eight incorrect InstCombine transformations during the development of Alive

File	# opts.	# translated	# bugs
AddSub	67	49	2
AndOrXor	165	131	0
Calls	80	0	0
Casts	77	0	0
Combining	63	0	0
Compares	245	0	0
LoadStoreAlloca	28	17	0
MulDivRem	65	44	6
PHI	12	0	0
Select	74	52	0
Shifts	43	41	0
SimplifyDemanded	75	0	0
VectorOps	34	0	0
Total	1,028	334	8

Table 3. The total number of optimizations, the number of optimizations that we translated into Alive, and the number of translated optimizations that were wrong, for each file in InstCombine

expression with one that is defined for a smaller range of inputs than was the original expression. There were four bugs in this category. We also found two bugs where the value of an expression was incorrect for some inputs, and two bugs where a transformation would generate a poison value for inputs that the original expression did not. Figure 8 provides the Alive code and the bug report numbers for the bugs that we discovered during our translation of LLVM InstCombine optimizations into Alive.

Alive usually takes a few seconds to verify the correctness of a transformation, during which time it may issue hundreds or thousands of incremental solver calls. Unfortunately, for some transformations involving multiplication and division instructions, Alive can take several hours or longer to verify the larger bitwidths. This indicates that further improvements are needed in SMT solvers to efficiently handle such formulas. In the meantime, we work around slow verifications by limiting the bitwidths of operands.

6.2 Preventing New Bugs

A few LLVM developers are already using Alive to avoid introducing wrong-code bugs. Also, we have been monitoring proposed LLVM patches and trying to catch incorrect transformations before they are committed to the tree. For example, in August 2014 a developer submitted a patch that improved the performance of one of the SPEC CPU 2000 benchmarks by 3.8%—this is obviously an interesting addition to a compiler. We used Alive to find bugs in the developer's initial and second proposed patches, and we proved that the third one was correct. This third and final patch retained the performance improvement without compromising the correctness of LLVM.

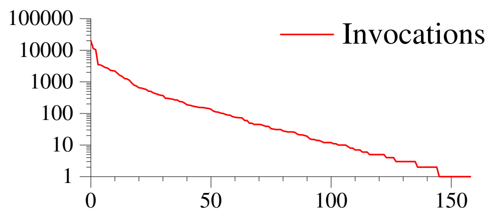

Figure 9. The number of times each Alive optimization was invoked during compilation of the LLVM nightly suite and SPEC 2000 and 2006. The x-axis is the Alive optimization number, sorted by decreasing number of invocations. The y-axis is the number of invocations. Only 159 of these optimizations were triggered during the experiment.

6.3 Inferring Instruction Attributes

Out of the 334 transformations we translated, Alive was able to weaken the precondition for one transformation and strengthen the postcondition for 70 (21%) transformations. The most strengthening took place for transformations in AddSub, MulDivRem, and Shifts, each with around 40% of transformations getting stronger postconditions. When a postcondition is strengthened, more instructions retain undefined behavior attributes, permitting additional optimizations to be performed. We have not yet quantified this effect.

6.4 Generating C++

We removed the InstCombine optimizer from LLVM 3.6 and replaced it with C++ generated by Alive; we refer to the resulting compiler as LLVM+Alive. We used LLVM+Alive to build the LLVM nightly test suite and also the SPEC 2000 and 2006 benchmarks, compiling about a million lines of code overall. All experiments were performed on an Intel x86-64 machine running Ubuntu 14.04. The code compiled by LLVM+Alive did not have any unexpected test case failures: as far as we can tell, it is free of miscompilation bugs.

Figure 9 reports the number of times that each optimization fired during compilation of the LLVM nightly test suite and SPEC benchmarks using LLVM+Alive at -O3. Alive optimizations fired about 87,000 times in total. Figure 9 also illustrates that a small number of optimizations are applied frequently. The top ten optimizations account for approximately 70% of the total invocations and there is a long tail of infrequently-used optimizations.

Compilation time. We measured the time to compile SPEC 2000 and SPEC 2006 using LLVM+Alive in comparison to LLVM 3.6 at -O3. Compilation using LLVM+Alive was on average 7% faster than LLVM because it runs only a fraction of the total InstCombine optimizations. Our C++ code generator is a proof of concept implementation and we have not yet implemented many optimizations such as reusing the results of redundant pattern-match calls.

Execution time of compiled code. We measured the execution time of the SPEC 2000 and 2006 benchmarks on their reference inputs. Code compiled with LLVM+Alive is (averaged across all SPEC benchmarks) 3% slower than code compiled with LLVM 3.6 -O3. Among these benchmarks, LLVM+Alive provides a speedup of 7% with gcc, and suffers a slowdown of 10% in the equake benchmark. The code generated with LLVM+Alive is slower with some benchmarks because we have only translated a third of the InstCombine optimizations.

7. Related Work

Prior research on improving compiler correctness can be broadly classified into compiler testing tools, formal reasoning frameworks for compilers, and domain specific languages (DSLs). DSLs for compiler optimizations are the most closely related work to Alive. These include languages based on graph rewriting [2, 21, 27], regular expressions [12], computation tree logic (CTL) [14], type systems [28], and rewrite rules [16, 36]. While Alive is perhaps most similar to high-level rewrite patterns [13, 20], it differs in its extensive treatment of undefined behavior, which is heavily exploited by LLVM and other aggressive modern compilers.

Peephole optimization patterns for a particular ISA can be generated from an ISA specification [6]. In contrast to compiler optimizations, optimized code sequences can be synthesized either with peephole pattern generation or through superoptimization [3, 11, 22, 30, 33].

Optgen [4] automatically generates peephole optimizations. Like Alive, Optgen operates at the IR level and uses SMT solvers to verify the proposed optimizations. While Alive focuses on verifying developer-created optimizations, Optgen generates all possible optimizations up to a specified cost and can generate a test suite to check optimizations not implemented in a given compiler. In contrast to Alive, Optgen handles only integer operations and does not handle memory operations, poison values, support any operation producing undefined behavior, or abstraction over bitwidths/types.

Random testing tools [15, 24, 37] have discovered numerous bugs in LLVM optimizations both for sequential programs and concurrent programs. These tools are not complete, as was shown by the bugs we found in optimizations that had previously been fuzzed.

An alternative approach to compiler correctness is translation validation [25, 26, 29] where, for each compilation, it is proved that the optimized code refines the unoptimized code. Many techniques and tools have been developed [10, 31, 32, 34, 38]. Translation validation appears to be a very promising approach, but it suffers from the drawback of requiring proof machinery to execute during every compilation. Our judgment was that the LLVM developers would not tolerate this, so Alive instead aims for once-and-for-all proof of correctness of a limited slice of the compiler.

The CompCert [17] compiler, for a subset of C, is an end-to-end verified compiler developed with the interactive proof assistant Coq. Vellvm [39, 40] formalizes the semantics of LLVM IR, SSA properties, and optimizations in Coq. Alive's treatment of `undef` values mirrors the treatment in Vellvm. In contrast to Vellvm, which concretizes values in memory, Alive maintains `undef` values in memory, handles poison attributes, and automates reasoning with an SMT solver. Recent efforts have explored formalizing the relations of optimizations with weak memory models in Coq [35]. Proving the correctness of trace optimizations (as used in, e.g., JIT compilers) has also been attempted recently [8].

8. Conclusion

We have shown that an important class of optimizations in LLVM—peephole optimizations—can be formalized in Alive, a new language that makes optimizations much more concise than when they are embedded in C++ code, while also supporting automated proofs of correctness. We designed Alive to resemble LLVM's textual format while also supporting abstraction over constant values, over bitwidths of operands, and over the presence of LLVM's undefined behavior attributes that modify instruction behavior. After an Alive transformation has been proved correct, it can be automatically translated into C++ that can be included in an optimization pass. Our first goal was to create a tool that is useful for LLVM developers. We believe this goal has been accomplished. Second, we would like to see a large part of InstCombine replaced with code generated by Alive; we are still working towards that goal.

Acknowledgments

The authors thank the LLVM developers for their continued support for the development of Alive, for discussions regarding LLVM IR's semantics, and for adopting Alive. A special thanks to David Majnemer for confirming and fixing the bugs we reported. Eric Eide and Raimondas Sasnauskas provided valuable feedback on this work. This paper is based upon work supported in part by NSF CAREER Award CCF–1453086, NSF Award CNS–1218022, and a Google Faculty Award.

References

[1] W. Ackermann. *Solvable Cases of the Decision Problem*. Studies in Logic and the Foundations of Mathematics, 1954.

[2] U. Aßmann. How to uniformly specify program analysis and transformation with graph rewrite systems. In *Proc. of the 6th International Conference on Compiler Construction*, pages 121–135, 1996.

[3] S. Bansal and A. Aiken. Automatic generation of peephole superoptimizers. In *Proc. of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 394–403, 2006.

[4] S. Buchwald. Optgen: A generator for local optimizations. In *Proc. of the 24th International Conference on Compiler Construction (CC)*, pages 171–189, Apr. 2015.

[5] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, Oct. 1991.

[6] J. W. Davidson and C. W. Fraser. Automatic generation of peephole optimizations. In *Proc. of the 1984 SIGPLAN Symposium on Compiler Construction*, pages 111–116, 1984.

[7] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[8] S. Dissegna, F. Logozzo, and F. Ranzato. Tracing compilation by abstract interpretation. In *Proc. of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 47–59, 2014.

[9] S. Guyer and C. Lin. Broadway: A compiler for exploiting the domain-specific semantics of software libraries. *Proceedings of the IEEE*, 93 (2), 2005.

[10] C. Hawblitzel, S. K. Lahiri, K. Pawar, H. Hashmi, S. Gokbulut, L. Fernando, D. Detlefs, and S. Wadsworth. Will you still compile me tomorrow? Static cross-version compiler validation. In *Proc. of the 9th Joint Meeting on Foundations of Software Engineering (FSE)*, 2013.

[11] R. Joshi, G. Nelson, and Y. Zhou. Denali: A practical algorithm for generating optimal code. *ACM Trans. Program. Lang. Syst.*, 28(6): 967–989, Nov. 2006.

[12] D. Kozen and M.-C. Patron. Certification of compiler optimizations using Kleene algebra with tests. In *Proc. of the 1st International Conference on Computational Logic*, pages 568–582, 2000.

[13] S. Kundu, Z. Tatlock, and S. Lerner. Proving optimizations correct using parameterized program equivalence. In *Proc. of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 327–337, 2009.

[14] D. Lacey, N. D. Jones, E. Van Wyk, and C. C. Frederiksen. Compiler optimization correctness by temporal logic. *Higher Order Symbol. Comput.*, 17(3):173–206, Sept. 2004.

[15] V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. In *Proc. of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 216–226, 2014.

[16] S. Lerner, T. Millstein, E. Rice, and C. Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *Proc. of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 364–377, 2005.

[17] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[18] LLVM Developers. LLVM Language Reference Manual. Available from `http://llvm.org/docs/LangRef.html`, 2014.

[19] N. P. Lopes and J. Monteiro. Weakest precondition synthesis for compiler optimizations. In *Proc. of the 15th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 203–221, 2014.

[20] N. P. Lopes and J. Monteiro. Automatic equivalence checking of programs with uninterpreted functions and integer arithmetic. *Int. J. Softw. Tools Technol. Transf.*, 2015.

[21] W. Mansky and E. Gunter. A cross-language framework for verifying compiler optimizations. In *Proc. of the 5th Workshop on Syntax and Semantics of Low-Level Languages*, 2014.

[22] H. Massalin. Superoptimizer: A look at the smallest program. In *Proc. of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 122–126, 1987.

[23] W. M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, Dec. 1998.

[24] R. Morisset, P. Pawan, and F. Z. Nardelli. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In *Proc. of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 187–196, 2013.

[25] G. C. Necula. Translation validation for an optimizing compiler. In *Proc. of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 83–94, 2000.

[26] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Proc. of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 151–166, 1998.

[27] N. Ramsey, J. Dias, and S. P. Jones. Hoopl: A modular, reusable library for dataflow analysis and transformation. In *Proc. of the 3rd ACM Symposium on Haskell*, 2010.

[28] A. Saabas and T. Uustalu. Program and proof optimizations with type systems. *The Journal of Logic and Algebraic Programming*, 77(1–2): 131–154, 2008.

[29] H. Samet. Proving the correctness of heuristically optimized code. In *Communications of the ACM*, 1978.

[30] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In *Proc. of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[31] T. A. L. Sewell, M. O. Myreen, and G. Klein. Translation validation for a verified OS kernel. In *Proc. of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 471–482, 2013.

[32] M. Stepp, R. Tate, and S. Lerner. Equality-based translation validator for LLVM. In *Proc. of the 23rd International Conference on Computer Aided Verification*, pages 737–742, 2011.

[33] R. Tate, M. Stepp, and S. Lerner. Generating compiler optimizations from proofs. In *Proc. of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2010.

[34] J.-B. Tristan, P. Govereau, and G. Morrisett. Evaluating value-graph translation validation for LLVM. In *Proc. of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 295–305, 2011.

[35] V. Vafeiadis, T. Balabonski, S. Chakraborty, R. Morisset, and F. Z. Nardelli. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In *Proc. of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2015.

[36] D. L. Whitfield and M. L. Soffa. An approach for exploring code improving transformations. *ACM Trans. Program. Lang. Syst.*, 19(6): 1053–1084, Nov. 1997.

[37] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proc. of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 283–294, 2011.

[38] A. Zaks and A. Pnueli. CoVaC: Compiler validation by program analysis of the cross-product. In *Proc. of the 15th International Symposium on Formal Methods*, pages 35–51, 2008.

[39] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *Proc. of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 427–440, 2012.

[40] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic. Formal verification of SSA-based optimizations for LLVM. In *Proc. of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 175–186, 2013.