Practical Verification of Peephole Optimizations with Alive

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

David Menendez
Rutgers University, USA
davemm@cs.rutgers.edu

Santosh Nagarakatte
Rutgers University, USA
santosh.nagarakatte@cs.rutgers.edu

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

ABSTRACT

Compilers should not miscompile. Peephole optimizations, which perform local rewriting of the input program to improve the efficiency of generated code, are a persistent source of compiler bugs. We created 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 the various kinds of undefined behavior. Alive has found numerous bugs in the LLVM compiler and is being used by LLVM developers.

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.

Random testing [7, 20] is one approach to improving 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). The state of the art in compiler verification requires a fresh compiler implementation and many person-years of proof engineering (e.g., CompCert [9]), making this appoach impractical in most production environments.

We developed Alive: a new language and tool for developing correct peephole optimizations; it is shown in Figure 1. Peephole optimizations in LLVM are performed by the instruction combiner (InstCombine) pass. 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 a satisfiability modulo theory (SMT) solver (or provides a counterexample), and it automatically generates C++ code that is similar to hand-written peephole optimizations such as those found in the instruction combiner. Alive's main contributions are in identifying a subset of peephole optimizations that can be automatically verified and in 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 undefined behavior, and with support for code generation.

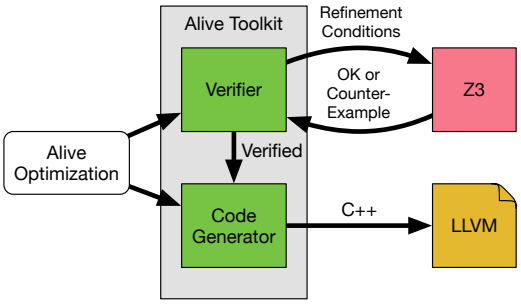

Figure 1: Overview of Alive. Optimizations expressed in Alive are automatically verified using the Z3 SMT solver. Verified optimizations are converted to C++ implementations for use in LLVM.

InstCombine transformations perform numerous algebraic simplifications that improve efficiency, enable other optimizations, and canonicalize LLVM code. InstCombine optimizations have been a persistent source of LLVM bugs [7, 20].

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 original version of this paper is titled "Provably Correct Peephole Optimizations with Alive" and was published in the proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2015.

In Alive the same optimization is:

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

The Alive specification is designed to resemble—both syntactically and semantically—the LLVM transformation that it describes. It is much more succinct than its equivalent C++ implementation, is not expressed in terms of LLVM's internal data structures and control flow, and can be automatically verified by the Alive tool kit. 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 twenty-three 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 and it is also available on-line at `http://rise4fun.com/Alive`.

2. THE ALIVE LANGUAGE

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

2.1 Syntax

An Alive transformation has the form $A \implies B$, where A is the *source template* (unoptimized code) and B is the *target template* (optimized code). Additionally, a transformation may include a precondition. Since Alive's representation, like LLVM's, is based on directed graphs of instructions in SSA form, the ordering of non-dependent instructions is irrelevant.

Alive implements a subset of LLVM's integer and pointer instructions. It also has limited support for branches: to avoid loops, they are not allowed to jump backwards. Alive supports LLVM's `nsw`, `nuw`, and `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.

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

```
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 $\%R$.

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, such as integers and pointers. LLVM uses arbitrarily-sized integers, with a separate type for each width (e.g., `i8` or `i57`). Alive has limited support for LLVM's pointer and array types, and does not support structures or vectors. Recent efforts have subsequently extended Alive with support for floating-point types [14, 16].

Unlike LLVM, Alive permits type annotations to be omitted and does not require values to have a unique type. This enables succinct specifications of optimizations in Alive, as many peephole optimizations are not type-specific. A set of possible types is inferred for each implicitly-typed value, and the correctness of an optimization is checked for each type assignment. Because LLVM has infinitely many integer types, we set an upper bound of 64 bits for implicitly typed integer values.

2.3 Built-In Predicates

Some peephole optimizations use 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 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.

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	$b \neq 0$

Table 1: The constraints for arithmetic instructions to be defined. $<_u$ is unsigned less-than. *INT_MIN* is the smallest signed integer value for a given bitwidth.

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

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

2.4 Undefined Behaviors in LLVM

To aggressively optimize well-defined programs, 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 and 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 `udiv` instruction is defined only when the dividend is non-zero. With the exception of memory access instructions (discussed in the original paper [10]), 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 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.

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 taint subsequent dependent instructions; unlike `undef`, poison values cannot be untainted by subsequent operations. The subtleties in the semantics of `undef` and poison values and its impact on either enabling or disabling optimizations are currently being explored [8].

Shift instructions, `shl`, `ashr`, and `lshr`, produce a poison value when their second argument, the *shift amount*, is larger than or equal to the bit width of the operation.

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 and 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 poison. 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.

3. VERIFYING OPTIMIZATIONS IN Alive

The Alive interpreter 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 and not exceeding the default limit of 64 bits.

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 valuation of the input variables, the value of any variable that is present in both the source and target must be identical. However, an equivalence check 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. 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. That is, a transformation may remove undefined behaviors but not add them.

When an optimization contains or may produce `undef` values, we need to ensure that the target never produces a value

```
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 3: Alive's counterexample for the incorrect transformation reported as LLVM PR21245

that the source does not produce. In other words, an `undef` in the source represents a set of values and the target can refine it to any particular value, but an `undef` in the target represents a set of values which must *all* be refinements of the source. 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 target produces a subset of the values produced by the source when the source is defined and poison-free.

To determine whether these conditions hold, we ask an SMT solver to find cases where they are violated. When found, these counter-examples are reported to the user, as shown in Figure 3. Conversely, if the SMT solver can show that no counter-example exists, then the conditions must hold and the optimization is valid.

3.1 Verification Condition Generation

Alive's verification condition generator (VC Gen) encodes the values, instructions, and expressions in a transformation into SMT expressions using the theory of bitvectors. The correspondence between LLVM operations and bitvector logic is very close, which makes the encoding straightforward. For each instruction, the interpreter computes three SMT expressions: (1) an expression ι for the result of the instruction, (2) an expression δ indicating whether the instruction is defined, and (3) an expression ρ indicating whether the result is free of poison. The first has a type corresponding to the return type of the instruction. The others are Boolean predicates. All three may contain free variables, corresponding to the uninterpreted input variables and symbolic constants in the optimization.

Typically, an instruction's result is encoded by applying the corresponding bitvector operation to the encoding of its arguments. Its definedness and poison conditions are the conjunction of the definedness and poison conditions, respectively, of its arguments along with any specific requirements for that instruction.

For example, consider the instruction `udiv exact %a, %b`, which is encoded as follows,

$$\iota = \iota_a \div_u \iota_b$$
$$\delta = \delta_a \wedge \delta_b \wedge \iota_b \neq 0$$
$$\rho = \rho_a \wedge \rho_b \wedge (\iota_a \div_u \iota_b) \times \iota_b = \iota_a,$$

where $\div_u$ is unsigned bitvector division. The unsigned division requires the second argument to be non-zero, and the `exact` attribute requires `%a` to be divisible by `%b`.

Encoding undef values.

Undef values represent sets of possible values. The VC Gen encodes them as fresh SMT variables, which are collected in a set $\mathcal{U}$. In particular, each reference to an undef value, direct or indirect, must receive a fresh SMT variable. The sets collected for the source and target will then be appropriately quantified over the correctness conditions.

Encoding preconditions.

Alive's precondition sublanguage provides comparison operators and a set of named predicates, along with conjunction, disjunction, and negation. Aside from the predicates, these have a straightforward encoding in SMT.

The encoding of named 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 $p \implies a \neq 0 \wedge a \,\&\, (a-1) = 0$.

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).

3.2 Correctness Criteria

Let ϕ be the encoding of the precondition, let ι_S and ι_T be the values computed by the source and target, respectively, and similarly let δ_S, δ_T, ρ_S, and ρ_T be the definedness and poison-free conditions.

Let $\mathcal{I}$ be the set of input variables, $\mathcal{P}$ be the Boolean variables used to encode analyses, and $\mathcal{U}_S$ and $\mathcal{U}_T$ be the sets of variables used to encode undef values in the source and target, respectively.

An Alive optimization is correct if and only if the following conditions hold for every feasible type assignment:

1. $\forall_{\mathcal{I},\mathcal{P},\mathcal{U}_T} \exists_{\mathcal{U}_S} : \phi \wedge \delta_S \implies \delta_T$
2. $\forall_{\mathcal{I},\mathcal{P},\mathcal{U}_T} \exists_{\mathcal{U}_S} : \phi \wedge \delta_S \wedge \rho_S \implies \rho_T$
3. $\forall_{\mathcal{I},\mathcal{P},\mathcal{U}_T} \exists_{\mathcal{U}_S} : \phi \wedge \delta_S \wedge \rho_S \implies \iota_S = \iota_T$

The first condition requires the target to be defined whenever the source is defined. The second condition requires the target to be poison-free whenever the source is defined and poison-free. The third condition requires the source and

target to compute the same result when the source is defined and poison-free. The constraints are only required to hold if the precondition is satified, because the optimization will not be applied otherwise.

Note that the variables used to represent undef values for the source and target are existentially and universally quantifed, respectively. When an undef term occurs in the target, the target must refine the source for all possible values the undef term might take. In contrast, an undef term in the source may be instantiated with any value which makes the optimization a refinement. The order of the quantifiers permit undef values in the source to have different instantiations, depending on the instantiation of the undef values in the target.

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, then the transformation is correct.*

3.3 Illustration of Correctness Checking

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

```
Pre: C1 != 0 && C2 %u C1 == 0
%m = mul nuw %a, C1
%r = udiv %m, C2
  =>
%r = udiv %a, C2 /u C1
```

This is encoded using the following definitions for `%r`:

$$\begin{aligned}
\phi &\equiv c_1 \neq 0 \wedge c_2 \bmod_u c_1 = 0 \\
\delta_S &\equiv c_2 \neq 0 \\
\delta_T &\equiv c_2 \div_u c_1 \neq 0 \\
\rho_S &\equiv ZExt(a, B) \times ZExt(c_1, B) = ZExt(a \times c_1, B) \\
\rho_T &\equiv \top \\
\iota_S &= (a \times c_1) \div_u c_2 \\
\iota_T &= a \div_u (c_2 \div_u c_1)
\end{aligned}$$

Note that ρ_S has propagated the poison-free condition for `%m`, and that the target is always poison free. The sets $\mathcal{U}_S$, $\mathcal{U}_T$, and $\mathcal{P}$ are empty, so the correctness conditions are:

$$\begin{aligned}
\forall_{a,c_1,c_2} : \phi \wedge \delta_S &\implies \delta_T \\
\forall_{a,c_1,c_2} : \phi \wedge \delta_S \wedge \rho_S &\implies \rho_T \\
\forall_{a,c_1,c_2} : \phi \wedge \delta_S \wedge \rho_S &\implies \iota_S = \iota_T.
\end{aligned}$$

The VC Gen tests these conditions by querying an SMT solver for counter-examples, using the negation of the conditions. These queries are

1. $\exists_{a,c_1,c_2} : \phi \wedge \delta_S \wedge \neg\delta_T$
2. $\exists_{a,c_1,c_2} : \phi \wedge \delta_S \wedge \rho_S \wedge \neg\rho_T$
3. $\exists_{a,c_1,c_2} : \phi \wedge \delta_S \wedge \rho_S \wedge \iota_S \neq \iota_T$.

Since an SMT solver can prove that these formulas are unsatisfiable, then no counter-examples exist and therefore the optimization is correct for this type assignment.

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()) {

  C3 = ConstantInt::get(I->getType(),
   C1->getValue() ^ C2->getValue());

  I->replaceAllUsesWith(
    BinaryOperator::CreateAdd(a, C3, "", I));
}
```

Figure 4: 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.

Example with `undef`.

The following simple optimization illustrates the nested quantifiers associated with `undef`:

```
%r = mul %x, undef
   =>
%r = undef
```

There is no precondition, and the source and target are always defined and poison-free, so we need only consider the third correctness condition:

$$\forall_{x,u_2} \exists_{u_1} : x \times u_1 = u_2,$$

where u_1 and u_2 encode to the `undef` values in the source and target, respectively. The corresponding SMT query is $\exists_{x,u_2} \forall_{u_1} : x \times u_1 \neq u_2$, which is satisfiable for $x = 2, u_2 = 1$ (because multiplying by two always yields an even number). Thus, this optimization is incorrect.

3.4 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 3 shows an example.

4. GENERATING C++ FROM Alive

Optimizations specified in Alive can be directly translated into an implementation using the same instruction pattern–matching library that InstCombine uses. The implementation checks whether a code fragment matches the pattern of the source template and whether the precondition holds. If so, it creates the instructions in the target template, replacing variables with their corresponding values from the code fragment. Figure 4 shows an Alive transformation and its corresponding C++ implementation.

Translating a source template.

The code generator uses LLVM's pattern-matching library to create a conditional which tests whether a code fragment matches the source template. For example, `match(I, m_Add(m_Value(b), m_ConstantInt(C2)))` returns true if the LLVM instruction `I` adds a value to a constant, and sets the variables `b` and `C2` to point to its arguments. Matching begins with the root instruction in the source template and recursively matches operands until all non-inputs have been bound.

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.

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 [4] to discharge both typing and refinement constraints. Alive is about 5,200 lines of open-source code.

The number of possible type assignments for a transformation is usually infinite. To ensure termination, Alive considers integer types up to 64 bits.

Refinement constraints are either over the BV or QF_BV (quantified/quantifier-free bitvector) theories. The constraints in Section 3.2 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.

6. EVALUATION

We translated hundreds of peephole optimizations from LLVM into Alive. We verified 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. The Alive-generated C++ code's compilation time and the performance of the resultant code compiled with it is similar to LLVM's unverified InstCombine pass [10].

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. During the initial testing of our prototype, we translated 334 InstCombine transformations to Alive. Of these, eight 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.

Subsequent efforts have used Alive to validate end-to-end transformations and extended the Alive language. These have lead to the discovery of at least fifteen additional bugs.

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 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 5 provides the Alive code and the bug report numbers for a sample of 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

Several LLVM developers use 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

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

```
Name: PR20186
%a = sdiv %X, C
%r = sub 0, %a
  =>
%r = sdiv %X, -C
```

```
Name: PR20189
%B = sub 0, %A
%C = sub nsw %x, %B
  =>
%C = add nsw %x, %A
```

```
Name: PR21242
Pre: isPowerOf2(C1)
%r = mul nsw %x, C1
  =>
%r = shl nsw %x, log2(C1)
```

```
Name: PR21243
Pre: !WillNotOverflowSignedMul(C1, C2)
%Op0 = sdiv %X, C1
%r = sdiv %Op0, C2
  =>
%r = 0
```

```
Name: PR21245
Pre: C2 % (1<<C1) == 0
%s = shl nsw %X, C1
%r = sdiv %s, C2
  =>
%r = sdiv %X, C2/(1<<C1)
```

```
Name: PR21255
%Op0 = lshr %X, C1
%r = udiv %Op0, C2
  =>
%r = udiv %X, C2 << C1
```

```
Name: PR21256
%Op1 = sub 0, %X
%r = srem %Op0, %Op1
  =>
%r = srem %Op0, %X
```

```
Name: PR21274
Pre: isPowerOf2(%Power) && hasOneUse(%Y)
%s = shl %Power, %A
%Y = lshr %s, %B
%r = udiv %X, %Y
  =>
%sub = sub %A, %B
%Y = shl %Power, %sub
%r = udiv %X, %Y
```

Figure 5: A sample of incorrect InstCombine transformations discovered during the development of Alive

performance of one of the SPEC CPU 2000 benchmarks by 3.8%—this is obviously an interesting addition to a compiler. Using Alive, we discovered that the developer's initial and second patches were wrong, 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 6 shows the initially proposed optimization, a counter-example demonstrating its invalidity, and the final precondition for the valid optimization. A recent work has proposed a learning technique for automatically inferring preconditions, which is useful to developers debugging an incorrect optimization [13].

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. Among them, Alive is perhaps most similar to high-level rewrite patterns [6, 11]. Alive differs in its extensive treatment of undefined behavior, which is heavily exploited by LLVM and other aggressive modern compilers, and its ability to generate code that is similar to LLVM's InstCombine pass.

Peephole optimization patterns for a particular ISA can be generated from an ISA specification [3]. In contrast to compiler optimizations, optimized code sequences can be synthesized either with peephole pattern generation or through superoptimization [12, 5, 19, 1].

Optgen [2] 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 [20, 15, 7] 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 [17, 18] where, for each compilation, it is proved that the optimized code refines the unoptimized code. Translation validation suffers from the drawback of requiring proof machinery to execute during every compilation. Alive aims for once-and-for-all proof of correctness of a limited slice of the compiler.

The CompCert [9] compiler for C is an end-to-end verified compiler developed with the interactive proof assistant Coq. Vellvm [21] reuses the memory model from the CompCert development and formalizes the semantics and SSA properties of the LLVM IR to reason about optimizations. Alive's treatment of `undef` values mirrors the treatment in Vellvm. In contrast to Vellvm, Alive handles poison values and automates reasoning with an SMT solver.

8. CONCLUSION

We have shown that an important class of optimizations in LLVM—peephole optimizations—can be formalized in Alive, a new language that specifies optimizations more concisely than C++ code, while also supporting automated proofs of correctness. We designed Alive to resemble LLVM's textual format while also supporting abstraction over types and constant values. 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, as LLVM developers are actively using it. Second, we would like to see a large part of InstCombine replaced with code generated by Alive; we are still working towards that goal.

9. 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.

10. REFERENCES

[1] 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.

```
Pre: isPowerOf2(C1 ^ C2)
%x = add %A, C1
%i = icmp ult %x, C3
%y = add %A, C2
%j = icmp ult %y, C3
%r = or %i, %j
  =>
%and = and %A, ~(C1 ^ C2)
%lhs = add %and, umax(C1, C2)
%r = icmp ult %lhs, C3
```

(a)

```
ERROR: Mismatch in values of i1 %r

Example:
%A i4 = 0x5 (5)
C1 i4 = 0x3 (3)
C3 i4 = 0x7 (7)
C2 i4 = 0x1 (1)
%x i4 = 0x8 (8, -8)
%i i1 = 0x0 (0)
%y i4 = 0x6 (6)
%j i1 = 0x1 (1, -1)
%and i4 = 0x5 (5)
%lhs i4 = 0x8 (8, -8)
Source value: 0x1 (1, -1)
Target value: 0x0 (0)
```

(b)

```
Pre: C1 u> C3 &&
     C2 u> C3 &&
     isPowerOf2(C1 ^ C2) &&
     isPowerOf2(-C2 ^ -C1) &&
     -C2 ^ -C1 == (C3-C2) ^ (C3-
C1) &&
     abs(C1-C2) u> C3
```

(c)

Figure 6: (a) A peephole optimization proposed by the developer. (b) A counter-example found by Alive. (c) A precondition that makes the optimization valid.

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

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

[4] 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.

[5] 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.

[6] 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*, 2009.

[7] 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*, 2014.

[8] 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 *Proc. of the 38th annual ACM SIGPLAN conference on Programming Language Design and Implementation*, 2017.

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

[10] N. P. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr. Provably correct peephole optimizations with Alive. In *Proc. of the 36th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 2015.

[11] N. P. Lopes and J. Monteiro. Automatic equivalence checking of programs with uninterpreted functions and integer arithmetic. *International Journal on Software Tools for Technology Transfer*, 18(4):359–374, 2016.

[12] 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)*, 1987.

[13] D. Menendez and S. Nagarakatte. Alive-infer: Data-driven precondition inference for peephole optimizations in LLVM. In *Proc. of the 38th annual ACM SIGPLAN conference on Programming Language Design and Implementation*, 2017.

[14] D. Menendez, S. Nagarakatte, and A. Gupta. Alive-FP: Automated verification of floating point based peephole optimizations in LLVM. In *Proc. of the 23rd International Symposium on Static Analysis*, 2016.

[15] 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*, 2013.

[16] A. Nötzli and F. Brown. LifeJacket: Verifying precise floating-point optimizations in LLVM. In *Proc. of the 5th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, 2016.

[17] 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.

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

[19] 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.

[20] 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*, 2011.

[21] 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*, 2012.